GNN Training Systems on Large Graphs: Optimization Strategies for Mini-batch Training

Taotao Wang,
taotaowang@link.cuhk.edu.cn, School of Data Science

***Abstract*—Recently, Graph Neural Networks(GNNs) have succeeded greatly in many applications and domains. Despite its effectiveness, it is still challenging for GNNs to efficiently scale to giant graphs. Mini-batch training has become a de facto way to train GNNs on large graphs. Some sample-based methods are proposed. This paper will introduce two kinds of strategies to optimize the mini-batch training process. They are cached-based methods and UVA-based methods. For cache-based methods, single-cache GNNLab and dual-cache DUCATI will be shown. GNNLab designs a pre-sampling caching policy and uses a factored design to optimize mini-batch training. DUCATI designs two caches, named Adj-cache and Nfeat-cache, respectively, to achieve better trade-offs than baselines. For UVA-based methods, a GPU-oriented data communication approach will be introduced. It achieves high host memory access efficiency by maximizing PCIe packet efficiency and overlaps data transfer with a well-designed asynchronous zero-copy access. Besides that, their experimental results will be compared; it is found that DUCATI performs better than the other two methods.**

***Index Terms*—Graph neural network, Mini-batch training, Data management system**

I. Introduction

GRAPH Neural Networks are powerful tools to handle problems modeled by the graph and have been utilized widely in various applications, such as social network analysis [1], bio-informatics(e.g., protein interface prediction) diction [2], recommendation [3], and others. GNNs are inclined to integrate the information of graph structure into deep learning model so that they can achieve significantly better results than traditional machine learning and data mining methods.

A GNN model generally contains multi-graph convolutional layers, where each vertex aggregates the latest states of its neighbors, updates the state of the vertex, and applies a neural network to the updated state of the vertex. Taking the traditional graph convolutional network (GCN) [4] as an example, in each layer, a vertex uses a sum function to aggregate the neighbor states and its own state, then applies a single-layer Multi-layer Perceptron (MLP) to transform the new state. Such procedures are repeated L times if the number of layers is L. The vertex states generated in the L-th layer are used by the downstream tasks, such as node classification, link prediction, and so on. In the past, many research works have made remarkable progress in the design of GNN models. Prominent models include GCN [4], GraphSAGE [5], GAT [6], GIN [7], and many other application-specific GNN models [3] [8]. On the other hand, to efficiently develop different GNN models, many GNN-oriented frameworks have been proposed based on various deep learning libraries [9]. However, these methods focus on training with a single machine, while the input graphs are very large for these downstream tasks. For example, the Ogbn-Papers100M [10] dataset contains over 111 million nodes and 1.6 billion edges.

As a result, mini-batch training has become the de-facto way to train GNNs on giant graphs. Given a large graph and corresponding node features, we sample the input graph to get a subgraph and then select the corresponding node features for this subgraph. From the perspective of the underlying data structure, sampling is conducted on the adjacency matrix and perform selection on the node features. The subgraph, together with its node features, forms a mini-batch which will be transferred to the GPU for training.

However, the preparation of such mini-batches is notoriously expensive. For example, when training GraphSAGE [5] on the Ogbn-Papers100M, the time for preparation for one mini-batch, which includes the time of generating the mini-batch and transferring it from CPU to GPU, is 240ms, far exceeding the actual computation time (about 10ms) for forward computation, backward propagation, and weights update [11].

To reduce the long preparation time of mini-batches, quite amount of efforts have been devoted to accelerating sample subgraphs and select node features. One line of the works propose to adopt the unified virtual addressing technique [12], which is called UVA method. Such technique can improve the efficiency of utilizing PCIe bandwidth when transferring graph data to the GPU. Moreover, UVA allows us to store the large graph data in the host memory while harnessing the GPU to select node features and sample subgraphs. Another line of works are cache-based systems which accelerate the preparation of mini-batches by leveraging the locality of graph data and the under-utilized GPU memory. On the one hand, the access to graph data during training exhibits a strong locality. For example, when training GraphSAGE on the Ogbn-Papers100M dataset containing over 53 GB of node features, 95% of node feature accesses during training occur in the most frequently accessed 2 GB of node features. On the other hand, up to 90% of the GPU memory would be left unused during training [13]. Based on these two lines of works, three methods will be introduced in the methods section.

The first introduced work is a single cache method, named GNNLab [14]. It is a facored system for sample-based GNN training in a single machine multi-GPU setup. On one hand, GNNLab adopts a space sharing design for multiple GPUs, i.e., one GPU loads either graph topological data or cached

features in its memory, and only conducts either graph sampling or model training based on its stored data. It eliminates GPU memory contention by leaving more GPU memory for both graph topological data and cached features. In this way, graph sampling and model training can be accelerated at the same time. However, this factored design may suffer from imbalanced loads between GPUs for graph sampling and model training. To solve this problem, GNNLab divides the GNN training pipeline into two kinds of executors, namely Sampler and Trainer, and bridges two kinds of executors asynchronously. A simple yet effective method is proposed to adaptively determine the appropriate GPU numbers for Samplers and Trainers. GNNLab further leverages dynamic switching from Samplers to Trainers to avoid idle waiting on GPUs if needed. On the other hand, to enhance the efficiency of GPU-based caching policies for various sampling algorithms and GNN datasets, we propose a universal caching strategy. This strategy is defined by two parameters: a hotness metric that estimates the frequency of vertex sampling during the graph sampling phase, and a cache ratio that determines the number of vertices that can be cached on GPUs. Existing caching policies can be naturally incorporated into this scheme. However, the hotness metrics used in previous studies, such as vertex out-degree, do not effectively capture the diversity of sampling algorithms and GNN datasets. To tackle this issue, we introduce a pre-sampling-based caching policy (PreSC), which is motivated by the observation that the most frequently sampled vertices exhibit significant overlap across different epochs. PreSC performs several sampling phases and uses the average visit count as the vertex hotness metric. We have found that PreSC achieves a high cache hit rate even with a small cache ratio and is resilient to diverse sampling algorithms and GNN datasets.

The second introduced work is a dual cache method named DUCATI [11]. It designs two caches to adapt to different workloads. This paper proposes that these single cache systems follow two assumptions: (1) node features selection(Nfeat-Selecting) consumes more time than adjacency matrix selection(Adj-Sampling). (2) Nfeat-cache can make full use of the spare GPU memory. However, these assumptions may not always hold in practice. Single-Cache systems cannot adapt to such diverse workloads since they unquestioningly optimize Nfeat-Selecting even when Adj-Sampling is more expensive. In addition, the Single-Cache systems may not fully utilize the GPU memory. They can waste up to several gigabytes of the GPU memory. Based on these findings and analysis, in addition to traditional Nfeat-cache, DUCATI further exploits the locality of the adjacency matrix by introducing a new Adj-Cache to accelerate Adj-Sampling. DUCATI devises a new cache structure for Adj-Cache, which fits the Adj-Sampling workflow and the adjacency matrix's characteristics. Besides that, DUCATI develops a workload-aware Dual-Cache Allocator, which adaptively finds the best allocation plan for Adj-Cache and Nfeat-Cache to minimize the total execution time.

The third work is a UVA-based method [12]. It demonstrates that traditional data access patterns make the traditional block data transfer method ineffective. Because each node can be connected to thousands of nodes in real-world graphs, we may need to access thousands of scattered locations in memory to collect relational information from those neighboring nodes. Therefore, a processor-oriented, software-defined data communication architecture is proposed to rely on DMA engines; GPU cores are programmed to access host memory directly using zero-copy memory access. This approach empowers application developers to direct GPU cores to the precise locations where the required computation data is stored. To optimize PCIe packet efficiency with zero-copy memory access, an automatic data access alignment optimization is introduced in the GPU data indexing kernel. This optimization enables zero-copy PCIe bandwidth to reach up to 93% of the block transfer PCIe bandwidth. A novel CUDA multi-process service (MPS) based resource provisioning optimization is proposed to minimize GPU resource consumption for zero-copy memory accesses. By examining the PCIe protocol and GPU architecture, it is determined that PCIe can be saturated even with a few GPU cores generating zero-copy accesses. Therefore, this optimization allocates only a tiny portion of GPU resources for zero-copy accesses, leaving the rest for computationally intensive workloads. Finally, an end-to-end zero-copy GCN training flow is implemented in PyTorch. To enable zero-copy memory access, a new class of tensors called "unified tensor" is introduced. This tensor provides an address mapping of host memory for GPUs, allowing them to access host memory directly with zero-copy accesses. The proposed GCN training flow supports zero-copy access in multi-GPU training environments by declaring multiple unified tensor instances for various GPUs.

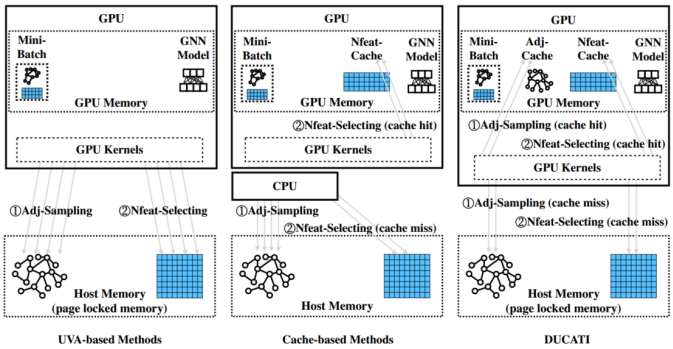

Fig. 1. Comparison of UVA-based method, single cache method, and dual cache method(DUCATI) on performing Adj-Sampling and Nfeat-Selecting on giant graphs.

This report introduces three methods that focus on the optimization of mini-batch training. Figure 1 shows the differences between these methods.

In summary, the main content of this paper is as follows:

- 3 methods are introduced to optimize the mini-batch training.
- The experimental results of these works are shown. It is found that the DUCATI performs better than the other introduced methods.

II. BACKGROUND

In this section, I briefly review the background of GNNs, the mini-batch training and two kinds of optimization methods.

A. Graph Neural Networks

Graph neural networks (GNNs) are a family of machine learning algorithms that apply neural network (NN) operations on graph-structured data. Initially, each vertex in a graph is associated with a high-dimensional feature vector. The goal of a GNN model is to learn a low-dimensional feature representation for each vertex, which can be further fed into various downstream tasks, e.g., node classification, link prediction and node clustering.

Similar to traditional neural networks, a GNN model stacks multiple GNN layers to update the vertex features iteratively, in which each successive layer uses the outputs of its previous layer as inputs. Here both the inputs and outputs of a layer contain the features of all vertices. In each GNN layer, the new feature of each vertex is computed by aggregating the features of its "neighbors" from the previous layer. Given a graph G with raw input features X_* for the vertices in G (e.g., word embeddings in the Reddit dataset), the computation within the k-th GNN layer can be expressed as follows:

$$a_v^{(k)} = AGGREGATE^{(k)}(h_u^{(k-1)} | u \in N(v)), \quad (1)$$

$$h_v^{(k)} = UPDATE^{(k)}(h_v^{(k-1)}, a_v^{(k)}), \quad (2)$$

where $AGGREGATE^{(k)}$ and $UPDATE^{(k)}$ are two operations conducted during the k-th GNN layer for k ¿ 1, $h_v^{(k)}$ and $a_v^{(k)}$ denote the feature vector and neighborhood representation of vertex v at the k-th layer, respectively, and $N(v)$ denotes the "neighbors" of v.

The operation $AGGREGATE^{(k)}$ is also referred to as neighborhood aggregation at layer k. Intuitively, it gathers the features of v's "neighbors" using accumulation functions to produce neighborhood representation $a_v^{(k)}$. This is followed by the invocation of $UPDATE^{(k)}$, which computes the new feature $h_v^{(k)}$ by combining v's previous feature $h_v^{(k-1)}$ and the newly computed $a_v^{(k)}$. Note that $AGGREGATE^{(k)}$ involves both graph propagation, i.e., features are propagated from the "neighbors" to v, and NN operations, while $UPDATE^{(k)}$ only includes NN perations.

B. Mini-batch Training of Graph Neural Networks

Graph Neural Networks can effectively model the graph data through iterative aggregation and transformation of node features on the graph topology. However, early GNNs such as GCN [4] are proposed with a full-batch training scheme, which means that we need to maintain the adjacency matrix, node features, and hidden representations in the memory of GPU(s). The core idea of GCN is to create node embeddings by iteratively aggregating neighboring nodes' attributes using neural networks. Due to its neighboring node's attribute lookup, training GCN requires accessing multiple scattered locations in memory. In Figure 2 (a), we show a simple example of GCN training. To generate the embedding of node 4, we traverse the input graph and aggregate node 4's features

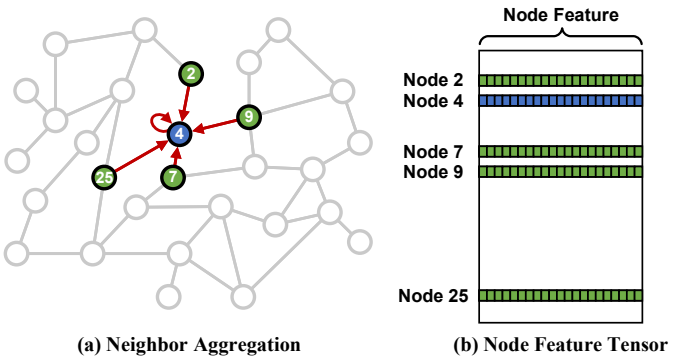

Fig. 2. (a) A simple example of GCN training on single node. (b) An illustration of node features in memory. The neighboring nodes' features are scattered in memory.

alongside the features of all neighboring nodes in the node feature tensor. The example that we show here is only a toy example.

It can be concluded that all graph topological data and features are kept in the host memory. For each mini-batch, graph sampling and feature extracting are performed on CPUs or GPUs sequentially; then the sampled vertices and their features are transferred to GPU memory for model training. Further, GNNs use data parallelism by default to enable multiple GPUs, as they commonly employ simple models with only 2 or 3 layers. Each GPU trains mini-batches independently and exchanges gradients among GPUs to update model parameters synchronously or asynchronously.

C. UVA-based Methods

To accelerate the preparation of mini-batches, some works [12] proposed the unified virtual addressing (UVA) technique for fast accessing and processing of irregular graph data. Specifically, [12] applied the UVA technique to Nfeat-Selecting.

Conventionally, the Direct Memory Access (DMA) engine is employed for data transfers from the host memory to the GPU. This engine is fine-tuned for moving large, contiguous blocks of data. Initiating a DMA transfer is costly, and the expense can only be justified when moving substantial amounts of data. Unfortunately, the indices of nodes in a sampled subgraph are typically random, leading to the data of a mini-batch being dispersed across many fragments. It is impractical to initiate a DMA call for each fragment due to the high cost. Consequently, the scattered data must first be aggregated by the CPU before transfer. This aggregation is both time-consuming and resource-intensive, as previous studies have shown. In contrast, the UVA technique is more suitable for transferring scattered data fragments from the host memory to the GPU. Specifically, the UVA technique moves the original adjacency matrix or node features to the page-locked host memory. Then, the UVA technique registers the page-locked memory with internal CUDA utilities, allowing the GPU to directly access the data stored in the page-locked memory. The GPU can then issue numerous CUDA threads to access the scattered data fragments in the page-locked memory

in parallel, which is much faster than the conventional DMA method. With the UVA technique, we can better utilize the PCIe bandwidth and save the time required to transfer the irregular data of the mini-batch from the host memory to the GPU. Additionally, since the GPU can directly access the large graph data stored in the host memory with the UVA technique, we can launch CUDA kernels on the GPU to perform fast Adj-Sampling and Nfeat-Selecting, bypassing the slow CPU execution. However, the UVA technique fails to leverage the locality of graph data.

D. Cache-based Methods

Cache-based approaches accelerate the preparation of mini-batches by exploiting the spatial proximity of node attributes and the underutilized GPU memory. These techniques suggest storing frequently accessed node attributes in the unused GPU memory to diminish the duration of Nfeat-Selection. Specifically, following the node IDs obtained from Adj-Sampling, Nfeat-Selection involves choosing the pertinent node attributes for the selected nodes. Cache-based methods detect a significant spatial proximity of the accessed node attributes across various mini-batches. Therefore, these methods propose retaining frequently accessed node attributes in the GPU to prevent redundant data transfer. These cache-based methods necessitate some preprocessing before the training commences. Initially, these methods identify the hot node attributes using diverse metrics. Then, these methods ascertain the available cache budget through several offline runs. Ultimately, these methods construct a Nfeat-Cache on the GPU encompassing the hottest node attributes given the cache budget. A corresponding lookup table is also constructed to indicate whether a specific node attribute is stored in the CPU or the GPU and the address to locate the node attribute. Once the training starts, these methods will also check the location of each selected node attribute during Nfeat-Selection. If Nfeat-Cache hits for a given node attribute, these methods directly fetch it on the GPU and omit the transfer from the CPU. If Nfeat-Cache misses, these methods fetch the corresponding node attributes from the CPU as before. Due to the spatial proximity of graph data, a significant portion of data transfer from the CPU to the GPU is saved. Hence, these methods can expedite Nfeat-Selection of the mini-batch preparation. Regarding Adj-Sampling, while some previous works proposed to accelerate Adj-Sampling with GPUs, their applicability is limited as they require maintaining the entire adjacency matrix in the GPU memory.

III. Related Work

A. Whole-graph GNN training.

To retain scalability, wholegraph GNN training divides a large graph into multiple partitions, and trains GNN models simultaneously on all vertices/edges with multiple machines/GPUs. Typical GNN systems that fall into this category include NeuGraph [15], ROC [16], FlexGraph [17] and Dorylus [18]. In whole-graph training, each vertex needs to consider its all neighbors while different vertices may have different neighbor sizes. Thus, it is hard to use dense tensor operations to express neighborhood feature aggregation since dense tensor operations require a regular input form. To address this problem, existing systems proposed different techniques, e.g., kernel fusion in DGL [19], sparse tensor operations in PyG [20] and hybrid aggregation in FlexGraph [17], to efficiently perform neighborhood feature aggregation operations.

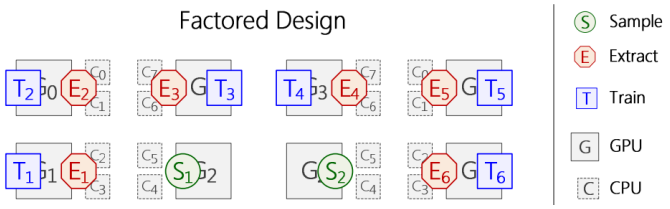

Fig. 3. An example of the factored design for sample-based GNN training over 8 GPUs and two 8-core CPUs.

B. GPU-based graph sampling

Graph sampling also takes a substantial portion of the end-to-end GNN training time [21]. Thus, leveraging GPUs to accelerate graph sampling has appeared in both academic and open-sourced projects, like NextDoor [21] and DGL [22]. The graph topological data is first loaded into GPU memory and then sampled on the GPU for each mini-batch. Next, the samples are returned to CPUs for extracting the features of sampled vertices to GPU memory. Finally, the GPU trains a GNN model with the sampled vertices and their features. Generally, the graph topological data is preloaded and kept in the GPU memory if possible [22].

C. GPU acceleration in GNN training

GPUs have been adopted to improve both whole-graph GNN training and sample-based training. For whole-graph training, GPUs have mostly been applied to accelerate the NN and graphrelated operations. For sample-based training, GPUs have been adopted to improve sampling, extracting, and training process. For the sampling, GPUs are applied to accelerate graph sampling due to their much higher parallelism and memory access bandwidth than CPUs. Typical work includes NextDoor [21], C-SAW [23] and DGL [22]. While PaGraph [13] adopts a degree-based caching policy to accelerate feature extraction, DGL [22] uses GPUs in the extracting process only if all feature data can be loaded in GPUs. For the training, PyG [20] and DGL [22] implement highly optimized GNN runtimes on GPUs.

D. Other Graph-Related Workload Acceleration on GPU.

There are several works which try to utilize GPUs in graph traversal workloads like PageRank [24] with large datasets. Due to the usage of sparse matrix format for the representation of graph structure, traversing graphs results in generating very irregular memory accesses. Considering that large graphs such as WDC14 [25] has about 64 billions edges, the graphs cannot be placed in GPU memory and therefore the graph traversal workloads face the similar issue in this paper. EMOGI [26]

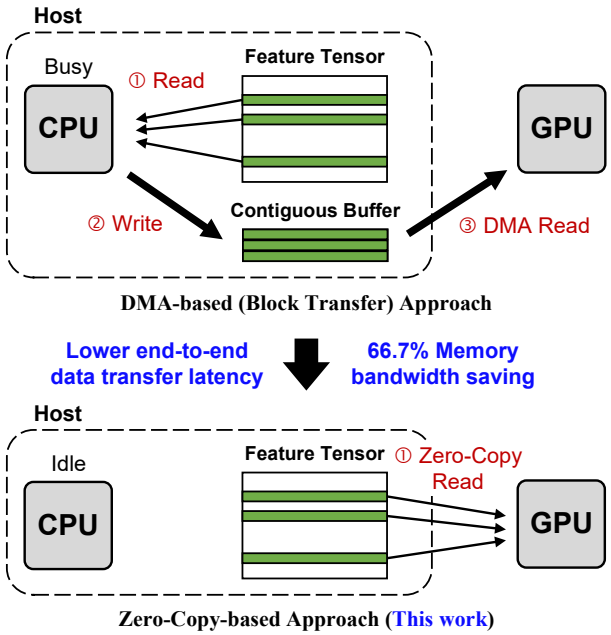

Fig. 4. Workload comparison between DMA-based method and the proposed zero-copy-based method.

utilizes zero-copy accesses to enable fine-grained host memory access during several graph traversal workloads. Halo [27] tries to ensure the spatial locality of graph nodes in the memory as well through extensive pre-processing. However, the effectiveness of this method is completely random depending on the shape of the input graph. Subway [28] uses a method very similar to the DMA-based method used in this work, which tries to utilize CPU as much as possible to gather scattered data for more efficient DMA.

IV. UVA-Based Method

I take the work [12] as an example and introduce its design and experimental results.

V. UVA-Based method

Due to the wide spread use of DMA-based data communication architecture, there are some number of system-level modifications that must to be established to support our GPU-oriented data communication architecture in the higher-level programming models. It first describes how to enable zero-copy accesses in PyTorch and then it discusses some of the technical aspects of zero-copy access to identify its weaknesses and how to overcome them. Finally, this method describes the end-to-end GCN training flow using zero-copy accesses.

A. Motivations

Conventional wisdom may still argue that since the node feature data is in host memory, CPU has significant bandwidth advantage over GPUs and therefore DMA should be a better option because CPU can quickly gather the sparse features on the fly. However, recent work has shown that the ability to issue a massive number of concurrent memory accesses enables GPUs to tolerate latency effectively when accessing complicated data structures like graphs that reside in host memory [26]. Therefore, in GCN training, if GPUs can make targeted fine-grain host memory accesses for sparse features while fully utilizing system interconnect (e.g., PCIe) bandwidth, the proposed approach can offer significant advantage over the DMA approach. The removal of CPU gathering stage not only shortens data access latency for GPUs, but also greatly reduces the CPU and host memory utilization (Figure 4). Offloading CPU workloads to GPUs also helps on training GCN with multiple GPUs as we can prevent the CPU becoming the bottleneck with increasing number of workers.

In order to propose the GPU-oriented data communication architecture for GCN training, three major questions are addressed in this work. First, can zero-copy memory access fully utilize PCIe bandwidth while training GCN considering the long latency for accessing host memory? Second, what would be the price of consuming GPU cores for zero-copy memory access? Finally, after resolving the above two questions, can we show real end-to-end application performance benefit from our method?

In this work, it answers all three questions.

B. Designs

In this section, I will show how this UVA-based method answers the three questions.

1) Improving Zero-Copy Efficiency Over PCIe.: For GCN training, we use PyTorch which is one of the most popular python-based ML frameworks. However, including PyTorch, there are no python-based ML libraries which naturally support zero-copy access for GPUs. To overcome such issue, it creates an extension of the existing PyTorch implementation with several modifications in its source code. In our design, we aim to aggressively avoid the implicit DMA data copy performed by PyTorch. It gives GPUs direct access to tensor data in the host memory by mapping the host-memory data pointers into the GPU address space. To achieve ourthe goal, it creates a new class of tensor with a new ”unified” context. A tensor with this new context can be declared from any existing CPU tensors. One of the common misconceptions of zero-copy access is its low data transfer efficiency compared to the DMA-based methods. The misconception is mainly coming from the fact that the users are treating the zero-copy without any specific care. However, as the zero-copy access requests are made over PCIe, it is important to understand how the zero-copy accesses interact with PCIe. In this section, it takes a deep-dive into the technical aspect of PCIe protocol and its interaction with GPUs. This work then presents two important techniques for maximizing the zero-copy efficiency during GCN training.

Aligned Memory Access Even though our purpose of using zero-copy is to make fine-grained memory accesses to the host memory, it is still desirable to make coarser-grained PCIe memory requests whenever possible for a couple of reasons. First, each PCIe packet has 12–16 bytes of header overhead. Therefore, to compensate the overhead, it is better

to increase the payload size by requesting a larger memory request. Second, PCIe devices have a hard limit on the number of outstanding requests they can create. Since the PCIe round trip time (RTT) is very long (1–5us, variable), it is necessary to submit multiple read requests in a pipelined fashion to fully occupy the interconnect. However, if we squander the capacity by generating too many small read requests, it becomes difficult to fully tolerate the latency and utilize the PCIe bandwidth. The numbers of maximum outstanding read requests for PCIe 3.0 and PCIe 4.0 are 256 and 768, respectively.

Now, with all that in mind, how do we generate coarser-grained PCIe requests? According to Min et al. [26], to make PCIe read requests more efficient, the same technique used for the GPU memory coalescing [29] can be used. In Figure 5, it explains two cases where (a) memory accesses from a warp are contiguous and aligned with the GPU cacheline, and (b) memory accesses from a warp are contiguous but misaligned with the GPU cacheline. In case of (a), the accesses from the threads in a warp are perfectly coalesced and the coalesced requests becomes a single 128B PCIe read request. In case of (b), the accesses from a warp are scattered over two GPU cachelines and they result in generating two separate PCIe read requests. The possible memory access granularities are 32B, 64B, 96B, and 128B, while 32B is a single sector size of GPU cacheline. Each GPU cacheline is composed of four sectors.

Of course, we would not need to worry about the misaligned accesses if the node feature objects always start at 128B boundaries and the sizes of node features are always multiples of 128B, but it is very unlikely to be so in reality. For example, if a certain dataset's node feature size is 480B, accessing the second node feature will start from accessing 480th byte in memory address. In this case, we are off by 32B from the closest 128B boundary (512B). To automatically resolve such issue, it adds a circular shift stage in the PyTorch indexing CUDA kernel. The shifting stage is aware of the GPU cacheline size and shifts the memory access indices by calculating the offset between the nearest 128B aligned location and the current indexing location. The visualization of our circular shift mechanism is shown in Figure 6. In this example, we want to access the second node feature with zero-copy access where each node feature size is 480B. Without the optimization, each warp start reading from misaligned locations and end up generating 8 PCIe requests. However, once its optimization is applied, the warps adjust their indexing locations and try to generate aligned memory accesses as much as possible. In this example, the total number of PCIe read requests is reduced to 5.

It does not apply the circular shift stage if the node feature size is less than the GPU cacheline size or if it is already a multiple of the GPU cacheline size. All these adjustments are transparent to the high-level programmers as a result of its modifications to PyTorch source code.

C. Asynchronous Operations and Resource Provisioning.

One important distinction of our design is that zero-copy accesses are done by GPU kernels. In other words, the other following GPU kernels need to wait until the zero-copy kernel is finished even if all it's doing is simply reading the host memory. However, like in many other ML algorithms, GCN can also greatly benefit from overlapping data communication time and training time, which naturally happens in DMA-based methods. To achieve the best training performance, we must devise a way to overlap the training GPU kernels and the zero-copy GPU kernels in our design.

TABLE I
NVIDIA RTX 3090 SPECIFICATIONS.

Category	Specification
PCIe Generation	4.0
Max # of Outstanding PCIe 4.0 Read Requests	768
# of Multiprocessors	82
# of Threads per Multiprocessor	1536
# of Threads per Warp	32

There is a hard limit on the number of outstanding PCIe read requests that PCIe devices can generate at a given moment. Therefore, if we can prove that we only need small amount of GPU resources to fully saturate the limit, it is worthwhile to seek for a way to achieve the concurrency. In Table I, it lists the specifications of NVIDIA RTX 3090 GPU which we use for our evaluations. At any given moment, the GPU cannot generate more than 768 outstanding PCIe read requests. To identify the portion of GPU resource it needs to generate 768 outstanding PCIe read requests, this work performs the following calculation. First, lets assume each warp's memory requests are coalesced to a single PCIe read request, and lets ignore the payload size for now. In this case, we need 768 warps available to the scheduler to reach the PCIe 4.0 limit. Since each streaming multiprocessor (physical processor) can hold up to 1,536 threads at a given moment, each multiprocessor can sustain up to 1,536 / 32 = 48 outstanding PCIe read requests. Now, we have 82 multiprocessors in RTX 3090, so the amount of GPU resource that we need to reserve for the zero-copy GPU kernel is about 16 / 82 = 19.5%. However, this is the upper bound for the extreme case. If we assume we can always generate 128B PCIe read requests, we can saturate the PCIe 4.0 bandwidth with much fewer outstanding requests. For example, the measured maximum PCIe bandwidth with `cudaMemcpy()` in RTX 3090 is 25.8GB/s and if we assume RTT (Round-Trip-Time) of PCIe is 1.5us [30], the number of outstanding requests that we need to sustain is (25.8GB/s) / (128B) $\times$ 1.5us = 324.6. That is, assuming all PCIe requests are 128B in size, we need to reserve only 8.2% of the total GPU resource for the zero-copy GPU kernel. In reality, since some of the requests will be smaller, this number is a lower bound and the actual number will be somewhat higher. In short, even if we try to maximize the zero-copy GPU kernel efficiency, there is at least 80% and up to 91.8% of the GPU resources available for other workloads.

Now, finally, since we realized how much of GPU resource should be allocated for the zero-copy kernel, we explore the method to enforce the limitation in practice. Fortunately, NVIDIA GPUs already provide support for limited execution

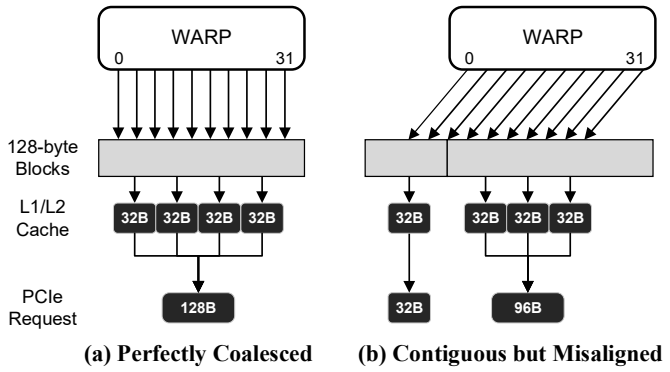

Fig. 5. (a) A perfectly coalesced 128-byte access from a warp. (b) A warp accessing a misaligned data needs to generate multiple PCIe requests.

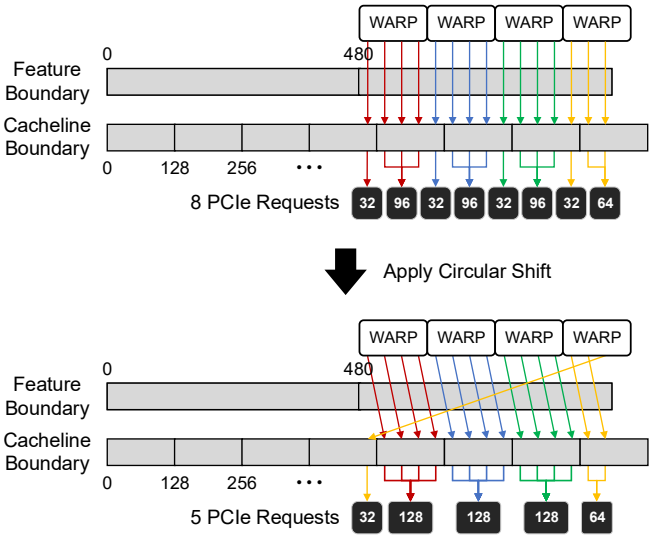

Fig. 6. Circular shift optimization explained. Circular shift transforms memory requests into a GPU cacheline-friendly way.

resource provisioning through CUDA multiprocessing service (MPS) [31]. MPS is originally designed to improve quality of service (QoS) between different clients' workloads, but we utilize this service to control the resource utilization of the zero-copy GPU kernel. To assign different resource limitations to different kernels, the kernels must be running in different processes. Since PyTorch already supports multiprocessing programming model, it is simple to launch the zero-copy GPU kernel and the training GPU kernel in two separate processes. Before we launch the zero-copy GPU kernel, we modify the GPU resource limitation to $X\%$ with the `nvidia-cuda-mps-control` utility. Next, before it launches the training GPU kernel, it also modifies the resource limitation to $(100 - X)\%$. In its PyTorch code, the whole process is scripted for an easier use. It would be more elegant if the resource limitation can be configured in the user CUDA code instead of the MPS utility, but currently CUDA does not support such functionality. Another side benefit of the multiprocessing approach is that the different GPU kernels running in different processes are not affected by the other processes' blocking CUDA API calls. With its approach, zero-copy accesses can saturate the PCIe bandwidth while leaving majority of GPU resources opened for other computationally intensive workloads.

With this optimization, it can basically transform the GPU cores into an intelligent DMA engine which can asynchronously perform complex data accesses such as data dependant index calculations and fine-grained host memory accesses. This optimization can be also useful for some of the workloads which utilize peer-to-peer GPU memory accesses with zero-copy accesses.

1) Workload Scheduling: In Figure 7 (a), it shows the initial tensor allocations during the initialization step. First, it maps the whole node feature tensor into the GPU address space by using the unified tensor. This unified tensor holds a memory pointer which GPU can use in its kernel to generate zero-copy accesses to the node feature tensor. Next, it creates two sets of ping pong buffers for interprocess communications. The goal of using ping pong buffers is to remove the usage of locking mechanisms between two different processes sharing data and to allow them to start working for the next minibatch immediately after finishing their current works. In its design, each process needs to be synchronized just once per minibatch.

After the initialization, the training pipeline begins from the sampler process randomly selecting nodes and collecting their neighbors' node indices (Figure 7 (b)). Once all the node indices are identified, the combined list is transferred to the producer process running on GPU for the zero-copy accesses. The list of node indices is transferred over DMA as it is contiguous and small. Once the node features are all gathered into one of the ping pong buffers, the producer notifies the consumer to train on the new minibatch data as soon as it is ready. Since the GPU ping pong buffers are located in the same GPU memory, it naturally makes sense for the consumer to directly access the buffer owned by the producer instead of copying it to its own space. To achieve this, yhis UVA-based method utilizes CUDA interprocess communication (IPC) APIs. With the CUDA IPC APIs, two different GPU kernels running on different processes can share the same GPU memory space without data copies in between. This specific GPU pointer sharing procedure is implemented in the PyTorch `Queue` class and we utilize it for our application. The ping pong buffers are statically located for the entire training process and therefore the pointer sharing needs to be done only once at the beginning of the producer process.

From the user's point of view, the training process is pipelined in a sampler $\rightarrow$ producer $\rightarrow$ consumer order (Figure 7 (c)). Except the unified tensor declaration, the rest of its end-to-end GCN training implementations is developed with the existing PyTorch functionalities, and this makes the method more accessible for the existing users. Its modifications are isolated into the data transfer portion of the GCN training and the algorithm remains unaffected.

To conclude, this methods show the contributions as follow:

- As opposed to the traditional DMA-based data communication architecture, it proposes GPU-oriented, software-defined data communication architecture with zero-copy memory accesses for efficient sparse accesses to graph

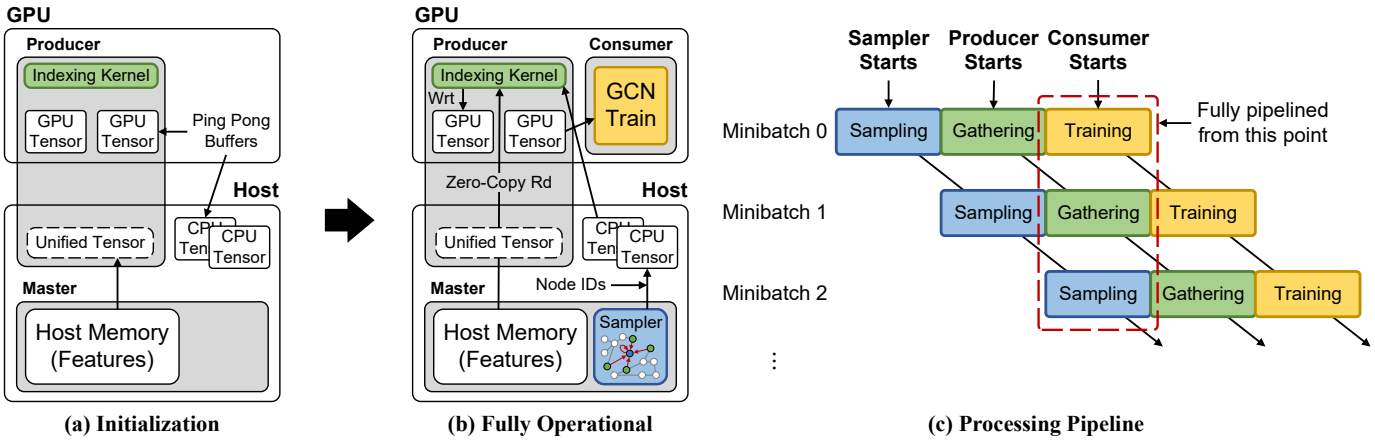

Fig. 7. GCN training flow with zero-copy accesses. Only the operations related to data accesses are shown. (a) This method setups unified tensor and the returned pointer is passed to GPU for zero-copy accesses. (b) The sampler generates node IDs used by the producer and the producer gathers scattered node features in the host memory. The consumer uses the gathered node features for training. (c) A visualization of processing pipeline.

node features in GCN training.

- To improve the efficiency of zero-copy memory access, it proposes automatic data alignment and a novel CUDA MPS based resource provisioning optimizations.
- It seamlessly integrate their modifications with the existing PyTorch framework for easier programming and show 65-92% of end-to-end training performance gain.

D. Experiments

This section evaluates the impact of its proposed design on GCN training. This work first takes a closer look at the improvements made by its optimizations one by one and then shows the overall training time reduction achieved.

1) Methodology: **Evaluation System** For evaluation, it uses the system described in Table II. The host system can hold two RTX 3090 GPUs, and both are operating in PCIe 4.0 mode. With PCIe 4.0 interconnects, both GPUs can achieve about 25.8GB/s of host to GPU DMA bandwidth in our microbenchmark. The measured aggregated bandwidth of the two GPUs performing DMA on host memory at the same time is about 51.7GB/s.

Application The unified tensor implementation and the indexing kernel modification are based on PyTorch 1.8.0-nightly version. For the GCN training, we use the GraphSAGE [5] implementation of DGL. It only modifies the data communication portion of the implementation. The sampling mechanism and the training algorithm remain unmodified.

(a) CPU-Only implementation only uses CPU for training GCN. In this case; there is no need for data transfer over PCIe since GPUs are not involved in the training.

(b) DMA-based implementation uses CPU to gather node features into a contiguous buffer. The gathered node features are transferred to GPUs by using DMA.

(c) NaÏ've Zero-Copy uses zero-copy as a main data transfer method but does not include any optimizations discussed in this paper. Unified tensors are used to allow GPUs to perform zero-copy accesses on host memory.

(c) Zero-Copy implementation enables zero-copy accesses and includes all optimizations we discussed in this paper.

TABLE II
EVALUATION SYSTEM CONFIGURATION.

Category	Specification
CPU	AMD Ryzen Threadripper 3960x 24C/48T
Memory	DDR4 3200 MHz 256GB in Quad Channel
GPU	2x NVIDIA Ampere RTX 3090 24GB
OS	Ubuntu 20.04.1 & Linux Kernel 5.8.0
S/W	CUDA 11.2 & PyTorch 1.8.0-nightly

(d) All-in-GPU implementation allocates the entire node feature array into each GPU memory before the training begins. This implementation is used to show the rough upper bound of the performance improvement we can achieve through the data transfer optimization. Due to the limited GPU memory capacity, we do not evaluate all datasets with this implementation. It explicitly denotes as ”out-of-memory (OOM)” for such cases.

Dataset In Table III, we show the datasets we used for the evaluation. `wikipedia` [**?**] network consists of the wikilinks of Wikipedia in English. Nodes are Wikipedia articles, and directed edges are Wikilinks. `amazon` [**?**] dataset is based on the Amazon product network connected by ”also viewed” and ”also bought” links. `ogbn-papers100M` dataset is a directed citation graph of 111 million papers indexed by MAG [**?**]. The above datasets are used for basic performance evaluations. `ogbn-products` [**?**] dataset is based on Amazon co-purchasing network [**?**] where nodes represent products sold in Amazon, and edges between two products indicate that the products are purchased together. `ogbn-products` is only used for the training time vs. node feature size sensitivity analysis.

E. Bandwidth Analysis

In Figure 8, it shows the comparison of the effective bandwidths we measured during the wikipedia dataset training. To observe the impact of the misaligned node feature access

TABLE III
EVALUATION DATASET.

Name	#Feature	#Node	#Edge	Size
ogbn-products	128 - 4096	2.4M	61.9M	-
wikipedia	315	13.6M	437.2M	17.1GB
amazon	578	14.7M	64.0M	34.0GB
ogbn-papers100M	128	111.1M	1.6B	56.9GB

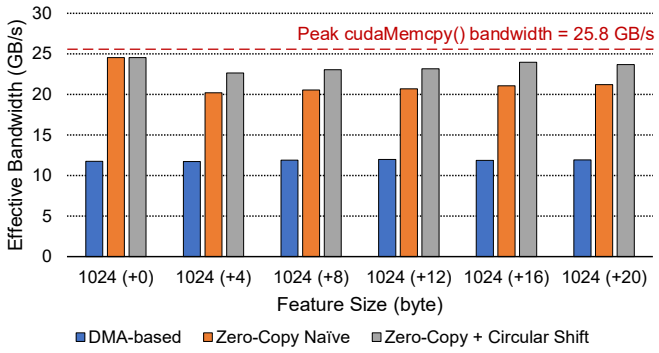

Fig. 8. Effective data transfer bandwidth measured during the wikipedia dataset training. We sweep feature size to observe the impact of misaligned zero-copy accesses over PCIe.

on the PCIe bandwidth, it sweeps the node feature size from 1024B to 1044B in this experiment. Zero-copy naïve approach does not implement the circular shift optimization. Throughout the experiment, the effective bandwidth of the DMA-based approach is only about half of the zero-copy approaches as it requires a long CPU gathering process.

When the node feature size is 1024B, regardless of the circular shift optimization existence, the zero-copy implementations show the best effective bandwidth numbers. Because the GPU cacheline size is 128B, in this case accessing any node features results in generating perfectly coalesced accesses. Considering that the best `cudaMemcpy()` bandwidth it achieved is about 25.8GB/s, it can roughly estimate the upper bound efficiency of zero-copy access is about 95.1%. With more misaligned accesses, the efficiency of the naïve zero-copy implementation drops to 78-82% while the optimized zero-copy implementation can achieve 88-93% of efficiency.

In general, the results re-emphasize the importance of making cacheline-aligned accesses whenever using zero-copy accesses. For savvy programmers, it expects them to understand the underlying hardware mechanism and to consider padding the input data if the overhead is not too big. However, even if they fail to do so, our optimizations would still reduce the performance penalty for them.

1) Overall Comparison: In Figure 9, we show the overall training performance comparison. Throughout the entire comparison, the CPU-only case shows the worst performance. By limiting the computation unit to CPU, there is no need to worry about efficient data transfers over PCIe but at the same time the computation power is severely limited. In general, the CPU-only method is 2.3–3.1× slower than the DMA-based method. This performance difference is an important motivation for moving the training into GPUs.

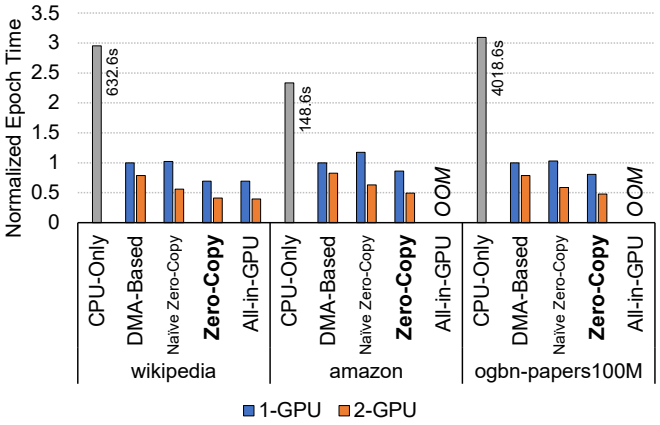

Fig. 9. GCN training time comparison. OOM denotes out-of-memory.

For the DMA-based method, doubling the number of GPUs does help reduce the overall training time. Still, additional GPU results in only 21-27% performance improvement across different datasets. Because the DMA-based method makes poor use of CPU resources and host memory bandwidth, increasing the number of workers quickly makes the entire training process throttled by them at this point, all GPUs are experiencing data starvation and have low utilizations.

For the naÏ've zero-copy method, we observe 2–17% of performance degradation compared to the DMA-based method in a single-GPU setup.This result also gives us an idea how the programmers can make a premature conclusion not further to investigate the usage of zero-copy accesses.

With two GPUs, the naïve zero-copy method shows much better performance as well as performance scalability than the DMA-based method. In a dual-GPU setup, the naïve zero-copy method becomes 30–41% faster than the DMA-based method. This is because, even without the optimizations, the zero-copy method by default much more efficiently uses the CPU resource and host memory bandwidth than the DMA-based method. However, this benefit is not visible until the number of GPUs increases.

Finally, with our zero-copy optimizations, we can now clearly see the benefit of zero-copy in all cases in a single-GPU setup; the optimized zero-copy method is 16–44% faster than the DMA-based method, and in a dual-GPU setup, it is 65–92% faster. More surprisingly, with all optimizations included, the performance of the zero-copy method matches with the all-in-GPU method for the `wikipedia` dataset training. Since the training process completely hides the data communication time in this case, there is no disadvantage compared to the all-in-GPU method. Overall, it observes a very significant benefit of using zero-copy accesses for GCN training.

VI. GNNLAB

GNNLab aims to design a SOTA system to accelerate the GNN mini-batch training process. It proposes 2 challeneges:

- The first challenge is how to eliminate contention on GPU memory between different stages of the SET model (Figure 10).

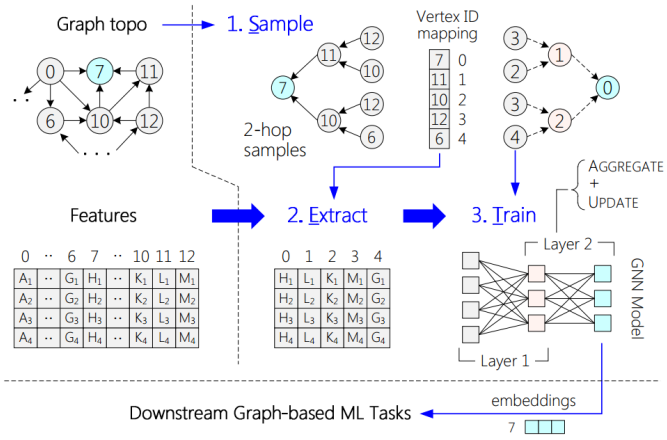

Fig. 10. An example of the SET model for sample-based training in a 2-layer GNN on V_7.

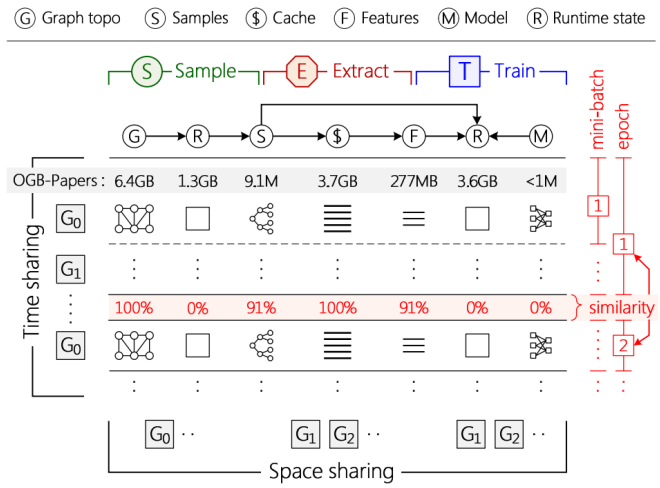

Fig. 11. A breakdown of memory usage and data similarity for different stages of the SET model when training OGB-Papers over multiple GPUs (G_0, G_1, ...) with 16GB of memory each.

- The second challenge is how to achieve optimal cache efficiency for diverse GNN datasets and sampling algorithms.

A. Motivations

GNNLab is motivated by an attractive observation that different training epochs in the same stage share a large amount or even all of the data, which means that sample-based GNN training has extremely good inter-task data locality. As shown in Figure 11, graph topology and feature cache, occupying more than 64% of the total 16GB GPU memory, are fully shared by the Sample and Extract stages in different epochs, respectively. This means that leveraging space sharing in the stage level, as shown vertically in Figure 11, can significantly reduce the cost of data transfer, which is the major obstacle to optimizing sample-based GNN training over GPUs.

B. Design of GNNLab

1) A Factored Design: GNNLab is based on the facored operating systems. Inspired by the factored operating system

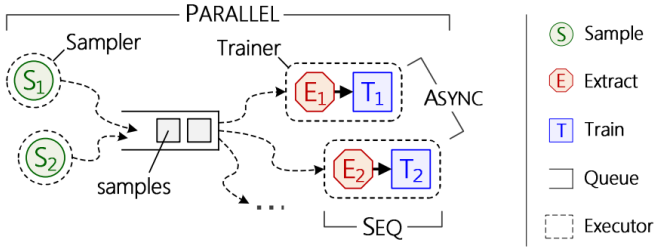

Fig. 12. The execution flow of GNNLab. This shows a simple working flow of GNNLab with sampler and trainer.

(fos), the key idea behind GNNLab is to perform each stage of the SET model (e.g., Sample) on dedicated processors (GPUs and/or CPUs) for different minibatches. GNNLab is a new factored system for sample-based GNN training over GPUs, in which space sharing replaces time sharing to improve performance significantly. Figure 3 illustrates a brief example of GNNLab that conducts two samplers, six extractors and six trainers on a machine with 8 GPUs and two 8-core CPUs.

2) GNNLab Architecture: To adapt to diverse workloads, GNNLab flexibly assigns GPUs to different stages and runs them in parallel. We observe that the Sample and Extract stages only share a small amount of data (i.e., samples). Thus GNNLab divides the training pipeline of the SET model into two kinds of individual executors, named Sampler and Trainer respectively, as shown in Figure 12. GNNLab uses a global queue in the host memory to link two kinds of executors asynchronously, which is flexible in supporting different numbers of executors. The concurrent queue would not be the bottleneck since the updates are infrequent. Figure 14 outlines the implementation of executors in GNNLab. GNNLab binds each Sampler to a GPU, and it will load graph topological data into GPU memory. The Sampler iteratively generates the samples for each mini-batch following a certain graph sampling scheme (e.g., k-hop random neighborhood sampling). To accelerate the pace of feature extraction, the sampled vertices will be deduplicated and reassigned with consecutive IDs (starting from 0). Finally, the samples will be sent to the Trainer asynchronously via a global queue. For multiple Samplers, a global scheduler assigns tasks (i.e., mini-batches) dynamically across them in order to achieve load balance without synchronization. For larger graphs that cannot fit in GPU memory, a simple approach is to divide the whole graph into multiple partitions and iteratively load a partition to the GPU memory for graph sampling.

On the other hand, GNNLab binds each Trainer to a GPU and several CPU cores. The Trainer sequentially executes the Extract and Train stages for each mini-batch. After receiving samples of a mini-batch, the Trainer will simultaneously extract their features from host memory and the feature cache in GPU memory (if any). Note that GNNLab adopts a static caching scheme, so each sampled vertex can be marked in the Sample stage whether its feature is cached in GPU memory or not (see §6 for more details). The Trainer then runs a forward pass that computes the output based on a certain GNN model (e.g., GCN), followed by a backward

TABLE IV
THE SIMILARITY (IN PERCENTAGE) OF ACCESS FOOTPRINT BETWEEN TWO EPOCHS FOR VARIOUS DATASETS AND SAMPLING ALGORITHMS.

Sampling algorithms	PR	TW	PA	UK
3-hop random	73.97	78.89	91.29	77.46
Random walks	78.16	72.68	87.14	64.40
3-hop weighted	77.69	66.64	89.57	72.96

pass that uses a loss function to compute parameter updates. Moreover, GNNLab employs a simple pipelining mechanism in the Trainer to overlap the Extract and Train stages. Note that existing GNN systems (e.g., DGL) already leverage the pipelining mechanism during model training, which is also enabled within the Train stage of GNNLab. For multiple Trainers, they do not interact with each other except for exchanging locally produced gradients to update GNN model parameters. To support pipelining, GNNLab updates model gradients with bounded staleness, which effectively mitigates the convergence problem.

3) A Pre-sampling Based Caching Policy: The existing degree-based caching policy only works well under certain assumptions. Thus a caching policy that is efficient and robust to diverse GNN datasets and sampling algorithms is highly favored.

GNNLab proposes a presampling-based feature caching policy (PreSC). Given a graph G, a sampling algorithm A and a training set T, PreSC conducts K sampling stages, starting from the vertices in T. Here K is a user-defined parameter. It records the visit count of the sampled vertices and uses the average count as the hotness metric h_v . We use PreSC#K to denote the variant of PreSC that conducts K sampling stages.

Cache ratio. The cache ratio α determines how many vertices can be cached in GPUs. Observe that a larger α usually implies a higher cache hit rate. However, due to the limited GPU memory, it is unfeasible to cache all vertex features. We also need to reserve enough memory for GNN model training. In general, the value of α for a given training task can be determined by two factors, the available GPU memory amount for feature cache and the vertex feature dimension. To determine GPU memory capacity for feature cache, we adopt the method proposed in PaGraph [13], where we simulate one-time model training for a mini-batch and record the peak memory usage for model training. Then the rest of the available GPU memory is allocated for feature cache.

Following this general scheme, GNNLab provides a builtin procedure load_cache(hotness_map,α) to enable GPU-based feature cache. Here hotness_map is a data structure that stores the hotness value of each vertex, and α is the cache ratio which can either be specified by users manually or determined as we have discussed above. The procedure identifies and loads the features of the top-ranked $\alpha|V|$ vertices w.r.t. h_v into GPUs. It also builds a hash table to indicate the location in feature cache of a given vertex. We can easily implement existing caching policies in GNNLab with this caching scheme. For example, to implement the degreebased caching policy adopted by PaGraph [13] in GNNLab, it suffices to compute the out-degree of each vertex v as h_v and construct the data structure hotness_map.

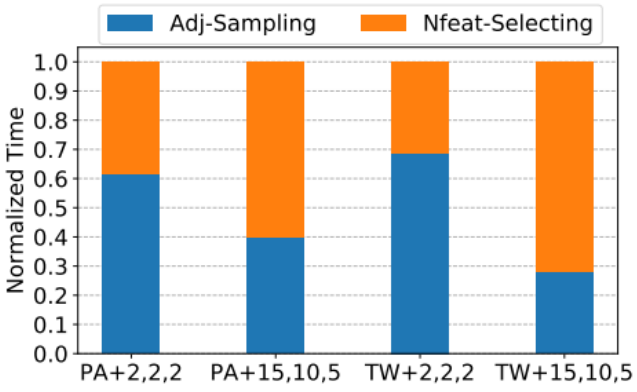

Fig. 13. Comparison of normalized time between Adj-Sampling and Nfeat-Selecting under different settings (datasets+fanouts). PA represents Ogbn-Papers100M and TW represents Twitter. (batch size=8000; nfeat dimension=128).

```
class Sampler(...):
  def run(self, dev, q, graph, ...):
    load(dev, graph)  # load graph to GPU memory
    khop3 = KHOP(graph, ...)  # define 3-hop sampling
    ...
    while (minibatch = get_minibatch())
      samples = khop3(minibatch)
      remap(dedup(samples))
      q.enque(samples)  # (async) send task
class Trainer(...):
  def run(self, dev, q, features, ...):
    model = GCN(...)  # define a 3-layer GCN
    loss_func = ...   # define a loss function
    ...
    while (samples = q.deque())  # (async) recv task
      in_feats = extract(features, samples)
      loss = loss_func(model(samples, in_feats), ...)
      loss.backward()
```

Fig. 14. Two kinds of executors in GNNLab.

GNNLab finds that a small number of sampling stages, i.e., K $\leq$ 2, already produce a decent hotness estimation and suffice for most training tasks. Thus it is feasible to compute the h_v 's of PreSC online, since (i) GPU-based graph pre-sampling is lightweight, e.g., on average it only takes 1.4× time of one epoch, and (ii) a typical GNN training pipeline usually has over 100 epochs. Specifically, we run the first K epochs of an end-to-end training pipeline for pre-sampling without features cache and determine which vertices should be cached. Then features of selected vertices are loaded into GPUs, and the rest of epochs can benefit from reduced data movement to GPUs. This presampling process can also be dealt with in an offline manner. In general, the benefits of pre-sampling based caching policy are two-fold: efficiency and robustness.

Efficiency. PreSC is very efficient in terms of cache hit rate. This is because the ideal hotness estimation metric $E[\hat{h}_v]$

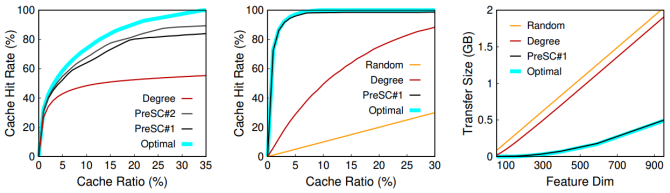

Fig. 15. The comparison among different caching policies for (a) Twitter with weighted sampling, (b) OGB-Papers with 3-hop neighborhood, and (c) OGB-Papers with the increase of feature dimensions. PreSC#K conducts K sampling stages.

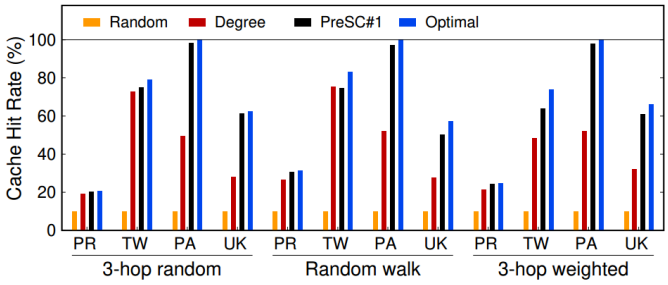

Fig. 16. Two kinds of executors in GNNLab.

captures the sampled frequency of a vertex in all epochs, and the hotness metric h_v of PreSC provides a good approximation of $E[\hat{h}_v]$. As shown in Figure 16, fixing cache ratio $\alpha = 10\%$, the cache hit rate of PreSC is almost as good as the Optimal policy, and is on average 1.5× (up to 2.2×) higher than that of the Degree policy. Recall that the Optimal policy defines an upper bound of cache hit rate for an fixed cache ratio, since it assumes that we can cache the actual most frequently sampled vertices in all epochs in advance.

Robustness. PreSC is robust to diverse datasets and sampling algorithms. As shown in Figure 16, on four GNN datasets and three sampling algorithms, PreSC constantly beats other baselines, including the Random policy and Degree policy adopted by PaGraph [13]. This is because, as oppose to prior work, the hotness metric hv of PreSC is computed by simultaneously taking the input graph G, the training set T and the sampling algorithm A into account. Observe that the performance of Degree policy is unstable. For example, for the 3-hop random neighborhood and random walks, the Degree policy has a similar cache hit rate as PreSC on the power-law graph TW . However, if we either use a non-power-law graph, e.g., PA, or use the weighted sampling, the cache hit rate of Degree drops very quickly, i.e., on average below 51% Instead, the performance of PreSC is stable and very close to Optimal in all 12 cases. These verify the robustness of PreSC.

The high efficiency and robustness of PreSC bring substantial advantages in processing large-scale graphs and high-dimensional features. To see this, in Figure 15(b) gnnlab first plots the cache hit rate to the cache ratio α on OGBPapers dataset with 3-hop random neighborhood sampling.

The cache hit rate of PreSC increases fast and reaches 96% when $\alpha = 5\%$. This verifies the effectiveness of PreSC to process large-scale graphs, i.e., PreSC is able to achieve a decent cache hit rate even with a very small α. In contrast, the cache hit rates of Random and Degree policies are below 5% and 29% when α = 5%, respectively. They increase much slower than PreSC. Observe that the hotness metric of PreSC takes the training set T into account, while the Random and Degree policies overlook the impact of T. For example, the Degree policy uses the static vertex out-degree as the hotness metric, which essentially assumes that the sampling operations are started from all vertices in the dataset. This largely reduces the cache utilization since some high degree vertices may never be sampled from a givenT. Fixing 5GB cache size, Figure 15(c) further shows the size of data moved to the GPU memory in one mini-batch as the feature dimension increases. We can see that as the dimension increases from 100 to 900, the transferred data size of PreSC increases much slower than Random and Degree policies. The transferred data size of PreSC is less than 500MB when the dimension is 900. Instead, Degree and Random need to move nearly 2GB data, which is 4× of that of PreSC.

To summarize, GNNLab presents the contributions as follows:

- A new factored space sharing design for sample-based GNN training that eliminates intra-task resource contention and unleashes inter-task data locality, and solutions to tackling the imbalanced load issues introduced by the factored design.
- A general GPU-based feature caching scheme, as well as a caching policy based on pre-sampling that is robust to diverse sampling algorithms and GNN datasets.
- An evaluation with various GNN datasets and models that shows the advantage and efficacy of GNNLab.

C. Experiments

The experiments were conducted on a GPU server that consists of two Intel Xeon Platinum 8163 CPUs (total 2 ×24 cores), 512GB RAM, and eight NVIDIA Tesla V100 (16GB memory, SXM2) GPUs. The software environment of the server was configured with Python v3.8, PyTorch v1.7, CUDA v10.1, DGL v0.7.1, and PyG v2.0.1.

Datasets. GNNLab used four datasets as listed in TableV for evaluation, including a social graph Twitter (TW), a web graph UK-2006 (UK), and two GNN datasets from Open Graph Benchmark (OGB) — a co-purchasing network OGB-Products (PR) and a citation network OGB-Papers (PA). Similar to prior work, GNNLab generated random features and labels for TW and UK since they originally had no features and labels. Both PR and PA from OGB provide an official training set. For TW and UK, which do not provide a training set, we followed a common practice that randomly selects a small portion of vertices as the training set. Note that the training set is selected offline once and shared across each run. The overhead to select the training set is trivial, e.g., less than 150 ms for the largest graph (UK).

Baselines. We compared GNNLab with PyG [20], DGL and T_{SOTA}, a state-of-the-art GNN system based on the conventional design, which extends DGL with a static GPUbased cache and a fast GPU-based sampler from scratch. As shown in Table V, PyG conducts graph sampling on CPUs, while DGL

TABLE V
DATASETS AND GNN SYSTEMS USED IN EVALUATION. $\#TS$ DENOTES THE SIZE OF TRAINING SET. Vol_G (RESP. Vol_F) IS THE DATA VOLUME OF GRAPH TOPOLOGICAL (RESP. FEATURE) DATA IN HOST MEMORY. N/A REPRESENTS THE GPU IS ONLY USED BY A SINGLE STAGE (I.E., TRAIN)

Dataset	#Vertex	#Edge	Dim.	#TS	Vol_G	Vol_F
PR	2.4M	124M	100	197K	481MB	934MB
TW	41.7M	1.5B	256	417K	5.6GB	40GB
PA	111M	1.6B	128	1.2M	6.4GB	53GB
UK	77.7M	3.0B	256	1.0M	11.3GB	74GB

System	Design	Sample	Extract	Train
PyG	N/A	CPU	No cache	GPU
DGL	Time S.	GPU	No cache	GPU
T_{SOTA}	Time S.	GPU w/ Opt.	Cache w/ Degree	GPU
GNNLab	Space S.	GPU w/ Opt.	Cache w/ PreSC	GPU

enables GPU-based sampling to accelerate graph sampling. T_{SOTA} is built upon the same codebase of GNNLab, and supports both GPUbased graph sampling and feature caching. Different from GNNLab, T_{SOTA} follows a time sharing design, i.e., each GPU conducts both graph sampling and model training, and adopts the degree-based caching policy. DGL also uses time sharing, but has no caching mechanism. Since DGL only supports synchronous gradient updates, for fair comparisons, GNNLab and other baselines employ synchronous gradient updates unless otherwise specified. All results were computed by calculating the averages over 10 epochs

1) Overall Performance: Table V reports the end-to-end training time of one training epoch for each GNN system. The number of GPUs allocated to Samplers (nS) is determined by the factored design. Note that for an 8-GPU machine, our flexible scheduling scheme already provides optimal GPU allocations for Samplers. Therefore, dynamic switching does not happen in this evaluation. It is mainly found that: (1) Overall, GNNLab outperforms DGL and PyG by up to 9.1× (from 2.4×) and 74.3× (from 10.2×), respectively. For systems using GPUs for graph sampling, only GNNLab can process the UK dataset in all cases, while other systems run out of memory (OOM) due to GPU memory contention. Note that PyG performs the worst in all experiments due to the high cost of graph sampling on CPUs and transferring features to GPUs. Due to space limitations, we do not report its experimental results in the rest of our evaluation. In general, the performance gain of GNNLab over its competitors mainly comes from three aspects: (A1) a new space-sharing design that unleashes the power of GPU-based sampling and caching, (A2) an efficient and robust caching policy (PreSC) and (A3) an efficient implementation of GPUbased graph sampling. (2) Compared with T_{SOTA}, GNNLab benefits from (A1) and (A2). Indeed, T_{SOTA} suffers from GPU memory contention and its inefficient degree-based caching policy. Specifically, T_{SOTA} needs to load graph topological data into each GPU, leaving only limited memory for the feature cache. As shown in Figure 15, PreSC is more efficient than the degree-based policy, especially when the cache ratio is small. Note that T_{SOTA} performs slightly better than GNNLab on PR. This is because all topological and feature data of PR can be loaded into a single GPU. Therefore, (A1) and (A2) cannot improve that case, while our factored design introduces little overhead in the Sample stage. Note that T_{SOTA} is built upon the same codebase of GNNLab. (3) The performance gain of GNNLab over DGL comes from (A1)˜(A3). DGL stores all feature data in the host memory and has no GPU-based caching mechanism. Thus, it transfers much more data to GPUs than T_{SOTA} and GNNLab. In addition, DGL only uses CPUs to extract features of sampled vertices, which also incurs a large number of random memory accesses. Therefore, apart from the more severe GPU memory contention problem, the limited memory access bandwidth shared by CPUs is another major bottleneck.

2) Performance Breakdown: A stage-level time breakdown analysis was conducted for the SET model on DGL, T_{SOTA}, and GNNLab with two GPUs (1S1T for GNNLab). The results are reported in Table VI. We mainly find the following.

(1) For the Sample stage (S), GNNLab and T_{SOTA} beat DGL on three models. We find that DGL adopts the Reservoir algorithm [32] for k-hop random neighborhood sampling on GPUs. The sampling complexity of each vertex is positively correlated with its in-degree, resulting in an unbalanced workload on GPU threads. Instead, GNNLab and T_{SOTA} implement a variant of the Fisher–Yates algorithm [33], which is GPU-friendly since the workload is more balanced for each vertex. Note that the performance gain of GNNLab and T_{SOTA} over DGL are larger on PinSAGE [34] than on GCN [4] and GraphSAGE [5]. After profiling, we find that invoking CUDA code from Python code in DGL incurs considerable runtime overheads. Meanwhile, compared with k-hop random neighborhood sampling, random walks has more complex vertex access patterns, making the runtime overheads more significant. Furthermore, compared to T_{SOTA}, GNNLab incurs additional overheads (less than 0.1 ms on average) for copying samples to a global queue in the host memory.

(2) For the Extract stage (E), the performance largely depends on cache size and caching policy. The former determines the cache ratio of features (R%), and the latter determines the cache hit rate (H%). Without caching, DGL must transfer all features of sampled vertices from host memory to GPU memory, which accounts for up to 85.0% (from 38.8%) of end-to-end time. By enabling GPU-based caching and the degree-based policy, T_{SOTA} performs much better, especially for small datasets (e.g., PR). However, the time sharing design greatly limits the available memory capacity for the feature cache, resulting in relatively low cache ratios for large graphs (e.g., only 1% for GCN on TW). Further, the degree-based caching policy is still far from efficient for many graphs and sampling algorithms. For example, the cache hit rate for GCN on PA is only 37%, when caching 7% of features. In contrast, GNNLab never gets bogged down by extracting features in all experiments, thanks to our space sharing design and presampling based caching policy (PreSC). First, the cache ratio is significantly improved in GNNLab, e.g., from 1% to 25% for GCN on TW. Second, PreSC demonstrates surprising efficiency. For example, on PA, caching less than 25% of features reduces data movement by more than 97%. However, compared to T_{SOTA}, GNNLab incurs additional overheads (less than 0.1 ms on average) for loading samples into GPU memory from the global queue. Overall, GNNLab outperforms

TABLE VI
THE RUNTIME BREAKDOWN (IN SECONDS) OF ONE EPOCH FOR DGL, T_{SOTA} AND GNNLAB. S, E, AND T REPRESENT SAMPLE, EXTRACT, AND TRAIN STAGES. G, M, AND C REPRESENT GRAPH SAMPLING, MARKING CACHED VERTICES, AND COPYING SAMPLES TO THE HOST MEMORY IN THE SAMPLE STAGE, RESPECTIVELY. R% AND H% REPRESENT THE CACHE RATIO OF FEATURES AND THE CACHE HIT RATE. GSG AND PSG ARE SHORT FOR GRAPHSAGE AND PINSAGE.

GNN	Dataset	DGL			T_{SOTA}			GNNLab		
		S	E	T	S = G + M	E (R%, H%)	T	S = G + M + C	E (R%, H%)	T
GCN	PR	0.35	2.81	1.22	0.30 = 0.29 + 0.01	0.04 (100, 100)	1.18	0.39 = 0.29 + 0.01 + 0.09	0.15 (100, 100)	1.18
	TW	0.74	9.44	1.48	0.29 = 0.26 + 0.03	3.68 (1, 29)	1.53	0.37 = 0.26 + 0.03 + 0.08	0.76 (25, 89)	1.51
	PA	1.20	10.70	4.00	0.79 = 0.70 + 0.10	3.64 (7, 38)	4.00	0.96 = 0.68 + 0.10 + 0.18	0.49 (21, 99)	3.82
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.56 = 0.39 + 0.03 + 0.14	3.06 (14, 70)	3.09
GSG	PR	0.13	1.92	0.23	0.16 = 0.15 + 0.01	0.03 (100, 100)	0.25	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	0.24
	TW	0.38	4.65	0.44	0.12 = 0.11 + 0.01	0.62 (15, 77)	0.44	0.16 = 0.11 + 0.01 + 0.03	0.41 (32, 89)	0.43
	PA	0.56	6.06	1.25	0.38 = 0.33 + 0.06	1.42 (11, 56)	1.18	0.46 = 0.31 + 0.06 + 0.08	0.28 (25, 99)	1.15
	UK	OOM	OOM	OOM	0.19 = 0.19 + 0.00	4.49 (0, 0)	1.08	0.26 = 0.18 + 0.02 + 0.06	1.39 (18, 72)	1.01
PSG	PR	0.40	1.64	1.75	0.16 = 0.16 + 0.01	0.03 (100, 100)	1.74	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	1.72
	TW	0.72	5.22	2.59	0.23 = 0.22 + 0.02	1.12 (4, 60)	2.60	0.28 = 0.21 + 0.02 + 0.05	0.51 (26, 86)	2.52
	PA	1.86	4.85	5.78	0.54 = 0.49 + 0.05	1.68 (6, 37)	6.09	0.61 = 0.47 + 0.04 + 0.09	0.33 (22, 97)	6.01
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.65 = 0.49 + 0.03 + 0.13	3.37 (13, 57)	7.00

T_{SOTA} by 4.2× on average in the Extract stage (except for PR), due to fetching over 84% of required features directly from the GPU cache.

(3) For the Train stage (T), GNNLab, T_{SOTA} and DGL have similar performance, as all three systems employ the same GNN execution runtime (i.e., DGL) in this stage. For flexible scheduling, in most cases, the training time is used to calculate the number of GPUs allocated to Sampler (N_s). For GCN and GraphSAGE on UK, the extracting time dominates the processing time of Trainer (T_t), so that it replaces the training time to calculate N_s.

VII. DUCATI

DUCATI is a workload-aware cache-based system tailored for the mini-batch training of GNNs on giant graphs. DUCATI leverages the spare GPU memory to cache frequently accessed parts of the graph data. Compared with existing Single-Cache systems, which only employ Nfeat-Cache, DUCATI is a Dual-Cache system that considers both Adj-Cache and Nfeat-Cache. The additionally introduced Adj-Cache can accelerate the time-consuming Adj-Sampling workload and mitigate the marginal effects of Nfeat-Cache. DUCATI also includes a workload-aware Dual-Cache Allocator to adapt to diverse settings. Given the current setting, our Dual-Cache Allocator will find the most prominent workload inside the mini-batch generation (Adj-Sampling or Nfeat-Selecting), and allocate more budget to the corresponding cache (Adj-Cache or Nfeat-Cache). Overall speaking, DUCATI will adaptively find the best plan to allocate the spare GPU memory to Adj-Cache and Nfeat-Cache so as to achieve the fastest training speed.

The overview of DUCATI is shown in Figure 17. DUCATI first extract essential information from the input using the Input Inspector. Then, with the extracted information, the Dual-Cache Allocator will determine the best allocation plan based on current workloads so as to achieve the fastest speed. Next, DUCATI construct Adj-Cache and Nfeat-Cache with the Cache Constructor. Finally, DUCATI perform the mini-batch training with the constructed two caches.

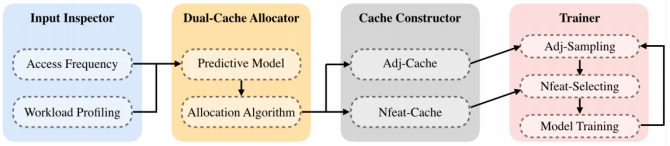

Fig. 17. Overview of DUCATI. It has 4 modules, named Input Inspector, Dual-Cache Allocator, Cache Constructor, Trainer respectively

A. Motivations

In this section, some provide empirical results are provided to illustrate the limitations of the existing Single-Cache systems and observations that are motivated to build a Dual-Cache system.

1) The data access to the adjacency matrix also has locality.: It is found that the data access to the adjacency matrix also exhibits locality like the node features. For example, when training GraphSAGE on the Ogbn-Papers100M dataset, 0.7 GB entries in the 12.9 GB of the adjacency matrix account for more than 98% of the total adjacency accesses.

2) Adj-Sampling can be more time-consuming than Nfeat-Selecting.: Depending on the input settings, the workloads of Adj-Sampling and Nfeat-Selecting are diverse as shown in Figure 13. According to the Amdahl's law, the overall speedup of existing Single-Cache systems is limited since they only optimize Nfeat-Selecting even when Adj-Sampling takes more time than Nfeat-Selecting.

3) Nfeat-Cache cannot utilize all spare GPU memory.: As illustrated in Figure 4, it is found that Nfeat-Cache exhibits the marginal effect as the cache size increases. Many real-world graphs follow a power law, and only a small fraction of node features are frequently accessed. This implies that DUCATI can get most of the benefit from Nfeat-Cache with only a

small cache budget. As observed in Figure 18, Nfeat-Cache can hardly benefit from more than 2.5 GB cache budget. Thus the leftover spare GPU memory is not utilized properly.

4) Opportunities.: Nfeat-Cache cannot exploit all spare GPU memory, and the time-consuming Adj-Sampling can be accelerated by introducing Adj-Cache. Inspired by these observations, DUCATI is proposed by combining Adj-Cache and Nfeat-Cache.

B. Design of DUCATI

1) Input Inspector: The Input Inspector will extract two pieces of crucial information from the input with offline sample runs. First, the access frequency of each entry is calculated. Such frequency is an estimation of the real access probability of each entry. Intuitively, entries with high access probability and low cost (small size) is inclined to be cached. Second, the workload information of Adj-Sampling and Nfeat-Selecting with profiling is collected. Also, statistics on how these two workloads benefit from the GPU cache is collected. Such workload information to build the workload-aware Dual-Cache Allocator will be used.

Information of entries' access frequency. The access frequency of each entry during several offline sample runs is calculated. DUCATI uses the access frequency to estimate the real access probability of each entry.

Information of workload profiling. Different inputs usually lead to different workloads of Adj-Sampling and Nfeat-Selecting. The same cache budget also leads to different improvements when allocated to Adj-Cache or Nfeat-Cache with different ratios. Such uncertainty makes it hard to solve the cache allocation problem. Such uncertainty is addressed by profiling in advance how Adj-Sampling and Nfeat-Selecting benefit from the GPU-based cache given the current input. Concretely, DUCATI will construct several Adj-Cache with randomly chosen entries and profile the corresponding running time of Adj-Sampling. DUCATI does the same with Nfeat-Selecting and Nfeat-Cache. Finally, DUCATI obtains data points in the format of (cached entries, running time).

2) Dual-Cache Allocator: The target of the cache allocator is to compare the benefit of caching each entry and select the most beneficial ones among all the candidates. The benefit of one entry is a measure of how much the caching of this entry can reduce the system running time. Under the single-cache setting, this task is easy since all the entries are homogeneous, i.e., all entries have the same shape and the same access pattern. Thus it is easy to compare the benefit of caching different entries and select the most beneficial ones. However, such a task is more challenging under the dual-cache setting because heterogeneous entries need to be considered. In DUCATI, both adjacency lists and node features are cached. Both types of entries exhibit heterogeneity in both the data characteristics and the access pattern. Specifically, the adjacency lists have variable lengths, while all the node features have the same length. And the access of adjacency lists is scattered in fine-grained computation, while the access of node features is a one-time effort inside one mini-batch. Such heterogeneity makes it hard to compare the benefit of caching different types of entries. In addition, many other factors make the allocation problem hard to analyze. On the hardware side, different utilization of the equipped PCIe lanes lead to different data transfer speeds. On the software side, diverse settings and inputs lead to dynamic workloads of Adj-Sampling & Nfeat-Selecting. All of these factors lead to different training efficiency even with the same amount of cache, which makes it challenging to develop a closed-form solution based on the known GNN training procedure and the graph structure. In DUCATI, the allocation problem is solved by introducing a benefit prediction model. Roughly, the benefit prediction model can capture all influencing factors with profiling and measure the benefit of caching different types of entries. Then the benefits of all entries can be compared and the most beneficial ones selected. With this model, the dual-cache allocation problem is transformed into a variant of the knapsack problem. Then an algorithm is proposed to solve the dual-cache allocation problem.

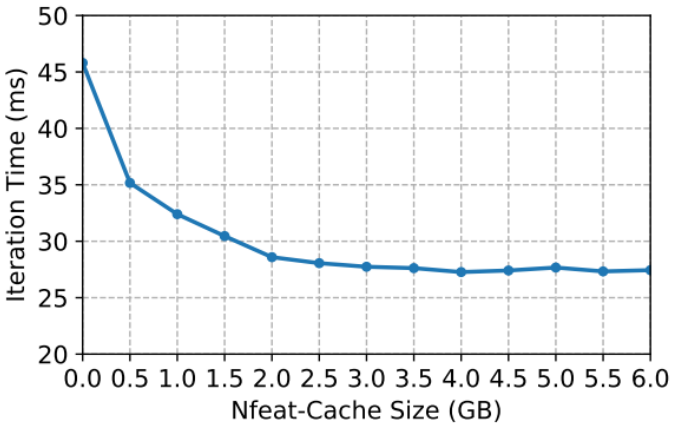

Fig. 18. The trend of iteration time with respect to the increase of Nfeat-Cache. DUCATI obtain the results on Ogbn-Papers100M dataset with GraphSAGE. (fanouts=15,10,5; batch size=8000; nfeat dimension=128)

The Benefit Prediction Model. In DUCATI, since DUCATI has two different types of entries to be cached, it needs to find a unified way to evaluate the benefit of caching each entry. Specifically, DUCATI wants to build a predictive model that maps the theoretical properties of one entry to the empirical reduction of the system running time if DUCATI caches the entry. It leverages one important observation that helps us to build a simple yet effective predictive model. As shown in Figure 19, DUCATI plots the running time of Adj-Sampling/Nfeat-Selecting with respect to different sizes of some randomly constructed Adj-Cache/Nfeat-Cache. DUCATI finds that (1) the running time of each task is almost (inversely) linearly related to the cache hit of the corresponding cache and (2) the slope of each line, which represents the decreasing speed of running time w.r.t. cache size, varies on different tasks/datasets.

3) Cache Constructor: **Adj-Cache.** The design of Adj-Cache is challenging due to: (1) the variable length of adjacency lists may require auxiliary data structure and procedures for the storing and the lookup of Adj-Cache and (2) the special workload of Adj-Sampling requires a lightweight cache lookup

mechanism. The data access of adjacency lists is interleaved with the fine-grained computation performed by CUDA kernels. To avoid lagging the execution of the fine-grained computation, we need to make the cache lookup lightweight and fast. These two challenges are addressed by adopting a CSC-like format for Adj-Cache combined with graph reordering. Our design requires no auxiliary data structure for Adj-Cache and a very lightweight cache lookup mechanism. Firstly, a graph reordering is performed so that these cached adjacency lists are at the front of all adjacency lists. Then, these elements are sliced at the front of three original CSC arrays and form three cached CSC arrays, namely col_index_cached, row_index_cache, values_index_cached

Nfeat-Cache. DUCATI constructs the same Nfeat-Cache as previous works [14]. Specifically, given the node features to be cached, DUCATI gathers these node features and store them as a GPU tensor. DUCATI also prepares a lookup table that stores the location information of all node features. During Nfeat-Selecting, DUCATI first checks all queried node features with the lookup table. When Nfeat-Cache hits, it fetches the data from the GPU using the address returned by the lookup table. Otherwise, Nfeat-Cache misses, and DUCATI fetches the data from the host memory with the UVA technique.

4) Trainer: Similar to previous works [14], DUCATI makes only minor changes to the programming interface of DGL. After constructing Adj-Cache and Nfeat-Cache, DUCATI abstracts the usage of two caches with two APIs, namely the sample and the load.

To summarize, DUCATI present the following contributions:

- A novel Dual-Cache training system DUCATI is proposed, which exploits the locality of the adjacency matrix in addition to that of the node features and makes better use of the GPU memory compared with existing Single-Cache systems.
- A novel workload-aware Dual-Cache Allocator is proposed which adaptively decides the best allocation plan for Adj-Cache and Nfeat-Cache to maximize training speed.
- The experimental results show that DUCATI is efficient and scalable for training GNNs on giant graphs compared with DGL and the state-of-the-art Single-Cache systems. The proposed Dual-Cache Allocator can better utilize the GPU memory with respect to diverse workloads.

C. Experiments

DUCATI was implemented on top of DGL v0.8 and PyTorch v1.9. The overall implementation consists of 1.6k lines of Python codes and 422 lines of CUDA and C++ codes. The experiments were run on a single machine with one NVIDIA GeForce RTX 2080 Ti GPU (11GB of HBM memory) with PCIe 3.0x16 and 256GB DDR4 memory and Intel(R) Xeon(R) Gold 6240 CPU@2.60GHz.

1) Datasets: DUCATI uses four billion-scale datasets for the evaluation of mini-batch training systems and two smaller datasets for the evaluation of full-batch training systems as listed in Table VII. Due to the worse scalability of full-batch systems, they encounter GPU OOM on billion-scale graphs, thus DUCATI compares solution with full-batch ones on the two smaller datasets. For billion-scale datasets, DUCATI uses two web graphs: UK-2006-05 (UK) and UK-Union (UU), a social graph: Twitter (TW), a citation graph in Open Graph Benchmark (OGB) Ogbn-Papers100M (PA). For smaller datasets, DUCATI uses a co-purchasing network Ogbn-Products (PR) from OGB, and an online posts network Reddit (RD). To test the scalability and adaptability on different settings, DUCATI generates two commonly used feature lengths (128 and 256) for each dataset. OGB provides an official training set for PA. For the other three datasets which do not provide a training set, it samples 1 percent of all vertices as the training set. DUCATI notices that vertices in the training set of PA have a higher average degree than that of all vertices, and hence for the other three datasets, it samples the training set with the normalized degree as the sampling probability. The training set selection is performed offline and shared across all experiments.

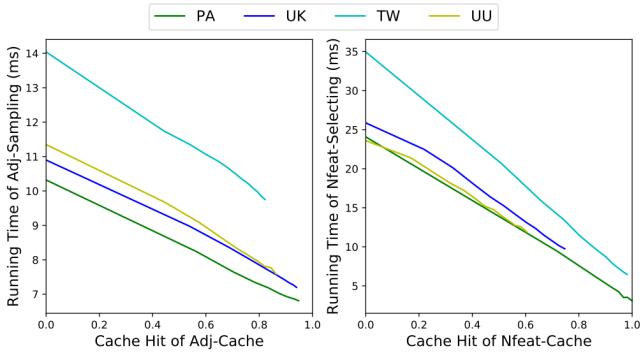

Fig. 19. System running time of Adj-Sampling and Nfeat-Selecting with respect to the cache hit of Adj-Cache and Nfeat-Cache

2) Baselines: DUCATI categorizes existing systems into five types:

- Base System: PyG [20]. PyG is the base system that uses the CPU for Adj-Sampling and Nfeat-Selecting.
- System with UVA: DGL [22]. The latest DGL v0.8 release supports the UVA technique. DGL has no cache mechanism.
- System with Single-Cache: PaGraph [13] & GNNLab [14]. PaGraph is built upon an older version (v0.4) of DGL. PaGraph only supports CPU-based sampling which is an order of magnitude slower than UVA-based sampling. GNNLab will OOM on all datasets in Table VII because it cannot scale to datasets with an adjacency matrix larger than GPU memory.
- System with UVA and Single-Cache: SOTA. SOTA is a synthetic system which is built with the same code base as DUCATI. SOTA employs UVA for Adj-Sampling, UVA for Nfeat-Selecting, and NfeatCache. SOTA constructs the same Nfeat-Cache as GNNLab, which is better than other NfeatCache.
- System with UVA and Dual-Cache: the DUCATI.

3) Evaluation of Overall Training Time: The overall training time per iteration of DUCATI and baselines are given in

TABLE VII
DATASET STATISTICS.

Dataset	Nodes	Edges	Feature	Size(adj)	Size(nfeat)
PA	111M	1.6B	128/256	12.9GB	54/108GB
UK	77.7M	3.0B	128/256	11.3GB	38/75GB
TW	41.7M	1.5B	128/256	22.7GB	20/40GB
UU	134M	5.5B	128/256	42.0GB	65/130GB
PR	2.4M	124M	100	0.94GB	0.91GB
RD	232K	114M	602	0.86GB	0.13GB

Table VIII. On average, DUCATI can speed up the training by 1.32 times compared with SOTA and 2.07 times compared with DGL. The speedup can be observed under all settings, which verifies the generalization ability of DUCATI.

(1) Compared with DGL, we can observe the benefit of caching frequently access graph data in GPU. When the fanouts and the node features dimension are small, the graph data of a mini-batch is relatively small, which makes the repeated transfer of graph data less obvious. Thus the speedups of SOTA and DUCATI are relatively small in such cases. However, under settings where the fanouts and the node features dimension are large, the UVA-based DGL will be dragged considerably by the repeated transfer of graph data. In such cases, the caching mechanism of SOTA and DUCATI can achieve more speedup than DGL.

(2) Compared with SOTA, we can observe the advantage of Dual-Cache paradigm over SingleCache paradigm. DUCATI uses the same amount of GPU cache as SOTA while achieving faster training speed, which demonstrates that DUCATI can make better use of spare GPU memory. The reasons behind this are that Nfeat-Cache cannot utilize all spare GPU memory and DUCATI's Adj-Cache can accelerate the time-consuming Adj-Sampling by leveraging the leftover space of GPU memory.

(3) By comparing three methods in the same setting, we can find that DUCATI usually achieves higher speedup over SOTA and DGL with the increase of the input size (fanouts and node features dimension). In other words, if we compare the performance of each method under increasingly larger input settings, we can find that the training time of DUCATI grows slower than that of SOTA and DGL. This observation demonstrates the good scalability of DUCATI.

TABLE VIII
ITERATION TIME OF ALL METHODS (UNIT: MS). SETTING REPRESENTS THE NEIGHBOR FANOUTS + NODE FEATURES DIMENSION.

Dataset	Setting	DGL	*SOTA*	DUCATI
PA	2,2,2+128	16.05(1.51x)	13.40(1.26x)	10.65
PA	2,2,2+256	18.48(1.82x)	12.75(1.26x)	10.13
PA	15,10,5+128	49.24(2.67x)	28.39(1.54x)	18.45
PA	15,10,5+256	71.79(3.33x)	31.16(1.44x)	21.59
UK	2,2,2+128	18.54(1.60x)	15.58(1.34x)	11.62
UK	2,2,2+256	23.63(1.65x)	18.79(1.31x)	14.35
UK	15,10,5+128	61.41(2.22x)	35.72(1.29x)	27.64
UK	15,10,5+256	99.61(2.15x)	61.56(1.33x)	46.37
TW	2,2,2+128	17.81(1.29x)	15.63(1.13x)	13.85
TW	2,2,2+256	19.86(1.50x)	15.44(1.17x)	13.22
TW	15,10,5+128	71.74(2.95x)	33.06(1.36x)	24.30
TW	15,10,5+256	119.89(3.00x)	53.43(1.33x)	40.03
UU	2,2,2+128	18.47(1.55x)	15.95(1.34x)	11.90
UU	2,2,2+256	23.56(1.61x)	19.87(1.36x)	14.63
UU	15,10,5+128	62.83(2.15x)	39.95(1.37x)	29.21
UU	15,10,5+256	100.00(2.04x)	66.95(1.36x)	49.11
	Avg. Speedup	2.07x	1.32x	1.00x

VIII. CONCLUSION

In this paper, 3 methods are introduced to tackle the problem that it is time-consuming when scaling GNNs to large graph with mini-batch training. The UVA-based method focuses on 1. The single cache method focuses on 2. The dual cache method focuses on fully utilizing the spare GPU memory and exhibiting poor adaptability to diverse workloads. The experimental results show that dual-cache performs better.

REFERENCES

[1] Q. Tan, N. Liu, and X. Hu, "Deep representation learning for social network analysis," *Frontiers in big Data*, vol. 2, p. 2, 2019.

[2] 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.

[3] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui, "Graph neural networks in recommender systems: a survey," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–37, 2022.

[4] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016. [Online]. Available: http://arxiv.org/abs/1609.02907

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

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

[7] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[8] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim, "Graph transformer networks," *Advances in neural information processing systems*, vol. 32, 2019.

[9] Y. Cen, Z. Hou, Y. Wang, Q. Chen, Y. Luo, X. Yao, A. Zeng, S. Guo, P. Zhang, G. Dai *et al.*, "Cogdl: An extensive toolkit for deep learning on graphs. arxiv 2021," *arXiv preprint arXiv:2103.00959*.

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

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

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

[13] 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.

[14] 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.

[15] 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 19)*, 2019, pp. 443–458.

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

[17] L. Wang, Q. Yin, C. Tian, J. Yang, R. Chen, W. Yu, Z. Yao, and J. Zhou, "Flexgraph: a flexible and efficient distributed framework for gnn training," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 67–82.

[18] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim *et al.*, "Dorylus: Affordable, scalable, and accurate {GNN} training with distributed {CPU} servers and serverless threads," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 495–514.

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

[20] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[21] 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.

[22] DGL Team, "Using gpu for neighborhood sampling," https://docs.dgl.ai/guide/minibatch-gpu-sampling.html, accessed: 2024-12-02.

[23] 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.

[24] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[25] Web data commons - hyperlink graphs. [Online]. Available: http://webdatacommons.org/hyperlinkgraph/index.html

[26] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W.-m. Hwu, "Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus," *arXiv preprint arXiv:2006.06890*, 2020.

[27] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, "Traversing large graphs on gpus with unified memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, p. 1119–1133, Mar. 2020.

[28] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: Minimizing data transfer during out-of-gpu-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[29] M. Harris. (2013) How to access global memory efficiently in cuda c/c++ kernels. [Online]. Available: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

[30] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding pcie performance for end host networking," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 327–341. [Online]. Available: https://doi.org/10.1145/3230543.3230560

[31] NVIDIA, "MULTI-PROCESS SERVICE," 2020. [Online]. Available: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

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

[33] R. A. Fisher, F. Yates *et al.*, *Statistical tables for biological, agricultural and medical research, edited by ra fisher and f. yates*. Edinburgh: Oliver and Boyd, 1963.

[34] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 974–983.