CoGNN: Efficient Scheduling for Concurrent GNN Training on GPUs

Qingxiao Sun*, Yi Liu*, Hailong Yang*, Ruizhe Zhang*, Ming Dun*, Mingzhen Li*, Xiaoyan Liu*,
Wencong Xiao†, Yong Li†, Zhongzhi Luan*, Depei Qian*
Beihang University*, Unaffiliated†, Beijing, China
{qingxiaosun,yi.liu,hailong.yang,19373568,dunming0301,lmzhhh,liuxiaoyan,zhongzhi.luan,depeiq}@buaa.edu.cn
{xiaowencong,relianceslee}@gmail.com

***Abstract*—Graph neural networks (GNNs) suffer from low GPU utilization due to frequent memory accesses. Existing concurrent training mechanisms cannot be directly adapted to GNNs because they fail to consider the impact of input irregularity. This requires pre-profiling the memory footprint of concurrent tasks based on input dimensions to ensure successful co-location on GPU. Moreover, massive training tasks generated from scenarios such as hyper-parameter tuning require flexible scheduling strategies. To address these problems, we propose *CoGNN* that enables efficient management of GNN training tasks on GPUs. Specifically, the *CoGNN* organizes the tasks in a queue and estimates the memory consumption of each task based on cost functions at operator basis. In addition, the *CoGNN* implements scheduling policies to generate task groups, which are iteratively submitted for execution. The experiment results show that the *CoGNN* can achieve shorter completion and queuing time for training tasks from diverse GNN models.**

***Index Terms*—Graph Neural Networks, GPU, Concurrent Training, Task Scheduling, Estimation Model**

I. Introduction

Deep learning (DL) has received ubiquitous adoption in many real-world application domains, ranging from object detection [64] and image classification [26] to language processing [5] and machine translation [46]. As a number of new deep neural networks (DNNs) are being explored, developers take advantage of hardware accelerators such as TPU [22] and GPU to accelerate DL tasks. Especially, the GPUs have become the dominating workhorse to provide computation power in mainstream server infrastructure [56]. The reason is that GPUs excel at handling highly parallelized matrix operations heavily used in DNNs [9].

Graph neural networks (GNNs) emerge at the front-line for graph-based prediction tasks [24], [49] due to the powerful node representation. GNNs learn data relationships by combining graph operations and neural computations. Due to the irregularity of the graph structure, it is challenging for GNNs to achieve high performance on GPUs. Many optimization strategies based on node partitioning and coalesced caching have been proposed to alleviate load imbalance and thread divergence [19], [20], [28]. However, graph-related operators implemented by GNN frameworks inevitably lead to GPU under-utilization due to the limited memory bandwidth. PyTorch Geometric (PyG) [11] separately updates the node features with message passing, and the frequent data movements cause computation stalls. Deep Graph Library (DGL) [51] applies SpMM-like kernels to achieve simultaneous updates, whereas the sparse fetches reduce memory efficiency.

To ease the model training, the most common approach is to set the minimum granularity of GPU allocation to the entire GPU [21]. Whereas due to the improvement of computation capability, it is difficult for a single DL task to fully utilize the GPU resources [54]. Especially for memory-intensive tasks such as GNNs, their performance becomes saturated with increasing computation resources allocated [47], [48], [63]. Multiple training tasks can be co-located onto the same GPU to improve resource utilization. In industry, Multi-Process Server (MPS) [31] and Multi-Instance GPU (MIG) [33] enable multiple CUDA processes to share one GPU by resource partitioning, both of which have highly frequent tensor allocations and deallocations that deteriorate performance. In academia, temporal sharing [3] reduces pipelining latency by overlapping data preprocessing and computations, while spatial sharing [27], [62] enables concurrent execution to provide higher GPU throughput. However, the above mechanisms are tailored towards DNNs with fixed-sized input and cannot be directly adapted to GNNs. GNNs are input-sensitive, whose memory consumption and computation complexity are closely related to the graph dimensions [53].

Moreover, there are complex scenarios dealing with batch training tasks in practice. AI companies and cloud providers manage GPU clusters in a multi-tenant fashion [8], where co-locating multiple tasks on shared GPUs can accommodate more computation demands to reduce the total cost of ownership (TCO) [57]. Hyper-parameter tuning generates a large number of training tasks to explore different hyper-parameter settings for one model [6], [41]. In such cases, it is necessary to design flexible scheduling mechanisms to reduce the queuing time and completion time [16]. Furthermore, the training tasks can be enabled with spatial sharing to improve GPU utilization and training throughput. The working set size of the training task becomes critical for successful spatial sharing. If the working set size of the training task exceeds the GPU memory capacity, it will cause task crashes or significant performance degradation with unified virtual memory (UVM) enabled [2]. Therefore, the memory consumption of training tasks needs to be estimated in advance to ensure safe co-location with spatial sharing enabled [12]. Likewise, the GNN-specific graph operators and the impact of the irregular graph input on layer

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

outputs need to be considered as well.

From the above analysis, fine-grained memory allocation and task scheduling are required to improve the overall training throughput for concurrent training tasks. Moreover, the scheduling for concurrent training of GNNs poses unique challenges : *1) the runtime memory consumption of graph operators is difficult to estimate, thus threatening the memory safety under co-location; 2) the graph irregularity dominates computation complexity, input agnostic task co-location may significantly degrade training performance due to ineffective resource allocation.* To address the above challenges, we propose a concurrent GNN training framework *CoGNN*, which enables efficient scheduling and management of GNN training tasks on GPUs. The *CoGNN* first packs the training tasks into a queue and extracts information about task input and network structure. After that, the *CoGNN* profiles the computation graph of each GNN task and quantifies the impact of its operators on GPU memory consumption. Finally, the *CoGNN* exploits several scheduling strategies to group tasks and iteratively allocate memory for execution. We evaluate *CoGNN* on various GNNs to prove its effectiveness in reducing the completion time and queuing time of training tasks.

The *CoGNN* applies spatial sharing and temporal sharing to intra and inter task groups, respectively. To the best of our knowledge, this is the first work that targets the scheduling optimization and memory management for concurrent GNN training. This paper makes the following contributions:

- We comprehensively analyze the underlying causes of GPU under-utilization for GNNs and illustrate the concurrent training opportunities to improve GPU utilization and training throughput.
- We propose a task management mechanism that automatically executes GNN training tasks through fine-grained memory allocation and worker dispatch. Besides, the scheduling policies for different optimization goals are designed to generate task groups for concurrent training.
- We propose a memory estimation strategy, which defines a memory cost function for each GNN-related operator, and then traverses the computation graph of the training process to estimate GPU memory consumption.
- We develop a concurrent GNN training framework *CoGNN* that efficiently schedules and manages GNN training tasks co-located on GPUs. The experiment results show that the *CoGNN* can complete the training tasks faster with low queuing delay.

The rest of this paper is organized as follows. Section II and Section III present the background and motivation. Section IV presents the details of *CoGNN* methodology. Section V presents the evaluation results of *CoGNN*. Section VI discusses the related work, and Section VII concludes this paper.

II. Background

A. Graph Neural Networks

Recently, there has been an increasing interest in applying deep learning to non-structured data such as graphs [55]. Unlike dense objects (e.g., image and text) handled by traditional deep learning models, graphs represent sparse and irregular connected links. GNN takes graph-structured data as input, where each node is associated with a feature vector. Edges between nodes indicate the graph topology, quantized with edge weights. GNN learns data relationships by combining the graph structure and feature vectors. Here we take several typical models to illustrate the common GNN operations. Table I lists the notations used in this paper.

TABLE I
IMPORTANT GNN NOTATIONS.

Notation	Definition
G	A graph $G = (V, E)$.
V, v	The set of nodes in a graph, a node $v \in V$.
E, e_{ij}	The set of edges in a graph, an edge $e_{ij} \in E$.
D_v	The degree of a node v.
N_v	The neighbor set of a node v.
h_v	The feature vector of a node v.
A	The graph adjacency matrix.
X	The feature matrix composed by feature vectors.
W, α, ε	The learnable model parameters.

Graph convolutional network (GCN) [24] is one of the most successful networks for graph learning, which alleviates the problem of overfitting on local neighborhood structures for graphs. It performs graph operation formulated as Equation 1:

$$h_v^k = \sigma\left(\sum_{u \in N(v) \cup v} W^k h_u^{k-1} / \sqrt{D_u \cdot D_v}\right) \tag{1}$$

SAGE [15] further adopts sampling to obtain a fixed number of neighbors for each node. Its graph operation is formulated as Equation 2, where $S_{\mathcal{N}(v)}$ is a random sample of the node v's neighbors. Graph attention network (GAT) [49] adopts attention mechanisms to learn the relative weights between two connected nodes. The graph operation according to GAT is formulated as Equation 3. The attention weight α_{vu}^k measures the connective strength between the node v and its neighbor u. Graph isomorphism network (GIN) [59] adjusts the weight of the central node by a learnable parameter ε^k. Its graph operation is formulated as Equation 4.

$$h_v^k = \sigma\left(\sum_{u \in N(v) \cup v} W^k \{h_u^{k-1} \forall u \in S_{\mathcal{N}(v)}\}\right) \tag{2}$$

$$h_v^k = \sigma\left(\sum_{u \in N(v) \cup v} \alpha_{vu}^k W^k h_u^{k-1}\right) \tag{3}$$

$$h_v^k = MLP\left((1+\varepsilon^k) h_v^{k-1} + \sum_{u \in N_v} h_u^{k-1}\right) \tag{4}$$

Based on the above analysis, the core computation of GNNs can be abstracted as Equation 5, where $\hat{A}$ is calculated from A and varies across models:

$$X^{k+1} = \sigma(\hat{A} X^k W^k) \tag{5}$$

Since A is ultra-large and sparse, the equation can be naively viewed as chained SpMMs [25]. However, it is difficult for SpMM-like operations to fully utilize GPU computation resources due to the limited memory bandwidth.

B. Computational Patterns of GNN

Figure 1 illustrates the computation flow of GNNs in one training iteration. A GNN layer typically consists of two phases that combine graph operations and neural computations. The *Aggregate* phase retrieves a feature vector from each neighbor of a node and aggregates these vectors into a new feature vector. The *Update* phase performs neural computations such as multi-layer perceptron (MLP) to transform the feature vector of each node. Popular GNN frameworks process graph operations according to the graph structure, where an edge indicates data transfer. DGL employs node-wise parallelization with a central-neighbor pattern. It fetches data from the feature matrix and then performs reduction with SpMM-like operations to update the node features simultaneously. PyG employs edge-wise parallelization with the *MessagePassing* abstraction. It explicitly generates messages on all edges via *MessagePassing* and then performs separate reductions.

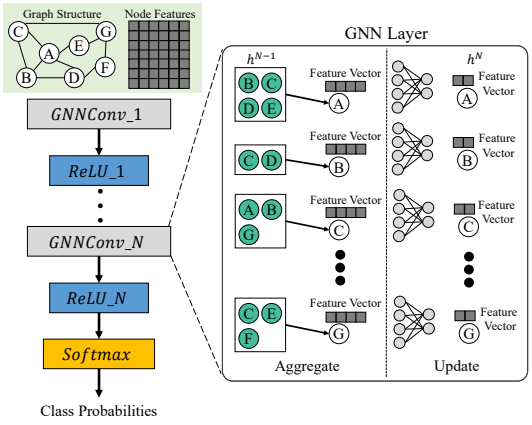

Fig. 1. The computation workflow of GNNs.

Both DGL and PyG are limited by insufficient utilization of GPU computation resources. DGL applies SpMM-like kernels to achieve simultaneous update of node features, but memory accesses dominate its performance due to the irregularity of the graph structure. PyG improves memory efficiency by implementing separate updates of nodes through aggregation kernels, but the time-consuming data movements cause computation stalls. To overcome this limitation, the co-location mechanism of GNN training tasks can be exploited to maximize GPU throughput. However, the memory consumption of training tasks shall be known in advance to avoid memory oversubscription. Unlike fixed-sized input neural networks, the output dimensions of GNN layers are strongly related to graph dimensions and feature length. Besides, the memory consumption of graph propagation also needs to be taken into account for accurate estimation.

C. GPU Sharing for Deep Learning

In industry, the mainstream practice is to set the minimum granularity of GPU allocation to the entire GPU [57]. While such a setup simplifies cluster resource management, it results in inefficient utilization of GPU resources. Therefore, GPU sharing is becoming a fundamental technique to co-locate more training tasks on GPU. For example, hyper-parameter tuning involves a large search space, where hyper-parameters are high-level properties exposed, such as learning rate and momentum. Popular tuning tools (e.g., *Hyperdriver* [41]) generate a large number of training tasks that explore different hyper-parameter settings for one model. For such scenarios, training tasks can be co-located into the same GPU to reduce queuing delay and improve GPU utilization.

Existing works propose mechanisms based on temporal or spatial sharing to achieve co-location of deep learning tasks on GPUs [3], [27], [56], [62]. Temporal sharing is highly flexible, dedicating GPU memory and cores to a single execution of a specific duration. *Gandiva* [56] improves the efficiency of deep learning tasks through coarse-grained time slicing and static memory partitioning. *PipeSwitch* [3] utilizes pipelined model transfers and active-standby workers to minimize switching overhead, thereby satisfying strict Service Level Objective (SLO) requirements for inference tasks. Although temporal sharing reduces latency by overlapping data preprocessing and computations, it can hardly improve the training throughout on GPU. For example, when processing language models composed of recurrent neural networks (RNNs), computation resources on GPU tend to be idle for a long time [38].

In contrast, spatial sharing can provide higher training throughput on GPU. One limitation of applying spatial sharing is the working set size of concurrent tasks. If the working set size exceeds the GPU memory, the system has to swap the data to the host, overshadowing spatial sharing performance benefits. *Salus* [62] designs scheduling strategies via two GPU sharing primitives, including fast job switching and memory sharing. However, *Salus* requires data of all processes to be preloaded into the GPU memory, thus restricting temporal sharing opportunities. *Zico* [27] overlaps the execution of concurrent tasks according to the cyclic patterns, thereby reducing the peak memory footprint. However, *Zico* only supports pairwise co-location and cannot handle massive tasks.

The above mechanisms target traditional neural networks with fixed-sized input, thus failing to consider the impact of irregular graphs on memory consumption. In addition, to support the co-location of massive training tasks, it is necessary to combine temporal sharing and spatial sharing to achieve more efficient scheduling strategies on GPUs.

III. Motivation

We make three key observations on the characteristics of GNN training on GPUs. These observations are consistent with existing GNN characterization [4], [60]. The experiment setup can be referred to Section V-A.

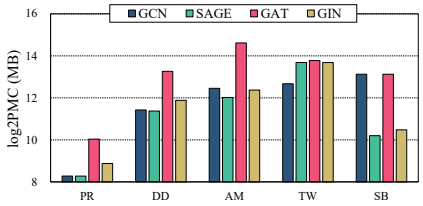

Fig. 2. Peak memory consumption of GNN training with different datasets on GPU.

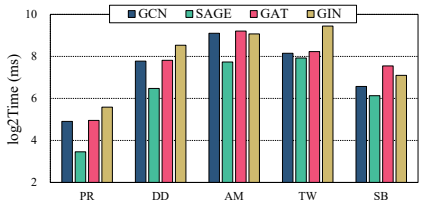

Fig. 3. Per-epoch execution time of GNN training with different datasets on GPU.

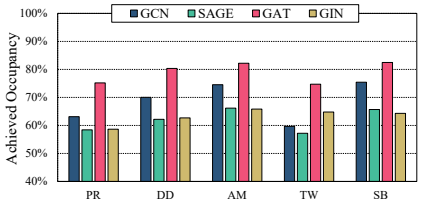

Fig. 4. Achieved occupancy of GNN training with different datasets on GPU.

A. Performance Impact of Graph Input

Figure 2 and Figure 3 show the peak memory consumption (PMC) and per-epoch time of GNN training with different datasets, respectively. The PMC of GNN training does not depend solely on the network design. Even with the same model structure, there is a large discrepancy in PMCs for GNN training with different datasets. This is because the graph dimensions and feature length dominate the memory usage of graph propagation and layer outputs. In addition, the irregularities of a graph may exponentially increase the per-epoch training time. For example, node degree is one of the graph irregularities that may lead to cache contention and load imbalance [17], [20], thus affecting the GNN training performance. From the above analysis, GNN training can be regarded as input-sensitive, whose working set size and execution time are closely related to the graph input. To enable successful co-location on GPUs with limited memory, graph information and network design shall be obtained in advance to estimate the PMCs of GNN training tasks.

B. Low GPU Utilization for GNN Training

Figure 4 shows the achieved occupancy[1] of GNN training with different datasets on GPU. The achieved occupancy is below 70% for most GNN training tasks. Specifically, the average achieved occupancies of GCN, SAGE, GAT, and GIN are 68.6%, 61.9%, 79.0%, and 63.3%, respectively. The relatively high achieved occupancy of GAT is due to the introduction of fully connected layers to compute the attention coefficients, which elevates the low utilization of graph operations. Note that both PyG and DGL have their limitations in fully utilizing GPU, such as low access efficiency and computation stalls. Except for optimizing a single GNN task for improved GPU utilization [19], [53], concurrently training multiple GNN tasks on a single GPU is also a promising direction to improve GPU utilization. This is well suited for scenarios such as hyper-parameter tuning, where massive training tasks can be submitted in a queue.

C. Opportunity for Concurrent Training

Figure 5 shows the GNN training performance of *MPS* normalized to that of default mode (*Default*) on GPU. The kernels are executed sequentially under *Default*, which means that only one kernel can occupy the GPU at a time. The UVM is enabled under *MPS* to avoid task crashes due to memory oversubscription. As seen, the *MPS* outperforms *Default* in most cases. Specifically, the *MPS* achieves average speedups of 1.92×, 1.06×, and 1.09× for GCN, SAGE, and GIN, respectively. The higher speedup of GCN is due to its highly memory-intensive nature, where the layer outputs are mainly computed by sparse operations. However, the *MPS* has highly frequent tensor allocations and deallocations, thus deteriorating the training performance. On the other hand, the *MPS* might suffer from GPU memory oversubscription due to the UVM overhead. For example, *DD-AM*, *AM-TW*, and *TW-SB* combinations of GAT obtain an average of 9.87× slowdown. Therefore, estimating the PMCs in advance is necessary for enabling concurrent training tasks. Besides, training tasks in the queue can be scheduled out of order to achieve better memory efficiency.

[1]Achieved occupancy [35] is an effective metric to diagnose performance issues on GPU. It indicates how many warps can be active at once per SMs.

IV. CoGNN Methodology

A. Design Overview

In this section, we propose a concurrent GNN training framework *CoGNN* that organizes GNN training tasks into a queue and enables efficient scheduling and management under GPU co-location. The queue structure facilitates *CoGNN* to group training tasks and execute them with out-of-order scheduling. As shown in Figure 6, the gray modules are designed or extended by *CoGNN*. The *CoGNN* consists of four important components, including memory manager, PMC profiler, task scheduler, and worker dispatcher. The memory manager maintains a unified memory pool and allocates memory on demand. The PMC profiler extracts runtime information to estimate the memory consumption of training tasks. The task scheduler determines the grouping and task execution order according to the scheduling policies. The worker dispatcher binds the training tasks to the specific workers and returns the results after execution. Note that a worker refers to a process responsible for task execution, which executes training tasks sequentially across different groups.

Figure 6 illustrates the design overview of *CoGNN*. The *CoGNN* integrates the CUDA allocator plugin into the DL backend for explicit GPU memory management. The *CoGNN* packs the training tasks implemented with the GNN framework into a task queue, where the PMC profiler extracts the details of model input and network structure. The PMC profiler represents the computation graph of a GNN model as a directed acyclic graph (DAG) and uses formulas to quantify

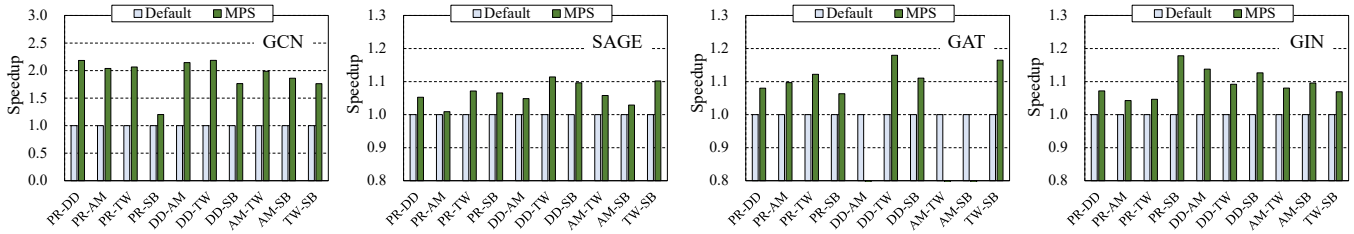

Fig. 5. GNN training performance of *MPS* normalized to that of default mode on GPU.

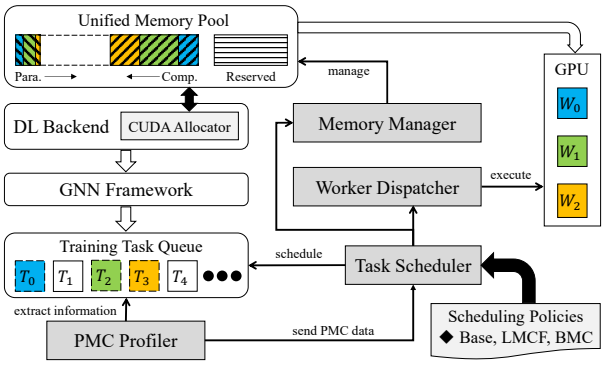

Fig. 6. The design overview of *CoGNN*.

the impact of each operator on memory consumption. After that, The PMC profiler sends the PMC information to the task scheduler, which uses a specific policy to group and rearrange training tasks. The task scheduler iteratively pops task groups from the queue. In each iteration, the memory manager and worker dispatcher receive signals to allocate shared GPU memory and execute training tasks.

The *CoGNN* combines spatial sharing and temporal sharing for more flexible task management. On the other hand, the *CoGNN* designs various scheduling policies to reduce makespan, job completion time (JCT) [16] or queuing time. In addition to GNNs, the *CoGNN* can support more general neural networks due to the versatility of its components.

B. Task Management Mechanism

The key to spatial sharing of concurrent tasks lies in the fine-grained management of GPU memory. We extend the unified memory pool of the DL backend (i.e., PyTorch) so that the *CoGNN* can insert task-occupied buffers into specific memory locations. In view of the characteristics of DL tasks, we partition the memory pool into reserved memory and allocated memory. The reserved memory stores framework-internal data such as CUDA context and model workspace, typically pre-allocated before task execution. The *CoGNN* inserts parameter buffers and computation buffers from both ends of the allocated memory to ensure full occupancy (Figure 6). The parameter buffer stores persistent tensors such as weights and biases, whose dimensions are known after loading the model. The Computation buffer stores tensors produced at task runtime, such as layer outputs. For the allocated memory, the *CoGNN* handles internal tensor fragmentation by padding extra memory according to the alignment requirements.

Figure 7 shows the overall workflow of task management in *CoGNN*. The task scheduler iteratively executes training tasks at the granularity of task groups. In such a way, it avoids memory fragmentation from batch tasks, which may reduce concurrent training opportunities while complicating memory maintenance. In each iteration, the *CoGNN* creates two processes that handle computation-related and parameter-related operations, respectively. Specifically, one process inserts the computation buffers into the allocated memory and dispatches workers for subsequent computations. Another process inserts the parameter buffers from the opposite end and conducts the Host-To-Device (H2D) parameter transmission. Note that the *CoGNN* overlaps the processes to improve pipelining efficiency and synchronize them afterward. The *CoGNN* then launches the workers to perform training tasks in a spatial-sharing manner. After the results are returned, the *CoGNN* clears all buffers and advances to the next group.

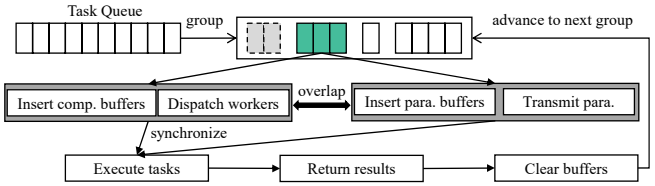

Fig. 7. The overall workflow of task management in *CoGNN*.

From the above analysis, the flexible management mechanism of *CoGNN* can support arbitrary grouping and out-of-order scheduling of tasks. This is necessary for improving the memory efficiency of input-sensitive GNNs, whose PMCs vary drastically with graph dimensions.

C. Memory Consumption Estimation

Inspired by [12], we formulate the computation graph (CG) of GNN training as a DAG (Equation 6), where each node op_i is an operator representing a mathematical invocation, and each edge ed_j specifies the execution dependency. With $TO = \langle ed_1, ed_2, ..., ed_m \rangle$ being the topological order dictated by the DAG, it can be pre-generated by referencing the topological order implementation within a framework [39]. The *CoGNN* utilizes TO to traverse the computation graph and update the PMC according to the allocation and deallocation of tensors.

$$CG = \langle \{op_i\}_{i=1}^{n}, \{ed_j = (op_x, op_y)\}_{j=1}^{m} \rangle \quad (6)$$

Figure 8 shows the DAG example for training a two-layer SAGE model. The SAGE layer (*SAGEConv*) consists of both graph and neural operators including *Propagate* and *Linear*. The activation functions (e.g., *ReLU*) are not shown in the figure due to their zero memory cost. For forward propagation, input graph data (*Data_X*) is fed through the neural network and manipulated by the above operators. The resulting outputs and input labels (*Data_Y*) are then back-propagated to compute the weight gradients. DL frameworks automatically insert auxiliary operators into the computation graph. For example, *SAGEConv_BP* operators compute output gradients to update the *SAGEConv* weights.

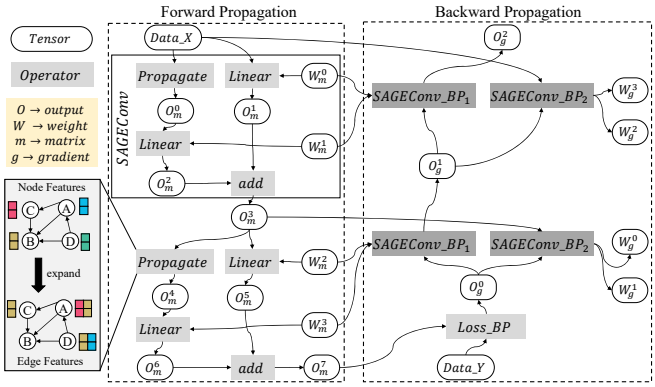

Fig. 8. Computation graph for training a two-layer SAGE model.

Estimating GPU memory consumption can be formalized as the memory required for each operator on the computation graph according to the topological traversal. We define a framework-independent memory cost function for each operator. The memory cost function (MCF) returns a set of allocated tensors with category and shape, which can be deduced by the input dimension and shape inference. The MCF of the operator op can be formulated as Equation 7, where W, O, G, and E are the sets of weight, output, gradient, and ephemeral tensors, respectively.

$$MCF(op) = W(op) \cup O(op) \cup G(op) \cup E(op) \quad (7)$$

The liveness of a tensor mainly depends on whether it will be used by the subsequent operators (i.e., an edge exists on the computation graph). However, weight tensors are exceptions because they are persistent in GPU memory for later updates. In addition, ephemeral tensors are only temporarily allocated inside an operator and released after the operator completes. Therefore, we can obtain the PMC of the GNN training task at the end of the graph traversal. The *CoGNN* currently requires the computation graph to be deterministic across training epoches. We will support dynamic GNNs [7] with graph evolution in future work.

Table II illustrates the tensors and tensor sizes of common operators in GNNs, where NE and NN denote the number of nodes and edges of the graph structure. Besides, FL_{in}^{i} and FL_{out}^{i} denote the input and output feature lengths of the i-th GNN layer, respectively. The *Matmul* is the core operator of GCN with sum-reduction, and its forward process can be abstracted as chained SpMMs. There are also customization details for GNNs that need to be considered. For example, GCN supplements self-loops so that the aggregated representation of the central node includes its own features. GAT involves *Leaky_ReLU* [29] and *Softmax* operators to generate coefficient messages. Even so, we still fail to take into account the temporary tensors inevitably created in GNN frameworks. Therefore, the *CoGNN* multiplies the estimated PMC by a fixed $threshold$ to guarantee that the inserted buffers are sufficient to meet the memory demands.

TABLE II
ALLOCATED TENSORS AND THEIR SIZES.

Operator Cate.	Operator	Tensor Cate.	Tensor Size
Graph	*Propagate*	Ephemeral	$NE \times FL_{in}^{i}$
		Output	$NN \times FL_{in}^{i}$
Matrix	*Matmul*	Weight	$FL_{in}^{i} \times FL_{out}^{i}$
		Output	$NN \times FL_{out}^{i}$
	Matmul_BP	Weight Gradient	$FL_{in}^{i} \times FL_{out}^{i}$
		Output Gradient	$NN \times FL_{out}^{i-1}$
Neural	*Linear*	Weight	$FL_{in}^{i} \times FL_{out}^{i}$
		Output	$NN \times FL_{out}^{i}$
	Linear_BP	Weight Gradient	$FL_{in}^{i} \times FL_{out}^{i}$
		Output Gradient	$NN \times FL_{out}^{i-1}$

The above abstraction and formalization are general for estimating the PMCs of various GNN frameworks. The *CoGNN* can also be adapted to other GNNs by modifying the computation graph or memory cost functions.

D. Scheduling Policies

The *CoGNN* opens up a large design space for scheduling policies due to its flexible management mechanism. In this paper, we implement three scheduling policies including *Base* policy, lowest-memory-consumption-first (*LMCF*) policy and balanced-memory-consumption (*BMC*) policy to improve memory efficiency. All policies work with "safety" condition to ensure that the memory usage of concurrent tasks does not exceed the GPU memory capacity. Fortunately, the *CoGNN* can estimate the PMCs of training tasks and accumulate them to keep each task group under "safety" condition. Moreover, the maximum size of a task group equals the number of workers. Next, we present the details of the scheduling policies.

1) Base policy: This policy is an extension of the first-in-first-out (FIFO) algorithm. Specifically, it follows in-order scheduling and greedily packs more tasks into the same group. However, packing too many tasks exceeding the GPU memory capacity will either crash the tasks or incur costly paging overhead with UVM enabled. Therefore, task packaging needs to accumulate PMC estimation results to meet "safety" condition.

2) LMCF policy: The *Base* policy might cause short-term tasks to suffer long queuing latencies while waiting for large ongoing tasks to complete. The shortest-job-first (*SJF*)

and shortest-remaining-time-first (*SRTF*) algorithms [13], [14] have been proposed to reduce the queuing time with and without preemption. However, the task duration or remaining time needs to be profiled offline, which affects usability and operation cost [16]. To address this issue, we propose the *LMCF* policy, given that the graph dimensions are positively correlated with computation complexity for GNN training. This policy adopts the non-preemptive mechanism based on out-of-order scheduling. Specifically, it sorts the task indices in the queue in ascending order of PMCs, and then traverses the sorting to generate task groups. Likewise, the size of a single task group cannot exceed the number of workers.

3) BMC policy: Although the *LMCF* policy alleviates the queuing latency of short-term tasks, grouping together training tasks with high PMCs may lead to increased resource conflicts. Therefore, we propose the *BMC* policy to balance makespan and queuing time. The principle is to group tasks with high and low computation complexity into a group, thereby reducing the performance interference of concurrent execution. Algorithm 1 illustrates the task grouping with *BMC* policy. We push the GNN tasks into a double-ended queue (*Deque*) [23] based on the ascending order of PMCs. We iteratively pop tasks from the right and left ends of the *Deque* and accumulate the PMCs (Line 6-11). If the group PMC reaches the allocated memory or the group size equals the number of workers, advance to the next group and initialize the counters (Line 12-17). The former steps are repeated until *Deque* is empty, and the policy obtains the final task groups.

Algorithm 1 Task grouping with *BMC* scheduling policy.

1: **Input:** ascending queue $Deque$, number of workers $nWorker$, memory allocated MA
2: **Output:** task group list $taskGroup$
3: $queSize \leftarrow Deque.size$ // original stack size
4: // initialize group PMC, group counter, and element counter
5: $gPMC$, $gCounter$, $eCounter \leftarrow 0$, -1, $nWorker$
6: **for** i in range $[0, queSize)$ **do**
7: **if** $i\%2 == 1$ **then**
8: $task \leftarrow Deque.pop()$ // pop a task from the right side
9: **else**
10: $task \leftarrow Deque.popleft()$ // pop from the left side
11: **end if**
12: $gPMC \leftarrow gPMC + task.PMC$ // accumulate PMCs
13: **if** $gPMC >= MA$ or $eCounter == nWorker$ **then**
14: $taskGroup.append([\,])$ // advance to next group
15: $gPMC$, $eCounter \leftarrow task.PMC$, 0
16: $gCounter \leftarrow gCounter + 1$
17: **end if**
18: $taskGroup[gCounter].append(task)$
19: $eCounter \leftarrow eCounter + 1$
20: **end for**

E. Implementation Details

We have implemented a system prototype of *CoGNN* in C++ and Python codes, and built it on top of PyG and PyTorch [36]. However, the ideas behind *CoGNN* are general to be adapted to other GNN frameworks or DL backends, and we leave such engineering efforts for future work.

We extend the PyTorch allocator module with CUDA IPC API to enable explicit management of GPU memory pool. We then add functions to insert buffers with specific CUDA streams and clear buffers from the memory pool. The buffer insertion function can be invoked multiple times to achieve spatial sharing of GNN training tasks. After the task group completes execution, the buffer clearing function will be invoked to clear memory allocations. Note that the actual size of memory allocated for tensors should meet certain alignment requirements. To address this issue for PyTorch, we leverage padding to align to multiples of 512 bytes.

The *CoGNN* consists of the scheduler process and worker process responsible for queue management and task execution, respectively. The scheduler process listens for task requests sent from the client through the TCP port. The requested tasks need to be in the list registered to the scheduler process so that the *CoGNN* can load the GNN models from disk to CPU memory. After receiving the requests, the scheduler process packs the tasks into a queue and loads the model structures. Next, the scheduler process profiles the PMCs through computation graph traversal to generate task groups. At each group iteration, the scheduler process sends the hash indices of the tasks to the worker process. The two processes are then overlapped to reduce pipelining latency.

The scheduler process inserts parameter buffers and transfers the model parameters to GPU memory with synchronization through CUDA events. The worker process identifies tasks according to the hash indices and dispatches them to worker threads. Each worker thread loads the corresponding GNN model and attaches it to the CUDA stream. In addition, each worker thread adds hooks to wait for the scheduler to transfer the parameters needed for model execution. After the results return, the worker thread will send a "finish" signal to the scheduler process through PyTorch Pipe API. When the number of received signals equals the group size, the scheduler process iterates to the execution of the next group.

V. Evaluation

A. Experiment Setup

1) Hardware and Software Configurations: The hardware specifications are presented in Table III. The experiments are conducted on Ubuntu 20.04 with GCC v9.3 and NVCC v11.1. The *CoGNN* is built on PyG v1.7 and PyTorch v1.8. In addition, PyTorch is modified to support explicit task co-location and memory management for *CoGNN*.

TABLE III
HARDWARE SPECIFICATIONS.

	CPU	GPU
Model	Intel Xeon E5-2680 v4 CPU	NVIDIA Tesla V100
Frequency	2.4GHz	1.5GHz
Cores	28	13440 (80 SMs)
Cache	32KB L1, 256KB L2, 35MB L3	6MB L2
Memory	378GB DDR4	32GB HBM2
Bandwidth	76.8GB/s	900GB/s

2) Graph Datasets and Task Queues: The graph datasets used for experiments are presented in Table IV. As seen, the graph datasets have diverse graph dimensions and feature lengths. Such diversity indicates significant differences in memory usage and computation complexity. We set the number of hidden units in each GNN layer to 64. The frameworks adjust the fan-outs of sampling-based GNNs by sampling ratios [11], [51]. Adopting PyG default settings, the sampling ratios of SAGE and GAT are set to 0.5 and 0.6, respectively. We set the number of layers in the range of $[4, 10]$ and train for 200 epoches. The layer number and graph dataset are combined to generate a task queue consisting of 20 models for each GNN. Furthermore, we sample 20 models from the combinations to obtain a task queue with distinct GNNs. To conclude, we perform experiments based on five task queues, including four queues with identical GNNs (i.e., GCN, SAGE, GAT, and GIN) and one queue with distinct GNNs (i.e., Mix). This variety of task organizations provides a comprehensive evaluation of the effectiveness of *CoGNN*.

TABLE IV
GRAPH DATASETS USED FOR EVALUATION.

Dataset	#Vertex	#Edge	#Feature	#Class
cora (CR)	2,708	10,858	1,433	7
citeseer (CT)	3,327	9,464	3,703	6
pubmed (PB)	19,717	88,676	500	3
PROTEINS (PR)	43,471	162,088	29	2
artist (AT)	50,515	1,638,396	100	12
soc-Blog (SB)	88,784	2,093,195	128	39
DD (DD)	334,925	1,686,092	98	2
amazon0601 (AM)	410,236	4,878,875	96	22
TWITTER (TW)	580,768	1,435,116	1,323	2
Yeast (YS)	1,714,644	3,636,546	74	2
OVCAR-8H (OV)	1,890,931	3,946,402	66	2

3) Comparison Methods and Metrics: We compare *CoGNN* with three scheduling policies (i.e., *Base*, *LMCF*, and *BMC*) against *Default*, *MPS*, *MIG*, and *PipeSwitch*. Note that *MIG* only supports Ampere and later GPU architectures (e.g., NVIDIA A100 in Section V-E). The *PipeSwitch* is aimed at fast preemptive switching of tasks that cannot be directly comparable. We extend *PipeSwitch* to execute GNN tasks in a temporal-sharing manner. The *MPS* and *MIG* execute GNN tasks in a spatial-sharing manner. For a fair comparison, the number of spatial-sharing tasks and *CoGNN* workers is both set to 2. For *MIG*, the fraction of memory and SMs are set to (4/8, 4/8) and (3/7, 4/7) [33]. The UVM is enabled to handle possible memory oversubscription with *MPS* and *MIG*. We have also tried to compare with *Salus* and *Zico*. However, *Zico* is not open source, and *Salus* fails to reproduce as described in [27] because of its outdated version. In addition, both of them are built on TensorFlow [1] and thus cannot be compared fairly with PyTorch-based methods. To determine task execution efficiency, we select four key metrics including SM utilization, makespan, JCT, and queuing time (QT) for comparison, which have been widely adopted in existing works [16], [27], [62]. We run each method 10 times and present the average results to isolate the effects of randomness.

B. Overall Performance Comparison

To evaluate the performance comparison of different methods, we present the makespans, average JCTs, and average QTs of task queues in Table V. The SM utilization comparison is shown in Figure 9. Overall, the *CoGNN* achieves better performance than *Default*, *MPS*, and *PipeSwitch* on all metrics. Below we analyze the experiment results in depth.

1) SM Utilization: Since PyTorch profiler [40] does not support obtaining the overall achieved occupancy of multiple tasks, we leverage NVIDIA System Management Interface [34] to obtain the SM utilization of different methods. The *CoGNN* and *MPS* achieve higher SM utilization due to the ability of spatial sharing. This indicates that a single GNN training task often cannot fully utilize GPU resources. Although the *MPS* achieves higher SM utilization than *CoGNN* for GAT, the memory oversubscription caused by *MPS* makes the kernels busy waiting for a large number of page migrations, other than performing useful computation. Such paging overhead significantly degrades the training performance.

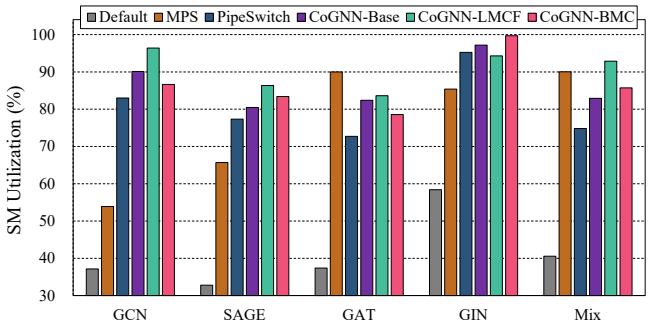

Fig. 9. SM utilization comparison of different methods on V100 GPU.

2) Makespan: The *MPS* achieves more than 2× slowdown for all task queues except GCN due to memory oversubscription. The *PipeSwitch* achieves an average speedup of 2.1× compared to *Default* with the same manner of sequentially executing tasks. The reason is that the memory pool maintained by *PipeSwitch* avoids frequent tensor allocation and deallocation operations. The *CoGNN* explores more optimization opportunities by combining spatial sharing and temporal sharing, thus almost all scheduling policies achieve shorter makespans than *PipeSwitch*. Specifically, tasks within a group improve overall training throughput through spatial sharing, whereas different groups reduce pipelining latency by overlapping data preprocessing and computation. Furthermore, the universal acceleration of *CoGNN* across all task queues demonstrates its generality for various GNNs. Among the policies, the *CoGNN-BMC* achieves the shortest makespans for most task queues (i.e., SAGE, GAT, and Mix). This is because the *BMC* policy places high and low complexity GNN tasks in the same group according to the estimation results, thus reducing resource contention and improving sharing efficiency.

TABLE V
PERFORMANCE COMPARISON OF DIFFERENT METHODS ON V100 GPU.

Method-policy	Makespan (min)					Average JCT (min)					Average QT (min)				
	GCN	SAGE	GAT	GIN	Mix	GCN	SAGE	GAT	GIN	Mix	GCN	SAGE	GAT	GIN	Mix
Default	12.14	11.57	18.40	19.10	15.02	7.47	7.04	10.47	14.20	8.86	6.86	6.46	9.55	13.25	8.11
MPS	4.26	10.55	44.00	23.12	44.69	3.27	8.05	39.47	20.24	40.96	2.89	7.02	35.21	18.21	36.74
PipeSwitch	3.54	3.53	7.02	11.67	6.07	2.89	2.87	4.93	10.56	4.34	2.71	2.69	4.58	9.97	4.04
CoGNN-Base	**3.41**	3.40	6.54	11.85	6.04	2.81	2.78	4.52	10.56	4.17	2.62	2.59	4.13	9.92	3.83
CoGNN-LMCF	3.43	3.45	6.88	**11.55**	5.65	**0.56**	**0.57**	**1.86**	**1.33**	**1.15**	**0.34**	**0.35**	**1.38**	**0.67**	**0.80**
CoGNN-BMC	3.47	**3.29**	**6.40**	13.76	**5.46**	2.86	2.73	4.62	11.38	4.33	2.67	2.54	4.24	10.67	4.01

3) JCT & QT: The *CoGNN* with *LMCF* policy achieves significant reductions for both average JCTs and QTs. Specifically, the *CoGNN-LMCF* achieves 9.9× and 15.1× reductions of JCT and QT on average compared to *Default* across all task queues. The *CoGNN-LMCF* also achieves 18.4× and 25.4× reductions compared to *MPS*, whereas it achieves 4.9× and 7.8× reductions compared to *PipeSwitch*. The reason is that the *LMCF* policy prioritizes the execution of tasks with small PMCs through out-of-order scheduling. In contrast, the FIFO scheduling of *MPS* and *PipeSwitch* may suffer from head-of-line blocking. This indicates that short-term tasks experience significant latencies by waiting for long-running tasks to complete. Figure 10 shows the CDF curves of JCTs for task queues with different methods. It can be observed that the JCT distributions are uneven due to the variability of graph datasets. Likewise, the *CoGNN-LMCF* significantly outperforms other methods or policies with smaller JCT for most tasks.

C. PMC Estimation Accuracy

Accurate estimation of PMCs is necessary to guarantee the memory safety of the task groups generated by *CoGNN*. We use the metric of relative error (RE) [61] to assess the estimation precision. Figure 11 shows the REs of PMC estimation for GNN training tasks. It can be observed that the *CoGNN* achieves a RE of less than 6% for all tasks. The reason is that the memory cost functions accurately calculate the memory consumption of GNN operators. Furthermore, the *CoGNN* generally achieves less than 1% RE for large graph datasets. This is because the output and ephemeral tensors of GNN operators occupy a large amount of memory space [58], which dilutes the estimation error caused by the implicit tensors from GNN frameworks. On the other hand, the *CoGNN* achieves stable accuracy results with changes in the number of GNN layers. This indicates that the DAG generated according to the network structure well represents the computation flow of single-epoch training. In such a way, PMC estimation can be adaptively scaled to an arbitrary number of network layers. From the above analysis, we empirically set $threshold$ for multiplication to 1.15, which is sufficient to ensure memory safety because of accurate PMC estimations.

D. Beyond Pairwise

Existing works such as *PipeSwitch* and *Zico* generally only support temporal sharing or pairwise spatial sharing. In contrast, the *CoGNN* can support spatial sharing of any number of tasks due to its flexible task management mechanism. Figure 12 shows the normalized performance of *CoGNN* with three workers compared to that with two workers. It can be observed that the performance change of increasing the number of workers is not consistent for different GNNs. For example, 3-workers *CoGNN* achieves 1.18×, 1.14×, and 1.10× increases for GCN in makespan, average JCT, and average QT, respectively. This is because the GCN layers are mainly composed of sparse computations, which may lead to severe cache conflicts when launching more workers. The performance degradation caused by cache conflicts is unpredictable, even leading to a 2.13× increase in makespan for SAGE with *BMC* policy.

On the other hand, 3-workers *COGNN* achieves 1.02×, 1.06×, and 1.11× reductions for GAT in makespan, average JCT, and average QT. The reason is that the fully connected operations in GAT layers are less memory-bound than sparse graph operations [60], and thus adding workers can make better use of memory bandwidth. In addition, spatial sharing of different GNNs may achieve better performance improvements. For example, 3-workers *CoGNN* achieves 1.01×, 1.04×, and 1.08× reductions for Mix in makespan, average JCT, and average QT. This is because GNNs have different demands on GPU resources, which alleviates the performance interference of concurrent tasks. However, the *CoGNN* still fails to identify resource contention among concurrent GNN training tasks. To address this issue, the PMC profiler can be extended to predict the resource usage of GNN operators and the latency of operator overlap [10]. We leave such extensions for future work.

E. Applying to other GPU Hardware

To demonstrate the generality of our approach, we evaluate *CoGNN* on NVIDIA A100 GPU with 108 SMs and 40GB memory. Table VI presents the makespans, average JCTs, and average QTs of task queues with different methods. It can be observed that *MIG* performs even worse than *Default* for most task queues. This is because the static resource partition of *MIG* causes severe load imbalance and thus exacerbates the UVM overhead. Consistent with the experiment results on V100 GPU, the *CoGNN* achieves better performance than *Default*, *MIG*, *MPS*, and *PipeSwitch* on all metrics on A100 GPU. The *CoGNN-BMC* achieves the shortest makespan for most task queues, while the *CoGNN-LMCF* always achieves

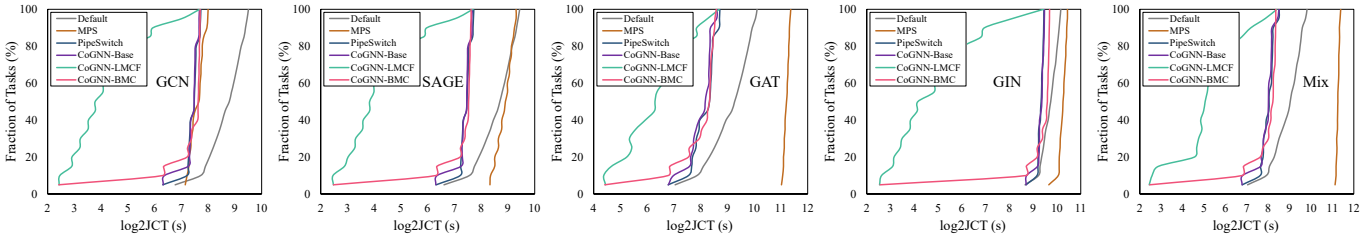

Fig. 10. Comparison of JCT distributions for task queues with different methods.

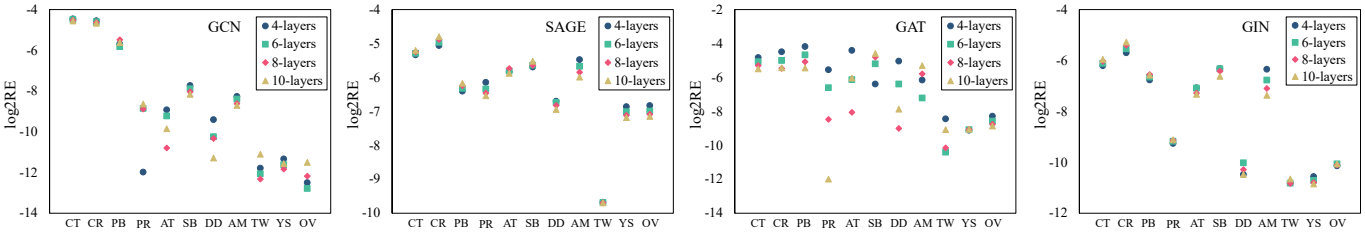

Fig. 11. Relative errors of PMC estimation for GNN training tasks with *CoGNN*.

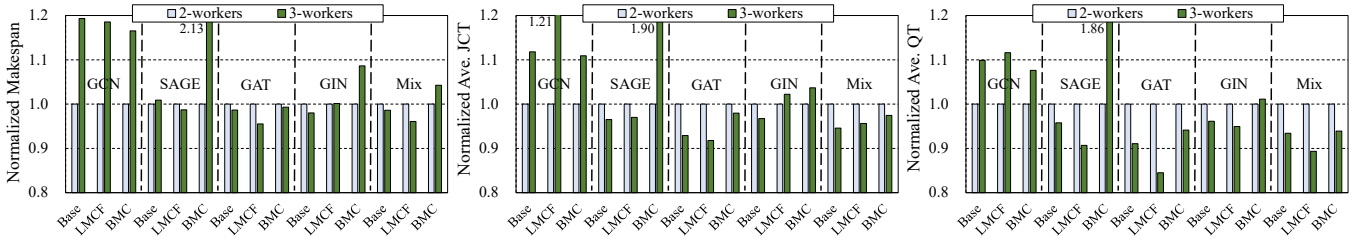

Fig. 12. Performance of *CoGNN* with three workers normalized to that with two workers.

the shortest average JCT and QT. However, there are differences in performance details on A100 compared to V100. For example, the *MPS* achieves significant performance improvements on A100 because the larger memory eliminates the need for frequent data swapping. In addition, the *CoGNN-BMC* achieves 1.11× and 1.12× makespan reductions compared to *PipeSwitch* for Mix queue on V100 and A100. The higher makespan reduction is that A100 has more cores and larger memory bandwidth, thereby alleviating performance interference for tasks with different resource requirements.

From the above analysis, we believe that the *CoGNN* has greater performance potential on future GPU architectures. For example, the latest NVIDIA H100 [32] has doubled the cores, memory size, and memory bandwidth due to the evolution of the manufacturing process. Moreover, H100 introduces thread block clusters to provide collaboration and data exchange across multiple SMs. This can further reduce the cache contention caused by concurrent GNN training to a certain extent.

F. Overhead Analysis

We normalize the *CoGNN* overhead to the duration of the task queue executing only one epoch. The *CoGNN* overhead can be divided into three parts, including PMC estimation, task grouping, and task scheduling. PMC estimation loads each model structure and traverses the computation graph through memory cost functions to update the PMC information. Task grouping re-orders the task queue and generates task groups according to the scheduling policy. Task scheduling iteratively allocates memory, dispatches workers, and synchronizes tasks within group. Note that the task grouping time is less than 0.001 epoches, which is negligible compared to GNN training. Figure 13 shows the breakdown of *CoGNN* processing, including PMC estimation and task scheduling. It can be observed that the total overhead of *CoGNN* is acceptable. PMC estimation and task scheduling take an average of 2.1 and 3.9 epoches across all task queues and scheduling policies, respectively. In production scenarios, thousands of epoches are often required to improve prediction accuracy. In sum, it is effective to improve GPU throughput by using *CoGNN* to schedule and manage massive GNN training tasks.

G. Auxiliary Analysis

1) Memory Safety: We evaluate the runtime memory consumption of *CoGNN* to demonstrate the effectiveness of the task grouping strategy. Figure 14 shows the PMC distribution of task groups with out-of-order policies. On V100 GPU, we empirically reserve 6GB memory for storing framework-internal data such as CUDA context and model workspace. It can be observed that task co-location never leads to memory oversubscription. Moreover, the PMCs of certain task groups

TABLE VI
PERFORMANCE COMPARISON OF DIFFERENT METHODS ON A100 GPU.

Method-policy	Makespan (min)					Average JCT (min)					Average QT (min)				
	GCN	SAGE	GAT	GIN	Mix	GCN	SAGE	GAT	GIN	Mix	GCN	SAGE	GAT	GIN	Mix
Default	11.54	10.87	17.66	18.07	14.29	7.15	6.67	10.09	13.49	8.46	6.57	6.13	9.21	12.59	7.74
MIG	10.03	9.87	84.54	25.02	80.23	6.29	6.28	75.05	20.54	72.33	5.36	5.35	70.15	18.79	67.94
MPS	3.78	6.78	22.84	13.56	9.48	2.66	4.23	18.33	10.48	5.69	2.31	3.58	16.11	9.33	4.94
PipeSwitch	3.17	2.77	6.10	10.29	4.93	2.52	2.20	4.14	9.25	3.47	2.36	2.06	3.84	8.74	3.23
CoGNN-Base	3.23	2.69	5.63	10.40	4.85	2.54	2.16	3.85	9.24	3.43	2.35	2.01	3.48	8.68	3.16
CoGNN-LMCF	3.22	**2.59**	5.76	**10.22**	4.54	**0.73**	**0.56**	**1.89**	**1.31**	**1.17**	**0.52**	**0.39**	**1.46**	**0.72**	**0.87**
CoGNN-BMC	**3.10**	4.05	**5.49**	11.13	**4.39**	2.47	3.11	3.86	9.40	3.38	2.27	2.88	3.50	8.81	3.12

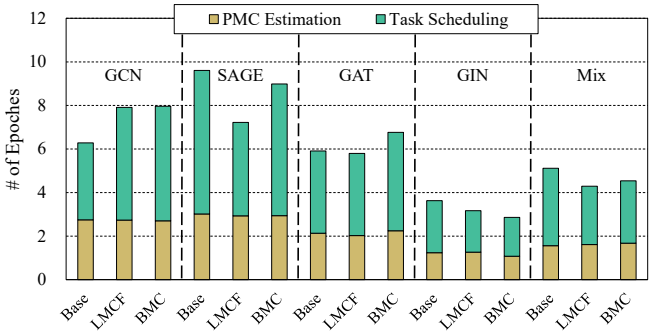

Fig. 13. Performance breakdown of *CoGNN* processing normalized to one-epoch execution time of task queues.

are close to the V100 capacity, indicating that the GPU memory is fully utilized. The *LMCF* and *BMC* policies exhibit opposite curves with group iterations due to different scheduling algorithms. Nevertheless, the *CoGNN* can accurately estimate PMCs and generate task groups adaptively to avoid working set size exceeding GPU memory capacity.

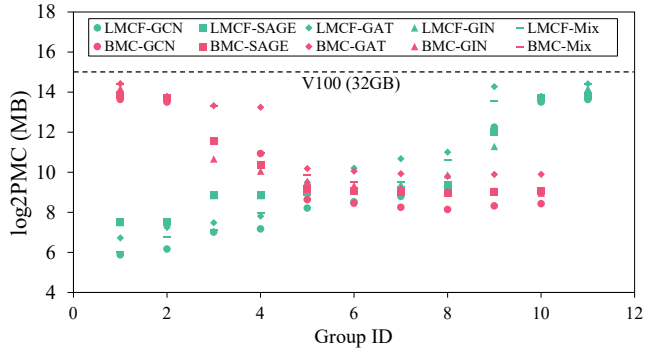

Fig. 14. PMC distribution of task groups with out-of-order policies.

2) PMC & Execution Time: The *CoGNN* assumes that PMC reflects computation complexity and further affects execution time. Larger graph dimension sizes and feature length tend to result in larger output and ephemeral tensors dominating the memory consumption during training. Moreover, larger dimension sizes indicate that each network layer contains more basic arithmetic operations (e.g., multiply and add), often leading to a longer execution time. Therefore, co-locating high-PMC GNN tasks may exacerbate cache thrashing and memory bandwidth contention. Figure 15 shows the PMCs and durations of 8-layer GNNs processing different graph datasets. It can be observed that PMC is strongly correlated with execution time, which is consistent with our assumption.

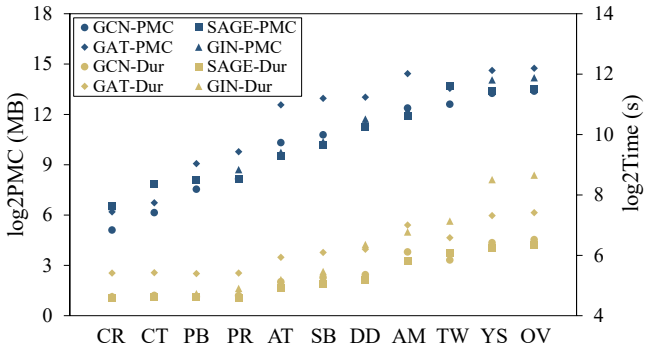

Fig. 15. Correlation of PMC and execution time for GNNs with eight layers.

VI. RELATED WORK

GNN Acceleration on GPUs. Recent research works deeply mine the computation features of GNNs to enable fine-grained optimizations for GPU architectures [17], [19], [20], [28], [30], [52], [53]. *GE-SpMM* [19] improved the access efficiency by using shared memory to cache sparse matrix rows and merging the workloads of different warps. *FeatGraph* [17] combined graph partitioning with feature dimension tiling to optimize cache utilization during GNN aggregation. *Huang et al.* [20] clustered central nodes by locality-sensitive hashing and further partitioned the workload by neighbor grouping to address load imbalance. *GNNAdvisor* [53] introduced warp-aligned thread mapping with neighbor and dimension partitioning to reduce thread divergence. *ES-SpMM* [28] proposed in-kernel edge sampling that downsized the graph to fit into shared memory and eliminated preprocessing overhead. *TC-GNN* [52] identified the non-zero tiles by sparse graph translation and adapted the input graph to the dense computations of tensor cores. The above works are orthogonal to this paper that targets concurrent GNN training scheduling.

Memory Management for Training. Since limited memory capacity restricts the scale of network training, research works generally utilize re-computation or swapping to manage GPU memory [18], [37], [42]–[45], [50], [65]. *vDNN* [44] performed data swapping at layer granularity, where data was

offloaded to the CPU in forward phase and prefetched at back-propagation. *SuperNeurous* [50] introduced unified tensor pool to reuse physical memory and re-computed less expensive layers at back-propagation. *Shriram et al.* [45] extended *vDNN* that applied zero-value compression and non-offload high-end allocation to reduce memory fragments. *SwapAdvisor* [18] used genetic algorithm to explore swapping decisions and designed data engine simulator to estimate the execution time. *Capuchin* [37] performed mini-batches to obtain access patterns of tensors and selected cheap operations for collective re-computation. *Sentinel* [42] coordinated OS and runtime profiling to enable co-allocating tensors with similar memory access frequency into the same pages. These approaches are effective against memory oversubscription and serve as references for processing larger graph inputs on GPUs.

VII. Conclusion and Future Work

In this paper, we propose a concurrent GNN training framework *CoGNN*, which can efficiently manage massive GNN training tasks co-located on GPUs. The *CoGNN* organizes training tasks into a queue and extracts the information of model input and network structure. After that, the *CoGNN* profiles the computation graph of each task and estimates the memory consumption according to the operator cost functions. Finally, the *CoGNN* exploits out-of-order scheduling policies to generate task groups that are iteratively submitted to GPU for execution. The experiment results show that the *CoGNN* can achieve shorter completion and queuing time for training tasks from diverse GNN models.

For future work, we would like to extend the PMC profiler of *CoGNN* to predict the resource usage of GNN operators and the latency of overlapped operators. In such a way, we can quantify the performance interference among spatially shared training tasks. Thanks to the flexible management mechanism, we would like to extend *CoGNN* to hybrid scenarios with both training and inference tasks. We can further explore how to guarantee the QoS of inference tasks and maximize the throughput of training tasks under GPU co-location. Moreover, we would like to extend *CoGNN* to support other deep learning frameworks such as Tensorflow and Mindspore[2], as well as to evaluate *CoGNN* on other GPU platforms such as AMD and Iluvatar Corex[3].

Acknowledgements

This work was supported by National Key Research and Development Program of China (No. 2020YFB1506703), National Natural Science Foundation of China (No. 62072018 and 61732002), Special Fund for Basic Scientific Research of Central Universities, and Iluvatar CoreX semiconductor Co., Ltd. Hailong Yang is the corresponding author.

[2]https://www.mindspore.cn/

[3]https://www.iluvatar.com/

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: {TensorFlow}: A system for {Large-Scale} machine learning. In: 12th USENIX symposium on operating systems design and implementation (OSDI 16). pp. 265–283 (2016)

[2] Allen, T., Ge, R.: Demystifying gpu uvm cost with deep runtime and workload analysis. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 141–150. IEEE (2021)

[3] Bai, Z., Zhang, Z., Zhu, Y., Jin, X.: {PipeSwitch}: Fast pipelined context switching for deep learning applications. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). pp. 499–514 (2020)

[4] Baruah, T., Shivdikar, K., Dong, S., Sun, Y., Mojumder, S.A., Jung, K., Abellán, J.L., Ukidave, Y., Joshi, A., Kim, J., et al.: Gnnmark: A benchmark suite to characterize graph neural network training on gpus. In: 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 13–23. IEEE (2021)

[5] Beltagy, I., Lo, K., Cohan, A.: Scibert: A pretrained language model for scientific text. arXiv preprint arXiv:1903.10676 (2019)

[6] Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. Advances in neural information processing systems **24** (2011)

[7] Chakaravarthy, V.T., Pandian, S.S., Raje, S., Sabharwal, Y., Suzumura, T., Ubaru, S.: Efficient scaling of dynamic graph neural networks. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–15 (2021)

[8] Chen, H.H., Lin, E.T., Chou, Y.M., Chou, J.: Gemini: Enabling multi-tenant gpu sharing based on kernel burst estimation. IEEE Transactions on Cloud Computing (2021)

[9] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)

[10] Cui, W., Zhao, H., Chen, Q., Zheng, N., Leng, J., Zhao, J., Song, Z., Ma, T., Yang, Y., Li, C., et al.: Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–15 (2021)

[11] Fey, M., Lenssen, J.E.: Fast graph representation learning with pytorch geometric. arXiv preprint arXiv:1903.02428 (2019)

[12] Gao, Y., Liu, Y., Zhang, H., Li, Z., Zhu, Y., Lin, H., Yang, M.: Estimating gpu memory consumption of deep learning models. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1342–1352 (2020)

[13] Grandl, R., Kandula, S., Rao, S., Akella, A., Kulkarni, J.: {GRAPHENE}: Packing and {Dependency-Aware} scheduling for {Data-Parallel} clusters. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 81–97 (2016)

[14] Gu, J., Chowdhury, M., Shin, K.G., Zhu, Y., Jeon, M., Qian, J., Liu, H., Guo, C.: Tiresias: A {GPU} cluster manager for distributed deep learning. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). pp. 485–500 (2019)

[15] Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. Advances in neural information processing systems **30** (2017)

[16] Hu, Q., Sun, P., Yan, S., Wen, Y., Zhang, T.: Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–15 (2021)

[17] Hu, Y., Ye, Z., Wang, M., Yu, J., Zheng, D., Li, M., Zhang, Z., Zhang, Z., Wang, Y.: Featgraph: A flexible and efficient backend for graph neural network systems. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–13. IEEE (2020)

[18] Huang, C.C., Jin, G., Li, J.: Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 1341–1355 (2020)

[19] Huang, G., Dai, G., Wang, Y., Yang, H.: Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. IEEE (2020)

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

[21] Jeon, M., Venkataraman, S., Phanishayee, A., Qian, J., Xiao, W., Yang, F.: Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). pp. 947–960 (2019)

[22] Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al.: In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th annual international symposium on computer architecture. pp. 1–12 (2017)

[23] Kashyap, B.R.: The double-ended queue with bulk service and limited waiting space. Operations Research **14**(5), 822–834 (1966)

[24] Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)

[25] Li, J., Louri, A., Karanth, A., Bunescu, R.: Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). pp. 775–788. IEEE (2021)

[26] Li, S., Song, W., Fang, L., Chen, Y., Ghamisi, P., Benediktsson, J.A.: Deep learning for hyperspectral image classification: An overview. IEEE Transactions on Geoscience and Remote Sensing **57**(9), 6690–6709 (2019)

[27] Lim, G., Ahn, J., Xiao, W., Kwon, Y., Jeon, M.: Zico: Efficient {GPU} memory sharing for concurrent {DNN} training. In: 2021 USENIX Annual Technical Conference (USENIX ATC 21). pp. 161–175 (2021)

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

[29] Maas, A.L., Hannun, A.Y., Ng, A.Y., et al.: Rectifier nonlinearities improve neural network acoustic models. In: Proc. icml. vol. 30, p. 3. Citeseer (2013)

[30] Min, S.W., Wu, K., Huang, S., Hidayetoğlu, M., Xiong, J., Ebrahimi, E., Chen, D., Hwu, W.m.: Large graph convolutional network training with gpu-oriented data communication architecture. arXiv preprint arXiv:2103.03330 (2021)

[31] NVIDIA: Sharing a gpu between mpi processes: multi-process service (2012)

[32] NVIDIA: Nvidia h100 tensor core gpu architecture. https://nvdam.widen.net/s/9bz6dw7dqr/gtc22-whitepaper-hopper (2022)

[33] NVIDIA: Nvidia multi-instance gpu user guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide (2022)

[34] NVIDIA: Nvidia system management interface. https://developer.nvidia.com/nvidia-system-management-interface (2022)

[35] NVIDIA, N.V.S.E.: 4.7 user guide (2022)

[36] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems **32** (2019)

[37] Peng, X., Shi, X., Dai, H., Jin, H., Ma, W., Xiong, Q., Yang, F., Qian, X.: Capuchin: Tensor-based gpu memory management for deep learning. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 891–905 (2020)

[38] Peng, Y., Bao, Y., Chen, Y., Wu, C., Guo, C.: Optimus: an efficient dynamic resource scheduler for deep learning clusters. In: Proceedings of the Thirteenth EuroSys Conference. pp. 1–14 (2018)

[39] PyTorch: The topological sorting algorithm for computation graphs in pytorch. https://github.com/pytorch/pytorch/blob/v1.8.0/caffe2/core/nomnigraph/include/nomnigraph/Graph/TopoSort.h (2021)

[40] PyTorch: Pytorch profiler. https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html (2022)

[41] Rasley, J., He, Y., Yan, F., Ruwase, O., Fonseca, R.: Hyperdrive: Exploring hyperparameters with pop scheduling. In: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference. pp. 1–13 (2017)

[42] Ren, J., Luo, J., Wu, K., Zhang, M., Jeon, H., Li, D.: Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). pp. 598–611. IEEE (2021)

[43] Ren, J., Rajbhandari, S., Aminabadi, R.Y., Ruwase, O., Yang, S., Zhang, M., Li, D., He, Y.: {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In: 2021 USENIX Annual Technical Conference (USENIX ATC 21). pp. 551–564 (2021)

[44] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., Keckler, S.W.: vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 1–13. IEEE (2016)

[45] Shriram, S., Garg, A., Kulkarni, P.: Dynamic memory management for gpu-based training of deep neural networks. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 200–209. IEEE (2019)

[46] Stahlberg, F.: Neural machine translation: A review. Journal of Artificial Intelligence Research **69**, 343–418 (2020)

[47] Sun, Q., Liu, Y., Yang, H., Jiang, Z., Qian, D.: Stencilmart: Predicting optimization selection for stencil computations across gpus. In: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 1–11. IEEE (2022)

[48] Sun, Q., Liu, Y., Yang, H., Luan, Z., Qian, D.: Smqos: Improving utilization and energy efficiency with qos awareness on gpus. In: 2019 IEEE International Conference on Cluster Computing (CLUSTER). pp. 1–5. IEEE (2019)

[49] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)

[50] Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S.L., Xu, Z., Kraska, T.: Superneurons: Dynamic gpu memory management for training deep neural networks. In: Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming. pp. 41–53 (2018)

[51] Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., et al.: Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315 (2019)

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

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

[54] Wu, X., Rao, J., Chen, W., Huang, H., Ding, C., Huang, H.: Switchflow: preemptive multitasking for deep learning. In: Proceedings of the 22nd International Middleware Conference. pp. 146–158 (2021)

[55] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y.: A comprehensive survey on graph neural networks. IEEE transactions on neural networks and learning systems **32**(1), 4–24 (2020)

[56] Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., et al.: Gandiva: Introspective cluster scheduling for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). pp. 595–610 (2018)

[57] Xiao, W., Ren, S., Li, Y., Zhang, Y., Hou, P., Li, Z., Feng, Y., Lin, W., Jia, Y.: {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). pp. 533–548 (2020)

[58] Xie, Z., Ye, Z., Wang, M., Zhang, Z., Fan, R.: Graphiler: Acompiler for graph neural networks (2021)

[59] Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? arXiv preprint arXiv:1810.00826 (2018)

[60] Yan, M., Chen, Z., Deng, L., Ye, X., Zhang, Z., Fan, D., Xie, Y.: Characterizing and understanding gcns on gpu. IEEE Computer Architecture Letters **19**(1), 22–25 (2020)

[61] Yu, G.X., Gao, Y., Golikov, P., Pekhimenko, G.: A runtime-based computational performance predictor for deep neural network training. arXiv preprint arXiv:2102.00527 (2021)

[62] Yu, P., Chowdhury, M.: Salus: Fine-grained gpu sharing primitives for deep learning applications. arXiv preprint arXiv:1902.04610 (2019)

[63] Zhao, X., Wang, Z., Eeckhout, L.: Classification-driven search for effective sm partitioning in multitasking gpus. In: Proceedings of the 2018 international conference on supercomputing. pp. 65–75 (2018)

[64] Zhao, Z.Q., Zheng, P., Xu, S.t., Wu, X.: Object detection with deep learning: A review. IEEE transactions on neural networks and learning systems **30**(11), 3212–3232 (2019)

[65] Zheng, B., Vijaykumar, N., Pekhimenko, G.: Echo: Compiler-based gpu memory footprint reduction for lstm rnn training. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). pp. 1089–1102. IEEE (2020)

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Hardware and Software Configurations. We evaluate CoGNN on Intel Xeon E5-2680 v4 CPU and NVIDIA V100/A100 GPUs. The experiments are conducted on Ubuntu 20.04 with GCC v9.3 and NVCC v11.1. The CoGNN is built on PyG v1.7 and PyTorch v1.8.

Graph Datasets and Task Queues. The graph datasets used for experiments have diverse graph dimensions and feature lengths. We set the number of layers in the range of [4, 10] and train for 200 epochs. The layer number and graph dataset are combined to generate task queues. We perform experiments based on five task queues including four queues with identical GNNs (i.e., GCN, SAGE, GAT, and GIN) and one queue with distinct GNNs.

Comparison Methods and Metrics. We compare CoGNN with three scheduling policies (i.e., Base, LMCF, and BMC) against Default, PipeSwitch and MPS. For a fair comparison, the numbers of MPS spatial-sharing tasks and CoGNN workers are both set to 2. CUDA UVM is enabled to handle memory oversubscription with MPS. We select four key metrics including SM utilization, makespan, JCT, and queuing time (QT) for comparison.

For more details on the environment and experiments, please refer to the READMEs in our github repository.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://github.com/guessmewho233/CoGNN_info_for_SC22;https://doi.org/10.5281/zenodo.6586262`

Artifact name: CoGNN Github repo/DOI

Artifact 2

Persistent ID: `https://drive.google.com/drive/folders/1Vq1nwAfVwBJfMVvepwW7UwtAx7PVRPaQ?usp=sharing`

Artifact name: CoGNN AE Docker Image

Reproduction of the artifact with container: 1. Download the Docker Image from Google Drive.

2. Follow the README to load the image and access the software environments.

3. Follow the instructions to reproduce the experimental results in our paper.

/home/sqx/reproduce/perf/README.md -> Tables 4-5, Figure 10, and Figure 12

/home/sqx/reproduce/pmc/README.md -> Figure 11

/home/sqx/reproduce/overhead/README.md -> Figure 13

Note that docker containers do not support MPS. MPS data can be reproduced via the instructions in our github repository, or an account will be provided later to access our machine.