PGSampler: Accelerating GPU-based Graph Sampling in GNN Systems via Workload Fusion

Xiaohui Wei[1], Weikai Tang[1], Hao Qi[2], Hengshan Yue[1,*]
[1]Jilin University, China
[2]Huazhong University of Science and Technology, China
Email: weixh@jlu.edu.cn, tangwk22@mails.jlu.edu.cn, theqihao@hust.edu.cn, yuehs@jlu.edu.cn

***Abstract*—Graph Neural Networks (GNNs) have demonstrated remarkable performance across various domains. Sample-based training, a practical strategy for training on large-scale graphs, often faces time-consuming graph sampling challenges. To address this, GPU-based graph sampling has been introduced, while there is still room for further efficiency improvements. Though several prior works have been proposed to accelerate the computation or memory access for GPU-based graph sampling, we show that the performance bottlenecks induced by small workload cannot be ignored. In this paper, we propose *PGSampler*, an efficient system for accelerating GPU-based graph sampling. First, *PGSampler* leverages a barrier-free execution mode to fuse workload, significantly improving the resource utilization. By altering the sampling execution mode, *PGSampler* also reduces the preprocessing time before kernel execution, thus accelerating the whole sampling process. Next, based on the new sampling execution mode, considering the dynamically generated nature of sampling tasks, *PGSampler* adopts a persistent kernel design and uses the task queue to assign tasks, achieving dynamic load balancing. Evaluations with diverse parameter settings show that *PGSampler* can achieve up to 2.22× performance speedup over the state-of-the-art GNN system DGL.**

***Index Terms*—graph neural networks, sample-based GNN training, GPU-based graph sampling, persistent kernel**

I. Introduction

In many practical scenarios, data is naturally organized into graphs, where entities are represented as nodes, and relationships between entities are represented as edges. The graph Neural Network (GNN) is a type of neural network particularly well-suited for tasks involving graph-structured data [1], it recursively updates vertices' features by aggregating information from their neighbors. GNNs have demonstrated compelling performance across a spectrum of domains, including social network analysis [2], recommendation systems [3], and molecule analysis [4]. In order to train GNN models more efficiently, based on widely used deep learning frameworks [5], [6], many GNN systems such as Deep Graph Library (DGL) [7] and Pytorch Geometric (PyG) [8] have been developed, providing robust support for various operations on graph. Real-world graphs, characterized by their large scale and high-dimensional features, pose substantial challenges when training GNN models with L layers. The computational and memory requirements grow exponentially as the model considers all neighbors within L hops for each training vertex, making the training process inefficient and even unfeasible.

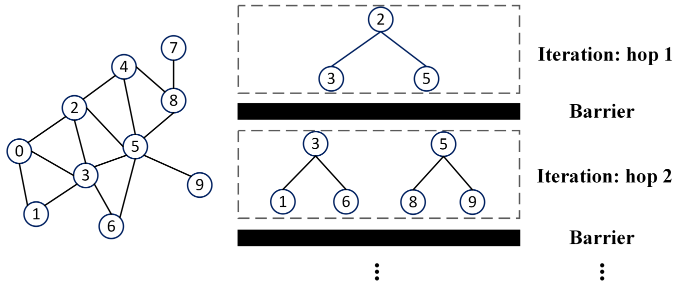

Fig. 1. The BSP execution model for graph sampling at each hop.

To scale GNN training to larger graphs, a typical approach is sample-based training. Various graph sampling algorithms, including node-wise [9] and layer-wise [10], [11] approaches, have been proposed. Through these methods, only the features of sampled vertices are utilized in subsequent mini-batch training, effectively reducing both computations and memory consumption. In this paper, we focus on uniform neighbor sampling [9], a fundamental algorithm for sample-based training, which uniformly selects a fixed number of neighbors for each seed vertex depending on the fanout parameter.

Despite the benefits of graph sampling, the process itself can be time-consuming on CPU [12]. To accelerate it, based on the observation that the graph topological data can be typically fitted into GPU memory, the GPU-based graph sampling has been introduced in the recent GNN systems [7], [13]. Similar to numerous parallel algorithms on GPU, the core of GPU-based graph sampling is implemented within a GPU kernel function, commonly referred to as a sampling kernel.

Although GPU-based graph sampling outperforms the CPU-based method, there is still considerable potential for improving the efficiency. First, in most existing GNN systems, the graph sampling at each hop is executed following the Bulk Synchronous Parallel (BSP) model [14], as shown in Fig. 1. The sampling kernel is executed once per iteration, and barrier synchronization should be performed between each iteration. We observe that due to the nature of neighbor sampling algorithm, as the iteration count rises, the workload (the number of vertices to be sampled) increases, but it remains relatively small especially at the first few hops. Kernels that process small workload cannot keep GPU computation and memory units busy enough, leading to low resource utilization.

[*]Hengshan Yue is the corresponding author.

Besides, with small workload, preprocessing can dominate the whole sampling process, rather than kernel execution. Second, in real-world graphs, the distribution of vertex degrees often follows a power-law distribution [15]. Simply assigning sampling tasks to a work unit (e.g., warp, block) on GPU may cause severe load imbalance during the execution of the sampling kernel.

In this paper, we propose *PGSampler*, a novel system that tackles the above challenges. In *PGSampler*, we alter the sampling execution mode, relaxing the barriers between iterations. Instead of processing single-hop sampling workload during each iteration, this approach allows us to fuse workload at each hop, asynchronously managing all workloads of *L*-hop sampling, leading to improved resource utilization. Besides, our execution mode requires only a single execution of the sampling kernel when processing *L*-hop sampling, achieving reduced preprocessing overhead. To manage the workload after fusion more efficiently, we adopt a persistent kernel design [16] to achieve dynamic load balancing, scheduling tasks through a task queue. This allows work units to check the queue for more tasks once they become idle, and continue this process until no task is left. So, all work units are always busy and can be expected to have balanced workloads, regardless of whether they handle numerous small tasks or a few large tasks during the kernel execution.

Recently, many optimizations have been proposed to improve resource utilization and achieve load balancing in GPU-based graph sampling. TurboGNN [17] optimizes atomic operations in sampling and uses a predefined task reassignment strategy to balance the workload; NextDoor [12] employs "transit-parallelism" and uses three types of GPU kernel, assigning vertices to different kernels based on their required threads. Both approaches yield substantial performance enhancements. However, they both ignore the performance bottlenecks caused by small workload, may result in limited improvements. And they both use static load balancing methods to assign tasks, which requires specialized scheduling strategies, may introduce additional overhead and lack adaptability to dynamically generated tasks.

In summary, we make the following contributions.

- We achieve the fusion of workload and the reduction of preprocessing time by altering the execution mode of the sampling process, which relaxes the barriers between iterations.
- We propose an efficient graph sampling kernel for fused workload, which achieves dynamic load balancing in a single persistent kernel.
- We finally implement *PGSampler* on top of the DGL. Experimental results show that *PGSampler* can achieve up to 1.57× speedup on the sampling kernel and up to 2.22× speedup on the whole sampling process compared to DGL.

II. Background and motivation

In this section, we provide a brief background of GNNs followed by an introduction to sample-based GNN training and GPU-based graph sampling. Finally, we discuss the motivation of our work.

A. Graph Neural Networks

Given a graph $G = (V, E)$, the input of a GNN model includes two components: the topological structure of the input graph and the dense feature vectors associated with each vertex. A GNN model consists of multiple layers, where each layer updates the feature of each vertex by gathering information from its neighbors. The core computations in a GNN layer can be summarized as follows:

$$\begin{aligned} a_v^{(k)} &= Aggregate^{(k)}\left(\left\{h_u^{(k-1)} \mid u \in \mathcal{N}(v)\right\}\right) \\ h_v^{(k)} &= Update^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right) \end{aligned} \tag{1}$$

As shown in (1), the feature vector of vertex v at layer k is denoted by h_v^k, and its neighbors are represented by $\mathcal{N}(v)$. The $Aggregate$ function determines how information from neighboring vertices is aggregated, while the $Update$ function determines how the feature vector of the current vertex is updated based on its aggregated information. These two functions vary across different GNN models [9], [18]–[20], aiming to achieve optimal performance in diverse scenarios.

B. Sample-based GNN Training

While GNNs offer powerful capabilities for learning from the graph-structured data, their training often encounters challenges. In traditional GNN training, each vertex gathers information from all neighboring vertices, this makes training hard to scale. On the one hand, real-world graphs can contain millions or even billions of vertices and edges, making it computationally expensive to process all the information associated with each vertex and its neighbors. On the other hand, vertices in real-world graphs often have high-dimensional feature vectors, which further increases the memory footprint of the GNN training. To address these issues, sample-based training is widely adopted. In each mini-batch training iteration, the whole process of this approach can be divided into four stages. First, a batch of training vertices is taken as seeds, and their neighbors are sampled in CPU based on a specific graph sampling algorithm to create a subgraph for each GNN layer. Next, feature vectors of the input vertices are extracted as input features. Then, subgraphs and input features are transferred into the GPU via PCIe. Finally, mini-batch GNN training is performed on the GPU with the sampled subgraphs and loaded features. Fig. 2 illustrates the pipeline of sample-based GNN training discussed above. This process will run iteratively until the model converges to the desired accuracy.

C. GPU-based Graph Sampling

The increasing scale of graph datasets has made graph sampling a critical bottleneck in the training process. Thus, leveraging GPU to accelerate graph sampling has been an effective approach in many GNN systems. In contrast to CPU-based sampling, GPU-based graph sampling involves loading graph topological data into GPU memory beforehand. The

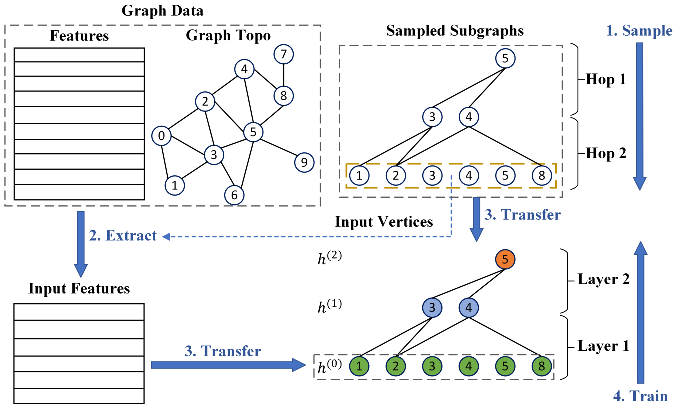

Fig. 2. Sample-based training for a 2-layer GNN on V_5, $h^{(k)}$ denotes the feature vectors at layer k. (sampling algorithm: uniform neighbor sampling, batch size = 1, fanout for each layer = 2)

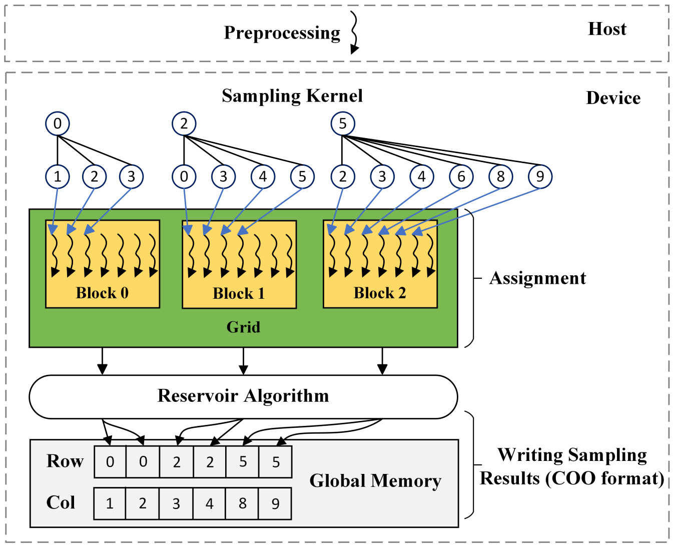

Fig. 3. The single-hop GPU-based graph sampling process implemented in DGL.

sampled results then need to be returned to the CPU for extracting the features of the sampled vertices. DGL is a widely-used framework that provides comprehensive graph operations, including GPU-based graph sampling. Due to its versatility, we center our discussion on DGL when introducing the implementation of GPU-based graph sampling. Fig. 3 shows the single-hop GPU-based graph sampling process implemented in DGL (V1.0.1). In the sampling kernel, each vertex is assigned to a GPU block, each thread within a block determines whether a neighbor should be sampled based on the Reservoir algorithm [21]. However, GPU-based graph sampling can still take a substantial portion of one mini-batch training iteration [17]. There are many factors that can influence the performance of the sampling kernel. As shown in Fig. 3, the complexity of the sampling task for each block varies due to differences in the degrees of the assigned vertices. This discrepancy can be quite significant especially when processing power-law graphs, leading to severe load imbalance. Moreover, uncoalesced memory access, warp divergence and atomic operations in the sampling kernel can cause low resource utilization. Therefore, prior works [12], [17], [22] have proposed their optimized hardware-friendly sampling kernels to further improve the efficiency of GPU-based graph sampling.

TABLE I
THE PROFILING RESULT OF THE SAMPLING PROCESS DURING ONE BATCH TRAINING ITERATION. WORKLOAD DENOTES THE NUMBER OF VERTICES TO BE SAMPLED. BATCH SIZE: 2048. FANOUT TUPLE: (10, 10, 10). DATASET: OGBN-PRODUCTS. (PRE: PREPROCESSING TIME. KERNEL: SAMPLING KERNEL EXECUTION TIME. UTILIZATION: THE PERCENTAGE OF THE PEAK VALUE.)

Hop	**Workload**	**Pre** (μs)	**Kernel** (μs)	**Utilization** (%)	
				Comp	Mem
Hop 1	2048	129.0	13.2	15.48	5.97
Hop 2	21633	127.7	92.4	48.68	19.49
Hop 3	179709	127.0	662.0	55.02	24.79

D. Motivation

Despite previous efforts achieving significant improvements in the resource utilization of the sampling kernel, their primary focus lies on computation or memory access patterns. The execution of their sampling process is still based on the BSP model at each hop, which is intuitive in programming but may introduce small workload during the sampling process. To attain a deep understanding of the performance bottlenecks caused by small workload, we perform 3-hop graph sampling using NeighborSampler [23] in DGL on Ogbn-products dataset [24] and profile the performance during one mini-batch training iteration via Nvidia Nsight tools [25], [26].

As shown in Table I, the resource utilization (including computation and memory throughput) at the first two hops is much lower compared to the last hop. The experimental results show that small workload cannot fully utilize the massively parallel resources of the GPU. We also measure the preprocessing time and sampling kernel execution time. The preprocessing overhead includes (1) the time required for data transfer between host and device (e.g. memory operation initiated by the CUDA API), (2) the execution of essential host function calls before kernel launch (e.g. data formatting), and (3) the kernel launch overhead. Even worse, with small workload, preprocessing can take much longer time than sampling kernel execution at the first few hops, causing GPU-based graph sampling inefficient. It is noteworthy that the issue of small workload is nearly inevitable, as increasing the batch size to a large value can lead to a considerable slowdown in convergence [27].

In fact, the sampling process of hop $(n+1)$ can begin as soon as seed vertices for it have been produced by hop n. That is, the sampling process of multiple hops can be executed almost simultaneously rather than sequentially processing single-hop sampling at a time. As shown in Fig. 2, V_3 is the sampled neighbor of V_5 during the first-hop sampling, it is also

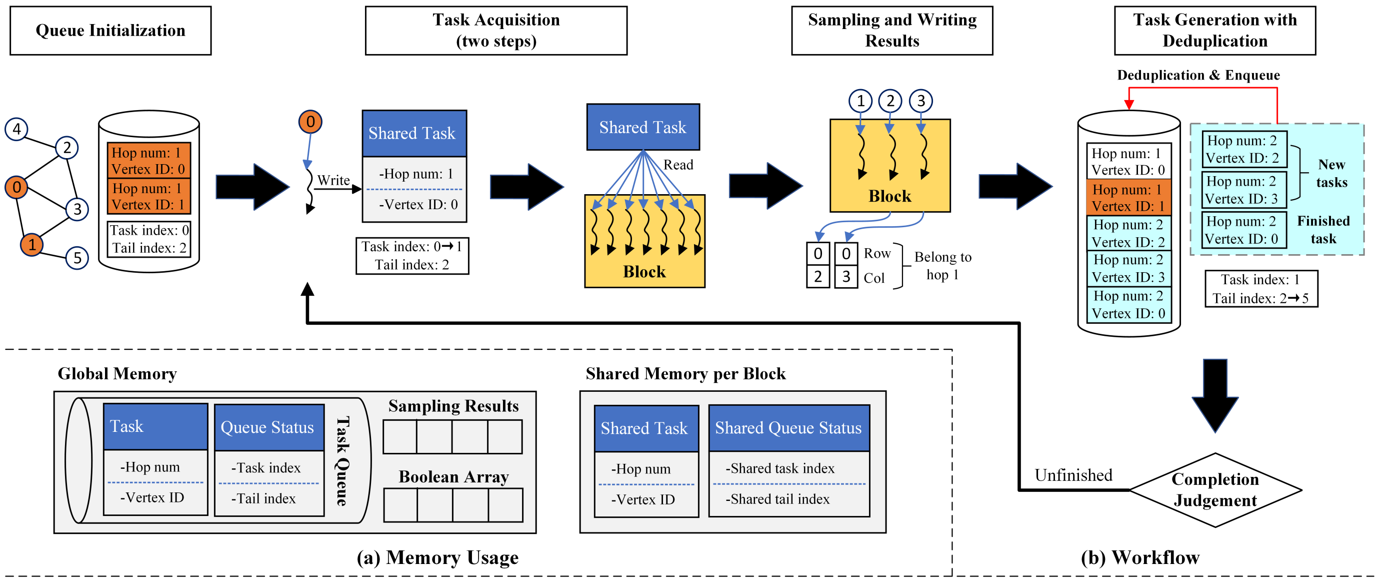

Fig. 4. The design of *PGSampler*.

one of the seed vertices for the second-hop sampling. Once V_3 is sampled, the second-hop sampling can begin immediately without the necessity to await the completion of the first-hop sampling. This observation presents an opportunity to relax the barriers between iterations. In contrast to the traditional execution mode, without the restrictions on iteration order, all workloads of L-hop sampling can be managed within a single kernel execution without additional synchronization. Because the entire multi-hop sampling is performed continuously on the GPU, data (e.g. intermediate results) can be reused for subsequent hops, further enhancing the efficiency of the sampling process.

While the introduction of workload fusion can significantly increase the workload of single sampling kernel to a considerable size, it also brings challenges to the task assignment. CUDA programming allows a thread to execute multiple times in a loop [28], so load balancing can be achieved through various static allocation methods. However, new tasks are dynamically generated as we fuse workload together. Many proposed static load balancing methods [17] rely on a predetermined number of tasks, which may introduce additional overhead and are unsuitable for dynamic task generation. Thus, our target is to achieve dynamic load balancing during task assignment in the sampling kernel.

III. PGSampler

In this section, we propose *PGSampler*, an efficient system designed to facilitate fast GPU-based graph sampling on various real-world graph datasets. We first present an overview of *PGSampler* and then introduce two key techniques in it: (1) a novel sampling execution mode for fusing workload and reducing preprocessing overhead, and (2) a persistent sampling kernel to achieve dynamic load balancing.

A. Overview

Fig. 4 illustrates the overview of *PGSampler*. We also choose the block as the work unit, and after initialization, each block will loop until all tasks are completed due to the persistent kernel design. The loop mainly encompasses four steps: (1) task acquisition; (2) sampling and writing results; (3) task generation with deduplication; and (4) completion judgement. As for memory usage, we leverage both global memory and shared memory. The task queue which contains tasks and queue status is stored in global memory. Additionally, sampling results and a boolean array for deduplication usage are stored in global memory. The task fetched for current iteration is stored in shared memory as a shared task, along with shared queue status, which will be updated every time before judging the completion of sampling. (see Section III-C for more details about deduplication and completion judgement)

B. Sampling Execution Mode

The current design of sampling kernels, such as in DGL, typically treats each sampled vertex as a task, without explicitly leveraging hop number information. As shown in Fig. 5a, because the L-hop sampling is executed within an iteration, the hop number information is implied in the iteration rounds. However, as discussed in Section II-D, with the BSP execution model at each hop, the issue of small workload can become the performance bottleneck of GPU-based graph sampling.

To address this issue, based on the observation that the sampling process of multiple hops can be executed almost simultaneously, we introduce a novel task definition to achieve barrier-free sampling execution mode in *PGSampler*, fusing the workload at each hop. Specifically, now each task is a structure with two attributes: vertex ID and hop number. As shown in Fig. 4, with seed vertices V_0 and V_1, the initial tasks

```python
class NeighborSampler(...):  # DGL
  def __init__(self, fanouts, ...):
    self.fanouts = fanouts
    ...
  def sample_blocks(self, graph, seed_nodes, ...):
    output_nodes = seed_nodes
    blocks = []
    # Iterate L times for L-hop sampling
    # Execute sampling kernel L times
    for fanout in reversed(self.fanouts):
      # Single-hop sampling
      frontier = sample_neighbors(graph, seed_nodes,
        fanout, ...)
      # Convert a sampled subgraph into a "DGL block"
      block = to_block(frontier, seed_nodes)
      # Update seed nodes for next-hop sampling
      seed_nodes = block.srcdata[NID]
      blocks.insert(0, block)
    return seed_nodes, output_nodes, blocks
```

(a) DGL

```python
class PersistentNeighborSampler(...):  # Ours
  def __init__(self, fanouts, ...):
    self.fanouts = fanouts
    ...
  def pers_sample_blocks(self, graph, seed_nodes, ...):
    output_nodes = seed_nodes
    blocks = []
    # Without iteration
    # Only execute sampling kernel once
    frontier_list = pers_sample_neighbors(graph, seed_nodes,
      self.fanouts, ...)
    # Convert sampled subgraphs into blocks
    ...
    input_nodes = blocks[0].srcdata[NID]
    return input_nodes, output_nodes, blocks
```

(b) Ours

Fig. 5. Our sampling execution mode compared to DGL.

Algorithm 1: The persistent sampling kernel

```
Input: Initialized task queue $queue$, queue status
       $taskIndex$ and $tailIndex$, The number of
       tasks fetched in each iteration $fetchSize$,
       graph $G$, the number of GNN layers
       $layerNum$, fanout at each hop $fanouts$,
       random seed $seed$
Output: Sampling results $resArray$
1  __shared__ int $sharedTaskIndex$;
2  __shared__ int $sharedTailIndex$;
3  __shared__ struct {
4      int hopNum;
5      int vertexID;
6  } $sharedTasks[fetchSize]$;
7  if $threadIdx.x == 0$ then
8      $sharedTaskIndex = blockIdx.x$;
9      $sharedTailIndex = tailIndex$;
10 end
11 __syncthreads();
12 while $sharedTaskIndex < sharedTailIndex$ do
13     if first iteration then
14         $sharedTasks[0] = queue[sharedTaskIndex]$;
15     else
16         for $i = 0$ to $fetchSize - 1$ in parallel do
17             $loc = sharedTaskIndex + i$;
18             $sharedTasks[i] = queue[loc]$;
19         end
20     end
21     __syncthreads();
22     foreach $task \in sharedTasks$ do
23         $neighbors = \textbf{Sample}(task, seed, G, fanouts)$;
24         $\textbf{WritingResults}(task, neighbors, resArray)$;
25         if $task.hopNum < layerNum$ then
26             $newTasks = \textbf{TaskGen}(task, neighbors)$;
27             $dTasks = \textbf{Deduplication}(G, newTasks)$;
28             foreach $dTask \in dTasks$ in parallel do
29                 $tail = \text{atomicAdd}(\&tailIndex, 1)$;
30                 $queue[tail] = dTask$;
31             end
32         end
33     end
34     if $threadIdx.x == 0$ then
35         $sharedTaskIndex =$
             $\text{atomicAdd}(\&taskIndex, fetchSize)$;
36         $sharedTailIndex = tailIndex$;
37     end
38     __syncthreads();
39 end
```

are two structures whose hop number attributes are both set to 1. That means every time a task is assigned to a block, we can not only get the vertex to be sampled, but also know which hop does this sampling task belongs to. Based on the hop number, the sampling results can be accurately written to the corresponding locations (see Section IV-A for more details about result format). Subsequently, new tasks can be generated by assigning the vertex ID as the sampled neighbors ID, and incrementing the hop number by 1. As shown in Fig. 4, with sampled neighbors V_2 and V_3, the new generated tasks are two structures whose hop number attributes are both set to 2. Therefore, instead of processing single-hop sampling workload during each iteration, the workloads at different hops have been effectively fused, allowing the management of all workloads in L-hop sampling within a single kernel execution. Based on our execution mode, now there is no need to follow the BSP model at each hop, iterating L times for L-hop sampling. As shown in Fig. 5b, the sampling kernel is executed only once, which means that only a single preprocessing step is required, thereby reducing preprocessing overhead.

C. Sampling Kernel

Following the alteration of the execution mode to achieve workload fusion, the sampling kernel should possess the capability to manage all workloads of L-hop sampling. A practical approach involves launching a sufficient number of blocks, then storing all tasks (including both initial and generated tasks) in global memory. Each block is responsible for one task before concluding its execution. However, this approach can cause load imbalance issue as previously mentioned in Section II-C. Additionally, when processing large-scale real-world

Algorithm 2: Deduplication algorithm

```
Input: Graph $G$, generated tasks $newTasks$, boolean
       array $boolArray$
Output: Tasks after deduplication $dTasks$
1  foreach $task \in newTasks$ in parallel do
2      $vertexID = task.vertexID$;
3      $hopNum = task.hopNum$;
4      $vertexNum = G.vertexNum$;
5      $loc = vertexID + vertexNum * (hopNum - 1)$;
6      if $boolArray[loc] == false$ then
7          $oldValue =$ atomicOr($boolArray + loc, 1$);
8          if $oldValue == false$ then
9              Append $task$ to array $dTasks$;
10         end
11     end
12 end
```

graphs, the execution order of millions of blocks depends on GPU hardware scheduling, leading to high scheduling overhead. Inspired by the persistent kernel design [16], the key idea of our sampling kernel is to achieve dynamic load balancing tailored for dynamically generated tasks through the utilization of a global task queue. The detail of our sampling kernel is described in Algorithm 1.

1) Persistent kernel design: Different from the traditional kernel design, where the number of launched blocks is related to the work size, a persistent kernel only launches enough blocks to reach the maximum occupancy of GPU Streaming Multiprocessors (SMs). So once all blocks are scheduled to the SMs, there will be no additional hardware scheduling overhead. During the entire kernel execution, these blocks remain resident and continuously work within a loop. In each iteration, tasks are assigned through a task queue. As shown in Fig. 4, every time a block completes the current task, it will try to fetch new tasks from the queue, unless all tasks have been completed. Unlike the manual assignment of tasks based on a predetermined strategy, which is common in many static load balancing methods, this queue-based scheduling approach is indifferent to which task a block is assigned. Its primary focus is on keeping the block busy, thereby achieving dynamic load balancing: a block can handle either numerous small tasks or a few large tasks. It is better suited for dynamically generated tasks, as the application of a static method might necessitate frequent execution of certain operations (e.g., sorting) on dynamically generated tasks, leading to potential performance degradation.

As for the implementation of the task queue, we utilize two global counters to track the status of the queue: $taskIndex$ (indicating the first task to be processed) for popping and $tailIndex$ (indicating the end of the task queue) for pushing. Modifications to both counters are based on atomic operations to guarantee data consistency in concurrent scenarios.

2) Task acquisition: Every time a block tries to fetch new tasks, we first choose a leader thread (line 7, line 34 in Algorithm 1) to store the starting index into shared memory. The initial value of $sharedTaskIndex$ in each block corresponds to the block ID (line 8). Initially, each block fetches only one task (line 13-14). Upon completing this task, the subsequent value of $sharedTaskIndex$ is determined using the $atomicAdd$ function (line 35). Each block then fetches $fetchSize$ tasks for the next iteration and stores them into the $sharedTasks$ array (line 16-19). Consequently, threads within the block can read the shared memory to access the same tasks when executing the reservoir algorithm for sampling (line 23).

3) Deduplication: The presence of common neighbors in the graph introduces the possibility of duplicate tasks. As shown in Fig. 4, V_2 is the common neighbor of V_0 and V_4. If both V_0 and V_4 sample V_2 at the first hop, only one new task with vertex ID set to 2 should be enqueued for the next hop sampling. So after new tasks are generated using the method described in Section III-B, deduplication should be performed before pushing them into queue. The pseudo-code of the deduplication algorithm is presented in Algorithm 2. We utilize a global boolean array to judge whether the same task is already present in the queue. An index is calculated first for each generated task (line 5). Subsequently, we check the corresponding value in the boolean array (line 6) and use $atomicOr$ function to ensure the correctness of the value in concurrent scenarios (line 7). If a task is not duplicate, it will be appended to the deduplication result array (line 9).

After deduplication, with index determined by $atomicAdd$ function, generated tasks will be pushed into the queue (line 28-30 in Algorithm 1).

4) Completion judgement: The simplest method to judge completion is to validate whether the number of completed tasks has reached a specified value. However, because of the stochastic nature of graph sampling, accurately estimating the total number of tasks in advance is challenging, thereby may complicate the judgment for task completion.

In fact, despite the presence of duplicate tasks, the number of new tasks enqueued by each block after deduplication in each iteration is still proportional to the $fanouts$ parameter. Therefore, as long as the value of $fetchSize$ is appropriately set, ensuring tasks are consumed slower than they are generated, $tailIndex$ will always increase faster than $taskIndex$. Since the sampled neighbors at the last hop are not used to generate new tasks (line 25 in Algorithm 1), $tailIndex$ will eventually stop to increase. Therefore, $taskIndex$ will catch up with $tailIndex$ only when all tasks are completed. In order to ensure that the judgment results of all threads in the block are consistent, besides $sharedTaskIndex$, we have introduced another shared variable, i.e., $sharedTailIndex$, whose initialization and update are performed simultaneously with those of $sharedTaskIndex$ (line 9, line 36). In summary, in *PGSampler*, when $sharedTaskIndex$ is not less than $sharedTailIndex$ (line 12), all tasks are guaranteed to be completed.

To determine the value of the $fetchSize$, a crucial constraint is making it less than $fanouts$. Based on our parameter settings (see Section V-A3) and considering the influence of

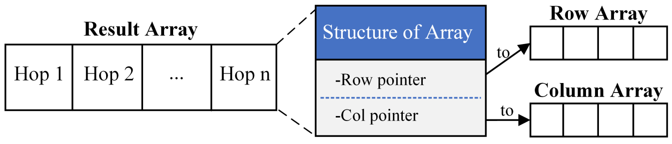

Fig. 6. Result array for the sampling of n-layer GNN.

```python
import dgl
import torch
from dgl.dataloading import (
  DataLoader,
  NeighborSampler,
  PGSampler
)

fanouts = [10, 10, 10]
# Create a sampler
# dgl_sampler = NeighborSampler(fanouts)
custom_sampler = PGSampler(fanouts)

# Create dataloader
train_dataloader = DataLoader(
  graph=g,
  indices=train_idx,
  # Choose a sampler
  # graph_sampler=dgl_sampler,
  graph_sampler=custom_sampler,
  device=torch.device('cuda:0'),
  batch_size=4096
)

# Training
for epoch in range(iter_num):
  # Mini-batch training iteration
  # Training with sampled vertices and bipartite graphs
  for input_nodes, output_nodes, blocks in train_dataloader:
  ...
```

Fig. 7. The use case of *PGSampler*.

duplicate tasks, $fetchSize$ is consistently set to 5 during evaluation.

IV. Implementation

We built *PGSampler* on the top of the DGL (v1.0.1) [7] and PyTorch (v1.13) [5]. In this section, we first delve into two key implementation details of *PGSampler*: the sampling result format and the reuse of finished tasks. We finally describe how our system integrates with DGL.

A. Result Format

Aligned with the implementation in DGL, in *PGSampler*, the final output of the sampling process should be multiple subgraphs represented as adjacency matrices in COO format. To achieve this, the sampling results of each vertex are written to an "array of structure of array", as shown in Fig. 6. The length of the array is equal to the number of GNN layers. Each element of the array is a structure, and each structure contains two pointers. Specifically, one pointer to a row array and another pointer to a column array, storing the sampling results for a single hop. This result format facilitates coalesced memory access during the results writing phase.

TABLE II
Details of graph datasets used for evaluation.

Dataset	#Vertex	#Edge	Feature	Class
Reddit	232,965	114,615,892	602	41
Flickr	89,250	899,759	500	7
Yelp	716,847	13,954,819	300	100
Ogbn-arxiv	169,343	1,166,243	128	40
Ogbn-products	2,449,029	61,859,140	100	47
Coauthor-physics	34,493	495,924	8,415	5

B. Reuse of Finished Tasks

In many GNN models, such as GraphSAGE [9], the update of vertices features needs to use their own features from the preceding layer. So DGL always includes the destination vertices (vertices to be sampled) themselves in the source vertices (sampled neighbors) [29]. In *PGSampler*, we also try to use the destination vertices themselves for the subsequent hop sampling if their corresponding tasks are not duplicate in the queue. As shown in Fig. 4, during task generation, the finished task may be re-introduced after deduplication. In this case, the vertex ID remains unchanged, while the hop number is incremented by 1.

C. Integration

The core of *PGSampler*, i.e., the sampling kernel, is implemented in CUDA and C++ for performance reasons. To seamlessly integrate the kernel implementation with the Python front-end interface provided by DGL, we leverage the DGL Foreign Function Interface (FFI) [30]. As shown in Fig. 7, by adopting this approach, programmers can use our custom sampler conveniently just like any DGL's existing samplers.

V. Evaluation

In this section, we first comprehensively evaluate the performance of *PGSampler* on real-world graph datasets. Following this, we analyze the effectiveness of our optimization techniques. Finally, we validate the correctness of *PGSampler* by examining the training convergence.

A. Experimental Setup

Unless specified otherwise, all results are reported as the average of 10 training epochs.

1) Environments: We conducted our experiments on a server that consists of two Intel Xeon E5-2680v4 2.40GHz CPUs, 125GB main memory and a single NVIDIA RTX 3090 (24GB memory) GPU, except for the experiments in Table III, which were performed on a NVIDIA Tesla P40 GPU. The server is installed with Ubuntu 20.04, GCC 9.5.0, CUDA library 11.7, DGL v1.0.1 and PyTorch v1.13.

2) Datasets: The graph datasets used for evaluation are listed in Table II, including a social network graph Reddit [9], Flickr and Yelp introduced in the GraphSAINT paper [31], two datasets from Open Graph Benchmark (OGB): a co-purchasing

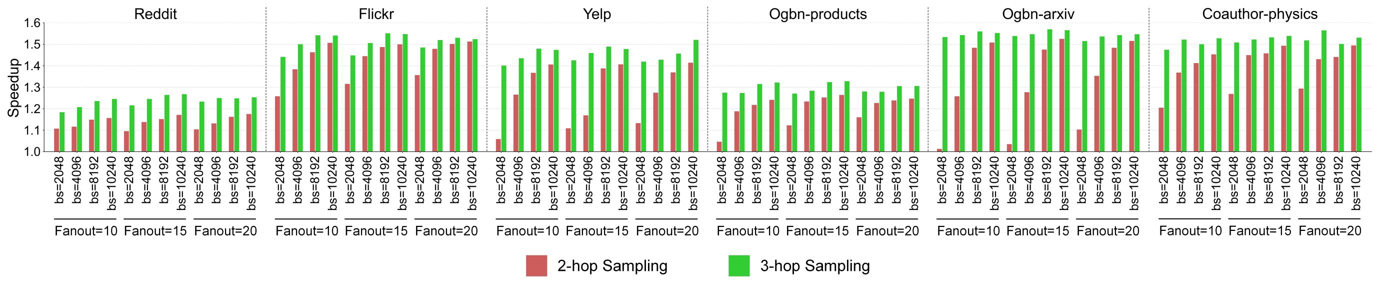

Fig. 8. The sampling kernel speedup over DGL.

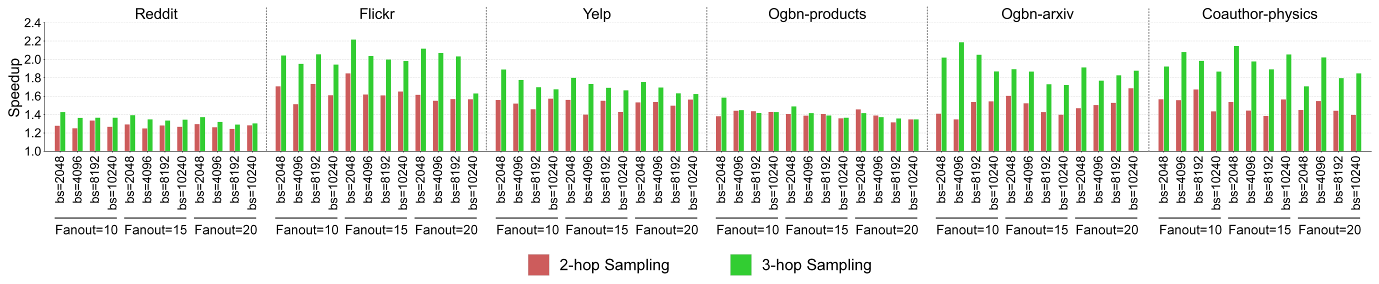

Fig. 9. The whole sampling process speedup over DGL.

TABLE III
THE SAMPLING KERNEL PERFORMANCE COMPARED TO TURBOGNN. BOTH SAMPLERS PERFORM 3-HOP SAMPLING. SOD DENOTES THE SPEEDUP OVER DGL.

Dataset	Sampler	Batch Size	Fanout	SoD
Reddit	PGSampler	4K	15	1.23×
	TurboGNN	4K	30	1.15×
Ogbn-products	PGSampler	10K	15	1.36×
	TurboGNN	200K	15	1.30×

network Ogbn-products [24] and a citation network Ogbn-arxiv [32], and the ”physics” part of the Coauthor dataset [33]. For the OGB datasets, the official training sets provided by OGB are employed, while for the remaining datasets, we follow the data splits provided by DGL.

3) Parameter settings: We run the experiments with multiple parameter settings, all of which are widely adopted to achieve optimal training performance in many GNN systems. [27]. The number of sampled vertices (batch size) tested in the experiments includes 2048, 4096, 8192 and 10240, while the choice of the number of neighbors to be sampled for each vertex (fanout) includes 10, 15 and 20. Besides, in the experiments, the fanout parameters for each layer are the same. For example, for a 3-layer GNN, the fanout tuple can be set to (10, 10, 10), (15, 15, 15) or (20, 20, 20).

4) Models: Within each mini-batch training iteration, GPU sampling and subsequent training are two completely separate stages in DGL. Therefore, the choice of the GNN model for training does not influence the evaluation of sampling performance. Accordingly, only GraphSAGE [9] is used as the representative GNN model when evaluating the performance of our custom sampler. To evaluate 2-hop and 3-hop neighbor sampling, we create 2-layer and 3-layer models, with the dimension of the hidden layers set to 256.

5) Baselines: We choose two baseline implementations for comparison: (1) DGL [7] is the state-of-the-art GNN system that supports multiple backends. We choose PyTorch version in this work. (2) TurboGNN [17] introduces a series of optimization techniques to enhance the end-to-end performance of the sampling-based GNN training pipeline. We compare with its GPU sampling module in the evaluation.

B. Performance Results

We first compare the sampling kernel performance of *PGSampler* with DGL’s neighbor sampler [23]. Our optimization techniques achieve significant improvements in resource utilization, with computation throughput up to 1.59× and memory throughput up to 2.01× over DGL. Fig. 8 shows the sampling kernel speedup over DGL on different datasets. Note that due to the difference in sampling execution modes, we compare the total sampling kernel execution time with DGL. Our results indicate that *PGSampler* outperforms DGL in all the cases and achieves up to 1.57× speedup. We observe that our method performs slightly better on Flickr (1.47× on average), Ogbn-arxiv (1.44× on average) and Coauthor-physics (1.46× on average). This is because for the sampling in DGL on small graphs, despite the workload increasing with the sampling hop, it tends to be quickly constrained by the total number of vertices. Thus, the issue of small workload can still be the main concern even at the last hop, leading to more performance gain from our workload fusion. For larger and more power-law graph datasets like Reddit, Yelp and Ogbn-

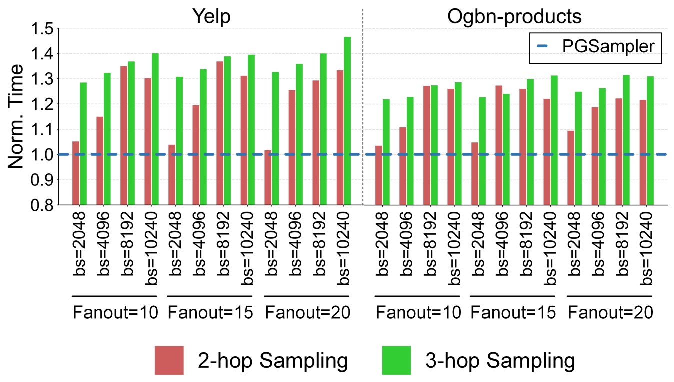

Fig. 10. Kernel execution time of the "DGL+Fusion" compared to *PGSampler*.

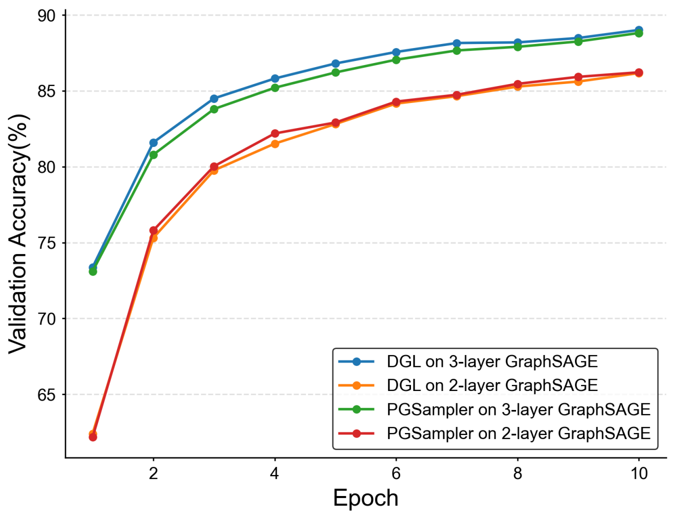

Fig. 11. Validation accuracy using *PGSampler* and DGL during training. Dataset:Ogbn-products. Batch size: 10240. Fanout tuple: (10, 10, 10).

products, load imbalance emerges as a primary challenge once the workload at each hop becomes large enough. Due to our implementation of the dynamic load balancing, significant performance improvements can also be achieved on these datasets (1.19×, 1.37× and 1.23× speedup on average on Reddit, Yelp and Ogbn-products, respectively).

We also find that speedup increases when batch size, fanout or the number of sampling hops increases. This phenomenon arises due to the larger workload at each hop, which results in a larger total workload after fusion. This achieves further improvement of resource utilization and amplifies the effectiveness of load balancing, leading to a more significant speedup.

As previously mentioned, our barrier-free sampling execution mode should also benefit the whole sampling process due to the reduction of preprocessing time. As shown in Fig. 9, *PGSampler* can achieve up to 2.22× speedup on the whole sampling process over DGL. Specifically, one can notice that across different datasets, the best performance speedup is always attained when performing 3-hop sampling with relatively small batch size and fanout. This is because at these points, the sampling kernel performance has been well optimized, while preprocessing still dominates the whole sampling process.

C. *Effectiveness of Workload Fusion*

Next, we compare *PGSampler* with the GPU sampling module presented in TurboGNN [17], which also outperforms DGL but still follows the BSP execution model for the sampling at each hop. TurboGNN has not open-sourced its implementation. In their paper, they use two datasets (Reddit and ogbn-products) to evaluate their sampling kernel. For a fair comparison, we (1) use the same two datasets and (2) use the GPU (Tesla P40) that is comparable with the GPU of TurboGNN (Tesla P100) in performance-critical factors. As shown in Table III, *PGSampler* outperforms TurboGNN on both two datasets. Specifically, *PGSampler* achieves more performance gain with a 2× reduction in fanout on Reddit and a 20× reduction in batch size on Ogbn-products. The results show that through workload fusion, *PGSampler* demonstrates improved performance with relatively small batch size and fanout, which are commonly utilized settings in GNN training to speed up convergence [27].

D. *Effectiveness of Dynamic Load Balancing*

To evaluate the effectiveness of dynamic load balancing, we first design a relevant baseline which only incorporates the workload fusion, denoted as "DGL+Fusion", and then compare *PGSampler* with DGL+Fusion on two graphs due to the space limit. As shown in Fig. 10, without dynamic load balancing, DGL+Fusion performs worse than *PGSampler* in all the cases, especially when processing 3-hop sampling with large batch size. This is due to the dependence on GPU hardware scheduling, as the total workload increases, DGL+Fusion will suffer from high block scheduling overhead, leading to significant performance degradation.

E. *Training Convergence*

To validate the correctness of our implementation, we evaluate the validation accuracy during GNN training with *PGSampler* and DGL's neighbor sampler on the Ogbn-products dataset. As shown in Fig. 11, on both 2-layer and 3-layer GraphSAGE, the convergence behavior of *PGSampler* is approximately the same as the DGL's built-in sampler, which confirms that the sampling results of *PGSampler* are as expected.

VI. Related Work

A. *GNN Systems*

The unique characteristics of graph data and the message-passing nature of GNN training necessitate specialized system design considerations compared to traditional DNNs. Apart from DGL [7] and PyG [8], many GNN systems [34]–[38] have been designed for training graph neural networks. Due to the prevalence of large-scale graphs, a single GPU becomes insufficient for processing millions, or even billions, of vertices and edges. Thus, these systems primarily concentrate on accelerating distributed training on multiple GPUs. For instance, NeutronStar [38] proposes a hybrid dependency management to improve communication and computation efficiency. To demonstrate the practical applicability of *PGSampler*, we integrate it into DGL, which is one of the most popular GNN systems. While our implementation is mainly oriented to single-GPU training, techniques such as graph partitioning [39] enable straightforward scaling of our optimizations to multi-GPU scenarios.

B. Optimizations in Sample-based GNN Training

Many works have been proposed to accelerate sample-based GNN training, targeting improvements on different stages of the training pipeline. For the sampling stage, typical works like C-SAW [22] and NextDoor [12] implement efficient GPU sampling kernels based on different parallelism paradigms. The GPU sampling module of TurboGNN [17] utilizes shared memory to optimize global atomic operations and achieves balanced workload in sampling. Our work also tries to accelerate the sampling stage. Unlike the aforementioned existing works, *PGSampler* concentrates on addressing the performance bottlenecks caused by small workload, thus achieving high-performance sampling. To accelerate feature extraction, both PaGraph [40] and GNNLab [13] propose efficient caching policies aimed at reducing data loading time. For the training stage, kernel optimizations [41]–[43] play a crucial role in overcoming the challenges of training GNNs on large-scale graphs.

C. Persistent Kernel

Persistent kernel design [16] has been widely adopted to benefit irregular applications, such as graph processing [44], [45]. Compared to traditional kernel design, persistent kernel offers the advantage of reducing kernel launch overhead and facilitates the implementation of dynamic load balancing [46]. Graph sampling for GNN training differs from traditional graph processing algorithms due to its distinctive layer-stacked characteristics and the inherent stochastic nature. Consequently, *PGSampler* introduces deduplication and completion judgment mechanisms in the persistent sampling kernel to ensure both correctness and efficiency.

VII. Conclusion

In this paper, we present *PGSampler*, a novel high-performance system for efficient GPU-based graph sampling. Specifically, we address performance bottlenecks by fusing workload, reducing preprocessing overhead and achieving dynamic load balancing in a single persistent kernel. We evaluate *PGSampler* with a series of parameter settings on 6 graph datasets. Experimental results show that *PGSampler* accelerates the sampling kernel by up to 1.57× and the whole sampling process by up to 2.22× compared to DGL, without compromising the training convergence.

VIII. Acknowledgment

We sincerely thank our shepherd and the anonymous reviewers for their insightful comments and feedback. This work is supported by the National Key Research and Development Program of China (Grant No. 2023YFB4502304) and National Natural Science Foundation of China (NSFC) (Grants No.62272190, No.62302190)

References

[1] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.
[2] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," *Advances in neural information processing systems*, vol. 31, 2018.
[3] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The world wide web conference*, 2019, pp. 417–426.
[4] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur, "Protein interface prediction using graph convolutional networks," *Advances in neural information processing systems*, vol. 30, 2017.
[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.
[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.
[7] 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.
[8] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
[9] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, vol. 30, 2017.
[10] J. Chen, T. Ma, and C. Xiao, "Fastgcn: fast learning with graph convolutional networks via importance sampling," *arXiv preprint arXiv:1801.10247*, 2018.
[11] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu, "Layer-dependent importance sampling for training deep and large graph convolutional networks," in *Advances in neural information processing systems*, vol. 32, 2019.
[12] 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.
[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] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.
[15] L. A. Adamic and B. A. Huberman, "Power-law distribution of the world wide web," *science*, vol. 287, no. 5461, pp. 2115–2115, 2000.
[16] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *2012 Innovative Parallel Computing (InPar)*, 2012.
[17] W. Wu, X. Shi, L. He, and H. Jin, "Turbognn: Improving the end-to-end performance for sampling-based gnn training on gpus," *IEEE Transactions on Computers*, vol. 72, no. 9, pp. 2571–2584, 2023.
[18] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017*, 2017.
[19] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018*, 2018.

[20] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[21] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[22] S. Pandey, L. Li, A. Hoisie, X. S. Li, and H. Liu, "C-saw: A framework for graph sampling and random walk on gpus," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[23] "NeighborSampler," accessed: Dec.18,2023. [Online]. Available: https://docs.dgl.ai/generated/dgl.dataloading.NeighborSampler.html

[24] "Open Graph Benchmark: The ogbn-products dataset," accessed: Dec.18,2023. [Online]. Available: https://ogb.stanford.edu/docs/nodeprop/#ogbn-products

[25] "NVIDIA Nsight Systems," accessed: Dec.18,2023. [Online]. Available: https://developer.nvidia.com/nsight-systems

[26] "NVIDIA Nsight Compute," accessed: Dec.18,2023. [Online]. Available: https://developer.nvidia.com/nsight-compute

[27] H. Yuan, Y. Liu, Y. Zhang, X. Ai, Q. Wang, C. Chen, Y. Gu, and G. Yu, "Comprehensive evaluation of gnn training systems: A data management perspective," *arXiv preprint arXiv:2311.13279*, 2023.

[28] "CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops," 2013, accessed: Dec.18,2023. [Online]. Available: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops

[29] "Introduction of Neighbor Sampling for GNN Training," accessed: Feb.28,2024. [Online]. Available: https://docs.dgl.ai/en/1.1.x/tutorials/large/L0_neighbor_sampling_overview.html

[30] "DGL Foreign Function Interface (FFI)," accessed: Feb.28,2024. [Online]. Available: https://docs.dgl.ai/en/1.1.x/developer/ffi.html

[31] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graph-SAINT: Graph sampling based inductive learning method," in *International Conference on Learning Representations*, 2020.

[32] "Open Graph Benchmark: The ogbn-arxiv dataset," accessed: Dec.18,2023. [Online]. Available: https://ogb.stanford.edu/docs/nodeprop/#ogbn-arxiv

[33] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann, "Pitfalls of graph neural network evaluation," *Relational Representation Learning Workshop, NeurIPS 2018*, 2018.

[34] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "Neugraph: Parallel deep neural network computation on large graphs," in *2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019, pp. 443–458.

[35] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with roc," in *Proceedings of Machine Learning and Systems*, vol. 2, 2020, pp. 187–198.

[36] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "Distdgl: distributed graph neural network training for billion-scale graphs," in *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2020, pp. 36–44.

[37] 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.

[38] Q. Wang, Y. Zhang, H. Wang, C. Chen, X. Zhang, and G. Yu, "Neutronstar: distributed gnn training with hybrid dependency management," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1301–1315.

[39] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.

[40] 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.

[41] 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.

[42] Q. Fu, Y. Ji, and H. H. Huang, "Tlpgnn: A lightweight two-level parallelism paradigm for graph neural network computation on gpu," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 122–134.

[43] Y. Zhou, J. Leng, Y. Song, S. Lu, M. Wang, C. Li, M. Guo, W. Shen, Y. Li, W. Lin *et al.*, "ugrapher: High-performance graph operator computation via unified abstraction for graph neural networks," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 878–891.

[44] Y. Chen, B. Brock, S. Porumbescu, A. Buluc, K. Yelick, and J. Owens, "Atos: A task-parallel gpu scheduler for graph analytics," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–11.

[45] Y. Chen, B. Brock, S. Porumbescu, A. Buluç, K. Yelick, and J. D. Owens, "Scalable irregular parallelism with gpus: getting cpus out of the way," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–16.

[46] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic load balancing on single-and multi-gpu systems," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–12.