Cherry: Breaking the GPU Memory Wall for Large-Scale GNN Training via Micro-Batching

Yan Wang[1], Qinghua Guo[1], Haoran Kong[2,3], Kai Sheng[1],
Zhen Xie[5], Hao Chen[4], Weile Jia[2,3], Dingwen Tao[2,3], Xin He[1]*
[1]Guangzhou Institute of Technology, Xidian University, Guangzhou, Guangdong, China
[2]SKLP, Institute of Computing Technology, CAS, Bejing, China
[3]University of Chinese Academy of Sciences, Beijing, China
[4]CSEE, Hunan University, Changsha, Hunan, China
[5]Binghamton University, Binghamton, NY, USA
{wang-yan,guo-qh}@stu.xidian.edu.cn,{hexin,kaisheng}@xidian.edu.cn
{taodingwen,jiaweile}@ict.ac.cn,zxie3@binghamton.edu,haochen@hnu.edu.cn,buaakhr@gmail.com

Abstract

Graph Neural Networks (GNNs) have shown remarkable performance across a variety of graph-related tasks. Recent efforts indicate that GNN performance can be enhanced through more sophisticated strategies, such as employing advanced aggregators, increasing aggregation depth, and utilizing larger sampling rates, etc. While these strategies yield promising results, it also incurs a significantly larger memory footprint that can easily surpass the GPU memory capacity. Micro-batching has emerged as a promising method to mitigate GPU memory bottleneck while preserving model accuracy. Nevertheless, integrating micro-batches into GNN training presents two significant challenges: (1) achieving load balancing while minimizing node redundancy, and (2) managing the high overhead associated with data preparation. Current solutions struggle to effectively address both load balancing and node redundancy while also inducing prohibitively high data preparation overhead, hindering the practicality and performance of applying micro-batching techniques to GNN training.

In this paper, we introduce Cherry, an efficient micro-batching method designed to make GNN training more accessible and scalable. The key enabling idea behind Cherry is to leverage an innovative combination of partitioning techniques aided by message-passing flow graphs and micro-batch based data loading mechanism, which collectively serve to reduce redundancy and load imbalance within the micro-batches and lower the overhead associated with data preparation in the training process. Our evaluation demonstrates that Cherry can enable billion-scale GNN training with sophisticated strategies on a single-GPU machine, significantly outperforming state-of-the-art baselines.

CCS Concepts

• **Computer systems organization → Single instruction, multiple data**.

Keywords

graph neural network, micro-batching, graph partitioning, load balancing, redundancy reduction.

1 Introduction

Graph Neural Networks (GNNs) have emerged as powerful tools for learning from graph-structured data and achieved remarkable performance across a variety of graph-related tasks, such as link prediction [35, 52], node classification [34, 44], and graph classification [18, 53].

Recent advancements indicate that the efficiency and accuracy of GNN training can be enhanced through several strategies, including the use of larger batch sizes (e.g., transitioning from mini-batch to full-batch training) [12, 15], employing more sophisticated aggregators (e.g., LSTM and attention networks) [24, 36], increasing aggregation depth (e.g., from 1 to 112) [20], utilizing larger sampling rates (i.e., incorporating more neighbors for aggregation) [55], and deploying deeper and wider neural encoders [20], etc. While

ICS '25, June 8–11, 2025, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/2025/06
https://doi.org/10.1145/3721145.3730429

*Corresponding author: Xin He.

yielding promising results, these strategies often incur significantly increased memory consumption. For instance, GNNs iteratively aggregate features from neighboring nodes to learn node vector representations. Increasing aggregation depth results in exponential growth in the number of feature vectors that must be loaded into memory for aggregation. Given that device memory on hardware accelerators like GPU is a scarce resource, using strategies like deep aggregation can easily run out of memory (OOM). As a result, this memory capacity wall restricts many methods to small-scale graphs (e.g., hundreds of thousands of nodes) or shallow architectures (e.g., fewer than three layers) in GNN models, limiting their ability to explore node dependencies before exhausting memory.

In an effort to break the GPU memory capacity wall, recent years have seen numerous research efforts on both algorithmic and system optimizations. Algorithmically, a typical method is sampling [9, 11, 49], which involves sampling a subset of neighbors to compute features for a specific node or subgraph. By reducing the sampling rate, the number of neighbors participating in aggregation decreases, leading to lower memory consumption. However, this approach necessitates careful consideration of the sampling strategy, as it may result in the loss of important neighbor information that ultimately harms the model accuracy [11]. On the system front, dedicated GNN frameworks like DGL [40], PyG [5], and NeuGraph [25] have been developed to boost GNN training efficiency by implementing highly efficient graph operation primitives. However, these frameworks encounter challenges in memory scalability when handling large graphs or sophisticated GNN architectures. An alternative is to partition and process large graphs in a distributed fashion using distributed frameworks like DistDGL [54], leveraging the combined memory of multiple GPUs and/or computing nodes to scale out model training. This approach, albeit effective, typically entails a significantly higher hardware cost, which is unrealistic for researchers and practitioners with limited hardware budgets.

Among the various methods, micro-batching has emerged as a promising technique in deep learning [14, 27, 30, 31]. It involves partitioning a batch into K smaller micro-batches and accumulating gradients from these micro-batches to update model weights. This approach reduces training memory footprint while maintaining model convergence and quality. Unlike traditional Neural Network (NN) models, GNNs exhibit more complex dependencies (e.g., N:M) between outputs/labels and input features within a batch, rather than a simple 1:1 mapping. Adapting micro-batching to GNN models presents unique challenges in terms of node redundancy, load imbalance, and data preparation overhead, as discussed in Section 3. Recent work, such as Betty [46], has explored micro-batching for GNNs. While it effectively reduces node redundancy, it fails to address the load balancing issue efficiently (as discussed in Section 3.2), leading to suboptimal reduction of peak memory consumption. Moreover, it suffers from high data preparation overhead (as discussed in Section 3.3), resulting in poor training efficiency.

In this paper, we present Cherry, an efficient micro-batching method to enable large-scale GNN training. The key enabling idea behind Cherry lies in combining partitioning techniques aided by message-passing flow graphs with a micro-batch-specific data loading mechanism.

To reduce redundancy and load imbalance, we utilize an out-degree centric partitioning approach aided by auxiliary graphs constructed using message-passing flow for graph partitioning. This method assigns the weight to edges based on the out-degrees of the source node. Leveraging the out-degree information embedded in the auxiliary graphs, this method can effectively group destination nodes sharing the same source node into the same micro-batch while maintaining load balance by distributing high out-degree nodes across micro-batches as evenly as possible.

To reduce data preparation overhead, we adapt the out-degree centric graph partitioning with global message-passing flow graph to avoid repetitive partitioning overhead. Additionally, we introduce a micro-batch-specific data loading mechanism to enhance data loading efficiency of micro-batching-based GNN training. These techniques collectively mitigate redundancy and load imbalance within micro-batches, and reduce data preparation overhead in micro-batching-based GNN training, making it feasible and efficient to train GNN with sophisticated strategies on large graphs.

In summary, we make the major contributions as follows.

- We conduct an in-depth analysis of the challenges and issues of micro-batching for GNN training as well as the limitations of current micro-batching framework.
- We propose an efficient out-degree centric partitioning technique that can effectively balance loads while reducing node redundancy among micro-batches.
- We introduce a global message-passing flow graph as an auxiliary graph for partitioning and develop a micro-batch-specific data loading mechanism, which collectively reduce high data preparation overhead.
- We build Cherry on top of DGL, and evaluate it with real-world large graphs. Our evaluation shows Cherry can enable efficient large-scale GNN training beyond GPU memory capacity while maintaining comparable model convergence and accuracy to full-batch training without requiring any changes to hyperparameters. Compared with the state-of-the-art (SOTA) method, Cherry achieves an average reduction of 10.8% (up to 18.8%) in memory consumption and 94.9% (up to 98.8%) in training time, respectively.

2 Background

2.1 Graph Neural Network

GNN models are a kind of neural networks designed to capture complex relationships among nodes in a graph, including node and edge attributes. In GNNs, nodes represent entities like users in a social network, each with unique feature vectors, while edges denote relationships between nodes, characterized by edge weights.

In a GNN model, a computation layer combines graph operations and neural operations. *Graph operations* include a central node gathering feature from neighbors, aggregating them, and updating its own features to encode the graph's structure. *Neural operations* are performed independently or following center-neighbor patterns, leveraging relationships within the neighborhood to conduct operations for each central node based on its neighbors' features.

The core of GNNs involves recursively aggregating neighbor information through message passing and applying feature transformation. For instance, in the GraphSAGE model [11], the training process for a single computation layer can be formulated as follows:

$$h_v^k = \sigma\left(W^k \cdot \text{Aggregate}\left(\left\{h_u^{k-1}, u \in N(v)\right\}\right) \parallel h_v^{k-1}\right) \quad (1)$$

where $N(v)$ denotes the set of neighbors of node v. In the k-th convolution layer, each node v aggregates its neighbors' feature vectors from the the $(k-1)$-th layer using an aggregator function (e.g., Sum, Mean, Pooling, LSTM). The aggregated result is concatenated with v's own feature vector from the previous layer, then multiplied by a learnable weight matrix W and transformed using a nonlinearity activation function σ (e.g., ReLU) to produce the hidden feature vector for node v in the k-th layer.

As the number of layers increases in a GNN, nodes must aggregate information from more distant neighbors (i.e., expanding from the i-hop neighbors to the j-hop neighbors for $j > i$). However, aggregating all neighbors can lead to prohibitive memory consumption. Hence, a fanout degree bound is often imposed by practitioners, restricting the number of neighbors considered for aggregation through graph sampling techniques. After training across all K layers, the final feature vector for each node v is fed into a task-specific mapping function for downstream tasks.

2.2 Batching Techniques for GNN Training

GNNs are essential for learning from graph-structured data and have succeeded in various graph-based applications. However, they face challenges in navigating trade-offs between training accuracy, model generalization, memory requirements, and computational efficiency. Various batching techniques, such as full-batching, mini-batching, and micro-batching, address these challenges by offering distinct solutions tailored to different aspects of GNN training.

Full-batching involves processing the entire graph in one forward and backward pass, leveraging the complete graph structure during optimization. This approach offers a comprehensive view of the graph, often leading to improved accuracy and generalization from the global context utilized. Full-batch training is ideal for small to medium-sized graphs, where memory and computational demands are moderate. However, its scalability is limited by graph size, as memory usage increases rapidly with the number of nodes and edges. Consequently, full-batch training becomes infeasible for large-scale graphs, necessitating alternative batching strategies.

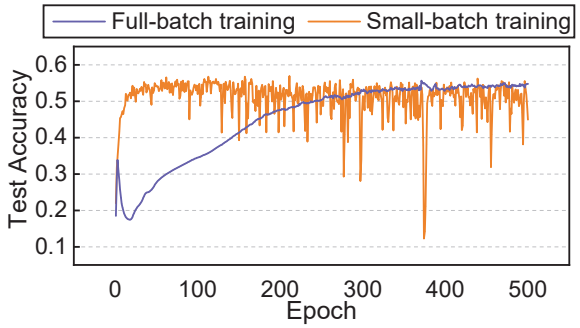

Figure 1: The convergence curves for full-batch and mini-batch training.

Mini-batching is a widely adopted approach that partitions the full-batch into multiple mini-batches and independently updates the model parameters for each mini-batch. While smaller mini-batch sizes can effectively decrease memory consumption, they can adversely affect model convergence and generalization. Figure 1 compares training convergence curves between full-batch and mini-batch training using GraphSAGE on the ogbn-products dataset with consistent hyperparameter settings. Although smaller mini-batch sizes lower memory consumption, the accuracy curves show significant differences. Specifically, mini-batch training exhibits a rapid increase in accuracy at the beginning, but after several epochs, the training curve experiences significant fluctuations and struggles to converge. This difference can be attributed to variations in effective batch size, a critical hyperparameter for GNN training that influences both convergence speed and model quality. Adjusting the effective batch size is generally non-trivial, often requiring modifications to other hyperparameters like learning rate schedules and weight decay to maintain convergence quality. Consequently, it is imperative to mitigate memory bottlenecks and enhance

the scalability of GNNs in a manner that remains transparent to users.

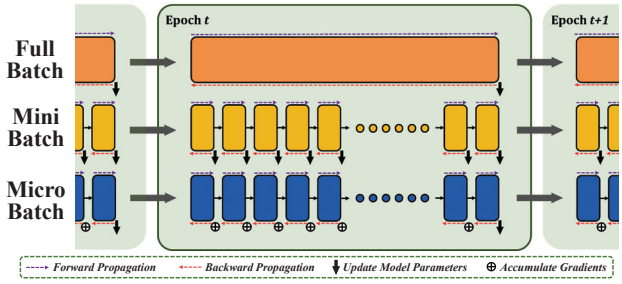

Figure 2: Workflow of different batching techniques. Each block shows an independent execution of forward and backward pass.

Micro-batching divides a full batch into multiple micro-batches for independent gradient computation, followed by accumulating the gradients from all micro-batches to perform a single update to the model parameters. Figure 2 shows the differences in computation and gradient updates among three batching techniques. Full-batching computes gradients and updates parameters once per epoch. Mini-batching does so for each mini-batch. In contrast, micro-batching computes gradients per micro-batch but accumulates them to update the model parameters once per epoch. This enables the model to train with a significantly reduced memory footprint while maintaining model convergence and quality without changes to hyperparameters or optimizers. This optimization is transparent to the training process, facilitating a more efficient training experience. Hence, in this work, we focus on exploring micro-batching for GNN training.

3 Motivation

In this section, we first analyze the memory performance of different batching techniques, which motivated our choice of micro-batch training. Then we dissect the challenges and issues of applying micro-batching to GNN training, including balancing load while reducing node redundancy and high overhead in data preparation. Additionally, we analyze the limitations of the SOTA approach, Betty, in tackling these challenges, which motivates our work.

3.1 Memory Capacity Bottleneck in GNNs

Despite the significant success of GNNs in various graph-based learning tasks, a major challenge is their substantial memory consumption, which often leads to out-of-memory (OOM) errors, even when using mini-batching. To illustrate the memory requirements of GNNs, we profile the full-batch and mini-batch (4 batches) training on the large dataset

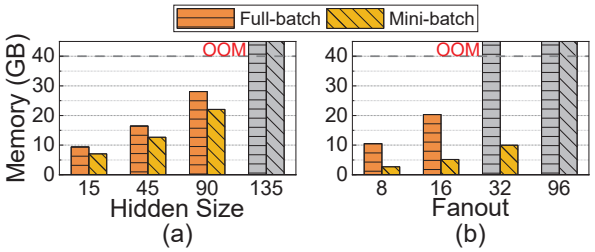

Figure 3: Full-batch & mini-batch in memory consuming cases.

Ogbn-Products using an NVIDIA A100 (40GB). Figure 3(a) depicts the training memory consumption under a three-layer GAT model for different hidden sizes, revealing that both full-batch and mini-batch approaches encounter OOM errors when the hidden size is set to 135. Figure 3(b) shows the memory consumption of a one-layer GraphSAGE model using the advanced LSTM aggregator with various fanout degrees. The fanout degree in GNN training regulates the number of neighboring nodes sampled at each layer, directly affecting memory requirements. The full-batch training experienced OOM at a fanout of 32, while mini-batch training faced OOM at a fanout of 96. Thus, although mini-batching reduces memory consumption compared to full-batching, it still poses significant memory demands during GNN training. Furthermore, adjusting the mini-batch size can jeopardize the convergence quality of the model, as discussed in Section 2.2. However, as shown in Figure 8, utilizing micro-batch training with Cherry effectively addresses these OOM cases and enable large-scale GNN training on a single GPU.

3.2 Challenges in Balancing Load While Reducing Node Redundancy

As discussed in Section 2.2, micro-batching can reduce memory consumption during training while ensuring the same convergence quality. However, there are two issues that impacts the memory consumption reduction and training efficiency: (1) **Node redundancy**: High redundancy not only increases memory usage but also leads to redundant computations, harming training efficiency. (2) **Load balancing**: Poor load balancing can increase the maximum memory footprint on the GPU, as one or a few micro-batches may have an unbalanced load, often due to the chosen partition strategies. Therefore, effectively addressing these issues is crucial for optimizing both memory usage and training efficiency in micro-batching.

Reducing node redundancy and load balancing are often competing objectives in graph partitioning. Reducing node redundancy typically involves grouping neighboring nodes

into the same partition to minimize cross-partition edges, potentially causing imbalanced workloads. Conversely, load balancing requires evenly distributing nodes and edges across partitions, often increasing redundancy due to the splitting of neighborhoods. Striking a balance between these objectives is critical, as it directly impacts both memory comsumption and computational efficiency in GNN training.

Current frameworks like DGL, typically use Metis for graph partitioning, trying to balance the workload across partitions but introduces significant redundancy. Betty, a SOTA micro-batching method, proposes a Redundancy Embedded Graph (REG) based partitioning algorithm to reduce redundancy (detailed in Section 7). However, Betty's approach does not directly address the load balancing but increases micro-batch counts to avoid OOM errors, which is computationally inefficient.

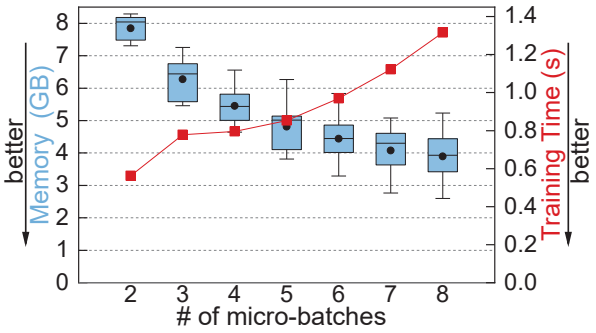

Figure 4: The tendency of memory consumption and training time (without data preparing) per epoch as the number of micro-batches increases in Betty.

Figure 4 shows memory consumption and training time trends for GraphSAGE on the ogbn-products dataset across varying numbers of micro-batches using Betty. While more micro-batches helps decrease memory consumption, it does not directly improve load balance within the micro-batches. Even worse, a consistent degradation in computational efficiency and long training times is observed as the number of micro-batches increases.

3.3 High Overhead in Data Preparation with Micro-Batching

While micro-batching can effectively mitigate GPU memory bottleneck, it introduces two types of overhead in data preparation. First, *graph partitioning* is a critical step in the micro-batching, inevitably adding time overhead. Both the partitioning object size and partitioning frequency during training contribute to time overhead. Second, *data loading* for micro-batching requires extracting data from the original batch, since micro-batches are subdivisions of it.

Table 1: The proportion of time spent on training (without data preparing) and data preparation (data extraction, index mapping, and partitioning) within a training epoch in Betty

Dataset	Training (%)	Data Preparing		
		Data Extraction (%)	Index Mapping (%)	Partitioning(%)
Reddit	0.45	3.37	3.70	92.46
Ogbn-Arxiv	0.67	7.21	10.00	82.10
Ogbn-Products	1.07	21.40	29.18	48.34

For instance, Table 1 shows profiling results detailing the average epoch time for training a 3-layer GraphSAGE model with a hidden size of 256 and 4 micro-batches across three datasets using an A100 GPU. Results show that data preparation—extraction, index mapping, and partitioning—significantly surpasses the actual computation time for training. This discrepancy arises because Betty's graph partitioning strategy mandates partitioning at each epoch, such re-partitioning overhead cannot be amortized over training epochs. Moreover, the auxiliary graph REG constructed by Betty is considerably more complex than the original input graph, with edges increased by 11.9x, 39.9x, and 48.3x for the Ogbn-Products, Ogbn-Arxiv, and Reddit datasets, respectively. Furthermore, Betty's data loading process is inefficient, involving layer-by-layer index mapping and data extraction to ensure alignment between the micro-batch data and the original batch data blocks.

4 Proposed Method

In this section, we introduce Cherry, a solution to address the challenges in micro-batch GNN training. We begin with an overview of our design, followed by detailing the proposed techniques, including out-degree centric partitioning, the global message-passing flow graph, and a micro-batch-specific data loading mechanism. To maximize memory consumption reduction during micro-batch training while minimizing the overhead associated with data preparation, we present Cherry, a novel micro-batching method specifically tailored large-scale GNN training. Cherry utilizes an auxiliary graph concept known as the message-passing flow graph (MFG) and incorporates two key techniques: (1) an out-degree centric partitioning algorithm to tackle redundancy and load imbalance(Section 4.2), and (2) a repetitive-partitioning elimination mechanism paired with a micro-batch-specific data loading mechanism to reduce both partitioning and data loading overhead (Section 4.3).

4.1 Overview

Figure 5 overviews Cherry. First, Cherry constructs the Global Message-passing Flow Graph (GMFG) by tracing the nodes and edges involved in the GNN's message-passing flow. Then, using an out-degree centric partitioning approach, Cherry

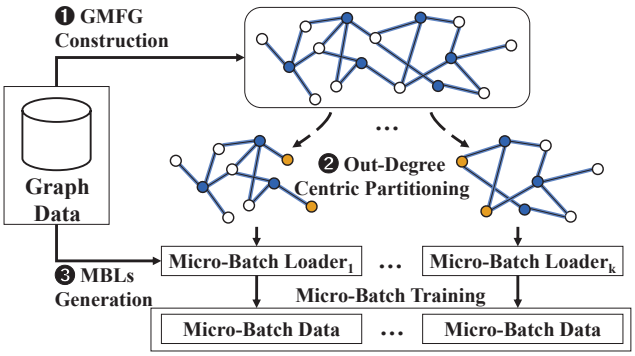

Figure 5: Overview of Cherry. ***Directions of edges in graphs are omitted for brevity.***

partitions the GMFG into subgraphs representing individual micro-batches based on a predetermined number of micro-batches. This partitioning minimizes redundancy and load imbalance within micro-batches and eliminates the repetitive partitioning across training epochs. Similar to Betty, Cherry employs a memory estimation-based technique to determine the optimal number of micro-batches before partitioning. Finally, Cherry uses the partitioned input node lists to instantiate a Micro-Batch Loader (MBL) to load the corresponding graph data independently for each micro-batch. which facilitates efficient data loading for the subsequent micro-batch training.

4.2 Reducing Redundancy and Load-Imbalance

In Section 3, we highlight the challenges of redundancy and load imbalance in current graph partitioning methods, which limit micro-batch training's effectiveness for GNNs in reducing memory usage and improving efficiency. Unlike prior approaches focusing on minimizing edge-cut weights to balance the number of edges or nodes in subgraphs without addressing redundancy, Cherry introduces an *Out-degree Centric Graph Partitioning* method to balance load while reducing redundancy for enhanced performance. The key innovation of our out-degree-centric graph partitioning approach lies using an MFG-based auxiliary graph alongside a corresponding MFG-aided graph partitioning algorithm. The MFG represents the GNN message-passing process, with directed edges indicating message flow. This representation effectively captures the computational processes and memory requirements during GNN training, enabling us to navigate the trade-off between redundancy and load balance. Aided by the out-degree information from the MFG, our method transforms redundancy and load imbalance into an edge-cut weight minimization problem. This allows Cherry to balance load while reducing redundancy within micro-batches. In Section 4.2, we use the Local Message-passing Flow Graph (LMFG) as the auxiliary graph for out-degree centric partitioning. This implementation of Cherry is referred to as Cherry-LMFG.

4.2.1 ***LMFG construction.*** The data shuffling operation in each training epoch alters the message passing paths, which subsequently changes the MFG for that epoch. To achieve optimal partitioning results, a straightforward approach is to partition the batches at each epoch, ensuring effective redundancy reduction and load balance. Consequently, we construct the MFG for each training epoch to assist in graph partitioning. Here we refer to the MFG as "local" in the sense that there is a distinct message-passing flow graph for each training epoch. The LMFG dynamically captures the directional relationships between nodes during the message aggregation process in each epoch. Specifically, for each layer in the GNN, the input nodes (dst_nodes) aggregate features from their neighboring nodes (src_nodes), and these src_nodes from one layer become the dst_nodes for the subsequent layer. By tracking this process, we can construct an LMFG (as illustrated in Figure 7) that reflects the directionalities utilized in the training process. In each training epoch, the LMFG changes with data shuffling.

4.2.2 ***Out-degree centric graph partitioning.*** As discussed in Section 3.2, the redundancy of nodes and the imbalance of load within micro-batches after partitioning substantially affect memory consumption and training efficiency. Consequently, developing an effective graph partitioning algorithm to create micro-batches is imperative. However, existing graph partitioning algorithms primarily focus on minimizing or maximizing cut edge weights (as shown in Equation 2), neglecting the inherent computational characteristics of message-passing in GNNs that could facilitate redundancy reduction and load balancing during the partitioning process.

$$\text{Minimize / Maximize} \sum_{(u,v)\in E_{\text{cut}}} w_{uv} \tag{2}$$

To leverage the structural and computational characteristics inherent in message-passing for graph partitioning, Cherry assigns the out-degree of source nodes in the MFG as edge weights. The out-degree effectively models the workload and redundancy between subgraphs. In terms of load balancing, both computational and storage loads are taken into account. The computational load correlates with the amount of message passing, while the storage load depends on the number of nodes and edges. Both factors are positively correlated with the out-degree of the subgraph (as demonstrated in Equation 3, where $L(G_i)$ represents the workload of subgraph G_i). Equation 4 outlines the objective of load

balancing: to minimize load differences between subgraphs by uniformly distributing out-degrees across them, in accordance with Equation 3.

$$L(G_i) \propto \sum_{v \in G_i} d_{\text{out}}(v). \tag{3}$$

$$\min \max_i L(G_i) - \min_i L(G_i) \tag{4}$$

In terms of redundancy reduction, the goal is to ensure that nodes, particularly high out-degree nodes connected to multiple neighbors, appear in as few subgraphs as possible to mitigate redundancy. The number of redundant nodes can be quantified using Equation 5, where cross_subgraph_count(v) indicates the number of subgraphs that node v appears in. For each node v, cross_subgraph_count(v) is positively correlated with its out-degree (as shown in Equation 6, where P is the probability of crossing subgraphs). Thus, minimizing the occurrence of high out-degree nodes across multiple subgraphs is crucial for reducing redundancy.

$$\text{Redundant nodes} = \sum_{v \in V} \text{cross\_subgraph\_count}(v) \tag{5}$$

$$\text{cross\_subgraph\_count}(v) \propto \text{d}_{\text{out}}(v) \cdot \text{P} \tag{6}$$

Utilizing the informative properties of MFG-based partitioning, Cherry-LMFG adopts an out-degree-centric approach to partition the LMFG at each training epoch, constructing micro-batches. This method effectively reduces peak memory consumption during training, as illustrated in Figure 9.

Algorithm 1: Out-Degree Centric Graph Partitioning

input : Message-Passing Flow Based Graph: *LMFG*
The number of micro-batches: K
output: batched_train_nodes_list

1 src_nids, dst_nids = Get_edges(*LMFG*);
2 $graph_{copy}$ = Graph_create(src_nids, dst_nids);
3 out_degrees = Get_out_degrees($graph_{copy}$, src_nids);
4 $graph_{out\_degree}$ = Set_weights($graph_{copy}$, out_degrees);
5 $graph_{out\_degree}$ = remove_non_output_nodes($self$);
6 $graph_{out\_degree}$ = remove_self_loop($self$);
7 *subgraphs* = Metis_partition($graph_{out\_degree}$, K);
8 **for** *part in subgraphs* **do**
9 *nids* = Get_dst_nids(*part*);
10 batched_train_nodes_list.*append*(*nids*);
11 **end**
12 **return** batched_train_nodes_list;

Algorithm 1 presents the out-degree centric graph partitioning process. Initially, the algorithm takes the LMFG here and the specified number of micro-batches as input. It begins by extracting the source nodes src_nids and destination nodes dst_nids from the LMFG (Line 1 in Algorithm 1), which are then used to create a graph structure (Line 2). Subsequently, the out-degree of each src_node is calculated and set as the edge weight in the graph (Lines 3-4), reflecting the influence of each source node on the training nodes. Nodes with higher out-degrees receive greater weights, indicating their higher impact on the peak memory consumption during training. The algorithm then refines the graph by eliminating non-output nodes and self loops (Lines 5-6), ensuring a focused and efficient partitioning process.

Next, an existing partitioning algorithm is applied to partition the weighted LMFG into subgraphs corresponding to individual micro-batches (Line 7). In our work, we use Metis for its scalability in handling large-scale graph partitioning tasks [54]. This step ensures equitable distribution of nodes with high out-degrees across micro-batches, promoting load balance and thus minimizing peak memory consumption. Finally, the algorithm iterates over the subgraphs, extracts the dst_nids, and appends them to the batched_train_nodes_list (Lines 8-12), serving as the output of the partitioning process. This list delineates the nodes assigned to each micro-batch, facilitating micro-batch training with reduced peak memory consumption and balanced computational load. Note that the construction and storage of the MFG-based auxiliary graphs, as well as the out-degree centric partitioning, are performed on the host side without incurring extra overhead on the GPU.

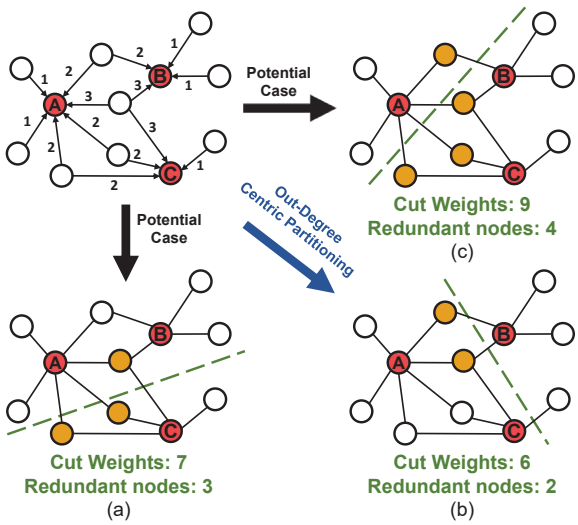

Figure 6: An example to depict out-degree centric partitioning.

Figure 6 illustrates a concrete example demonstrating the partitioning of a specific LMFG into two micro-batches with minimal redundant nodes. The figure showcases the actual partitioning results achieved by our method (Figure 6(b)). For comparative analysis, the figure also displays two additional partitioning outcomes corresponding to alternative partitioning approaches that do not prioritize minimizing edge-cut weights (Figure 6(a) and Figure 6(c)). Our out-degree centric partitioning technique, focused on minimizing the aggregate weight of cut edges, successfully cuts edges with a total weight of 6. This process results in two distinct subsets of training nodes, denoted as set AC and set B, with only 2 redundant nodes. In contrast, the two alternative partitioning methods yield cut edges with total weights of 7 and 9, leading to the creation of 3 and 4 redundant nodes, respectively. This example showcases the efficiency of our approach in reducing redundancy.

4.3 Reducing Partitioning and Data Loading Overhead

Current micro-batch GNN training methods like Betty suffer from high data preparation overhead due to repetitive partitioning and inefficient data loading across training epochs (as discussed in Section 3.3).Cherry-LMFG faces same issues due to the repetitive LMFG-based partitioning in each epoch and using Betty's data loading mechanism. To tackle the two problems, we introduce the Global Message-Passing Flow Graph (GMFG) ,an improved version of LMFG to aid in out-degree centric partitioning and data loading. The GMFG offers a constant global representation during training while encompassing the complete information of the original graph. Leveraging the GMFG, we propose two remedies: (1) adapting the out-degree centric graph partitioning to enable one-time partitioning results for reuse across epochs, eliminating repetitive partitioning, and (2) crafting a micro-batch-specific data loading mechanism based on the partitioning results to enhance data loading efficiency in micro-batch GNN training.

4.3.1 ***Eliminating the repetitive-partitioning overhead.*** In Betty, the input graph's dynamic topology requires fresh partitioning in each training epoch due to data shuffling across epochs, which improves the model's generalization by altering neighboring nodes and edges associated with each training node. Notably, Cherry-LMFG introduced in Section 4.2 faces similar demands. While Cherry-LMFG, leveraging the auxiliary graph LMFG, effectively mitigates redundancy and load imbalance, it still mandates a fresh partitioning step with the updated LMFG in each epoch.

The need for re-partitioning stems from two main limitations of the LMFG: (1) The LMFG is dynamic, only capturing a snapshot of the MFG for the current epoch and undergoes changes due to shuffling operations between epochs. (2) The LMFG offers an incomplete or potentially biased representation of the original graph structure and node distribution. Essentially, as a subgraph sampled from the original graph, it often uses methods like neighborhood sampling, which select a fixed number of destination nodes per source node. This can result in uniform out-degrees across all nodes, disrupting the original degree distribution, such as the long-tail distribution in power-law graphs [45].

To address this limitation, we construct a global graph representation, the GMFG, which remains constant during training and encompasses the original graph's complete information. Similar to the LMFG, the GMFG follows the MFG's layer-wise construction by tracking training nodes in the message-passing path. However, unlike the LMFG, which only monitors the current epoch's message-passing path, the GMFG sampling all training nodes, thereby tracking all the message passing paths.

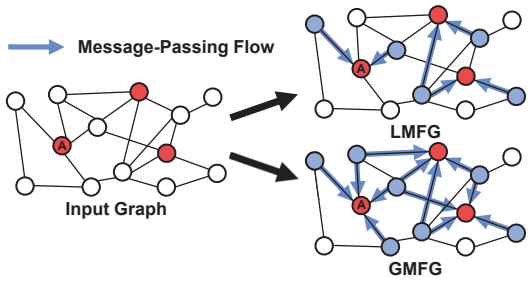

Figure 7: The construction process of LMFG and GMFG.

For instance, in Figure 7, node A has only two message passing paths in the LMFG, while it has four in the GMFG. This comprehensive sampling ensures that the GMFG is solely dependent on the dataset itself and the number of model layers, where the dataset provides the original graph information and the number of layers dictates the hops of message passing. Furthermore, comprehensive sampling guarantees that the edge information of nodes in the GMFG remains consistent with the edge information of the corresponding nodes in the original graph, thereby preserving the original degree distribution. Consequently, we adapt our Out-Degree Centric Graph Partitioning methodology based on GMFG.

In this refined approach, instead of utilizing a new LMFG for each epoch, we apply the proposed graph partitioning algorithm directly on the GMFG to create a batched list of reusable micro-batch training nodes. This strategy allows one-time partitioning results can be reused across all training epochs, eliminating the repetitive partitioning during the

micro-batch GNN training process. Importantly, the transition from LMFG to GMFG does not compromise the effectiveness of our partitioning technique in reducing redundancy and load imbalances, as the GMFG preserves the same essential partitioning-aided properties as LMFG.

4.3.2 ***Optimizing the data loading efficiency.*** As discussed in Section 3.3, Betty's data loading is inefficient due to cumbersome data extraction and index mapping. To address this, Cherry-GMFG introduces a micro-batch specific data loading mechanism to streamline the data loading process. Leveraging DGL's optimized data loader, we create a dedicated Micro-Batch Loader (MBL) for each micro-batch. As mandated by DGL, constructing a DGL's *DataLoader Class* requires fixed training nodes [40] which can be easily satisfied by GMFG-based out-degree centric partitioning.

Algorithm 2 illustrates the generation process of MBLs, The GMFG is constructed first with the training nodes and the input graph (Line 1). With the constructed GMFG, Cherry-GMFG performs the out-degree centric partitioning to obtain the *batched training nodes list* (Line 2). Then, the algorithm iterates through this list to instantiate a dedicated MBL for each micro-batch (Line3-6), and finally returns a list of MBLs (Line 7). These MBLs can be reused across training epochs for micro-batch based data loading.

Algorithm 2: The Generation of Micro-Batch Data Loaders

```
   input  : Input Graph: g
            training nodes: t_nids
            The number of micro-batches: K
   output : micro_batch_loaders
1  GMFG = Full_MFG_construction(g, t_nids);
2  batched_train_nodes_list =
     Out_Degree_Centric_Partitioning(GMFG, K);
3  for node_list in batched_train_nodes_list do
4  |   dataloader = gen_dataloader(node_list, g);
5  |   micro_batch_loaders.append(dataloader);
6  end
7  return micro_batch_loaders;
```

Within Cherry-GMFG, each MBL operates as an iterable entity, shuffling data independently to segregate data loading for distinct micro-batches. This ensures that Cherry's data loading is independent of the full-batch data loader, eliminating cumbersome data extraction and index mapping procedures as in Betty. All MBLs are stored in host memory and work on the same input graph as the full-batch data loader. We observe that MBLs add only tens of megabytes of extra memory overhead, which is acceptable in practical that host memory resources are generally abundant.

Notably, using MBLs, the same training node may aggregate features from distinct neighbors compared to using the full-batch loader due to their different shuffling process for training nodes and neighboring nodes. Each MBL shuffles nodes independently, whereas the full-batch loader shuffles all nodes at once. Despite this difference, since MBLs operate on the same input graph as the full-batch loader, the neighboring nodes and their features for each training node remain consistent across methods, ensuring similar model convergence (validated in Section 6.3).

5 Implementation

Cherry[1] is implemented using DGL 2.0.0 [40] and PyTorch 2.0.1 [28] for GNN training. It utilizes DGL's APIs for GMFG construction, out-degree centric partitioning, and micro-batch data loading. Cherry can be used as a Python module compatible with DGL, enabling large-scale GNN training beyond the memory capacity wall.

GMFG construction. Cherry constructs the GMFG by tracing the message-passing process for all training nodes. It creates a dgl.dataloader with sampling nodes covering all neighbors, enabling iteration through the dataloader layer by layer to retrieve neighboring nodes used in message-passing for training nodes. This process forms a dgl.graph representing the GMFG.

Out-degree centric graph partitioning. Cherry uses dgl.graph.edges()[0] to obtain source node indices and calculates their out-degrees with dgl.graph.out_degrees(). These out-degrees serve as edge weights after removing self-loops by dgl.rm_self_loop(). Subsequently, the graph is partitioned via dgl.metis(), and the partitions are iteratively handled to extract training nodes for micro-batches.

Micro-batch data loader. Cherry utilizes training nodes and the original graph to create a dgl.dataloader for each micro-batch and are then aggregated into a list of micro-batch data loaders, facilitating efficient retrieval of data for each micro-batch layer by layer during micro-batch training.

Multi-GPU support. Since we focus on using single-GPU scenario to demonstrate Cherry's ability in breaking the memory capacity wall, the experimental version of Cherry is single-GPU based. Note that Cherry is not restricted to single-GPU setup. Cherry can be easily adapted to a data-parallel multi-GPU environment due to the independence between MBLs. Specifically, we need to convert the model into a torch.distributed model and utilize dgl.dist_graph and dgl.dist_dataloader to construct MBLs. Subsequently, we must establish the correspondence between MBLs and GPUs, ensuring that each MBL is exclusively assigned to a single GPU. Other computational processes remain consistent with those on a single GPU.

[1] https://github.com/cherryWangY/Cherry

6 Evaluation

In this section, we compare *Cherry-LMFG* and *Cherry-GMFG* to three micro-batching based methods: DGL-Random, DGL-Metis, Betty. We first describe the experimental setup and then we test the overall performance of all methods in terms of memory and time consumption. Then we evaluate the accuracy and convergence of micro-batch training. Next, we evaluate node redundancy, load balance, computation efficiency. Finally, we compare our micro-batching method with mini-batching. Notably, we only compare Cherry-LMFG and Cherry-GMFG in Section 6.2. In other experiments, we use Cherry-GMFG due to its better overall performance.

6.1 Experimental Setup

Platform and Tools. We use a system with a 16-core AMD EPYC 7543 CPU at 2.80 GHz and an Nvidia A100 GPU. The system runs on CentOS 7.9, offering 40GB of GPU memory and 256GB of host memory. We use CUDA 11.8/cuDNN 8.7 for GPU computation. For model training, we use Python 3.10, Pytorch v2.0.1, and DGL v2.0.0.

Table 2: Training datasets

Dataset	#Node	#Edge	Avg Degree	Feature	Classes
Reddit	232,965	114,615,892	492	602	41
Ogbn-Arxiv	169,343	2,315,598	13.7	128	40
Ogbn-Products	2,400,608	123,718,024	51.5	100	47
Amazon	1,569,960	264,339,468	168	200	107
Ogbn-Papers100M	111,059,956	1,615,685,872	14.5	128	172

Datasets. As shown in Table 2, our evaluation includes five real-world large datasets: *Reddit*, *Ogbn-Arxiv*, *Ogbn-Products*, *Amazon*, and *Ogbn-Papers100M*, each with distinct features relevant to its domain. The *Avg Degree* column indicates the average node degree, reflecting the graph's density by the ratio of edges to nodes.

Models. We use three representative GNN models: GraphSAGE [11], GCN [17], and GAT [38]. The neighbor sampling algorithm [11] is utilized for graph input sampling. The aggregator used in GraphSage is Mean, unless specified otherwise.

Baselines.

We compare Cherry-GMFG, incorporating all the proposed techniques, against four micro-batch baselines: (1) The **DGL-Random** splits full-batch nodes into n sequential random segments for micro-batches, implemented using DGL as an oracle partition method. (2) **DGL-Metis [16]** ,a well-known graph partitioning algorithm for its parallel processing efficiency in handling large graph. We use DGL' API to apply Metis in our experiments. (3) **Betty [46]** ,the SOTA micro-batch GNN training method, with redundancy-embedded graphs and memory-aware partitioning to mitigate redundancy and load imbalance. We use its open-source code in our evaluation. (4) **Cherry-LMFG**(Section 4.2) uses LMFG based out-degree centric partitioning and the same data loading technique as Betty.

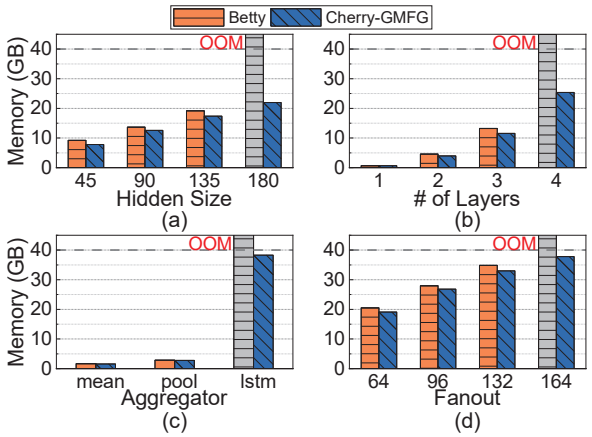

Figure 8: Cherry vs. Betty in memory-consuming cases.

6.2 Overall Performance

6.2.1 Breaking the memory capacity wall. We evaluate Cherry-GMFG against Betty in four memory-consuming cases, each employing an advanced GNN training method: wider hidden size, deeper aggregation, sophisticated aggregator, or larger fanout degree. In all experiments shown in Figure 8, we use Ogbn-Products dataset and 4 micro-batches both for Betty and Cherry. In Figure 8(a), with a 3-layer GAT model, Betty fails with a hidden size of 180, while Cherry trains successfully. In Figure 8(b), with a GAT model and hidden size of 80, Betty encounters OOM error in the 4-layer model, while Cherry enables this case. In Figure 8(c), with a 2-layer GraphSAGE model and a hidden size of 192, Cherry is able to train with the LSTM aggregator while Betty fails. In Figure 8(d), with a 1-layer GraphSAGE model, an LSTM aggregator, and a hidden size of 256, Cherry enables training with a fanout degree of 164 while Betty encounters OOM error. These results highlight Cherry's superiority in overcoming memory constraints.

6.2.2 Peak Memory Consumption. Figure 9 compares the peak GPU memory consumption during training for Random, Metis, Betty, Cherry-LMFG, and Cherry-GMFG across three GNN models (GraphSAGE, GCN, GAT) and five datasets. We use torch.max_memory_allocated to record the maximum GPU memory allocated during training, a critical metric for large-scale GNN training on GPU. Each model employs a three-layer architecture with a hidden size of 256 for GCN and GraphSAGE, and 4 attention heads with a hidden size of 128 for GAT. All experiments use 8 micro-batches.

Figure 9 shows that both Cherry-LMFG and Cherry-GMFG consistently exhibit the lowest memory consumption across

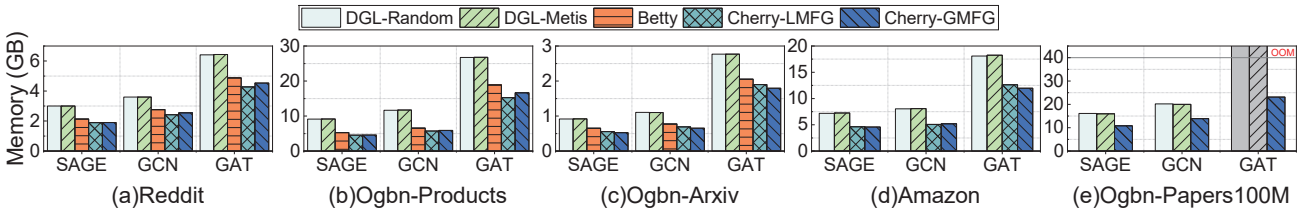

Figure 9: The peak memory consumption of DGL-Random, DGL-Metis, Betty, Cherry-LMFG, and Cherry-GMFG for training three GNN models on five datasets.

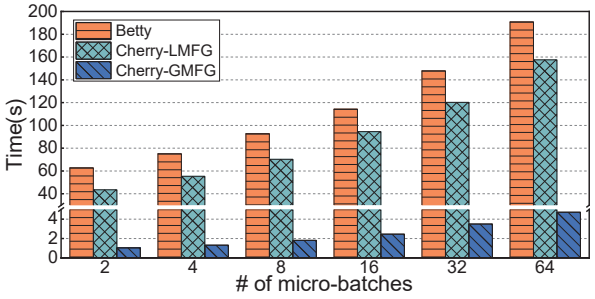

Figure 10: The training time (with data preparing) per epoch of Betty, Cherry-LMFG, and Cherry-GMFG.

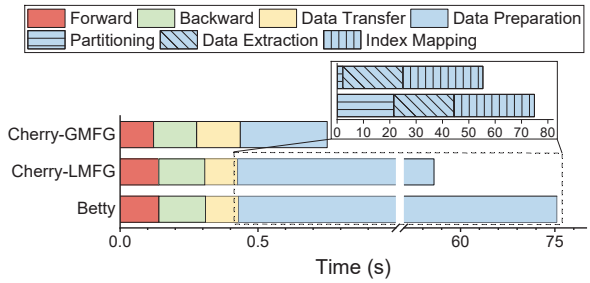

Figure 11: The breakdown of training time (with data preparing) per epoch for Cherry and Betty.

all cases. Despite minor variations in some cases, their overall memory consumption remains comparable. Specifically, Cherry-GMFG reduces memory usage by 37.6% and 37.4% on average compared to Random and Metis, and by 10.8% on average (up to 18.8%) compared to Betty. This demonstrates Cherry's superiority in reducing peak memory usage thanks to its efficient partitioning strategy addressing node redundancy and load imbalance.

Notably, in our experiments, Betty fails to handle the two largest datasets, Amazon and Ogbn-Papers100M, because (1) it exhausts 256GB of main memory during graph partitioning on the Amazon, likely due to the excessive redundant edges as discussed in Section 3.3. (2) Betty's data loading technique encounters an index mapping issue on Ogbn-Papers100M due to node length mismatches from errors in Betty's bookmarking process related to the large dataset size. Consequently, Cherry-LMFG, which employs the same data loading technique, also fails on Ogbn-Papers100M. In contrast, Cherry-GMFG successfully manages all datasets, demonstrating Cherry's practicability for GNN training on real-world large graphs.

6.2.3 Training Time. We evaluate the training efficiency of three memory-efficient methods (Betty, Cherry-LMFG and Cherry-GMFG) in terms of training time. Figure 10 compares the per-epoch training time (with data preparation) using a three-layer GraphSAGE on ogbn-products across different batch numbers. We run each experiment 50 epochs and discard the first 10 warm-up epochs to average the results. Cherry outperforms all baselines in all cases significantly. Specifically, Cherry-GMFG reduces training time by 90.5% on average (up to 96.2%) compared to Cherry-LMFG and by 94.9% on average (up to 98.8%) compared to Betty. This highlights Cherry's superior training efficiency, making it a more practical micro-batch training solution for large-scale GNN training.

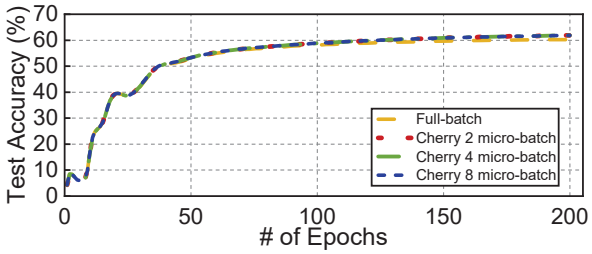

Figure 12: Convergence curves for full-batch and micro-batch training with three different numbers of batches.

To further dissect Cherry's significant training time reduction, we break down the training time per epoch for

Cherry-GMFG, Cherry-LMFG, and Betty using the Ogbn-Products dataset with a 3-layer GraphSAGE model, hidden size of 256, and 4 micro-batches. Figure 11 shows the results. Given that Cherry's one-time partitioning overhead is amortized across successive training epochs, we do not show the partitioning time for Cherry. We see that Cherry has a far lower data preparation time than both Betty and Cherry-LMFG. Such huge data preparation time reduction is mainly because Cherry circumvents those time-consuming operations (repetitive partitioning, data generation, and index mapping) involved in Betty's data preparation phase by employing a micro-batch specific data loading and a repetitive-partitioning elimination mechanism. Moreover, Cherry reduces the time for forward pass and backward pass by 18.8% and 14.1%, respectively, compared to Betty. This mainly comes from Cherry's more sufficient node redundancy reduction as verified in Section 6.4.2.

6.3 Training Accuracy and Convergence

Table 3: DGL vs. Cherry in training accuracy

Dataset	Model	DGL/Acc(%)	Cherry/Acc(%)
Reddit	GraphSAGE	86.14 ± 0.28	86.59 ± 0.37
	GCN	94.34 ± 0.61	94.13 ± 0.32
	GAT	89.64 ± 1.85	89.71 ± 1.97
ogbn-arxiv	GraphSAGE	71.32 ± 0.09	71.07 ± 0.05
	GCN	68.51 ± 0.33	68.79 ± 0.08
	GAT	69.67 ± 0.21	69.31 ± 0.17
ogbn-products	GraphSAGE	74.39 ± 0.33	76.18 ± 0.17
	GCN	75.51 ± 0.24	75.87 ± 0.13

Table 3 presents the training accuracy of DGL using full-batch training and Cherry using micro-batch training across different datasets and models. As GAT cannot use ogbn-products dataset, no results are shown for it. Also, no results are shown for Amazon and Ogbn-Papers100M datasets as DGL fails on our platform. The results indicate that micro-batch training enabled by Cherry achieves comparable training accuracy to full-batch training, as they are mathematically equivalent.

To evaluate whether Cherry affects convergence, we sample accuracy per epoch to show GCN's convergence curves on Ogbn-Arxiv. Figure 12 compares Cherry's micro-batch training with 2, 4, and 8 batches to full-batch training under identical hyperparameters. The figure shows nearly identical convergence curves for both methods. This confirms Cherry's micro-batch training maintains full-batch convergence behavior without losing accuracy. Similar results are observed across all other datasets and models.

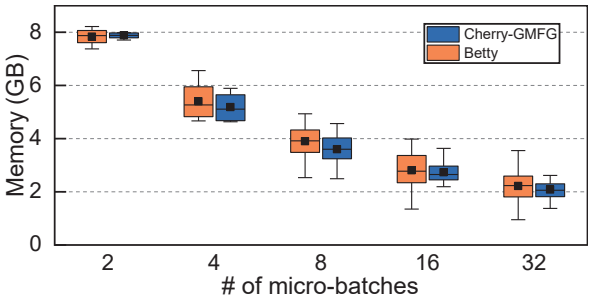

Figure 13: Betty vs. Cherry in load balancing with different numbers of micro-batches.

6.4 Load Balance and Redundancy Reduction

6.4.1 Load Balance. We evaluate Cherry's effectiveness in load balancing. Figure 13 shows memory consumption variation across micro-batches using GraphSAGE on Ogbn-Products. Cherry consistently shows narrower memory fluctuation ranges, indicating lower variation within micro-batches than Betty. This advantage grows as batch numbers increases. For instance, from 16 batches to 32 batches, Cherry shows reduced memory fluctuation, while Betty shows a slightly larger fluctuation. This indicates Cherry's superiority in load balancing. By balancing memory consumption across micro-batches, Cherry reduces peak memory usage significantly, facilitating larger-scale GNN training without encountering OOM errors, as shown in Figure 8. In contrast, Betty requires more batches to avoid OOM errors in this case, sacrificing training efficiency.

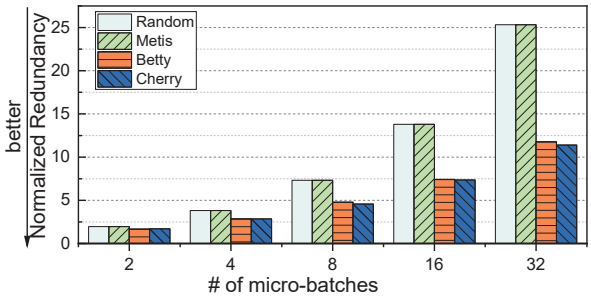

Figure 14: Normalized redundancy of Random, Metis, Betty, and Cherry across different numbers of micro-batches.

6.4.2 Redundancy Reduction. We evaluate Cherry's effectiveness in reducing redundancy. Figure 14 shows normalized redundancy results using a three-layer GraphSAGE model on Ogbn-Products. Normalized redundancy, ratio of the sum of nodes in micro-batches to the total nodes in the full batch,

indicates node redundancy after partitioning. Cherry-GMFG reduces redundant nodes by 13.2% to 54.9% compared to Random and Metis. Even compared to Betty, Cherry achieves a 3.7% average reduction. Such advantage becomes particularly pronounced as the number of batches increases. Overall, Cherry maintains lower normalized redundancy across all configurations, showcasing its superiority in reducing redundancy, especially with more micro-batches.

6.5 Computation Efficiency

We evaluate Cherry's computation efficiency against Betty. Computation efficiency is defined as the inverse of computation overhead. The fluctuation in computation efficiency across different epochs during the training process reflects the computation stability, i.e., less fluctuation indicates better computation stability. Figure 15(a) compares computation efficiency of Cherry-GMFG and Betty over 50 epochs when training GraphSAGE model on the Ogbn-Products dataset with 4 batches. We see that Cherry's computation efficiency remains steady at around 0.473, while Betty's fluctuates significantly between 0.448 and 0.474 due to the million-level computation nodes variations. This indicates that Cherry has significantly more stable and higher computation efficiency over Betty.

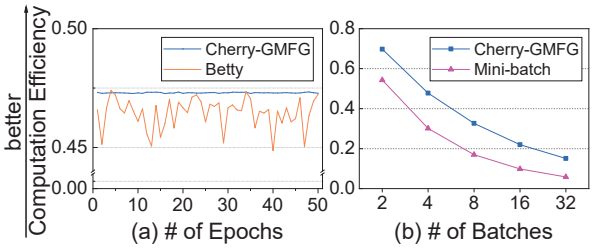

Figure 15: Comparison of computation efficiency among Cherry, Betty, and mini-batch.

6.6 Time and Memory Reduction Comparing with Mini-batch

Table 4 illustrates the performance differences between mini-batch and Cherry-enabled micro-batch under the same batch size, utilizing the Ogbn-Products dataset with a 3-layer GraphSAGE model as a case study. Compared to mini-batch training, micro-batch training achieves an average reduction of 32.7% (up to 53.9%) in training time (with data preparing), and 34.0% (up to 40.1%) in memory consumption. Moreover, these advantages become increasingly pronounced as the number of batches increases. Figure 15(b) also shows that Cherry achieves an average of 1.9x (up to 2.6x) improvement computational efficiency compared to mini-batch training. The reduction in memory consumption primarily results from decreased node redundancy and improved load balancing, while the training time reduction is attributed to improved computational efficiency, efficient data loading and the elimination of re-partitioning. This highlights the effectiveness of Cherry-enabled micro-batch training.

Table 4: Time and memory consumption between micro-batch and mini-batch training.

	Training time per epoch (sec)		CUDA Memory consumption (GB)	
# of batches	Mini-batch	Micro-batch	Mini-batch	Micro-batch
2	1.019	1.031	10.748	7.982
4	1.743	1.311	8.959	5.883
8	2.933	1.815	7.271	4.567
16	4.728	2.457	6.061	3.632
32	7.584	3.494	4.791	3.216

7 Related Work

GNN Frameworks and Graph Partitioning. A number of GNN systems [1, 4, 5, 10, 13, 15, 23, 25, 26, 40, 54, 56] have emerged in recent years, with DGL [40] being a popular framework. DistDGL [54], its distributed version, utilizes the Metis partition algorithm [16] to reduce inter-GPU communication. Cluster-GCN [4] constructs small clusters using METIS for SGD updates. AliGraph [56] and DistGNN [26] follow similar strategies for improved performance. NeuGraph [25] maximizes edge numbers in diagonal chunks using the Kernighan-Lin algorithm. Roc [15] achieves balanced partitioning with a linear-regression-based approach but may lead to unbalanced computational loads. WiseGraph [13] explores partitioning graph data and GNN operations jointly. PaGraph [23] partitions graphs based on a training vertex's neighborhood but incurs high memory overhead. DGCL [1] distributes physical edges and features among machines using a partitioning algorithm. PyG [5] employs feature-based partitioning for distributed GNN learning.

The above graph partitioning efforts mostly focus on balancing loads between GPUs. In contrast, Cherry identifies the significant impact of both node redundancy and load imbalance on GNN training efficiency, and minimizes both simultaneously via an out-degree centric partitioning strategy.

Memory Optimization for NN Training. Various methods [2, 6, 29, 32, 33, 37] have been proposed to tackle memory challenges in traditional NN training. Capuchin [29] enhances memory management by monitoring tensor access dynamically. ZeRO-Offload [33] offloads tasks to the CPU for large models. ZeRO-Infinity [32] scales NN model training using GPU, CPU, and NVMe memory. Zico [22] tracks memory usage during GPU computation for efficient management. FlexNN [21] optimizes memory utilization through joint planning of slicing-loading-computing. However, their

direct applicability to GNNs operating on large-scale graphs is limited.

Several studies use off-chip memory to reduce GPU memory usage for GNN training. P3 [7] minimizes network communication overhead to utilize host memory for graph storage. MariusGNN [39] utilizes the entire storage hierarchy, including SSDs, for large-scale GNN training. BeaconGNN [43] enhances training throughput with a multi-tier flash memory architecture. PyTorch-BigGraph [19] reduces GPU memory usage by swapping partition embeddings to the hard drive. HongTu [41] stores vertex data in CPU memory and offloads training to the GPU for billion-scale full-graph training. In contrast, Cherry focuses on micro-batching to lower GPU memory consumption, which is orthogonal to these systems.

Graph Sampling. Graph sampling algorithms are essential in GNNs to reduce input size and computational complexity. Node-wise sampling methods like GraphSAGE [11] and VR-GCN [48] sample neighboring nodes. Layer-wise sampling techniques, such as FastGCN [3] and AS-GCN [50], sample a fixed number of nodes per layer using pre-computed probabilities. Subgraph-based sampling approaches like GraphSAINT [51] and Cluster-GCN [4] partition graph for training. However, these sampling methods may reduce memory requirements at the cost of potential model accuracy loss.

Micro-Batch Training. GPipe [14] and μ-cuDNN [27] first introduce micro-batching to optimize traditional NN training via a pipeline approach. Works like MBP [30], Melon [42] and Sage [8] extend micro-batching to for memory efficiency and larger batch sizes for NN models. However, applying micro-batching to GNNs faces challenges due to complex dependencies (e.g., N:M) between outputs/labels and input features within a batch, unlike the 1:1 mapping in traditional NNs. Betty [46] is the first method to apply micro-batching in GNN training, effectively addressing the memory capacity bottleneck. It formulates micro-batch partitioning as a multi-level bipartite graph partitioning problem and proposes a graph partitioning algorithm based on a redundancy-embedded graph to reduce node redundancy. However, as mentioned in Section 3, Betty improves load balancing by sacrificing computational efficiency and suffers from high data preparation overhead. Buffalo [47] is built upon Betty and solves Betty's bucket explosion problem through bucket partitioning. However, due to their shared design paradigm, Buffalo still suffers from the same repartitioning and data extraction overhead as Betty.

8 Conclusion

Despite the widespread success of GNN in graph-based applications, training GNN with sophisticated strategies on large graphs has long been a challenging endeavor, primarily due to the memory scalability and computational efficiency concerns. This paper introduces Cherry, a novel micro-batching method tailored to GNN that makes large-scale GNN training more efficient and accessible. Cherry incorporates message-passing flow graph-aided partitioning techniques and micro-batch-specific data loading mechanism, effectively alleviating the redundancy and load imbalance issues as well as reducing the overhead associated with data preparation, which traditionally plague micro-batch based GNN training. Evaluation results show the superiority of Cherry over existing training methods, highlighting its potential to significantly enhance the scalability and efficiency of large-scale GNN training.

9 Acknowledgment

We sincerely thank our anonymous reviewers for their valuable comments. We thank Dr. Wenqian Dong and Dr. Mingzhen Li for the valuable feedback to the paper. This work is supported by the following funding: Natural Science Foundation of Guangdong Province (2022A1515110073), National Natural Science Foundation of China (Grant Nos. 62032023, T2125013, 92270206, 62372435, 61972377, and T2293702), Innovation Funding of ICT, CAS (Grant No. E461050), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-005).

References

[1] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: An efficient communication library for distributed GNN training. In *Proceedings of the 16th European Conference on Computer Systems*. 130–144.

[2] Hao Chen, Yuanchen Bei, Qijie Shen, Yue Xu, Sheng Zhou, Wenbing Huang, Feiran Huang, Senzhang Wang, and Xiao Huang. 2024. Macro graph neural networks for online billion-scale recommender systems. In *Proceedings of the ACM on Web Conference 2024*. 3598–3608.

[3] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247* (2018).

[4] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 257–266.

[5] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[6] Matthias Fey, Jan E Lenssen, Frank Weichert, and Jure Leskovec. 2021. Gnnautoscale: Scalable and expressive graph neural networks via historical embeddings. In *International conference on machine learning*. 3294–3304.

[7] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation*. 551–568.

[8] In Gim and JeongGil Ko. 2022. Memory-efficient dnn training on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 464–476.

[9] Ping Gong, Renjie Liu, Zunyao Mao, Zhenkun Cai, Xiao Yan, Cheng Li, Minjie Wang, and Zhuozhao Li. 2023. gSampler: General and efficient

GPU-based graph sampling for graph learning. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 562–578.

[10] Daniele Grattarola and Cesare Alippi. 2021. Graph neural networks in tensorflow and keras with spektral application notes. *IEEE Computational Intelligence Magazine* 16, 1 (2021), 99–106.

[11] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[12] Yaochen Hu, Amit Levi, Ishaan Kumar, Yingxue Zhang, and Mark Coates. 2021. On Batch-size Selection for Stochastic Training for Graph Neural Networks. (2021).

[13] Kezhao Huang, Jidong Zhai, Liyan Zheng, Haojie Wang, Yuyang Jin, Qihao Zhang, Runqing Zhang, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2024. WiseGraph: Optimizing GNN with Joint Workload Partition of Graph and Operations. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1–17.

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[15] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems*, Vol. 2. 187–198.

[16] George Karypis. 1997. METIS: Unstructured graph partitioning and sparse matrix ordering system. *Technical report* (1997).

[17] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[18] John Boaz Lee, Ryan Rossi, and Xiangnan Kong. 2018. Graph classification using structural attention. In *Proceedings of the 24th ACM International Conference on Knowledge Discovery & Data Mining*. 1666–1674.

[19] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-BigGraph: A Large Scale Graph Embedding System. In *Proceedings of Machine Learning and Systems*, Vol. 1. 120–131.

[20] Guohao Li, Chenxin Xiong, Ali Thabet, and Bernard Ghanem. 2020. Deepergcn: All you need to train deeper gcns. *arXiv preprint arXiv:2006.07739* (2020).

[21] Xiangyu Li, Yuanchun Li, Yuanzhe Li, Ting Cao, and Yunxin Liu. 2024. FlexNN: Efficient and Adaptive DNN Inference on Memory-Constrained Edge Devices. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 709–723.

[22] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. 2021. Zico: Efficient GPU memory sharing for concurrent DNN training. In *2021 USENIX Annual Technical Conference*. 161–175.

[23] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 401–415.

[24] Zhilong Lu, Weifeng Lv, Zhipu Xie, Bowen Du, and Runhe Huang. 2019. Leveraging graph neural network with lstm for traffic speed prediction. In *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 74–81.

[25] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference*. 443–458.

[26] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. 2021. Distgnn: Scalable distributed training for large-scale graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[27] Yosuke Oyama, Tal Ben-Nun, Torsten Hoefler, and Satoshi Matsuoka. 2018. Accelerating deep learning frameworks with micro-batches. In *2018 IEEE International Conference on Cluster Computing*. IEEE, 402–412.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[29] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 891–905.

[30] XinYu Piao, DoangJoo Synn, JooYoung Park, and Jong-Kook Kim. 2021. Micro Batch Streaming: Allowing the Training of DNN Models to Use a large Batch Size in Memory Constrained Environments. *arXiv preprint arXiv:2110.12484* (2021).

[31] XinYu Piao, DoangJoo Synn, JooYoung Park, and Jong-Kook Kim. 2023. Enabling Large Batch Size Training for DNN Models Beyond the Memory Limit While Maintaining Performance. *IEEE Access* (2023).

[32] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–14.

[33] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. Zero-offload: Democratizing billion-scale model training. In *2021 USENIX Annual Technical Conference*. 551–564.

[34] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. 2019. Dropedge: Towards deep graph convolutional networks on node classification. *arXiv preprint arXiv:1907.10903* (2019).

[35] Andrea Rossi, Denilson Barbosa, Donatella Firmani, Antonio Matinata, and Paolo Merialdo. 2021. Knowledge graph embedding for link prediction: A comparative analysis. *ACM Transactions on Knowledge Discovery from Data* 15, 2 (2021), 1–49.

[36] Chenyang Si, Wentao Chen, Wei Wang, Liang Wang, and Tieniu Tan. 2019. An attention enhanced graph convolutional lstm network for skeleton-based action recognition. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 1227–1236.

[37] Zhanhong Tan, Zijian Zhu, and Kaisheng Ma. 2024. Cocco: Hardware-Mapping Co-Exploration towards Memory Capacity-Communication Optimization. *arXiv preprint arXiv:2402.00629* (2024).

[38] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[39] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2023. Mariusgnn: Resource-efficient out-of-core training of graph neural networks. In *Proceedings of the 18th European Conference on Computer Systems*. 144–161.

[40] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural

networks. *arXiv preprint arXiv:1909.01315* (2019).

[41] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. 2023. HongTu: Scalable Full-Graph GNN Training on Multiple GPUs. *Proc. of the ACM on Management of Data* 1, 4 (2023), 1–27.

[42] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. 2022. Melon: Breaking the memory wall for resource-efficient on-device machine learning. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 450–463.

[43] Yuyue Wang, Xiurui Pan, Yuda An, Jie Zhang, and Glenn Reinman. 2024. BeaconGNN: Large-Scale GNN Acceleration with Out-of-Order Streaming In-Storage Computing. In *Proceedings of 2024 IEEE International Symposium on High-Performance Computer Architecture*. 330–344.

[44] Shunxin Xiao, Shiping Wang, Yuanfei Dai, and Wenzhong Guo. 2022. Graph neural networks in node classification: survey and evaluation. *Machine Vision and Applications* 33, 1 (2022), 4.

[45] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed power-law graph computing: Theoretical and empirical analysis. *Advances in neural information processing systems* 27 (2014).

[46] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. 2023. 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*. 103–117.

[47] Shuangyan Yang, Minjia Zhang, and Dong Li. 2025. Buffalo: Enabling Large-Scale GNN Training via Memory-Efficient Bucketization. In *Proceedings of the 2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[48] Rui Ye, Xin Li, Yujie Fang, Hongyu Zang, and Mingzhong Wang. 2019. A vectorized relational graph convolutional network for multi-relational network alignment.. In *Proceedings of International Joint Conference on Artificial Intelligence*. 4135–4141.

[49] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 974–983.

[50] Zhizhi Yu, Di Jin, Ziyang Liu, Dongxiao He, Xiao Wang, Hanghang Tong, and Jiawei Han. 2021. AS-GCN: Adaptive semantic architecture of graph convolutional networks for text-rich networks. In *2021 IEEE International Conference on Data Mining*. 837–846.

[51] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931* (2019).

[52] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *Advances in neural information processing systems* 31 (2018).

[53] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[54] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE, 36–44.

[55] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4582–4591.

[56] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. Aligraph: A comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730* (2019).