MorphGL: Towards Hardware-Adaptive Training for Graph Neural Networks on Giant Graphs

Xin Zhang
HKUST
Hong Kong, China
xzhanggb@ust.hk

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

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

Lei Chen
HKUST(GZ)
Guangzhou, China
leichen@ust.hk

***Abstract*—Graph Neural Networks (GNNs) have achieved great success in many applications and mini-batch training has become the de-facto way to train GNNs on large-scale graphs. When developing mini-batch GNN training systems on the CPU-GPU platform, existing works adopt *static* workload-processor binding strategy, which fixes the training workload distribution over CPU and GPU, and encounter two problems: (i) inconsistent training efficiency across different machines, and (ii) resource under-utilization on common hardware. To address the problems, we develop a novel hardware-adaptive GNN training system named MorphGL with a *dynamic* workload-processor binding strategy. MorphGL adaptively partitions the mini-batch preparation workload to the CPU and the GPU, ensuring that the workload distribution matches the relative computation capabilities of the heterogeneous processors. MorphGL further employs a Dual-Buffer Scheduling algorithm to schedule the training phases on CPU, PCIe, and GPU, maximizing the resource utilization. Extensive experiments on three real large graphs with billions of edges and three machines with different CPU&GPU configurations demonstrate that MorphGL consistently outperforms the state-of-the-art GNN training systems, achieving up to 2.76x and 2.20x speedup than SALIENT and DUCATI, respectively.**

***Index Terms*—Graph Neural Network, mini-batch training, hardware adaptive, workload scheduling, large-scale graph**

I. Introduction

In recent years, Graph Neural Networks (GNNs) 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 [1]–[3], biological molecules property prediction [4]–[6], and weather forecasting [7], [8]. Similar to other deep learning models [9], [10], GNNs typically employ mini-batch training 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; 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 [11]–[19]. *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 [14], [16], [20], [21] 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 [14], [16], [20], reducing redundancy by reusing previously sampled neighbors [16], and optimizing node feature selection with improved data structures [14], [16]. The systems in Category II [12], [15], [17], [18] 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 [18], DSP [17], and Quiver [15] propose to cache frequently accessed graph data in the GPU memory to reduce data transfer volume over PCIe.

Unfortunately, we find that both categories of GNN training systems suffer from poor adaptivity across various real-world hardware configurations. To demonstrate this, we analyze the normalized batching speed of SALIENT [14] and DUCATI [18], representing Category I and Category II systems respectively, on machines with varying CPU cores to GPU ratios. According to the results in Figure 1, we have two key observations. (1) *Efficiency variability*: both DUCATI and SALIENT show inconsistent efficiency across different hardware configurations. Their training efficiency can degrade by up to 3.5 times when the underlying hardware changes. Moreover, the comparative efficiency of the two systems fluctuates considerably, depending on the specific hardware in use. (2) *Suboptimal efficiency on common hardware*[1]: on typical hardware setups, both SALIENT and DUCATI fall short of achieving optimal efficiency due to insufficient resource utilization. That is, one processor becomes a bottleneck while the other remains underutilized[1].

The inconsistent and suboptimal efficiency in current GNN

[1]Details and definitions will be given in Section III.

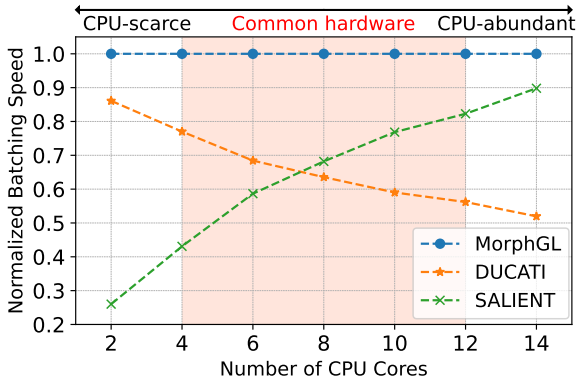


Fig. 1: Normalized batching speed of DUCATI, SALIENT and our proposed MorphGL, on diverse machines represented by different number of CPU cores per GPU.

training systems stems from their inflexible approach to distributing the mini-batch preparation workload. Existing systems typically adopt a *static* workload-processor binding strategy, assigning the whole mini-batch preparation task to either CPU or GPU exclusively. This rigid approach leads to the mismatch between preparation workload distribution and processors' computing resources, which eventually causes inconsistent and suboptimal training efficiency (see detailed discussion in Section III). Inspired by this, we propose to develop a *dynamic* workload-processor binding strategy that accommodates the preparation workload to the running hardware adaptively. Then we can achieve good utilization of both processors for diverse machine setups and address the aforementioned two limitations of the existing systems.

In this paper, we introduce MorphGL, a novel hardware-adaptive GNN training system. The system consists of two major components, namely the Partitioner and the Scheduler. To conquer the efficiency variability, the Partitioner adaptively partitions the total mini-batch preparation workload so that the workload distribution always matches the processors' setup of running hardware. To maximize resource utilization and training speed, the Scheduler devises asynchronized execution plans that overlap workloads on CPU, PCIe, and GPU as much as possible. At a high level, MorphGL adaptively adjusts several operational aspects, including batching workload distribution, data transferring techniques, and scheduling patterns. The adjustments are made based on the real-time system performance, allowing MorphGL to optimize its operations for each specific hardware environment automatically.

We formulate the hardware-adaptive training problem, which encompasses two challenges in workload partitioning and scheduling, and prove the problem is NP-hard. The Partitioner and the Scheduler of MorphGL solve the problem in an iterative way. For the Partitioner, the key challenge is: **how to effectively divide the mini-batch preparation workload on diverse machines**? The Partitioner takes in statistics collected with offline profiling on the given hardware and outputs an initial workload partitioning plan based on necessary relaxations. The Partitioner later finetunes the initial plan with the downstream feedback. Then, the Scheduler addresses the

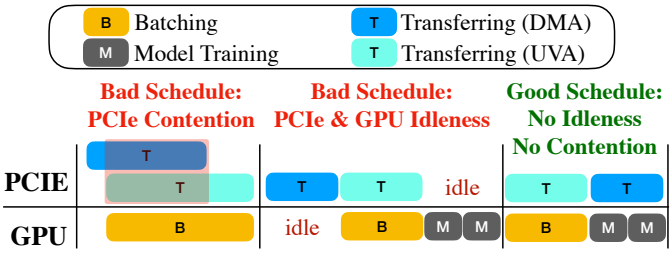


Fig. 2: An illustration of resource contention and idleness with different schedules.

second challenge: **how to optimally coordinate batching, data transferring, and model training across CPU, PCIe, and GPU**? As shown in Figure 2, inappropriate scheduling would result in resource contention and idleness on PCIe and GPU. More detailed schedules will be given in Section III. The Scheduler uses the Dual-Buffer Scheduling algorithm which maintains two buffers, one residing in main memory and the other in device memory, and carefully arranges the moment of transferring and model training to avoid both resource contention and idleness. The Scheduler also provides downstream feedback on workload partitioning that helps the Partitioner refine the old partitioning plan and output a new one. In this way, the Partitioner and the Scheduler work together and iteratively optimize the problem until they converge on the final solution. We also theoretically establish the approximation guarantee of the proposed solution.

We conduct extensive experiments on three machines with enterprise/consumer-level GPUs and different CPU setups. Experimental results on three mainstream GNNs and three large-scale graphs with billions of edges demonstrate that MorphGL adapts to diverse machines and consistently outperforms state-of-the-art Category I and Category II systems (i.e., SALIENT and DUCATI) by up to 2.76x and 2.20x, respectively.

To summarize, the major contributions of the paper are:

- We propose a hardware-adaptive GNN training framework named MorphGL with the *dynamic* workload-processor binding strategy, which adaptively distributes mini-batch preparation workload to diverse processor setups and carefully schedules workloads on CPU, PCIe, and GPU to avoid both resource contention and idleness.
- We prove the hardware-adaptive batching and scheduling problem is NP-hard and propose a tunable workload partitioning 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 three hardware settings with the most common CPU:GPU ratios. Experimental results demonstrate that MorphGL can achieve universal speedup against the baselines in diverse settings.

II. Background

In this section, we provide the background on GNN training and review the existing works that are closely related.

A. 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 [22], 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 handle 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 GPU memory.

To cope with large-scale graphs, GraphSAGE [4] proposes a mini-batch training approach consisting of three major phases. (1) In the **batching phase**, we prepare a mini-batch 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 phase**, we transfer the prepared mini-batch from main memory to device memory via the PCIe. (3) In the **model training phase**, 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 device memory. Only the GNN model's weights, activations, gradients, and optimizer states, reside in device memory. These components typically occupy less than one gigabyte in total. Hence, mainstream GNN models [22], [23], including those leading the OGB leaderboard [24] and the key examples in DGL and PyG [20], [21] frameworks, employ single-GPU training. In this paper, we also focus on the single-GPU mini-batch GNN training setting.

B. Category I Systems with CPU-based Preparation

Category I systems [14], [16], [20], [21] advocate binding mini-batch preparation workload to CPU. PyG (CPU) [21], which represents the CPU-batching mode of PyG, and DGL (CPU) [20], two of the most popular GNN Libraries, provide multiprocessing-based batching and asynchronized mini-batch transferring. However, the additional overhead of data sharing between multiple processes harms the training efficiency. SALIENT [14] and MariusGNN [16] propose multithreading-based batching to avoid the overhead and achieve better 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 over 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, SALIENT only achieves 26% GPU utilization on a typical commercial server to be introduced in Section V-D due to the bottleneck on CPU batching.

TABLE I: 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 IV.

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

C. Category II Systems with GPU-based Preparation

In the presence of limited CPU resources, Category II Systems [12], [13], [15], [18], [20], [21], [25] propose to bind all or the majority of the batching workload to GPU instead. Specifically, PyG (GPU) [21], DGL (GPU) [20], GNNLab [13] and NextDoor [25] 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 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, PyTorch-Direct [12], Quiver [15], DSP [17], and DUCATI [18] 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 also 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, these systems cannot leverage the CPU computing power when applied to machines with considerable CPU resources. For example, DUCATI wastes 87% of CPU resources when training on the typical commercial server.

Some recent works focus on enhancing the heterogeneous hardware utilization in query processing with the morsel-driven parallelism [26]–[28]. However, the problems faced by them and our `MorphGL` are different and their solutions cannot be applied here. Specifically, these works split one query's input table into morsels with tunable sizes for parallel processing while `MorphGL` schedules hundreds of mini-batches with fixed sizes for independent model training. Besides, their solutions do not consider the UVA technique, which occupies GPU and PCIe simultaneously during execution. Therefore, their solutions cannot properly schedule the mini-batch GNN training to avoid hardware contention and idleness.

Summary. We summarize key designs of existing works in Table I. Due to the *static* workload-processor binding strategy for mini-batch preparation, existing systems are not adaptive regarding different machines and suffer from resource under-utilization on common hardware.

TABLE II: Distribution of average CPU cores per GPU of 141 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%

III. Motivation

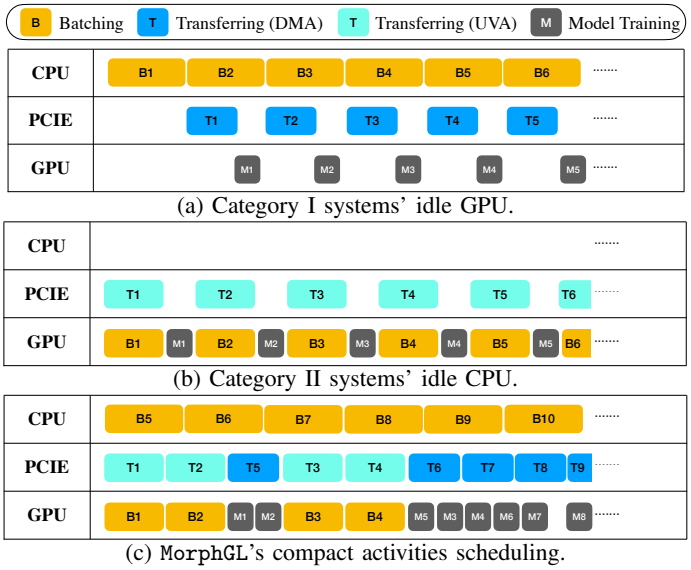
In this section, we provide relevant empirical results to illustrate the limitations of the existing systems and present the key observations that motivate the development of MorphGL.

Common hardware. In practice, it is desirable to ensure a GNN training system in accordance 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,[2] of all 141 GPU instances to date on the three largest cloud providers, namely Amazon Web Service (AWS) [29], Microsoft Azure (Azure) [30], and Google Cloud Platform (GCP) [31]. In Table II, 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 3, we provide the simplified training profiles of Category I and Category II systems on Machine A, a server with the most common CPU:GPU ratio to be introduced in Section V-A.We find that the performance bottleneck is the capability of the processor, either the CPU or the GPU, whichever undertakes the batching phase. Specifically, we find two resulting symptoms of suboptimal efficiency: (1) Category I systems are faced with the slow CPU batching and the idle GPU symptom as shown in Figure 3a and (2) Category II systems suffer from prolonged GPU batching and the idle CPU symptom as shown in Figure 3b. PCIe bandwidth is underutilized by both kinds of systems due to the bottleneck in the batching phase. 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 GNN training systems and common hardware, there is an opportunity to build a new GNN training system that is adaptive to common hardware. That is, by breaking down the mini-batch preparation workloads to both CPU and GPU and carefully scheduling workloads on CPU, PCIe, and GPU, we can avoid the resource under-utilization problem of existing systems and achieve faster training speed as shown in Figure 3c. However, such an attempt is faced with two key challenges as introduced in Section I, namely how to divide the mini-batch preparation workload and how to coordinate the execution of workloads on CPU, PCIe, and GPU. We will illustrate in detail how we conquer these two challenges in Section IV.

[2](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 influencing specifications will also be taken into consideration by our system through profiling as will be introduced in the next section.



(a) Category I systems' idle GPU.

(b) Category II systems' idle CPU.

(c) MorphGL's compact activities scheduling.

Fig. 3: Simplified illustration of CPU/GPU batching, DMA/UVA transferring, and model training phases' execution on the three hardware. X-axis: time. Y-axis: major activities on the three hardware. The length of each rectangle represents the duration of a certain activity of one mini-batch on the corresponding hardware.

IV. MorphGL: Hardware-Adaptive GNN Training

In this section, we first formulate the *Hardware-Adaptive Batching and Scheduling Problem* in Section IV-A and prove the problem is NP-hard. Then we present the design and implementation of our hardware-adaptive GNN training system MorphGL in Section IV-B. MorphGL consists of the Profiler to collect essential profiling information (Section IV-B1), the Partitioner for partitioning the batching workload among the CPU and the GPU (Section IV-B2), the Scheduler for scheduling the partitioned workload on CPU, PCIe, and GPU accordingly (Section IV-B3), and the Executor that performs the final training given the final scheduling plan (Section IV-B4).

A. Problem Formulation

In this section, we first introduce necessary notations and the constraints for the *Hardware-Adaptive 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 phases 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 phase X_i as $\Gamma(X_i)$ and the duration of X_i as $Dur(X_i)$. We denote the hardware requirement of each training phase X_i as $\zeta(X_i)$, where $X_i \in \{B_i, T_i, M_i\}$:

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

To avoid resource contention, we have **hardware occupation constraints** $\Psi^{hw}(\Gamma(X_i), r_i)$ given in Constraints 2, namely the same hardware can only process one training phase at one timestamp:

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

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 phase T_i starts only after its batching phase B_i is finished. For the mini-batch following Route 1, its transferring phase and batching phase are executed simultaneously with GPU and PCIe due to the mechanism of UVA. For all mini-batches, the model training phase M_i cannot precede the end of the corresponding transferring phase T_i since the model training cannot start before the training data is ready on GPU:

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

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:

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

Definition 1 (Hardware-Adaptive 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 phase 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:

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

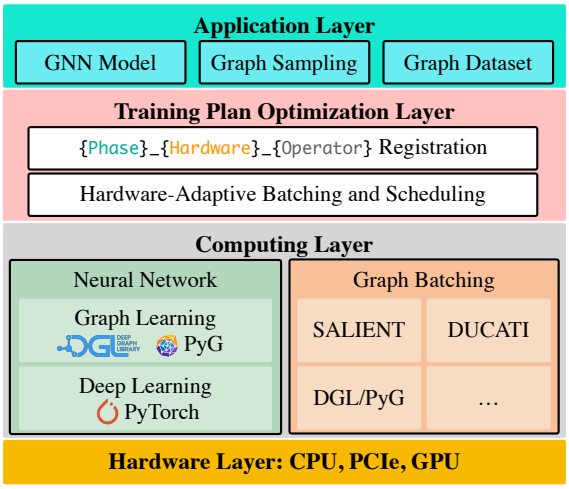


Fig. 4: Abstraction layers of GNN training systems. We propose the novel Training Plan Optimization Layer where MorphGL improves the workload partitioning and scheduling.

Theorem 1. The optimization problem stated 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 [33], 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 0 and apply the scheduling result to the JSSP problem by treating CPU, PCIe, and GPU as the three machines, n mini-batches as n jobs to complete, and different training phases 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 [33], 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 partitioning and training phases scheduling, to minimize C. We will illustrate the details in the next section.

B. Design and Implementation of MorphGL

We adhere to two principles when designing MorphGL. First, from the implementation perspective, we emphasize hardware generality by designing MorphGL in a modular way. All required operators of MorphGL should be easy to upgrade or substitute on new hardware. Second, from the application perspective, we advocate user-friendliness by encapsulating the

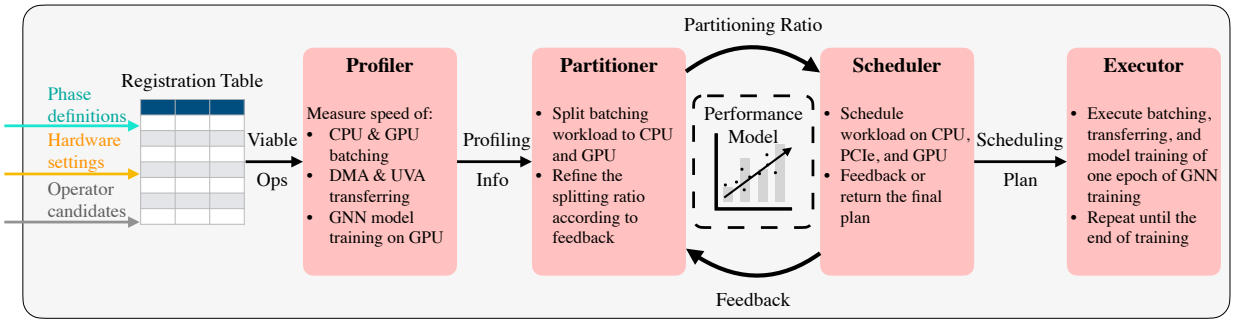


Fig. 5: Overall architecture of `MorphGL`.

TABLE III: `{Phase}_{Hardware}_{Operator}` Table.

Phase	Hardware	Operator
Batching	CPU	SALIENT.fast_sampler
	GPU+PCIe	DUCATI.NeighborSampler
	NPU+PCIe	To be developed
Transferring	PCIe	torch.to
	GPU+PCIe	dgl.gather_pinned_tensor_rows
	NPU+PCIe	To be developed
Training	GPU	dgl.nn
	NPU	To be developed

low-level workload partitioning and scheduling mechanism. Users only need to provide high-level configurations such as sampling arguments and model definitions to `MorphGL`.

Following the principles, we provide the general abstraction layers of GNN training systems in Figure 4. There are four layers involved, namely the Hardware Layer for machine setups, the Computing Layer for neural network and graph batching computation, the Training Plan Optimization (TPO) Layer we proposed for workload partitioning and scheduling, and the Application Layer for training tasks. In the novel TPO Layer, we (1) propose the `{Phase}_{Hardware}_{Operator}` abstraction for hardware generality and (2) support the hardware-adaptive batching and scheduling with `MorphGL`. Specifically, we maintain a registration table as in Table III. For each training phase, we bind the phase to viable operators for later use in `MorphGL`. For example, Given the batching phase, we first lookup the registration table and get two viable operators, namely the CPU batching operator from SALIENT and the GPU batching operator from DUCATI. Then, we provide the two viable operators as input to `MorphGL` for profiling, workload partitioning, and scheduling. With such abstraction, once necessary operators are supported on other accelerators such as NPU, we can update the registration table and `MorphGL` can be applied as well.

We present the overall architecture of `MorphGL` in Figure 5. Specifically, technical contributions of `MorphGL` are posed in the novel TPO Layer with four major components, namely the Profiler, the Partitioner, the Scheduler, and the Executor. Given the viable operators, The Profiler captures details of the running hardware and software configurations with profiling. The Partitioner is responsible for dividing the mini-batch preparation workload to CPU and GPU. The Scheduler arranges the execution of all training phases on CPU, PCIe, and GPU. The Executor performs GNN model training with

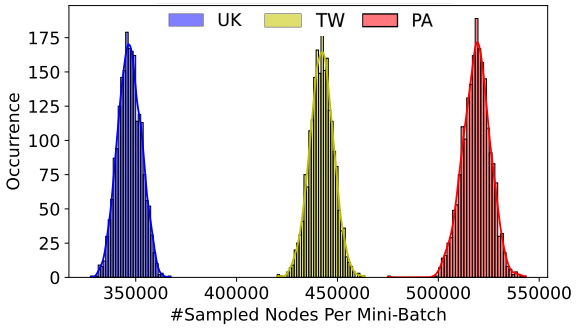


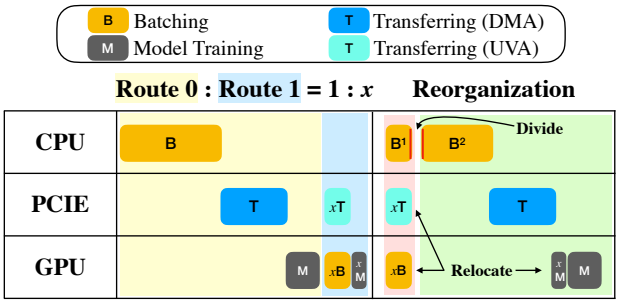
Fig. 6: Distribution of number of sampled K-hop neighbor nodes per mini-batch of the three datasets.

the final partitioning and scheduling plan.

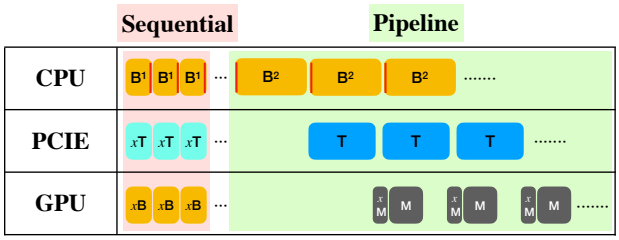
We next elaborate the details for each of the components, and present how to solve the *hardware-adaptive batching and scheduling problem* in an iterative way. Briefly, given the profiling results of the Profiler, the Partitioner bootstraps the iteration by applying necessary relaxations to find a crude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

1) Profiler: The Profiler extracts essential information from the running hardware and training configurations. The smallest indivisible unit in partitioning and scheduling is a phase (among CPU/GPU batching, DMA/UVA transferring, and GPU model training) of a mini-batch. It is the phase duration that directly influences the workload partitioning and scheduling decisions. Instead of collecting indirect specifications and building a cost model to predict the phase duration, which inevitably introduces additional overhead and errors, we directly measure the duration of training phases with profiling.

Specifically, as shown in Figure 6, the total number of sampled nodes in mini-batches, which determines the execution duration of training phases, 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



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



(b) Optimal scheduling under relaxations.

Fig. 7: Initial partition plan derivation. X-axis: time. Y-axis: major activities on the three hardware. Activity duration (rectangle's length) is scaled by a factor of x for Route 1.

phases 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 phase for k steps, where k is a hyperparameter, and calculate the mean duration as $T_{phase} = \frac{1}{k}\sum_{i=1}^{k} Dur(phase), phase \in \{CBT, GBT, DMA, UVA, Model\}$. This information is sufficient for us to guide the systems' efficiency tuning.

2) Partitioner: The Partitioner takes in the profiling information from the Profiler and outputs the workload partitioning ratio x, representing that the number of mini-batches prepared by CPU and GPU is $1 : x$. The Partitioner determines the partitioning of mini-batch preparation workload over CPU and GPU in a tunable way. The Partitioner first obtains an initial guess by relaxing some constraints. Then the Partitioner refines the initial guess by incorporating the downstream feedback provided by the Scheduler. We illustrate the initialization step and the finetuning step of the Partitioner in detail as follows.

Initial Partition. We bootstrap the iteration and derive the first partition plan by relaxing the *buffer size constraints* and assume that (1) we have unlimited buffers in both main memory and device memory and (2) the batching workload is infinitely divisible. Considering the batching phase, we assign all the mini-batches to CPU (Route 0) and GPU (Route 1) with a ratio of $1 : x$ as shown in the left of Figure 7a. Then, we divide the CPU batching workload of Route 0 and relocate the workload of Route 1 so that the reorganized route has two stages in the pink and green background as shown in the right of Figure 7a. The pink stage has three equal-length workloads on CPU, PCIe, and GPU. Then, we arrange an optimal execution which comprises a part of sequential execution with pink background and a part of pipeline execution with green background in Figure 7b, both of which make the most of

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';

$buf\_c \leftarrow$ deque(size=cbs) ▷ CPU buffer
$buf\_g \leftarrow$ deque(size=gbs) ▷ GPU buffer
$opt \leftarrow optim\_c(M$.params(), ...) ▷ optimizer
def train_batch(batch, label, event):
  s_0.wait_event(event) ▷ ensure async transfer is finished before training
  loss = loss_func(M(batch), label)
  opt, loss $\cdots$ ▷ calculate gradients and update params
while true **do**
  if $iter\_g$.end() **or** $iter\_c$.end() **then**
    $\cdots$ ▷ dispose of remaining mini-batches
    break
  end if
  if buf_c.not_full() **and** buf_g.not_full() **then**
    buffers_filling($buf\_c, buf\_g, iter\_c, iter\_g$)
  else if buf_c.not_full() **or** buf_g.not_full() **then**
    buffers_blocking($buf\_c, buf\_g, iter\_c, iter\_g$)
  else
    buffers_flushing(buf_c, buf_g)
  end if
end while
return M'

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 trivial to minimize.

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

Under the relaxations, we can simplify the NP-hard optimization on C to an easy optimization problem on C', which we minimize and obtain the initial partitioning ratio x. However, in practice we often have limited buffers and the training phase is not divisible. Hence, we rectify such a crude partitioning plan with downstream feedback in the finetuning step.

Finetuning Step. The above initial guess disobeys some constraints which inevitably leads to suboptimal scheduling. Therefore, we need to tune the partitioning plan with more downstream information. As will be introduced later in Section IV-B3, the Scheduler will provide feedback information about whether the current partitioning plan places too much batching workload on CPU or GPU side. With such feedback, the Partitioner 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-

Algorithm 2 The buffers_filling function.

Input: CPU buffer buf_c; GPU buffer buf_g; CPU batching Iterator $iter\_c$; GPU batching Iterator $iter\_g$.

```
gb, gl = iter_g.next()
buf_g.append((gb, gl, None))
cb, cl = iter_c.next(non_blocking=true)
buf_c.append((cb, cl))
```

batches distributed to GPU will be provided to the Partitioner. Then the Partitioner adjusts the partitioning plan accordingly and provides the updated plan to the Scheduler for the next round of evaluation.

3) Scheduler: Given the partitioning ratio, the Scheduler devises a schedule plan that arranges the execution of all training phases on CPU, PCIe, and GPU. Depending on the internal evaluation, the Scheduler either (1) provides feedback to the Partitioner to finetune the current partitioning plan, 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 main memory holding mini-batches generated by the CPU and the **GPU buffer** in 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 partitioning ratio between CPU and GPU as we introduced earlier in the Initial Partition 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 need to block the mini-batch generation on the buffer-full hardware and wait

Algorithm 3 The buffers_blocking function.

Input: CPU buffer buf_c; GPU buffer buf_g; CPU batching Iterator $iter\_c$; GPU batching Iterator $iter\_g$.

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

for the other buffer[3]. 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 Partitioner. If the majority of the blocking records are due to busy GPU and idle CPU, it means that the Partitioner assigns too much batching workload to GPU in the current partitioning plan. Therefore, we provide a feedback to the Partitioner on decreasing the number of mini-batches distributed to GPU. With the repetition of such feedback and re-partitioning, 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-partitioning the batching workload anymore, which marks the convergence of the iterative feedback and re-partitioning 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 GNN 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 this period, CPU-batcher is preparing mini-batches for the next *overlap*. As described, we leverage separated CUDA streams and two buffers for asynchronous execution and keep all the three hardware busy in the *buffers flushing phase*.

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 Partitioner can partition the batching workload ideally and shorten the *buffers blocking phase* as much as possible. Then,

[3]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.

```
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)
        ce = s_1.record_event()        ▷ async transfer cpu batch on s_1
        buf_g.append((cb,cl,ce))
    end if
    train_batch(gb, gl, ge)            ▷ async train on s_0
endwhile
```

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 IV-B1, 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 Partitioner and predicts the training time under the current partitioning plan and scheduling plan.

Extreme Cases We find that in some rare cases there already exists a trivial optimal training plan, namely the naive 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 naive 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 [34]. Although we haven't seen such rare contexts in our experiments, `MorphGL` still supports the naive 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 Partitioner and the Scheduler has an approximation rate of $3 + \frac{Bdw_p}{Bdw_g}$ for the problem in Definition 1.

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) Executor: The Executor absorbs the final scheduling plan and other training configurations and trains the GNN model for a user-defined number of epochs. Listing 1 provides the APIs and the example usage of `MorphGL`. The user-defined training context information such as software and hardware settings are processed by the `Profiler` API, which obtains and stores the batching/transferring/training speed information. Then the `Partitioner` and the `Scheduler` APIs are called in a loop until the iterative partitioning and scheduling converge. Finally, we train with the `Executor` API for a number of epochs times.

```
import MorphGL
# instantiate samplers and model with registration
# table and user-provided information
input_dict = {}
input_dict['device'] = ...
input_dict['CPU_sampler'] = ...
input_dict['GPU_sampler'] = ...
input_dict['model'] = ...
# Profiler measures the speed and stores them
MorphGL.Profiler(input_dict)
# Partitioner and Scheduler
part_plan, feedback = None, None
while True:
    part_plan = MorphGL.Partitioner(
            feedback, part_plan)
    feed_back, sched_plan, converge =
            MorphGL.Scheduler(part_plan)
    if converge:
        break
# Executor
For e in range(num_epochs):
    MorphGL.Executor(input_dict, sched_plan)
```

Code Listing 1: APIs of `MorphGL`.

V. Experiments

We implemented `MorphGL` with 1.3k lines of code based on SALIENT, DUCATI, and DGL (v0.8) and open source it on the github[4]. We run the experiments with different hardware, GNN models, and datasets in the following sections.

It is noteworthy that `MorphGL` does not change the content of mini-batches and model implementations 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.

[4]https://github.com/CSLabor/morphGL

TABLE IV: 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

A. Experimental Setup

Hardware. As stated in Section III, 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 [35], 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 [29], 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. We use another two machines to verify the the influence of more device memory and the hardware generality in Section V-G and Section V-H. Specifically, we choose a machine with NVIDIA V100 (32GB) for the former and a machine with the Huawei Ascend 910B NPU, a leading NPU in the Chinese market for the later. In summary, we test `MorphGL` and baselines 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.
- **Machine E**: Huawei Ascend 910B (32G), six/eight/twe-lve 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 IV. Specifically, we use a web graph: UK-2006-05 (UK) [36], a social graph: Twitter (TW) [37], and an academic citation graph in Open Graph Benchmark (OGB) [24]: Ogbn-Papers100M (PA). Similar to previous works [13], [14], 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 [24], we add bidirectional edges for the topology data of all datasets before training. OGB provides official training sets for PA. For the other two datasets that do not provide a training set, we follow the practice of previous works [11], [13] and randomly sample 1 percent of all vertices as the training set. Following the

TABLE V: Epoch training time comparison (unit: s). The red figures denote the maximum speedup of MorphGL.

	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 ×

previous work [14], we store the node features in float16 and the topology data in int64 CSC format.

GNN Models. We do experiments with three mainstream GNN models, namely GCN [22], GraphSAGE [4], and GAT [38]. Following previous works [11], [12], 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 [22], [38] and previous works [14], [18]. In terms of computing complexity, 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 II, 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.

B. Main Results: Epoch Time Comparison

We compare the training time of one epoch of all systems and present the results in Table V. Overall, MorphGL outperforms SALIENT and DUCATI by up to 2.76 times (on average 1.55 times) and 2.20 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 hardware-adaptive design. As introduced in Section II-B, 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 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 hardware-adaptive design.

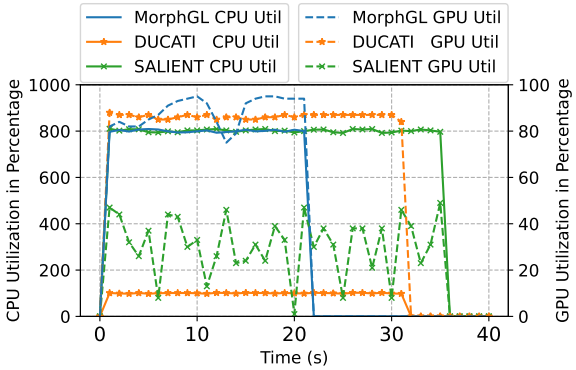


Fig. 8: Detailed CPU and GPU utilization for the three systems in one training epoch (UK dataset, GraphSAGE, Machine B)

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 partitioning. For example, for SALIENT experiments on Machine C, when training GCN instead of GAT, the model training on GPU is less time-

TABLE VI: 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 VII: Scheduling plans of MorphGL.

	Machine A	Machine B	Machine C
	(CPU buffer, GPU buffer) sizes in #batches		
PA	(15, 10)	(5, 10)	(11, 10)
TW	(20, 10)	(8, 10)	(24, 10)
UK	(30, 10)	(12, 10)	(28, 10)

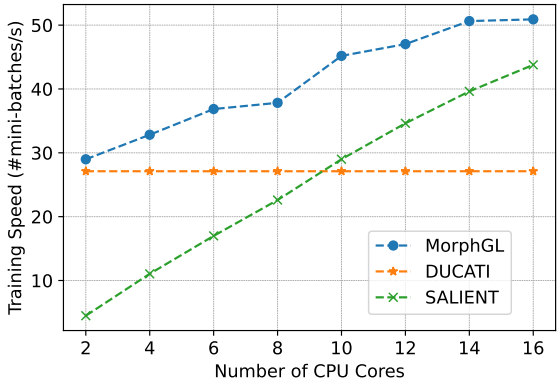


Fig. 9: The training speed comparison of three systems with different numbers of CPU cores per RTX3090.

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 dataset's locality. PA has a manually labeled training set containing high-degree nodes with dense connection [24] and, thus, strong locality [13], [18]. 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.

C. 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 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.

D. Hardware Utilization

We compare the hardware utilization of CPU, PCIe, and GPU of the three systems and present results in Table VI. 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 partitioning and scheduling. In a nutshell, MorphGL improves the utilization of hardware, resulting in the speedup of GNN training as we observed in Section V-B.

E. Scheduling and Partitioning 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 VII. Overall, the proportion of the CPU batching workload to total batching workload ranges from 33% to 75%, which demonstrates that

TABLE VIII: 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

TABLE IX: Epoch training time (unit: s) comparison on Machine D with different device memory sizes. T_{GBT}: Average GPU Batching phase 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

MorphGL can assign batching workload to CPU and GPU adaptively in a hardware-adaptive way. 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.

F. Preprocessing Time

We put together the time of the Profiler, the Partitioner, and the Scheduler as the preprocessing time and compare it to the epoch training time as in Table VIII. We obtain the results with the GraphSAGE model. The results are similar in other settings. As shown, the preprocessing time is usually less than five epoch of training. As pointed out in previous works [13], [39], [40], 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.

G. 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 IX. 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 [18], 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 [41], 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.

TABLE X: Epoch training time (unit: s) comparison on Machine E with different CPU:NPU ratios. Last column: scheduling plans (CPU buffer size, NPU buffer size) of MorphGL.

CPU:NPU	SALIENT	DUCATI	MorphGL	
12:1	69.53 (1.45x)	76.43 (1.59x)	48.03 (1.00x)	(10, 8)
8:1	102.51 (1.83x)	76.43 (1.36x)	56.07 (1.00x)	(7, 10)
6:1	135.32 (2.20x)	76.43 (1.24x)	61.61 (1.00x)	(5, 10)

H. Support of Other Accelerators Like NPU

In Table X, we present the epoch training time and final scheduling plans of the three systems on Machine E equipped with the Huawei Ascend NPU. As shown, MorphGL adaptively partitions the batching workload and achieves universal speedup in different CPU and NPU settings, which demonstrates the generality of MorphGL.

Note that since DGL, SALIENT, and DUCATI are all incompatible with the Huawei NPU platform, we develop dummy operators as the substitution for them. The GPU to NPU transplant of involved operators, such as the UVA transferring, dual-cache batching, and graph convolution operators, requires months of teamwork and is out of the workload partitioning and scheduling scope of this work. MorphGL is hardware-neutral and can be applied to NPU as well as other accelerators once related GNN operators are ready.

VI. Conclusion

In this paper, we present MorphGL, a hardware-adaptive GNN training system to speed up mini-batch GNN training on billion-scale graphs. We identify two problems of existing systems, namely the inconsistent training efficiency on different machines and the resource under-utilization on common hardware. MorphGL addresses the two problems by adaptively partitioning the batching workload and scheduling training phases on CPU, GPU, and PCIe to minimize the training time. We prove the hardware-adaptive batching and scheduling problem is NP-hard, and propose an iterative method, which includes a tunable partitioning scheme and the `Dual-Buffer Scheduling` algorithm, with a theoretical bound to solve it. We evaluate the performance of MorphGL on three common machines, three sizes of GNN models, and three billion-scale datasets. Experimental results show that MorphGL has strong hardware adaptivity and consistently outperforms the state-of-the-art GNN training systems, namely SALIENT and DUCATI, by up to 2.76x and 2.20x speedup on training time.

References

[1] L. Yang, Z. Liu, Y. Dou, J. Ma, and P. S. Yu, "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*, 2021, pp. 2141–2145.

[2] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui, "Graph neural networks in recommender systems: a survey," *ACM Computing Surveys*, 2020.

[3] X. Li, Y. Shen, and L. Chen, "Mcore: Multi-agent collaborative learning for knowledge-graph-enhanced recommendation," in *2021 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2021, pp. 330–339.

[4] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[5] K. Yang, K. Swanson, W. Jin, C. Coley, P. Eiden, H. Gao, A. Guzman-Perez, T. Hopper, B. Kelley, M. Mathea *et al.*, "Analyzing learned molecular representations for property prediction," *Journal of chemical information and modeling*, vol. 59, no. 8, pp. 3370–3388, 2019.

[6] H. Stärk, D. Beaini, G. Corso, P. Tossou, C. Dallago, S. Günnemann, and P. Liò, "3d infomax improves gnns for molecular property prediction," in *International Conference on Machine Learning*. PMLR, 2022, pp. 20 479–20 502.

[7] J. Fan, J. Bai, Z. Li, A. Ortiz-Bobea, and C. P. Gomes, "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, no. 11, 2022, pp. 11 873–11 881.

[8] J. Li, Y. Shen, L. Chen, and C. W. W. Ng, "Ssin: Self-supervised learning for rainfall spatial interpolation," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–21, 2023.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[11] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "Pagraph: Scaling gnn training on large graphs via computation-aware caching," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 401–415.

[12] S. W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W.-m. Hwu, "Pytorch-direct: Enabling gpu centric data access for very large graph neural network training with irregular accesses," *arXiv preprint arXiv:2101.07956*, 2021.

[13] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou, "Gnnlab: a factored system for sample-based gnn training over gpus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 417–434.

[14] T. Kaler, N. Stathas, A. Ouyang, A.-S. Iliopoulos, T. Schardl, C. E. Leiserson, and J. Chen, "Accelerating training and inference of graph neural networks with fast sampling and pipelining," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 172–189, 2022.

[15] Z. Tan, X. Yuan, and C. H. et. al, "Quiver: Supporting gpus for low-latency, high-throughput gnn serving with workload awareness," 2023.

[16] R. Waleffe, J. Mohoney, T. Rekatsinas, and S. Venkataraman, "Mariusgnn: Resource-efficient out-of-core training of graph neural networks," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 144–161.

[17] Z. Cai, Q. Zhou, X. Yan, D. Zheng, X. Song, C. Zheng, J. Cheng, and G. Karypis, "Dsp: Efficient gnn training with multiple gpus," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 392–404.

[18] X. Zhang, Y. Shen, Y. Shao, and L. Chen, "Ducati: A dual-cache training system for graph neural networks on giant graphs with the gpu," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–24, 2023.

[19] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo, "{BGL}:{GPU-Efficient}{GNN} training by optimizing graph data {I/O} and preprocessing," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 103–118.

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

[21] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[22] M. Welling and T. N. Kipf, "Semi-supervised classification with graph convolutional networks," in *ICLR*, 2017.

[23] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graphsaint: Graph sampling based inductive learning method," *arXiv preprint arXiv:1907.04931*, 2019.

[24] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *Advances in neural information processing systems*, vol. 33, pp. 22 118–22 133, 2020.

[25] A. Jangda, S. Polisetty, A. Guha, and M. Serafini, "Accelerating graph sampling for graph machine learning using gpus," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 311–326.

[26] A. Kroviakov, P. Kurapov, C. Anneser, and J. Giceva, "Heterogeneous intra-pipeline device-parallel aggregations," *20th International Workshop on Data Management on New Hardware*, 2024.

[27] V. Rosenfeld, S. Bress, and V. Markl, "Query processing on heterogeneous cpu/gpu systems," *ACM Computing Survey*, vol. 55, no. 1, Jan. 2022.

[28] V. Leis, P. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 743754. [Online]. Available: https://doi.org/10.1145/2588555.2610507

[29] Amazon, "Aws gpu instances." [Online]. Available: https://aws.amazon.com/ec2/instance-types

[30] Microsoft, "Azure gpu instances." [Online]. Available: https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/overview?#gpu-accelerated

[31] Google, "Gcp gpu instances." [Online]. Available: https://cloud.google.com/compute/docs/gpus

[32] "Morphgl technical report." [Online]. Available: https://github.com/CSLabor/morphGL/blob/main/report.pdf

[33] Y. N. Sotskov and N. V. Shakhlevich, "Np-hardness of shop-scheduling problems with three jobs," *Discrete Applied Mathematics*, vol. 59, no. 3, pp. 237–266, 1995.

[34] T. K. Rusch, M. M. Bronstein, and S. Mishra, "A survey on oversmoothing in graph neural networks," *arXiv preprint arXiv:2303.10993*, 2023.

[35] J. P. Research, "Shipments of graphics add-in boards decline in q1 of 24 as the market experiences a return to seasonality." [Online]. Available: https://www.jonpeddie.com/news/shipments-of-graphics-add-in-boards-decline-in-q1-of-24-as-the-market-experiences-a-return-to-seasonality/

[36] P. Boldi and S. Vigna, "The webgraph framework i: compression techniques," in *Proceedings of the 13th WWW*, 2004, pp. 595–602.

[37] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 591–600.

[38] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[39] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 551–568.

[40] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, "{GNNAdvisor}: An adaptive and efficient runtime system for {GNN} acceleration on {GPUs}," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 515–531.

[41] P. Documents, "Memory management." [Online]. Available: https://pytorch.org/docs/stable/notes/cuda.html#memory-management/