2025 IEEE International Symposium on High Performance Computer Architecture (HPCA) | 979-8-3315-0647-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPCA61900.2025.00083

Buffalo: Enabling Large-Scale GNN Training via Memory-Efficient Bucketization

Shuangyan Yang Minjia Zhang†[1] Dong Li[1]
University of California, Merced, University of Illinois Urbana-Champaign†
syang127@ucmerced.edu, minjiaz@illinois.edu, dli35@ucmerced.edu

***Abstract*—Graph Neural Networks (GNNs) have demonstrated outstanding results in many graph-based deep-learning tasks. However, training GNNs on a large graph can be difficult due to memory capacity limitations. To address this problem, we can divide the graph into multiple partitions. However, this strategy faces a memory explosion problem. This problem stems from a long tail in the degree distribution of graph nodes. This strategy also suffers from time-consuming graph partitioning, difficulty in estimating the memory consumption of each partition, and time-consuming data preparation (e.g., block generation). To address the above problems, we introduce Buffalo, a GNN training system. Buffalo enables flexible mapping between the nodes and partitions to address the memory explosion problem, and enables fast graph partitioning based on node bucketing. Buffalo also introduces lightweight analytical modeling for memory estimation, and reduces block generation time by leveraging graph sampling. Evaluating large-scale real-world datasets (including billion-scale datasets), we show that Buffalo effectively addresses the memory capacity limitation, enabling scalable GNN training and outperforming prior works in the compute-vs-memory efficiency Pareto frontier. With a limited memory budget, Buffalo achieves an end-to-end reduction of training time by 70.9% on average, compared to state-of-the-art (DGL [73], PyG [12], and Betty [93]).**

I. Introduction

Graph neural network (GNN) frameworks such as DGL [34] and Pytorch Geometric [12] have been widely adopted for GNN training. These frameworks let users write GNN training programs using a set of graph-neighborhood aggregation operators and deep learning (DL) operators without worrying about the implementation of low-level message-passing primitives and their interactions with DL models.

Problems. Although current frameworks provide numerous operators and optimizations for training GNNs, training advanced GNNs over large-scale graphs is still quite challenging because the memory complexity of aggregation scales *exponentially* with the depth of GNNs. This makes them inefficient for several important use cases: (i) GNN training that exploits multi-hop information [42], [76], [103], and (ii) GNN training that explores advanced aggregators [16]. Unfortunately, the most common way to overcome the complexity of memory consumption is via sampling. Systems such as DGL [73] and Pytorch Geometric [12], provide sampling-based methods, where a graph is sampled and used in each training iteration. However, using a low sampling rate can lead to biased results and loss of the GNN model accuracy [70]. Also, sampling cannot be applied to certain datasets and tasks. For example, in molecule structure prediction, dropping connections in between atoms would lead to completely different molecules. Furthermore, batch-based training that utilizes sampling can still encounter out-of-memory (OOM) issues. To tackle the GPU memory capacity problem, another strategy in addition to sampling is to divide the input graph into multiple partitions. Each partition fits into the GPU memory, and the GPU processes the partitions in order. However, the partition-based approach faces multiple problems.

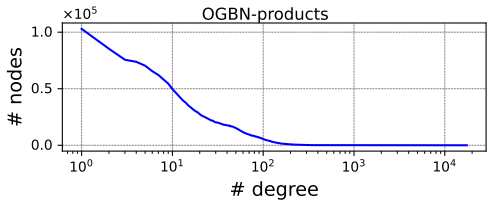

Fig. 1. The degree frequency of all nodes in the OGBN-products dataset.

First, the partition-based approach can suffer from a so-called *bucket explosion* problem. We observe that the number of nodes with various degrees often follows a power-law distribution, which is characterized by a long tail. This problem is commonly seen in many datasets (e.g., OGBN-papers [29], MAG [21], IGB [29], and others [78], [104]). Figure 1 shows the degree frequency of all nodes (i.e., for a specific node-degree, the number of nodes with that degree). This figure, using the OGBN-products dataset, supports the above observation. When bucketing the nodes according to their degrees for efficient processing (called *degree bucketing*), as the existing system does [73], the distribution with such a long tail can lead to a large bucket that accommodates the long tail. This bucket is much larger than the other buckets, which not only causes load imbalance across buckets but also invalidates the effectiveness of the partition-based approach to reduce memory consumption.

Second, the graph partitioning can be time-consuming. Existing efforts employ either graph-level partitioning [74], [101] or batch-level partitioning [4], [84], [93], both of which commonly employ METIS [26]. METIS iteratively simplifies, partitions, and refines the graph. This process is slow and computationally intensive, because it explores how to exchange nodes between partitions based on repeated analysis of node dependency across partitions. To accelerate the METIS-based partitioning, Betty [93] applies METIS to a smaller graph of output nodes rather than the larger graph of input nodes

[1]Equal advisorship.

979-8-3315-0647-6/25/$31.00 ©2025 IEEE
DOI 10.1109/HPCA61900.2025.00083

as other solutions [12], [90], [101]. However, Betty must explicitly embed node-dependency information into the graph of output nodes in order to minimize redundancy across the partitions. This embedding process can take a few minutes for a billion-scale graph (e.g., OGBN-papers). In general, the METIS-based graph partitioning is time-consuming. As a result, the graph partitioning must happen offline before the GNN training.

Third, estimating the memory consumption of each partition to avoid underutilization of GPU memory or memory overflow is challenging. The memory estimation for each partition must consider the variance of degree across nodes, data dependencies, and complex graph topology. As a result, the memory consumption of a partition is not a simple linear relationship to the number of nodes, which is challenging to predict.

Fourth, the block generation needed by the GNN training is time-consuming. A block in a GNN framework represents a structure that summarizes the connectivity and features of a subset of nodes and edges from the graph. Using blocks can enhance the training process because each block bundles connectivity information into a single object, enabling efficient data transfer. To generate the block, node connectivity must be examined from one node to another and from one GNN layer to another following a specific order. During training, the neighbors of a node can change across iterations because of node sampling in each iteration. Without sampling, node neighbors remain unchanged, risking overfitting in GNN training. Thus, we need to assess node connectivity in each iteration, which is time-consuming.

Solutions. To address the above problems, we introduce Buffalo, an *online* GNN training system. Buffalo does not change GNN. With Buffalo, the GNN convergence result is exactly the same as that when sufficient GPU memory is provided to accommodate the whole graph. Buffalo is general and supports full-batch and mini-batch training, because it allows a batch to be partitioned to fit into GPU memory.

To address the bucket explosion problem, Buffalo splits and groups buckets into bucket groups. Each group is a list of small-sized buckets with varying degrees of nodes or a portion of a large-sized degree-bucket. This bucket re-organization enables a flexible mapping between nodes and the partitions.

Buffalo performs graph partitioning based on buckets (i.e., *bucket-level partitioning*), which is much faster than using METIS. Using bucket-level partitioning effectively exploits the clustering structure of the graph, hence avoiding repeated analysis on node dependency and significantly saving time. In particular, the real-world graph typically exhibits clustering characteristics, where the nodes with similar degrees are clustered together [2], [15], [28], [50], [79] , which aligns with the principle of degree bucketing. Since there are fewer connections across clusters (compared with nodes within each cluster), using bucket-level partitioning, there are fewer connections across partitions. Hence, the bucket-level partitioning does not need to check node dependency across partitions, but simply splits and groups buckets, which is lightweight.

Buffalo introduces lightweight analytical modeling to estimate memory consumption of each partition. The modeling uses graph characteristics (e.g., the degree of the buckets and average clustering coefficient) to quantify the redundancy across partitions and efficiently predict memory consumption.

To reduce block generation time, Buffalo samples all neighbors of the center nodes in the subgraph (after sampling), thereby avoiding repeated connection checks to confirm which neighbors are selected from the original graph (before sampling). Also, the neighborhood checks occur in parallel at the node level. As a result, Buffalo significantly decreases block generation time by 10x, making online training feasible.

Besides the above solutions, Buffalo is featured with a scheduling algorithm to improve load balance across partitions. The algorithm minimizes the number of bucket groups by modeling the bucket-level partitioning as a knapsack problem and employing a greedy search algorithm. The analytical modeling is used to estimate memory consumption, ensuring that the bucket group does not violate the GPU memory constraint. Buffalo significantly enhances the efficiency and feasibility of training large-scale graphs on individual GPUs.

Results. We evaluate Buffalo on a wide range of datasets with representative GNN models and compare it with state-of-the-art GNN systems (DGL [73], PyG [12], and Betty [93]). Our results show that Buffalo can successfully address the OOM issue from bucket explosion, compared to DGL and PyG. Buffalo achieves up to 70.9% speedup compared to Betty under the same memory budget. Besides, we demonstrate Buffalo's ability to enable the training of complex GNN models and large datasets: it achieves a 74.4% speedup for GNN models with an aggregation depth of 2 and a hidden size of 1,024 on the OBGN-arxiv dataset. Additionally, it attains a 72.5% speedup for models with an aggregation depth of 2 and hidden size 128 on the OGBN-products dataset. Notably, Buffalo also trains on the billion-scale OGBN-papers dataset in just tens of seconds per iteration with a single GPU, drastically cutting the time taken by state-of-the-art methods [40], [69], [93] that typically require minutes to tens of minutes.

II. Background and Preliminaries

A. GNN Memory Overhead

GNNs have demonstrated success in many traditional graph analytic tasks [3], [32], [62], [66], [80], [102]. Unlike DNNs, GNNs take graphs as inputs, consisting of nodes (entities with feature vectors) and edges (relationships between nodes). Each layer performs two major operations: *message-passing* (i.e., neighborhood aggregation) and *permutation-invariant function* (i.e., DNN compute).

Except the input features of large-scale graphs [5], [7], [93], message passing also can result in huge memory overload, especially with multi-hop aggregation and memory-intensive aggregators (e.g., LSTM). To see the problem, we analyze the memory consumption of a widely-used GNN model, GraphSAGE [16] using two million-scale datasets (OGBN-product and OGBN-arxiv [17]). We use a server with an NVIDIA RTX6000 GPU (24GB memory) and 192GB CPU memory. Section V has more details about the hardware

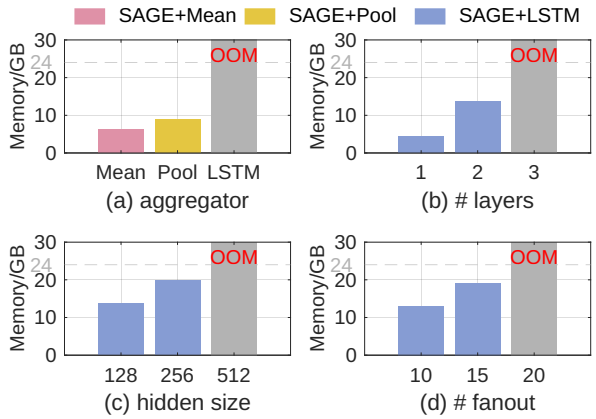

Fig. 2. Large-scale GNN training is memory intensive and can easily lead to OOMs. The figure shows the memory consumption of training GraphSAGE over OGBN-products and OGBN-arxiv.

setup. Figure 2 reveals that scaling GNNs easily hits the GPU memory capacity limit, resulting in OOMs with either (a) more advanced aggregators (e.g., from Pool to LSTM), (b) more aggregation depths (e.g., from 2-hop to 3-hop aggregation), (c) larger hidden dimensions (e.g., from 256 to 512), or with (d) a larger sampling rate (e.g., from 15 to 20). In general, memory capacity is a major bottleneck preventing scientists and practitioners from exploring more advanced GNN training.

B. GNN Framework

Multiple GNN frameworks have been created to mitigate the huge memory overhead of GNN training via distributed training (Table I). A common approach is the graph-level partitioning using METIS [25] to reduce communication between nodes, employed by DGL [101], PyG [12], AliGraph [91], and others [45], [47], [90], [97]. ParGNN [35], GLISP [104], and others [14], [22], [23], [71]) also use balanced workload partitioning and pipelining. Buffalo improves memory efficiency on individual GPUs via fine-grained bucketing-level partitioning and scheduling, and hence complements the existing efforts.

Existing works, such as BGL [40] and cuGraph [11], focus on reducing I/O and sampling overhead in GNN training. In contrast, our work addresses the memory bottleneck while maintaining computational efficiency. Unlike the complex disk-based design [69], both Betty [93] and Buffalo work for individual GPUs. Different from Betty, Buffalo solves the bucket explosion problem and enables online training.

C. Zero Padding via Degree Bucketing

Degree bucketing, aka *bucketization*, is a technique commonly employed by GNN frameworks (e.g., DGL [34] and DEMO-Net [81]) to make massage passing more efficiently. In particular, to exploit hardware efficiency, DNN operators often take a batch of inputs to execute in parallel. Since batched inputs require input shape to be identical, DNN operators often presume fixed-sized inputs. However, this assumption cannot always hold in GNN, as the size of GNN inputs can vary significantly. In GNN, each input comprises a group of nodes, and the input size reflects the number of neighboring nodes

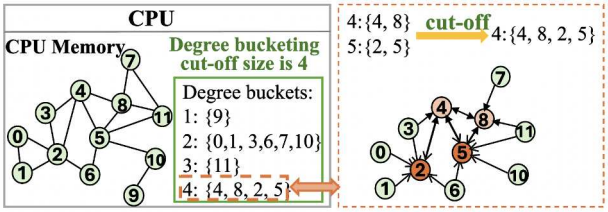

Fig. 3. An illustration of degree bucketing. The nodes with the degrees of 1-3 are grouped into three degree-buckets according to their degrees. The nodes with the degrees of 4 and 5 are grouped into another degree bucket. These four degree-buckets are fed to GNN for training.

Algorithm 1: Degree-bucketing based Training Iteration.

Input : A batch (i.e., a sampling subgraph): $\mathcal{G}$;
Aggregation depth (layer): L;
GNN model weights: W;
Output: Updated GNN weights: W'

1 $degree\_buckets \leftarrow \{\}$;
2 $blocks \leftarrow \{\}$;
3 **for** $l \leftarrow 1$ **to** L **do**
4 $\quad blocks[l] \leftarrow$ `BlockGenerate`$(\mathcal{G}, l)$;
5 $\quad degree\_buckets[l] \leftarrow$ `Bucketing`$(blocks[l], l)$;
$\quad$ **foreach** *bucket* $b \in degree\_buckets[l]$ **do**
6 $\quad\quad hidden_l \leftarrow$ Aggregate$(blocks[l], b)$;
7 $\quad\quad hidden_l \leftarrow$ Update$(hidden_l, W_l)$
8 $loss \leftarrow$ `Loss`$(hidden_L, labels)$;
9 Compute gradients w.r.t. model parameters W;
10 $W' \leftarrow$ Optimizer.step(), Update W;
11 **return** W'

connected to the group of nodes. A solution to address the above problem is to pad all nodes to match the maximum degree of neighbors. However, this brings in significant memory overhead and redundant computation on wasted padding.

To minimize the padding overhead, existing frameworks group nodes possessing identical degree into so-called *degree bucket*. The degree bucketing efficiently handles large, diverse graph datasets, and is crucial for memory-intensive aggregators (e.g., LSTM [16], transformer [36], [98], and MLP [18], [60]).

With degree bucketing, if a node's degree is less than F (i.e., the cut-off degree), then this node and other nodes with the same degree are grouped into the same bucket. The nodes with a larger degree than F are grouped together into a single bucket (called the degree-F bucket). The above bucketing strategy effectively groups nodes. F is usually determined by the user. Figure 3 illustrates the concept of degree bucketing.

Algorithm 1 depicts GNN training based on degree bucketing. At each training iteration, the algorithm processes a subgraph to produce a list of buckets for each GNN layer (or each aggregator layer). See Line 1. The training proceeds through L GNN layers (where L is the aggregation depth) at a layer-by-layer basis. At each layer, GNN performs message-passing to aggregate node-neighbor information, and updates node representations. Once all layers have been processed, the algorithm calculates the loss, back-propagates the gradients, and the optimizer states to update the GNN model parameters. The training iteration repeats until the GNN converges.

TABLE I
COMPARISON OF GNN TECHNIQUES.

Framework/ Technique	Type	Fine Granularity?	Dynamic Real-time Partition?	Traditional Method METIS-based?	Redundancy Aware?	Complex Design?	Convergence Complete Match?	Need Explicit Caching?
DistDGL [101]	Distributed	N	N	Y	N	Y	N	N
PyG [90] , AliGraph [91]	Distributed	N	N	Y	N	Y	N	Y
DistGNN [47]	Distributed	N	N	Y	N	Y	N	Y
NeuGraph [45]	Distributed	Y	N	N	N	Y	N	Y
SUGAR [90],ParGNN [35]	Distributed	N	N	Y	N	Y	N	N
CLISP [104]	Distributed	N	N	Y	Y	Y	N	N
G3 [71]	Distributed	Y	N	N	N	Y	N	N
BGL [40]	Distributed	Y	N	Y	N	Y	N	Y
Dask cuGraph [11]	Distributed	N	N	Y	N	Y	N	Y
Cugraph [11]	Individual GPUs	N/A	N/A	N/A	N/A	Y	N/A	Y
MariusGNN [69]	Individual GPUs	N	N	N	N	Y	N	Y
Betty [93]	Individual GPUs	N	N	Y	Y	N	Y	N
Buffalo	Individual GPUs	Y	Y	N	Y	N	Y	N

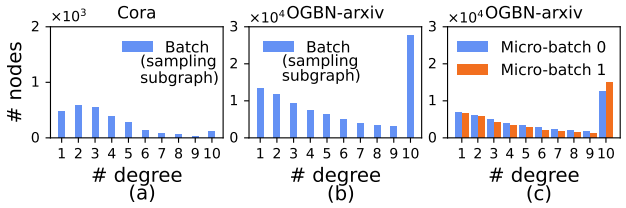

Fig. 4. The bucket-volume distribution across buckets.

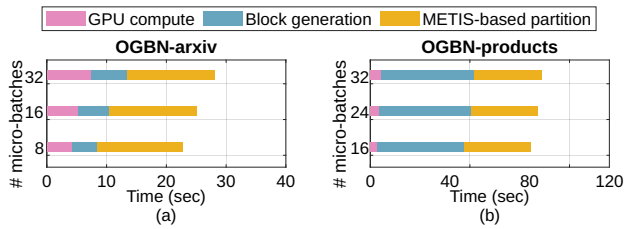

Fig. 5. Time comparison of different execution phases. Each horizontal bar shows the total time for partitioning, block generation, and GPU compute in one iteration. We use two datasets, OGBN-arxiv and OGBN-products.

III. MOTIVATION

We conduct preliminary analysis on the memory capacity challenges faced by GNN training to motivate our study.

Bucket explosion on large-scale graphs. When the size of a batch (i.e., sampling subgraph) is small, the max degree in a batch is typically limited, and the number of nodes in a bucket (i.e., the bucket volume) is relatively balanced across buckets. See the degree distribution of the Cora dataset in Figure 4.a as an example. However, as the graph scales up in size, the bucket volume becomes significantly skewed across buckets, leading to the so-called *bucket explosion problem*. Figure 4.b depicts the bucket explosion problem on the OGBN-arxiv dataset with F=10. Because of the long-tail distribution of degrees, nodes whose degrees are equal or higher than F are all gathered into the same bucket, leading to an explosion of the last bucket.

Memory-inefficiency from bucket explosion. Bucket explosion reduces memory efficiency because the explosion bucket not only has a high volume, but nodes within that bucket also have more neighbor embeddings involved in message passing which further increases GPU memory usage. The existing batch-level partitioning strategies cannot address this problem. For example, Figure 4.c shows that even after batch-level partitioning using a start-of-the-art solution Betty [93] on OGBN-arxiv, each micro-batch still suffers from the bucket explosion problem: the last bucket (the bucket 10) is much larger than any other bucket. The reason is that bucket explosion is tightly related to the long tail distribution of node degrees, whereas a batch-level partitioned subgraph still has long tail distribution, mitigating but not eliminating the bucket explosion problem. The bucket explosion can further introduce load imbalance across micro-batches, shown in Figure 4: the memory cost of micro-batch 1 exceeds that of micro-batch 0 by 20%. Such load imbalance largely comes from the imbalance between the last buckets in the two micro-batches.

Graph partitioning is time-consuming per training iteration. Graph partitioning usually takes tens of seconds to minutes to finish [40], [47], [58], [90], which makes it infeasible to be used for online training. Integrating the graph partitioning into the training process – essentially enabling online partitioning – brings the benefits of being adaptive to changes in the graph structure during training.

To study the overhead of graph partitioning, we apply the METIS-based partitioning to the subgraph in each training iteration. Figure 5 illustrates execution times of different execution phases (i.e., partitioning, block generation, and GPU compute) on two datasets. The results indicate that the METIS-based partitioning method requires significantly more time than GPU compute. For instance, partitioning OGBN-products takes 33.36 seconds, while GPU compute time for training takes only 3.43 seconds, shown in Figure 5.b.

Data preparation time is non-negligible for micro-batch generation. One challenge often missed in the existing work is the data preparation cost after partitioning. Data preparation is needed to create micro-batches. Such data preparation generates block representations for micro-batches [6]. To generate a block, we must track the neighbor dependencies of that block, which would be extremely time-consuming, given that the dependencies between nodes/edges are complicated in large graphs. Figure 5 shows that the block generation accounts for non-trivial amount of overhead (54.3%). As one increases the

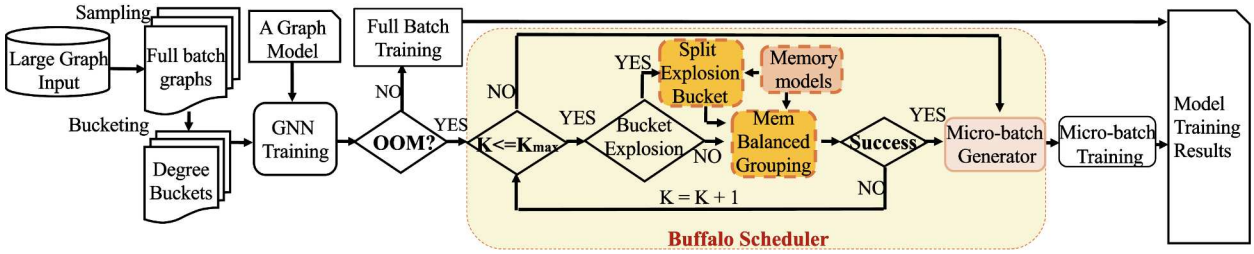

Fig. 6. Overview of Buffalo. The major components of Buffalo are included into a light-yellow box.

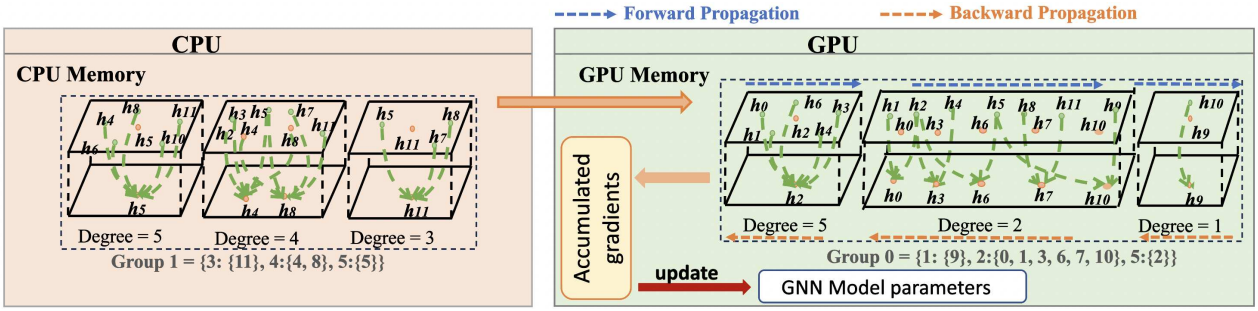

Fig. 7. Scheduling of bucket groups with Buffalo.

number of micro-batches, the overhead of block generation increases. Without reducing this overhead, the computing efficiency of batch-level partitioning is low.

IV. Design

A. Overview

Buffalo aims to enable large-scale GNN-training using a single GPU. To achieve this, Buffalo introduces an effective bucketization method, which provides fine-grained control of buckets to mitigate the memory-inefficiency issue caused by the bucket explosion problem. Figure 6 provides a design overview of Buffalo. Different from the normal GNN training with bucketing, Buffalo introduces two transformation operations at the bucket level: bucket *split* and *grouping*. The *split* operation partitions a bucket (i.e., a degree-bucket), e.g., the bucket that causes the bucket explosion problem, into smaller degree-buckets (called *micro-buckets*). The *grouping* operation allows one to combine micro-buckets and the non-split degree-buckets into *bucket groups*. Figure 7 shows how the graph in Figure 3, represented as {1: {9}, 2: {0, 1, 3, 6, 7, 10}, 3: {11}, 4: {4, 8}, 5: {2, 5}}, is partitioned into two bucket groups and how the two groups are scheduled during GNN training.

Each bucket group is then transformed into a GNN micro-batch, where Buffalo provides a highly efficient implementation to collect all dependent nodes/edges for a given bucket group to construct a micro-batch. To maximize the usage of GPU memory and minimize the number of bucket groups, we formulate the grouping problem into a knapsack problem, and introduce a greedy search algorithm to find the solution, which is lightweight and works surprisingly well in practice.

Additionally, we introduce analytical models for memory estimation. The models consider the impact of graph characteristics (e.g., the average clustering coefficient and the average node-degree of the graph) and node redundancy between bucket groups, and they help determine whether any grouping plan violates the GPU memory constraint. The analytical models are lightweight and accurate in comparison to profiling from actual GPU training.

B. Bucket-Level Partitioning

Challenge: Complexity of computation dependency. Since the GNN bucketing mechanism faces a bucket explosion issue, which we aim to mitigate this issue during GNN computation. While splitting the graph into independent micro-batches is straightforward for DNNs (due to the 1:1 correspondence of inputs and labels), scheduling buckets is complex(using "splitting" and "grouping" of buckets, depicted in Section IV-C1). This complexity arises from the neighborhood aggregation operation, which samples neighboring nodes, creating dependencies among partitioned buckets. Consequently, depending on at which layer we schedule the degree buckets, we have different schedule complexity. We use Figure 8 as an example to explain the above problem.

Figure 8.a shows the computation dependency in a multi-layer aggregation graph. Each degree-bucket at a layer depends on a list of buckets with varying degrees in its previous layer, to perform neighborhood aggregation. For example, the degree-5 bucket in Layer 1 has neighbors in the buckets of degree-1, degree-3, and degree-5 in Layer 0. This means that each node in the degree-5 bucket′ has 5 neighbors to perform neighborhood aggregation in Layer 1, such as the neighbor nodes v_1, v_2, v_3, v_4, and v_5. Node v_1 has only one neighbor involved in neighborhood aggregation in Layer 0. Nodes v_2 and v_3, each of which has three neighbors, are placed in the degree-3 bucket. Meanwhile, the nodes v_4 and v_5 both have 5 neighbors, hence the two nodes belong to the degree-5 bucket.

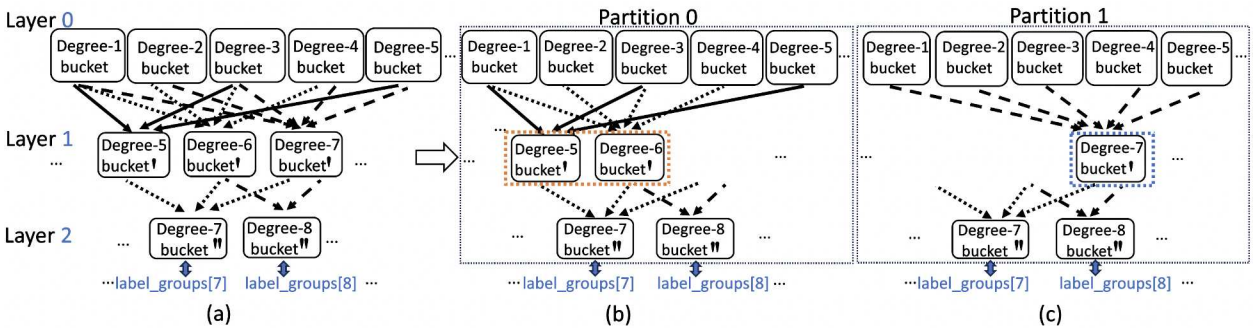

Fig. 8. An example of bucket partitioning at a non-output layer (Layer 1).

In the example of Figure 8, assuming that we aim to partition all degree buckets into two partitions, we must decide which layer to partition. If we partition buckets at the non-output layer, e.g., Layer 1, it can generate two partitions shown in Figure 8.b and Figure 8.c, respectively. With the Layer 1-based partitioning, in the partition 0, the degree-8 bucket$''$ in Layer 2 misses a dependency (i.e., the degree-7 bucket$'$ in Layer 1); in the partition 1, the degree-7 bucket$''$ in Layer 2 also misses dependencies (i.e., the degree-5 bucket$'$ and degree-6 bucket$'$ in Layer 1). These missing dependencies prevent GNN training from doing gradient accumulation and releasing activation memory.

Solution: Buffalo partitions the degree buckets at the output layer. The output layer relies on the information from the previous layer for its computations but does not serve as a dependency for any subsequent layers, as it is the final layer. As such, the loss calculation (Line 8) in Algorithm 1 can be done at a micro-batch level (i.e., a subset of output nodes and their dependencies). In the example of Figure 8, if we partition at Layer 2 (the output layer), each partition no longer has dependencies on other partitions, and all activation memory associated with one partition can be released after gradient calculation has been done. Hence, the partition at Layer 2 is better than at Layer 1.

In Buffalo, we partition the hierarchical aggregation graph into K subgraphs, where the nodes at the output layer of these subgraphs are disjoint sets. With this partition, we extend the traditional degree-bucketing algorithm (Algorithm 1). The major extension is highlighted in blue in Algorithm 2. With the extended algorithm, after bucket groups are generated by the Buffalo Scheduler (Section IV-C), Buffalo loads individual bucket groups from the host memory to the GPU memory, and gradients are generated after each bucket groups is processed. The gradients then get accumulated across consecutive bucket groups; the GNN parameters are updated once *all* bucket groups have been processed. This method makes the Buffalo training as effective as the original training because it does not impact the training convergence. In the next subsection, we describe how the Buffalo Scheduler works.

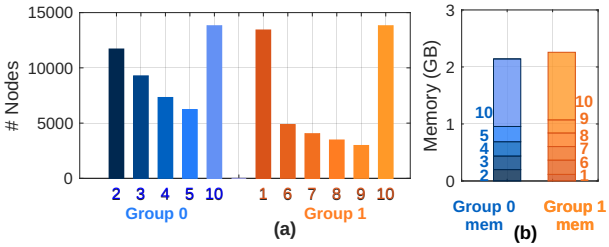

Fig. 9. The schedule of degree buckets for Figure 4(a).

Algorithm 2: Degree-bucketing based Training with Buffalo. The algorithm extension is highlighted in blue.

Input : A batch (i.e., a sampling subgraph): $\mathcal{G}$;
The aggregation depth: L;
Memory constraint: M_{ctr};
GNN model weights: W;
Output: Updated GNN weights: W'

1 $\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_K \leftarrow$ **Buffalo_Scheduler**$(\mathcal{G}, L, M_{ctr})$;
2 $partitioned\_degree\_buckets \leftarrow \{\{\}\}$;
3 **for** $k \leftarrow 1$ **to** K **do**
4 **for** $l \leftarrow 1$ **to** L **do**
5 $block_{kl} \leftarrow$ **BlockGenerate** $(\mathcal{G}_k, l)$;
6 $partitioned\_degree\_buckets[k][l] \leftarrow$ **Bucketing**$(block_{kl}, l)$;
7 **foreach** *bucket* $b \in$ $partitioned\_degree\_buckets[k][l]$ **do**
8 $hidden_{kl} \leftarrow$ Aggregate$(block_{kl}, b)$;
9 $hidden_{kl} \leftarrow$ Update$(block_{kl}, W_l)$;
10 $partial\_loss \leftarrow$ Loss$(hidden_{kL}, label_groups[k])$;
11 Backward pass, compute gradients w.r.t.model parameters;
12 AccumulatePartialGradients$(\mathcal{M})$;
13 $W' \leftarrow$ Optimizer.step(), Update W;
14 **return** W'

C. Buffalo Scheduling

1) Background: Scheduling a degree-bucket list into bucket groups performs two operations. (1) Splitting degree buckets into micro-buckets. When Buffalo performs this operation, each micro-bucket must be small enough to fit into the GPU memory. However, they must not be too small to fully utilize GPU resources (including GPU memory and thread-level parallelism). (2) Grouping micro-buckets and non-split degree-buckets. The grouping results are bucket groups. Based on the bucket groups, we build micro-batches, each including

Algorithm 3: Buffalo Scheduler

Input : A batch (i.e., a sampling subgraph): $\mathcal{G}$;
The aggregation depth: L;
Memory constraint: M_{ctr};
Sampling size (cut-off degree): F;
Output: A list of subgraphs corresponding to K micro-batches: $\mathcal{G}_1$, $\mathcal{G}_2$,...,$\mathcal{G}_K$;

1 $degree\_buckets_L \leftarrow$ **DegreeBucketing**($\mathcal{G}$, L) ;
2 $K \leftarrow 1$;
3 **while** $K \leq K_{max}$ **do**
4 **if** *bucket explosion detected* **then**
5 $degree\_buckets_L[F : F+K-1] \leftarrow$ **SplitExplosionBucket**($degree\_buckets_L[F]$, K) ;
6 $success$, $\mathcal{B}_1$, $\mathcal{B}_2$,...,$\mathcal{B}_K \leftarrow$ **MemBalancedGrouping**($degree\_buckets_L$, K, M_{ctr}, F) ;
7 **if** $success$ **then**
8 break;
9 **else**
10 K ← K + 1;
11 $\mathcal{G}_1$,...,$\mathcal{G}_K \leftarrow$ **MicroBatchGenerator**($\mathcal{B}_1$, $\mathcal{B}_2$,...,$\mathcal{B}_K$) ;
12 **return** $\mathcal{G}_1$,$\mathcal{G}_2$,...,$\mathcal{G}_K$

Algorithm 4: MemBalancedGrouping

Input : Degree bucket list of the output layer: $degree\_buckets_L$;
Bucket group size: K;
Memory constraint: M_{ctr};
Sampling size (cut-off degree): F;
Output: success or fail;
Bucket groups: $\mathcal{B}_1$, $\mathcal{B}_2$,...,$\mathcal{B}_K$;

1 $\mathcal{B}_1$, $\mathcal{B}_2$,...,$\mathcal{B}_K \leftarrow \{\}$;
2 $M_{est} \leftarrow$ **BucketMemEstimator**($degree\_buckets_L$);
3 $M_{\text{est\_sorted}} \leftarrow$ sort(M_{est}, descending);
4 **while** $M_{est\_sorted} \neq \emptyset$ **do**
5 $cur\_bucket \leftarrow M_{\text{est\_sorted}}$.pop();
6 M_1, M_2,...,$M_K \leftarrow$ **RedundancyAwareMemEstimator**($\mathcal{B}_1$, $\mathcal{B}_2$,...,$\mathcal{B}_K$) ;
7 find the bucket group $\mathcal{B}_k$ with the lowest memory estimation;
8 $\mathcal{B}_k$.add(cur_bucket);
9 **end**
10 M_1, M_2,...,$M_K \leftarrow$ **RedundancyAwareMemEstimator**($\mathcal{B}_1$, $\mathcal{B}_2$,...,$\mathcal{B}_K$) ;
11 **if** any $M_k > M_{ctr}$ **then**
12 **return** $fail$, _
13 **else**
14 **return** $success$, $\mathcal{B}_1$, $\mathcal{B}_2$,...,$\mathcal{B}_K$
15 **end**

a bucket group plus the group's node dependencies.

Figure 9 gives an example to show the results after scheduling (using splitting and grouping). These results correspond to the graph (the dataset OGBN-arxiv) shown in Figure 4.b. In this example (shown in Figure 9.a), the degree-10 bucket is split; the non-split degree-{2,3,4,5} buckets, along with the first micro-bucket of the split degree-10 bucket, form group 0, while the non-split degree-{1,6,7,8,9} buckets, along with the second micro-bucket of the split degree-10 bucket, constitute group 1. After the scheduling, each group generates one micro-batch. Figure 9.b shows the corresponding memory cost of the micro-batches of the two groups.

To maximize the benefits of bucket scheduling, we must consider the possibility of grouping the micro-buckets with non-split degree-buckets. However, the solution space for grouping the micro-buckets and degree buckets is large, making it challenging to find an optimal solution. Assuming K_{MB} is the number of micro-buckets and K_{DB} is the number of non-split buckets, there are $K_{\text{MB}} \cdot K_{\text{DB}}!/(K_{\text{DB}} - K_{\text{MB}})!$ possible solutions in this space. We introduce an algorithm for the splitting and grouping operations.

2) Scheduling Algorithm: The algorithm accepts three input variables: a subgraph representing a training batch, the aggregation depth, and a GPU memory constraint, as shown in Algorithm 3. The algorithm output is K micro-batches. Each micro-batch respects the constraint of GPU memory capacity. The memory consumption is balanced across the micro-batches to avoid the waste of GPU memory. The algorithm also minimizes K in order to reduce the overhead of data preparation and loading.

The scheduling algorithm is generally depicted in Algorithm 3. Following the degree bucketization process (Line 1 in Algorithm 3), which generates a bucket list for the output layer, Buffalo uses a bucket-group plan generator, shown in Lines 4—6 in Algorithm 3, to generate a list of bucket groups $B_1, B_2, .., B_K$. The group generation includes splitting (Line 5) and grouping (Line 6), which are described in more details below. Once the bucket group plan is generated, Buffalo uses a `RedundancyAwareMemEstimator` (more details in Section IV-D) to estimate the memory consumption of each bucket group ($M_1, M_2, ..., M_K$). If any bucket group's memory consumption exceeds the memory capacity, Buffalo increments K by one and repeats the process until a valid set of K subgraphs is found. In the particular case, where the total estimated memory of the entire buckets (e.g., $K = 1$) is less than the memory constraint, we do not do anything and treat the original subgraph ($\mathcal{G}$) as the micro-batch.

SplitExplosionBucket (Line 5 in Algorithm 3) evenly splits a degree bucket into micro-buckets when a degree bucket faces the bucket explosion problem. As a result, each micro-bucket has roughly the same number of output nodes after splitting.

In addition to splitting the explosion bucket into micro-buckets, Buffalo also supports flexibly regrouping micro-buckets and non-split buckets to form bucket groups. Specifically, Buffalo formulates the degree-bucket grouping into a load-balanced bin packing problem [67] and utilizes a **MemBalancedGrouping** algorithm (Line 6 in Algorithm 3) to group micro-buckets and non-split buckets into memory-balanced bucket groups. The load-balanced bin packing problem consists of packing items of different sizes into a finite number of bins, achieving well-balanced packing while minimizing the number of used bins.

In our approach, we treat each bucket as an item and have its value equal to its weight. Both the weight and value of the item equate to the estimated memory of the corresponding bucket. This is intuitive because the goal here is to ensure that the cumulative weight of the items, i.e., the memory consumption of a group of buckets, does not surpass the bin's capacity, i.e., the GPU memory constraint. The load needs to be balanced so

there is no waste of GPU memory, and we reduce the overhead of data preparation and loading by minimizing the number of bins (i.e., bucket groups).

we introduce a greedy approximation method to solve the bucket grouping. At the beginning, we sort the buckets, including both micro-buckets and non-split buckets, in descending order by their memory estimation from *BucketMemEstimator()* (Line 3 in Algorithm 4). Then, starting from the bucket with the largest memory estimation, the algorithm iterates through the sorted micro-buckets and non-split buckets and places the bucket in the bucket group that has the lowest memory estimation so far. The memory estimation for the bucket group is provided by *RedundancyAwareMemEstimator()*, which we discuss in more details in the next section. This method ensures that the most valuable item (the largest bucket) is considered first, in order to accelerate the solution-finding process. During the iteration, Buffalo accumulates the memory consumption of the buckets. If any of the bucket groups has an accumulated memory size larger than the GPU memory constraint, the grouping algorithm returns a fail status, where a larger K will be tested by the Buffalo scheduler in the next iteration. Otherwise, Buffalo returns a list of bucket groups using the accumulated buckets (Line 14 in Algorithm 4).

D. Memory Estimation

Challenges of memory estimation. Existing memory estimation techniques, like the one in [93], can reasonably estimate the working memory of individual buckets during GNN training. However, estimating the overall GNN memory consumption is complex because simply summing the memory estimates of individual buckets does not accurately predict the actual consumption of a bucket group, which we call the *non-linear relationship*. The non-linear relationship incurs major challenges for Buffalo's grouping algorithm. If the memory estimation of a bucket group is inaccurate, then a proposed bucket-group partition plan may still lead to OOM or under-utilized GPU memory during actual training.

The core reason for this non-linear relationship in GNN memory consumption is because of node redundancy across micro-batches built from bucket groups. When a micro-batch i loads target nodes and their neighbors into GPU memory, some of these neighbors may be required by another micro-batch j, which creates duplication for memory estimation when the micro-batch j loads neighbors. Such duplication happens more often in the L-hop aggregation layer than (L-1)-hop layer because neighbors tend to grow exponentially as the hop increases. This redundancy problem commonly exists in many GNN training methods [93], [104].

To illustrate this non-linear relationship, we train a GraphSAGE model with an LSTM aggregator and an aggregation depth 2 on the OGBN-arxiv dataset. The cut-off degree (F degree) for the 1-hop neighbors and 2-hop neighbors are 25 and 10 respectively. The hidden size of the LSTM aggregator is 128. Without bucket splitting and grouping, the memory consumption of a batch is 13.68GB. When splitting the batch across the 1-hop and 2-hop buckets to generate two micro-batches, the overall memory consumption of each micro-batch is 8.57GB and 10.95GB, which is 25% and 60% larger than half of the memory consumption of the original batch (6.84GB). This example highlights how node redundancy across micro-batches leads to a non-linear increase in memory consumption. We propose a lightweight memory-estimation method that accounts for redundancy.

Quantifying redundancy is the core of the memory estimation, represented by a redundancy-aware grouping ratio R_{group} in Buffalo. R_{group} quantifies the impacts of node connections in a bucket group on the memory consumption of a micro-batch after grouping. The grouping ratio R_{group} is calculated by considering how input nodes are related to output nodes, how many connections each node has within a bucket, and how interconnected the nodes are within the buckets. Hence, R_{group} is related to the number of input nodes (I), the number of output nodes (O), the neighbor degree of the bucket (D), and the average clustering coefficient (C, a metric to quantify the node clustering of the input graph, which is discussed later). A higher R_{group} indicates a closer to the linear relationship of memory consumption when merging two buckets, and vice versa.A larger O, D, or C leads to a lower grouping ratio. In contrast, a larger I leads to a higher grouping ratio. The grouping ratio should be at most 1 (which means a perfect linear relationship).

Based on the above discussion, we define the grouping ratio for a given bucket i in Equation 1.

$$R_{\text{group}}[i] = min(1, \frac{I_i}{O_i \times D_i \times C}) \tag{1}$$

Given n entities (including micro-buckets and non-split buckets) for grouping and their degrees ($\{i_1, i_2, .., i_n\} \in \{1, 2, ..., F, F+1, ..., F+K\}$), the memory estimation of a bucket group is calculated with the following equation.

$$\sum_{i=i_1}^{i_n} M_{\text{est}}[i] \times R_{\text{group}}[i] \tag{2}$$

Note that when $R_{group}[i]$ equals 1, the memory consumption of a bucket group becomes the linear addition of the memory estimation of individual buckets. However, given that $R_{group}[i]$ is most likely less than 1, the memory consumption of a bucket group is usually much smaller than the sum of individual buckets in that group. Next, we discuss why I, O, D, and C are related to the redundancy ratio as follows.

About I and O. A higher ratio of input nodes (I) to output nodes (O) within a bucket indicates a more significant number of input nodes associated with the output nodes in the bucket. Hence, the bucket tends to have more overlap with other buckets in terms of nodes, leading to high redundancy.

About D. The redundancy ratio is divided by $degree$ (D). The rationale behind this is as follows: D is related to node dependency. Grouping buckets introduces node redundancy because of the existence of node dependencies. D quantifies the potential of the node redundancy.

TABLE II
THE INFORMATION FOR THE TRAINING DATASETS AND THEIR CHARACTERISTICS.

Dataset	Feat. Dim.	Nodes	Edges	Avg. Deg.	Avg. Coef.	Power Law
Cora	1433	2.7K	10K	3.9	0.24	✗
Pubmed	500	19K	88K	8.9	0.06	✗
Reddit	602	0.2M	114.6M	492	0.579	✓
OGBN-arxiv	128	0.16M	2.31M	13.7	0.226	✓
OGBN-products	100	2.45M	61.86M	50.5	0.411	✓
OGBN_papers	128	111.1M	1.6B	29.1	0.085	✓

About C. C is a term borrowed from the graph theory and used to quantify how closely connected a node's neighbors are, which reflects the tendency of nodes to form clusters. A higher C signifies a well-connected graph with many clusters, while a lower C suggests a more sparsely connected and less clustered graph. In the context of a degree bucket with the degree of cut-off, especially in cases where there is bucket explosion, C of the bucket that stores nodes with the degree of cut-off closely resembles that of the original input graph. This observation suggests that as the subgraph grows large enough, C becomes a reasonable representation of the entire graph's characteristics. Hence, C is an essential parameter to estimate the actual number of input nodes after grouping buckets. To get the actual number of input nodes in the input layer, we need to eliminate the redundancy within the input layer. Hence, we divide C to estimate the grouping ratio.

I, O and D can be obtained during the micro-batch generation. C is a graph characteristic and can be obtained by offline graph analysis, utilizing the equation presented on page 142 of reference [27]. Hence, obtaining I, O, D, and C do not bring any computation overhead. Our method is general and applicable to any aggregators in GNN.

E. Data Preparation

To reduce the data preparation time required for micro-batch generation from the subgraph (the subgraph is generated by *sampling* of the original graph as in the traditional approaches), we accelerate block generation using two techniques. (1) We track all neighbors of the center nodes in the subgraph following the sampling order, hence avoiding repeated connection check in the subgraph for block generation. (2) Tracking neighbors happens in parallel at the node level instead of micro-batch level, hence offering more parallelism. In particular, before block generation, we use a Compressed Sparse Row (CSR) matrix to represent the adjacency relationships of a subgraph after sampling. For each CSR row, neighbor sampling for the center nodes is performed in parallel, which involves concurrent operations to gather the required neighbors for some center nodes in a micro-batch. As the CSR matrices provide efficient row accesses and parallel processing allows simultaneous sampling of neighbors, we significantly reduce the overall time needed for data preparation.

V. EVALUATION

Implementation. We implement Buffalo in DGL [73], which is an open-source state-of-the-art GNN library. We choose DGL due to its excellent performance from various optimizations against single-GPU efficiency, such as fused message passing kernels, shared-memory graph store, and many more [73]. DGL shows superior performance compared to other frameworks. For example, for a 2-layer GCN model with the dataset Reddit on an NVIDIA P100 GPU, the training throughput of DGL is 2x better than PyG [10]. Hence, DGL provides a high bar for studying performance and memory saving.

Platform. We use a machine equipped with two Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz, RAM 192GB, each with 24 cores. The machine has an NVIDIA Quadro RTX 6000 GPU with 24GB memory. We use another machine equipped with two Intel(R) Xeon(R) Platinum 8380 CPU @ 2.30GHz, RAM 512GB, each with 24 cores. This machine has an NVIDIA A100GPU with 80GB memory. We use CUDA 12.1, cuDNN 8.9, Python 3.10, and PyTorch [52] 2.1.0.

Workloads. We use five datasets from DGL benchmarks [73], each representing a distinct input graph. Table II summarizes these datasets. Due to disk space limitations, we are unable to test the IGB [29] dataset. We employ GNN models GraphSAGE [16] and GAT [68], which are commonly used in existing work [39], [46], [51], [99].

We choose shallow GNNs (2-5 layers) from DGL benchmarks [73] because they are commonly used in both research and production to avoid issues like over-smoothing in deeper models, as highlighted by recent studies and practices at companies like Pinterest and Twitter [9], [59], [94]–[96].

Baseline. We use DGL, PyG, and Betty for performance comparison. We choose DGL and PyG because they are widely used GNN training frameworks. We choose Betty because it achieves memory-efficient training of GNNs on a single GPU. We also look into cuGraph [11]. Although cuGraph is an effective tool for accelerating GNN training, we cannot use it as a baseline or to speed up our data preprocessing because of the following reasons: First, cuGraph's `cuGraphSAGEConv` is limited to basic aggregation functions, including mean, sum, min, and max, and does not accommodate memory-intensive advanced aggregators like LSTM, Meta [24], [33] . However, these advanced aggregators are necessary for developing advanced GNN models. Second, these large-scale graphs formatted in cuGraph for GNN training cannot be accommodated by individual GPUs.

A. Improvements of Compute-vs-Memory Efficiency

We evaluate compute-vs-memory efficiency. We control the availability of GPU memory and measure the end-to-end time per training iteration. The time includes data preparation time, data transfer time (from CPU memory to GPU memory), and training time on GPU. Figure 10 shows the results. The figure shows that Buffalo excels DGL, PyG, obtaining better Pareto frontier in terms of compute-vs-memory efficiency. While DGL and PyG achieve lower execution time on the small

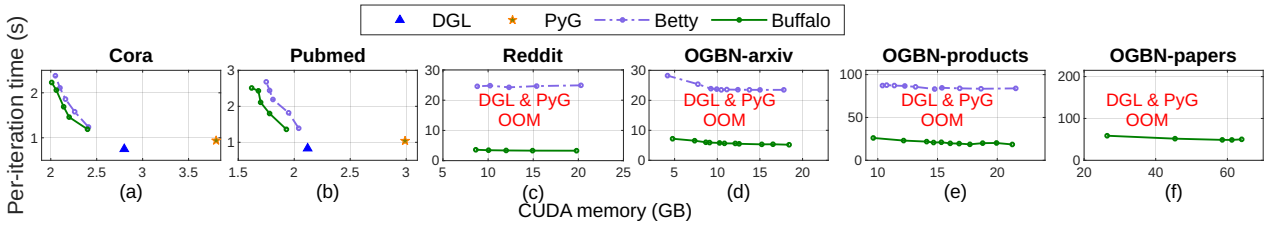

Fig. 10. Training time and CUDA memory cost with varying numbers of micro-batches for GraphSAGE.

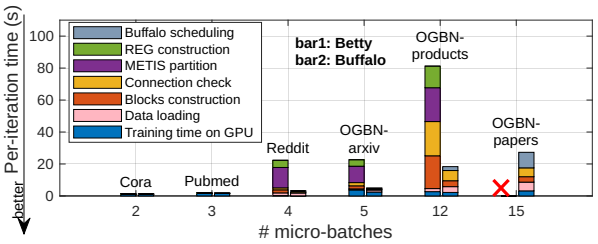

Fig. 11. Training time variance when we use different numbers of batches. We use GraphSAGE with the aggregator LSTM and all datasets.

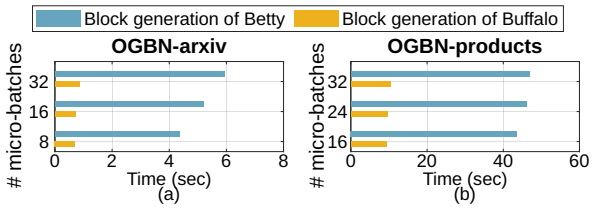

Fig. 12. Time comparison between block generation time of Buffalo and Betty. Each horizontal bar represents the block generation in one iteration. We use two datasets, OGBN-arxiv, and OGBN-products.

dataset Cora, they both suffer from OOM when training on the large data sets Reddit, OGBN-arxiv, and OGBN-products on tested models and configs. In contrast, Betty and Buffalo do not have the OOM issue. Compared with Betty, Buffalo reduces the iteration time by 70.9% on average. This large performance benefit comes from Buffalo's bucket-level splitting and grouping mechanism with various optimizations, which effectively mitigate the bucket explosion problem on large datasets while achieving better compute-vs-memory efficiency without incurring the expensive graph partitioning overhead.

B. Execution Time Breakdown

Figure 11 shows how different components contribute to the end-to-end training time of Betty and Buffalo across various datasets and micro-batch sizes. We report the end-to-end time per epoch, which includes the following components: (1) Buffalo scheduling: the execution time of Buffalo scheduler, (2) REG construction: Betty's process for embedding node redundancy information into the graph, (3) METIS partition: Betty's graph partitioning a given graph (REG) using METIS [25], (4) connection check: tracking dependencies between nodes and these nodes' neighbors, (5) block construction: generating blocks based on the connection check, (6) data loading: the transfer time of data from CPU to GPU, and (7) training time on GPU: the time spent on forward, backward, and step functions. We do not report the time for DGL and PyG, as they do not use graph partition on individual GPUs.

We make several observations. First, as the graph size increases, the training per iteration time also increases, as larger graphs often have bigger batch sizes. On average, Buffalo reduces the end-to-end training time by 70.9%, compared with Betty. The improvement mostly comes from avoiding the expensive graph partitioning, e.g., REG construction and `METIS partition`, which takes 46.8% of the end-to-end training time on average in Betty. In contrast, Buffalo does not have this time cost, and Buffalo scheduling only incurs minimal overhead. Other improvement comes from the optimization of block generation implemented with C++ (see Section V-C).

Second, as graph size increases, we see an interesting shift of performance bottlenecks. For smaller graphs like Cora and Pubmed, the GPU compute time dominates overall training time. However, for larger graphs such as Reddit and OGBN-arxiv, graph partitioning overhead becomes the main contributor to execution time. Here, Buffalo significantly reduces execution time compared to Betty by avoiding costly graph partitioning. As the graph moves to a million scale, e.g., OGBN-products, the overhead from connection checks and block construction rises sharply. This is because Betty must partition a batch into more micro-batches in order to fit each micro-batch to the device's memory. Additionally, Betty does not support block generation for billion-scale OGBN-papers because Betty cannot process nodes with zero in-edges (shown as no data in Figure 11 for OGBN-papers).

C. Optimization of Block Generation

We compare the block generation performance of Buffalo and Betty. Figure 12 shows that Buffalo takes significantly less time than Betty for block generation, up to 8 times faster. For instance, Betty requires 5.21 seconds to generate blocks for OGBN-arxiv to produce 16 micro-batches, while Buffalo takes only 0.70 seconds, as illustrated in Figure 12.b.

D. Reduction of Peak Memory Consumption

We re-evaluate the cases presented in Figure 2 using Buffalo. Figure 13 shows the results. With Buffalo, we successfully address all OOM issues. Shown in Figure 13.a, Buffalo enables a sophisticated aggregator (LSTM) using fifteen micro-batches. Additionally, Buffalo allows us to execute GNN models with more layers (3 and 4 layers, using 2 and 5 micro-batches, respectively). Moreover, Buffalo allows us to

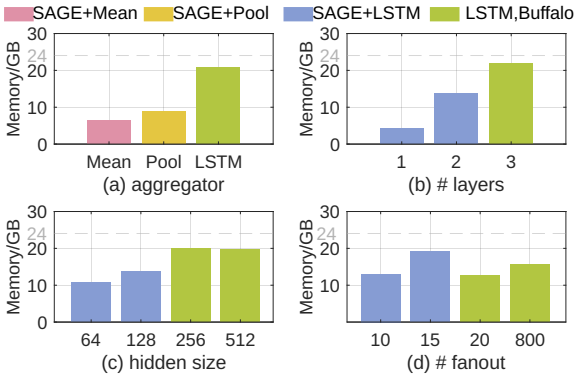

Fig. 13. Buffalo breaks the memory capacity wall shown in Figure 2. This figure uses the same configurations as Figure 2.

execute GNN models with larger hidden sizes, such as 512, using 2 micro-batches. Notably, we achieve this while also increasing the fanout to 20 and 800 using 2 and 13 micro-batches, respectively. This demonstrate the effectiveness of Buffalo in overcoming the memory capacity wall.

E. Load Balance after Applying Buffalo

We measure the memory consumption of each micro-batch after using Buffalo. Figure 14 shows the result for three datasets — OGBN-arxiv (split into 4 micro-batches), OGBN-products (12 micro-batches), and OGBN-papers (8 micro-batches). For each dataset, the memory consumption of micro-batches is almost the same. The difference in memory consumption across micro-batches is only 4%-6%.

F. Sensitivity Analysis for Memory Budget and Bucket Size

We examine how the memory budget is related to the bucket group size (i.e., the number of output nodes in a bucket group). To assess performance with various bucket group sizes, we utilize four different GPU memory budgets: 16GB, 24GB, 48GB, and 80GB, along with OGBN-products. We perform our analysis using a 2-layer GraphSAGE model with an LSTM aggregator. Figure 15 shows the performance of this model with different GPU memory budgets and bucket group sizes. We use NVIDIA GPU A100. The data points labeled 2, 4, 12, and 18 in the figure indicate the number of micro-batches generated. Given a GPU memory budget of 80GB, the training memory cost is 76.65GB, and the end-to-end time is 9.37 seconds. In general, as the GPU memory budget increases, the size of the bucket group increases, resulting in shorter training times. This implies that the GNN operates more efficiently when more GPU memory is available.

G. Performance on Multi-GPU

To employ Buffalo on multiple GPUs, we create micro-batches using Buffalo scheduling with the consideration of GPU memory capacity on each GPU. Then, we use data parallelism to train GNN. We repeat the evaluation in Figure 15 using two A100 GPUs in a machine connected by PCIe instead of one GPU. The CUDA memory budget per GPU is still 16GB, 24GB, 48GB, and 80GB. With two GPUs, the

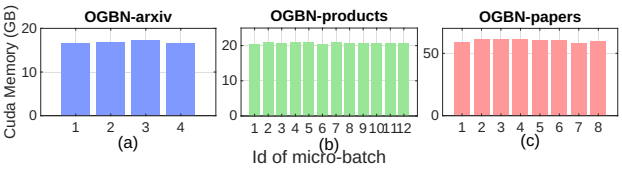

Fig. 14. Memory consumption of micro-batches after applying Buffalo.

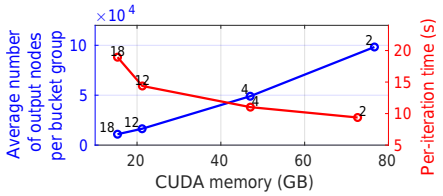

Fig. 15. Bucket group size v.s. memory budget.

iteration times are reduced slightly by 3%-5%. This happens because each iteration includes both the time to generate micro-batches and the time for training. The training time is making up only 9%-12% of the total time. While multiple GPUs can speed up training, the time for generating micro-batches stays the same. Additionally, the extra communication time between the GPUs adds 0.9%-1.2% to the total time, leading to a small overall reduction in iteration time.

H. Computation Efficiency

We evaluate the computation efficiency of Buffalo. The computation efficiency is defined as the total number of nodes across all micro-batches divided by the end-to-end training time per training iteration. In this evaluation, we also compare Buffalo with several alternative batch-level partitioning strategies: Random, Range, METIS, and Betty. These four partition strategies are applied on the subgraph only contains output nodes. The Range partition method sequentially and evenly splits the 1D space of output nodes, whereas the Random partition method does so evenly but randomly. For example, given the indices of output node {10, 35, 46, 79, 105, 123, 254, 328}, when split into two partitions, the Range partition results in {10, 35, 46, 79} and {105, 123, 254, 328}. In contrast, the Random partition might yield {328, 79, 35, 123} and {254, 105, 10, 46}. Buffalo completes training using 12 micro-batches(Figure 14.b), while the Random and Range requires 14 micro-batches. The Random and Range methods don't effectively reduce redundancy, which leads to larger partition sizes than Buffalo. METIS divides output nodes via partitioning the graph. , while Betty first transforms the graph to a new graph including node-redundancy information before using METIS. Figure 16 shows that computation efficiency remains stable for Range, Random, METIS, and Betty as the number of micro-batches increases. However, Buffalo outperforms the best of these by 36.4%, indicating Buffalo can handle more node computations per epoch.

I. Evaluation of Memory Estimation

Table III shows the memory estimation error of Buffalo on all datasets, using GraphSAGE with LSTM and mean aggregator. GraphSAGE has a hidden size of 256 and 2 layers.

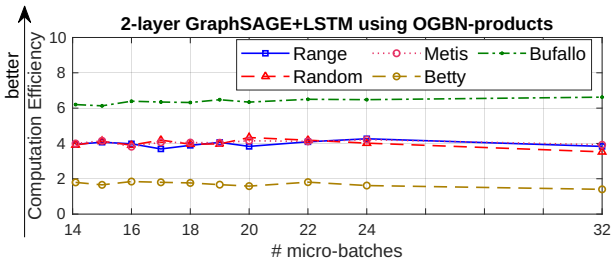

Fig. 16. Evaluation of computation efficiency.

TABLE III
MEMORY ESTIMATION ERROR FOR LSTM AND MEAN AGGREGATOR.

Dataset	Cut-off	LSTM		mean	
		# batch	Error rate %	# batch	Error rate%
Cora	10,25	4	4.3	4	10.02
Pubmed	10,25	4	5.7	4	5.16
Reddit	10,25	4	0.16	4	0.29
OGBN-arxiv	10,25	4	0.5	4	1.82
OGBN-products	10,25	16	7.6	8	0.34
OGBN-papers	10,25	16	4.1	8	3.6

TABLE IV
TRAINING WITH DGL V.S. TRAINING WITH BUFFALO

Dataset	Model	DGL / Loss	Buffalo/ Loss
Cora	SAGE	0.0017 ± 0.0010	0.0018 ± 0.0011
	GAT	1.8931 ± 0.0100	1.9005 ± 0.0097
Pubmed	SAGE	0.0003 ± 0.0001	0.0003 ± 0.0001
	GAT	1.0916 ± 0.0067	1.0911 ± 0.0045
Reddit	SAGE	OOM	0.2107 ± 0.0044
	GAT	OOM	2.7091 ± 0.0027
OGBN-arxiv	SAGE	0.9786 ± 0.0043	0.9691 ± 0.0039
	GAT	OOM	3.0569 ± 0.0010
OGBN-products	SAGE	OOM	0.3519 ± 0.0088
OGBN-papers	SAGE	OOM	1.4548 ± 0.0036

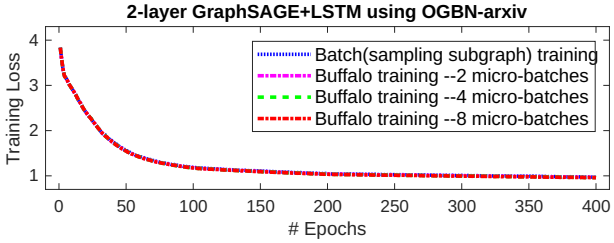

Fig. 17. Convergence curves for batch(sampling subgraph) training and micro-batch training using Buffalo.

Our estimation exhibits a low error rate: in all cases, the rate is below 10.02%.

J. Training Convergence and Loss

We examine model convergence with Buffalo. Figure 17 presents the convergence curve for GraphSAGE on OGBN-arxiv, comparing batch and micro-batch training across three batch sizes with identical hyperparameters. The curves are closely aligned, indicating that the model convergence is not impacted by Buffalo and micro-batch training. Using all other datasets, we see the same trend.

Table IV presents the training loss of DGL with batch (sampling subgraph) training and Buffalo using micro-batch training. Overall, the training loss of Buffalo is almost the same as that of DGL. This is because Buffalo only changes the schedule of training at the micro-batch level and the micro-batch training is mathematically equivalent to the batch (sampling subgraph) training.

VI. RELATED WORK

Joint optimization for graph data and operations. GNNAdvisor [77] focuses on optimizing data patterns from the graph structure, utilizing both the graph's inherent structure and model-specific information simultaneously to enhance performance. WiseGraph [19] is designed to optimize both the partitioning of graph data and GNN operations concurrently. ByteGNN [100] focuses on locality-aware partitioning and partial code execution to minimize data movement and copying overhead. GraphPart [44] emphasizes the selection of representative nodes within graph partitions for active learning. Kim et al. [31] leverage a performance model to guide data division and introduce locality-aware neighbor sampling to reduce data movement during training. Song et al. [63] introduce a locality-aware neighbor sampling technique to further minimize data movement overhead.

Data-preprocessing for graph analysis. There are many data-preprocessing techniques, including [1], [8], [30], [40], [65], [89], [97]. Our approach differs from them because of the fine-grained partitioning method to address the bucket explosion problem.

Load balance for graph analysis. There are many load-balance efforts for graph analysis, including [13], [41], [48], [49], [61], [64], [72]. However, those efforts focus on the balance of number of nodes across partitions, not the balance of memory consumption like Buffalo. Hence, Buffalo makes more effective usage of GPU memory.

Memory tiering. Memory tiering [20], [37], [38], [43], [53]–[58], [75], [82], [83], [85]–[88], [92] combines multiple memory components to address memory capacity problems. Buffalo is a solution to leverage tiered memory.

VII. CONCLUSION

In this paper, we reveal a bucket explosion problem that leads to OOM and limits the effectiveness of existing methods to train GNN on single GPUs. We introduce Buffalo, a system addressing the bucket explosion and enabling load balancing between graph partitions for GNN training. Our comprehensive evaluation demonstrates that Buffalo significantly improves compute-vs-memory efficiency, successfully mitigates out-of-memory challenges, reduces end-to-end training time by 70.9%, and outperforms existing methods by 36.4% in terms of computation efficiency, advancing the capability of GNN training on a single GPU.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This work was partially supported by U.S. National Science Foundation (2104116 and 2316202) and the Chameleon Cloud.

APPENDIX

A. Abstract

This artifact includes source codes and expected experimental data to replicate the evaluations in this paper.

We used Figure 2 to denote the OOM situation of the current advanced GNN training, and Figure 13 to illustrate how Buffalo breaks the memory wall. Figure 6 presents our estimate of memory consumption during the workflow of Buffalo. Figure 10 shows the trend for maximum memory consumption and training time per epoch as the number of micro-batches increases. Finally, Figure 17 provides evidence that model convergence is not affected by Buffalo, confirming the effectiveness of micro-batch training. .

The framework of Buffalo is developed based on DGL (pytorch backend). The requirements are as follows: Pytorch >= 2.1, and DGL >= 2.2. The other software dependency includes sortedcontainers, pyvis, pynvml, tqdm, pymetis, and seaborn.

The results of our experiments, as presented in the paper, were obtained using a machine equipped with an RTX 6000 GPU (24 GB memory) and an Intel® Xeon® Gold 6126 CPU @ 2.60 GHz. You can use a different configuration as long as it includes at least one GPU.

B. Artifact check-list (meta-information)

- **Model:** In artifact evaluation, we use primarily the GraphSAGE model to show the performance of Buffalo.
- **Data set:** The datasets used are ogbn-arxiv and ogbn-products, which can be downloaded directly from the Open Graph Benchmark(OGB) website.
- **Runtime environment:** Ubuntu20.04, CUDA 12.1 pytorch >= 2.1, and DGL >= 2.2. Details can be found in the github repo https://github.com/PASAUCMerced/Buffalo.git. Python 3.9 is just one option in the requirement. You can also use another Python version, e.g., Python3.11, but you need to configure the corresponding PyTorch and DGL versions.
- **Hardware:** At least one GPU.
- **Metrics:** GPU memory usage and execution time.
- **Experiments:** To save time, we choose Figure 10, 11, 2&13, 17 to denote that Buffalo can effectively reduce the maximum memory consumption without changing the training convergence.
- **How much disk space required (approximately)?:** 60GB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** a few hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache-2.0 license.
- **Data licenses (if publicly available)?:** MIT License and U.S. Government Work license.
- **Archived (provide DOI)?:10.5281/zenodo.14676525**

C. Description

1) How to access.: You can obtain the artifact from https://github.com/PASAUCMerced/Buffalo.git.

2) Hardware dependencies.: The results presented were collected from a machine equipped with a single RTX6000 GPU (24 GB memory). If you use a GPU with different memory capacity, the problem of out-of-memory (OOM) during mini-batch training may vary. Running the benchmark requires up to 60 GB of disk space. Aside from this, there is no other special hardware requirement.

3) Software dependencies.: Ubuntu20.04, CUDA 12.1, Python 3.9, PyTorch 2.1 or higher, and DGL 2.2 or higher. The main software include DGL, PyTorch, sortedcontainers, pyvis, pynvml, tqdm, and pymetis. The software might have compatibility issues, so please be cautious when installing software.

4) Data sets.: The datasets (OGBN-arxiv and OGBN-products) we used in the artifact can be downloaded from Open Graph Benchmark(OGB) Dataset.

5) Models. : The model used in artifact is GraphSAGE with different aggregators, number of layers, hidden size, and fan out size.

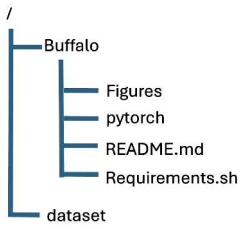

Fig. 18. Directory structure of the artifact.

D. Installation

Obtain the artifact (see Section C1), and extract the archive files.

Next, download the benchmarks and generate the full batch data into the folder `/dataset/`. You can execute the bash file `gen_data.sh` located in the folder `/Buffalo/pytorch/micro_batch_train/` with fanout 10 and 25. After this, you will find the folder `/dataset/fan_out_10,25` containing the pickle files of the full batch data after sampling.

E. Experiment Workflow

We present the directory structure of our artifact in Figure 18. The directory `pytorch` contains all necessary files for the micro-batch training and mini-batch training. In the folder `bucketing/`, `bucket_partitioner.py` includes our implementation of the degree-bucket scheduler. `bucket_dataloader.py` is designed to construct the micro-batch based on the scheduling results of Buffalo. The folder `Figures/` contains important figures for analysis and performance evaluation. In Section F, we explain how these scripts replicate the results shown in those figures to evaluate the performance.

F. Evaluation and Expected Results

We select scripts for certain figures to replicate the evaluation, and the commands that need to be executed are located in the folder `Figures/`. For example, in `Figure 10/`, you can execute the bash file to get the

results for full batch size as well as 2, 4, 8, 16, and 32 micro-batches. The results will be saved in the folder `/Buffalo/Figures/Figure 10/log/`. After that, run `data_collection.py` to summarize the maximum memory and time consumption data of micro-batches training. These data are stored in a table shown in README.md in `Figure 10/`.

The output of the execution will be stored in `log/` folder in each figure folder. The logs of expected results are stored in `log/bak/` folder, and the figures and/or tables of expected results are displayed in each figure folder.

More details can be found in the `README.md` file.

REFERENCES

[1] Y. Bai, C. Li, Z. Lin, Y. Wu, Y. Miao, Y. Liu, and Y. Xu, "Efficient data loader for fast sampling-based gnn training on large graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 10, pp. 2541–2556, 2021.

[2] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[3] M. Ciortan and M. Defrance, "Gnn-based embedding for clustering scrna-seq data," *Bioinformatics*, vol. 38, no. 4, pp. 1037–1044, 2022.

[4] M. T. Dearing and X. Wang, "Analyzing the performance of graph neural networks with pipe parallelism," *arXiv preprint arXiv:2012.10840*, 2020.

[5] G. V. Demirci, A. Haldar, and H. Ferhatosmanoglu, "Scalable graph convolutional network training on distributed-memory systems," *arXiv preprint arXiv:2212.05009*, 2022.

[6] DGL, "Deep Graph Library," https://www.dgl.ai/.

[7] B. Du, J. Liu, Z. Luo, C. Wu, Q. Zhang, and H. Jin, "Expediting distributed gnn training with feature-only partition and optimized communication planning," in *IEEE International Conference on Computer Communications (INFOCOM) 2024 (20/05/2024-23/05/2024, Vancouver)*, 2024.

[8] V. P. Dwivedi, C. K. Joshi, A. T. Luu, T. Laurent, Y. Bengio, and X. Bresson, "Benchmarking graph neural networks," *Journal of Machine Learning Research*, vol. 24, no. 43, pp. 1–48, 2023.

[9] C. Eksombatchai, P. Jindal, J. Z. Liu, Y. Liu, R. Sharma, C. Sugnet, M. Ulrich, and J. Leskovec, "Pixie: A system for recommending 3+ billion items to 200+ million users in real-time," in *Proceedings of the 2018 world wide web conference*, 2018, pp. 1775–1784.

[10] Exxact. (2023) Pytorch geometric vs deep graph library. Accessed: 2025-01-13. [Online]. Available: https://www.exxactcorp.com/blog/Deep-Learning/pytorch-geometric-vs-deep-graph-library

[11] A. Fender, B. Rees, and J. Eaton, "Rapids cugraph," in *Massive Graph Analytics*. Chapman and Hall/CRC, 2022, pp. 483–493.

[12] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019. https://doi.org/10.48550/arXiv.1903.02428.

[13] K. Fu, Q. Chen, Y. Yang, J. Shi, C. Li, and M. Guo, "Blad: Adaptive load balanced scheduling and operator overlap pipeline for accelerating the dynamic gnn training," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–13.

[14] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021. https://api.semanticscholar.org/CorpusID:236992607, pp. 551–568.

[15] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[16] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017. https://doi.org/10.48550/arXiv.1706.02216.

[17] 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.https://doi.org/10.48550/arXiv.2005.00687.

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

[19] K. Huang, J. Zhai, L. Zheng, H. Wang, Y. Jin, Q. Zhang, R. Zhang, Z. Zheng, Y. Yi, and X. Shen, "Wisegraph: Optimizing gnn with joint workload partition of graph and operations," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 1–17.

[20] Y. Huang and D. Li, "Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems," in *IEEE International Conference on Cluster Computing*, 2017.

[21] Y. Huang, Y. Bu, Y. Ding, and W. Lu, "Number versus structure: Towards citing cascades," *Scientometrics*, vol. 117, pp. 2177–2193, 2018.

[22] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken, "A Distributed Multi-GPU System for Fast Graph Processing," *Proc. VLDB Endow.*, vol. 11, no. 3, p. 297–310, 2017.

[23] 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.

[24] Y. Jing, Y. Yang, X. Wang, M. Song, and D. Tao, "Meta-aggregator: Learning to aggregate for 1-bit graph neural networks," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 5301–5310.

[25] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997. https://hdl.handle.net/11299/215346.

[26] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[27] A. Kemper, *Valuation of network effects in software markets: A complex networks approach*. Springer Science & Business Media, 2009.

[28] D. Y. Kenett, M. Perc, and S. Boccaletti, "Networks of networks–an introduction," *Chaos, Solitons & Fractals*, vol. 80, pp. 1–6, 2015.

[29] A. Khatua, V. S. Mailthody, B. Taleka, T. Ma, X. Song, and W.-m. Hwu, "Igb: Addressing the gaps in labeling, features, heterogeneity, and size of public graph datasets for deep learning research," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 4284–4295.

[30] B. Khemani, S. Patil, K. Kotecha, and S. Tanwar, "A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions," *Journal of Big Data*, vol. 11, no. 1, p. 18, 2024.

[31] T. Kim, C. Hwang, K. Park, Z. Lin, P. Cheng, Y. Miao, L. Ma, and Y. Xiong, "Accelerating gnn training with locality-aware partial execution," in *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2021, pp. 34–41.

[32] D. P. Kingma and J. Ba, "Graph Attention Networks," in *International Conference for Learning Representations (ICLR)*, 2018.

[33] K.-H. Lai, D. Zha, K. Zhou, and X. Hu, "Policy-gnn: Aggregation optimization for graph neural networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 461–471.

[34] M. Li, J. Zhou, J. Hu, W. Fan, Y. Zhang, Y. Gu, and G. Karypis, "Dgl-lifesci: An open-source toolkit for deep learning on graphs in life science," *ACS omega*, vol. 6, no. 41, pp. 27 233–27 238, 2021.

[35] S. Li, J. Gu, J. Wang, T. Yao, Z. Liang, Y. Shi, S. Li, W. Xi, S. Li, C. Zhou *et al.*, "Poster: Pargnn: Efficient training for large-scale graph neural network on gpu clusters," in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2024, pp. 469–471.

[36] J. Liu, F. Tang, and X. Hou, "Label aggregation with self-supervision enhanced graph transformer." in *ECAI*, 2023, pp. 1513–1520.

[37] J. Liu, D. Li, and J. Li, "Athena: High-Performance Sparse Tensor Contraction Sequence on Heterogeneous Memory," in *International Conference on Supercomputing (ICS)*, 2021.

[38] J. Liu, J. Ren, R. Gioiosa, D. Li, and J. Li, "Sparta: High-Performance, Element-Wise Sparse Tensor Contraction on Heterogeneous Memory," in *Principles and Practice of Parallel Programming*, 2021.

[39] J. Liu, G. P. Ong, and X. Chen, "Graphsage-based traffic speed forecasting for segment network with sparse data," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 3, pp. 1755–1766, 2020.

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

[41] W.-X. Liu, J. Cai, Y.-H. Zhu, J.-M. Luo, and J. Li, "Load balancing inside programmable data planes based on network modeling prediction using a gnn with network behaviors," *Computer Networks*, vol. 227, p. 109695, 2023.

[42] S. Luan, C. Hua, Q. Lu, J. Zhu, M. Zhao, S. Zhang, X. Chang, and D. Precup, "Is heterophily A real nightmare for graph neural networks to do node classification?" *CoRR*, vol. abs/2109.05641, 2021.

[43] B. Ma, J. Ren, S. Yang, B. Francis, E. Ardestani, M. Si, and D. Li, "Machine Learning-Guided Memory Optimization for DLRM Inference on Tiered Memory," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2025.

[44] J. Ma, Z. Ma, J. Chai, and Q. Mei, "Partition-based active learning for graph neural networks," *arXiv preprint arXiv:2201.09391*, 2022.

[45] 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. https://www.usenix.org/conference/atc19/presentation/ma, pp. 443–458.

[46] W. Marfo, D. K. Tosh, and S. V. Moore, "Enhancing network anomaly detection using graph neural networks," in *2024 22nd Mediterranean Communication and Computer Networking Conference (MedComNet)*. IEEE, 2024, pp. 1–10.

[47] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha, "Dist-gnn: Scalable distributed training for large-scale graph neural networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021. https://doi.org/10.48550/arXiv.2104.06700, pp. 1–14.

[48] S. Mondal, S. D. Manasi, K. Kunal, S. Ramprasath, Z. Zeng, and S. S. Sapatnekar, "A unified engine for accelerating gnn weighting/aggregation operations, with efficient load balancing and graph-specific caching," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 12, pp. 4844–4857, 2022.

[49] S. Mondal, S. D. Manasi, K. Kunal, and S. S. Sapatnekar, "Gnnie: Gnn inference engine with load-balancing and graph-specific caching," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 565–570.

[50] M. E. Newman, "Assortative mixing in networks," *Physical review letters*, vol. 89, no. 20, p. 208701, 2002.

[51] F. S. Nurkasyifah, A. K. Supriatna, and A. Maulana, "Supervised gnns for node label classification in highly sparse network: Comparative analysis," in *2024 IEEE International Conference on Evolving and Adaptive Intelligent Systems (EAIS)*. IEEE, 2024, pp. 1–8.

[52] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019. https://dl.acm.org/doi/10.5555/3454287.3455008.

[53] J. Ren, J. Luo, I. Peng, K. Wu, and D. Li, "Optimizing Large-Scale Plasma Simulations on Persistent Memory-based Heterogeneous Memory with Effective Data Placement Across Memory Hierarchy," in *International Conference on Supercomputing (ICS)*, 2021.

[54] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[55] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "ZeRO-Offload: Democratizing Billion-Scale Model Training," in *USENIX Annual Technical Conference*, 2021.

[56] J. Ren, D. Xu, J. Ryu, K. Shin, D. Kim, and D. Li, "MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory Systems," in *European Conference on Computer Systems*, 2024.

[57] J. Ren, S. Yang, D. Xu, J. Li, Z. Zhang, C. Navasca, C. Wang, G. H. Xu, and D. Li, "DyNN-Offload: Enabling Large Dynamic Neural Network Training with Learning-based Memory Management," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2024.

[58] J. Ren, M. Zhang, and D. Li, "HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory," in *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[59] S. Sajadmanesh, A. S. Shamsabadi, A. Bellet, and D. Gatica-Perez, "{GAP}: Differentially private graph neural networks with aggregation perturbation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3223–3240.

[60] A. Santoro, D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, and T. Lillicrap, "A simple neural network module for relational reasoning," *Advances in neural information processing systems*, vol. 30, 2017.

[61] Y. Shao, H. Li, X. Gu, H. Yin, Y. Li, X. Miao, W. Zhang, B. Cui, and L. Chen, "Distributed graph neural network training: A survey," *ACM Computing Surveys*, vol. 56, no. 8, pp. 1–39, 2024.

[62] W. Shi, J. M. Lemoine, A.-E.-M. A. Shawky, M. Singha, L. Pu, S. Yang, J. Ramanujam, and M. Brylinski, "Bionoinet: ligand-binding site classification with off-the-shelf deep neural network," *Bioinformatics*, vol. 36, no. 10, pp. 3077–3083, 2020.

[63] S. Song and P. Jiang, "Rethinking graph data placement for graph neural network training on multiple gpus," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–10.

[64] Q. Su, M. Wang, D. Zheng, and Z. Zhang, "Adaptive load balancing for parallel gnn training," 2021.

[65] R. Sun, H. Dai, and A. W. Yu, "Does gnn pretraining help molecular representation?" *Advances in Neural Information Processing Systems*, vol. 35, pp. 12 096–12 109, 2022.

[66] S. Thais, P. Calafiura, G. Chachamis, G. DeZoort, J. Duarte, S. Ganguly, M. Kagan, D. Murnane, M. S. Neubauer, and K. Terao, "Graph neural networks in particle physics: Implementations, innovations, and challenges," *arXiv preprint arXiv:2203.12852*, 2022.

[67] A. Trivella and D. Pisinger, "The load-balanced multi-dimensional bin-packing problem," *Comput. Oper. Res.*, vol. 74, pp. 152–164, 2016.

[68] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

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

[70] X. Wan, K. Chen, and Y. Zhang, "Dgs: Communication-efficient graph sampling for distributed gnn training," in *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. IEEE, 2022. doi: 10.1109/ICNP55882.2022.9940348., pp. 1–11.

[71] X. Wan, K. Xu, X. Liao, Y. Jin, K. Chen, and X. Jin, "Scalable and efficient full-graph gnn training for large graphs," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–23, 2023.

[72] 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.

[73] 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.

[74] Q. Wang, Y. Chen, W.-F. Wong, and B. He, "Hongtu: Scalable full-graph gnn training on multiple gpus," *Proceedings of the ACM on Management of Data*, vol. 1, no. 4, pp. 1–27, 2023.

[75] X. S. Wang, J. Liu, J. Wu, S. Yang, J. Ren, B. Shankar, and D. Li, "Performance Characterization of CXL Memory and Its Use Cases," in *International Parallel and Distributed Processing Symposium*, 2025.

[76] Y. Wang, K. Yi, X. Liu, Y. G. Wang, and S. Jin, "ACMP: allen-cahn message passing with attractive and repulsive forces for graph neural networks," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[77] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, "{GNNAdvisor}: An adaptive and efficient runtime system for {GNN} acceleration on {GPUs}," in *15th USENIX symposium on operating systems design and implementation (OSDI 21)*, 2021, pp. 515–531.

[78] Z. Wang, Z. Wei, Y. Li, W. Kuang, and B. Ding, "Graph neural networks with node-wise architecture," in *Proceedings of the 28th ACM*

SIGKDD conference on knowledge discovery and data mining, 2022, pp. 1949–1958.

[79] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[80] B. Wu, X. Yang, S. Pan, and X. Yuan, "Adapting membership inference attacks to gnn for graph classification: Approaches and implications," in *2021 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2021, pp. 1421–1426.

[81] J. Wu, J. He, and J. Xu, "Net: Degree-specific graph neural networks for node and graph classification," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 406–415.

[82] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[83] K. Wu, J. Ren, and D. Li, "Runtime Data Management on Non-volatile Memory-based Heterogeneous Memory for Task-parallel Programs," in *International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2018.

[84] Y. Xia, Z. Zhang, H. Wang, D. Yang, X. Zhou, and D. Cheng, "Redundancy-free high-performance dynamic gnn training with hierarchical pipeline parallelism," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, 2023, pp. 17–30.

[85] Z. Xie, W. Dong, J. Liu, I. Peng, Y. Ma, and D. Li, "MD-HM: Memoization-based Molecular Dynamics Simulations on Big Memory System," in *International Conference on Supercomputing (ICS)*, 2021.

[86] Z. Xie, J. Liu, J. Li, and D. Li, "Merchandiser: Data Placement on Heterogeneous Memory for Task-Parallel HPC Applications with Load-Balance Awareness," in *Proceedings of the Symposium on Principles and Practices of Parallel Programming (PPoPP)*, 2023.

[87] D. Xu, Y. Feng, K. Shin, D. Kim, H. Jeon, and D. Li, "Efficient Tensor Offloading for Large Deep-Learning Model Training based on Compute Express Link," in *36th ACM/IEEE International Conference for High Performance Computing, Performance Measurement, Modeling and Tools*, 2024.

[88] D. Xu, J. Ryu, J. Baek, K. Shin, P. Su, and D. Li, "FlexMem: Adaptive Page Profiling and Migration for Tiered Memory," in *30th USENIX Annual Technical Conference (ATC)*, 2024.

[89] H. Xu, Z. Duan, Y. Wang, J. Feng, R. Chen, Q. Zhang, and Z. Xu, "Graph partitioning and graph neural network based hierarchical graph matching for graph similarity computation," *Neurocomputing*, vol. 439, pp. 348–362, 2021.

[90] Z. Xue, Y. Yang, and R. Marculescu, "Sugar: Efficient subgraph-level training via resource-aware graph partitioning," *IEEE Transactions on Computers*, 2023.

[91] H. Yang, "Aligraph: A comprehensive graph neural network platform," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019. https://doi.org/10.1145/3292500.3340404, pp. 3165–3166.

[92] S. Yang, K. Wu, Y. Qiao, D. Li, and J. Zhai, "Algorithm-directed crash consistence in non-volatile memory for hpc," in *IEEE Cluster Computing*, 2017.

[93] S. Yang, M. Zhang, W. Dong, and D. Li, "Betty: Enabling large-scale gnn training with batch-level graph partitioning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023. 10.1145/3575693.3575725, pp. 103–117.

[94] 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.

[95] S. Yun, S. Kim, J. Lee, J. Kang, and H. J. Kim, "Neo-gnns: Neighborhood overlap-aware graph neural networks for link prediction," *Advances in Neural Information Processing Systems*, vol. 34, pp. 13 683–13 694, 2021.

[96] L. Zangari, R. Interdonato, A. Calió, and A. Tagarelli, "Graph convolutional and attention models for entity classification in multilayer networks," *Applied Network Science*, vol. 6, no. 1, p. 87, 2021.

[97] L. Zhang, Z. Lai, Y. Tang, D. Li, F. Liu, and X. Luo, "Pc-graph: Accelerating gnn inference on large graphs via partition caching," in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 2021, pp. 279–287.

[98] P. Zhang, Y. Yan, X. Zhang, C. Li, S. Wang, F. Huang, and S. Kim, "Transgnn: Harnessing the collaborative power of transformers and graph neural networks for recommender systems," in *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2024, pp. 1285–1295.

[99] T. Zhang, H.-R. Shan, and M. A. Little, "Causal graphsage: A robust graph method for classification based on causal sampling," *Pattern Recognition*, vol. 128, p. 108696, 2022.

[100] C. Zheng, H. Chen, Y. Cheng, Z. Song, Y. Wu, C. Li, J. Cheng, H. Yang, and S. Zhang, "Bytegnn: efficient graph neural network training at large scale," *Proceedings of the VLDB Endowment*, vol. 15, no. 6, pp. 1228–1242, 2022.

[101] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "Distdgl: distributed graph neural network training for billion-scale graphs," in *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2020. doi: 10.1109/IA351965.2020.00011., pp. 36–44.

[102] F. Zhou, C. Cao, K. Zhang, G. Trajcevski, T. Zhong, and J. Geng, "Meta-gnn: On few-shot node classification in graph meta-learning," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019. https://doi.org/10.48550/arXiv.1905.09718., pp. 2357–2360.

[103] J. Zhu, Y. Yan, L. Zhao, M. Heimann, L. Akoglu, and D. Koutra, "Beyond homophily in graph neural networks: Current limitations and effective designs," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[104] Z. Zhu, B. Jing, X. Wan, Z. Liu, L. Liang *et al.*, "Glisp: A scalable gnn learning system by exploiting inherent structural properties of graphs," *arXiv preprint arXiv:2401.03114*, 2024.