Betty: Enabling Large-Scale GNN Training with Batch-Level Graph Partitioning

Shuangyan Yang
University of California, Merced
Merced, California, USA
syang127@ucmerced.edu

Minjia Zhang
Microsoft Research
Redmond, WA, USA
minjiaz@microsoft.com

Wenqian Dong
University of California, Merced and Florida International University
Miami, Florida, USA
wdong@fiu.edu

Dong Li
University of California, Merced
Merced, California, USA
dli35@ucmerced.edu

ABSTRACT

The Graph Neural Network (GNN) is showing outstanding results in improving the performance of graph-based applications. Recent studies demonstrate that GNN performance can be boosted via using more advanced aggregators, deeper aggregation depth, larger sampling rate, etc. While leading to promising results, the improvements come at a cost of significantly increased memory footprint, easily exceeding GPU memory capacity.

In this paper, we introduce a method, Betty, to make GNN training more scalable and accessible via batch-level partitioning. Different from DNN training, a mini-batch in GNN has complex dependencies between input features and output labels, making batch-level partitioning difficult. Betty introduces two novel techniques, redundancy-embedded graph (REG) partitioning and memory-aware partitioning, to effectively mitigate the redundancy and load imbalances issues across the partitions. Our evaluation of large-scale real-world datasets shows that Betty can significantly mitigate the memory bottleneck, enabling scalable GNN training with much deeper aggregation depths, larger sampling rate, larger training batch sizes, together with more advanced aggregators, with a few as a single GPU.

CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Computing methodologies** → **Neural networks**.

KEYWORDS

Efficient training method; Heterogeneous memory; Graph neural network; Graph partition; Redundancy elimination; Load balancing

ACM Reference Format:
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 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575725

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575725

1 INTRODUCTION

Graph neural network (GNN) has emerged as an effective paradigm to learn rich relation and interaction information in irregular graph-based structures. Since its debut, it has led to accuracy breakthroughs in a wide range of tasks such as link prediction [16], node and graph classification [37, 41], visualization [2], and clustering and community detection [25].

Recent efforts show that GNN training efficiency or accuracy can be improved by using larger batch sizes (e.g., from mini-batch training to full-batch training) [10, 13], training with more sophisticated aggregators (e.g., LSTM and attention networks) [23, 38], increasing aggregation depth (e.g., from 1 to 112) [20], using a larger sampling rate (i.e., to include more neighbors for aggregation) [45], or using deeper and wider neural encoders [20]. However, despite leading to promising results, the improvements often come at a cost of significantly increased memory consumption. For example, GNNs learn vector representations of nodes by recursively aggregating features of their neighboring nodes, increasing the aggregation depth would make the number of feature vectors that need to be loaded into memory for aggregation grow exponentially. Given that device memory of the hardware accelerator (e.g., GPU) is a scarce resource, the methods such as deep aggregation can easily run into scalability challenges. As such, many methods are applied to only small-scale graphs (e.g., hundreds of thousands of nodes) or with a shallow structure (e.g., less than three layers) in GNN models to explore the node dependencies before running out of memory.

To work around the memory capacity bottleneck, prior work explored both algorithmic and system optimizations. On the algorithm side, a popular method is sampling [21, 42, 44], which samples a subset of neighbors to compute the feature for a given node/subgraph. By reducing the sampling rate, the number of neighbors participating in aggregation is reduced, leading to reduced memory consumption. However, this method requires careful consideration of the sampling strategy and may cause loss of important neighbor information that hurts the final model accuracy.

On the system side, researchers and practitioners have also built specialized GNN frameworks that optimize the GNN training efficiency. For example, state-of-the-art GNN training frameworks, such as DGL [3], PyTorch Geometric [30], and NeuGraph [24], support convenient and highly efficient graph operation primitives (e.g., aggregators) in terms of compute and memory efficiency. However, as the batch size or aggregation depth increases, especially when using more memory intensive aggregators, GNN training can still run out of memory. Frameworks such as DGL also support distributed training for GNN (DistDGL [44]), where they partition the graph across multiple GPUs and/or multiple nodes, and leverage the aggregated memory from multiple GPUs to scale out the GNN training. While being an effective approach, these methods often increase the hardware cost of training GNN models significantly. For example, training Relational Graph Convolution Network (RGCN) [36] with mag-lsc (a large graph with 7B edges 240M vertices [8]) needs 32 NVIDIA T4 GPUs [45] (16 GB memory and more than $2,000 per GPU), which could be a barrier for many model scientists who lack access to many GPUs.

In this work, we analyze GNN scalability bottlenecks and identify that feature vectors and their corresponding hidden maps involved in aggregation as a major source of memory consumption for GNN training. To reduce the memory usage, one effective method is applying batch-level partitioning, where a mini-batch is partitioned into K micro-batches. The GNN model then calculates partial gradients for each micro-batch and uses accumulative gradients of all micro batches to update the whole model weights. The benefit of the batch-level partitioning is that it allows the model to train the same effective batch size but with a reduced memory footprint, and the model convergence and quality will not be affected by the change of batch sizes.

However, the batch-level partitioning is challenging for GNN. Unlike traditional deep learning training, where a mini-batch consists of inputs and labels that fall into a 1:1 mapping relationship, the output/labels and input features in GNN have much more complicated dependencies (e.g., N:M). In particular, this complex dependency creates two challenges: (1) *high redundancy*. A naive partitioning scheme would create a large amount of redundancy across micro-batches, because an input node might be duplicated into multiple micro-batches if it is a shared neighbor across several nodes that get partitioned into different micro-batches. (2) *Load imbalance*. Most micro-batches would have a similar amount of memory consumption, but there could be one or a few micro-batches that have an unbalanced load (usually caused by partition strategies we selected), which will lead to an increase of the maximum memory footprint on device.

To tackle these challenges, we formulate the batch-level partitioning problem for GNN as a multi-level bipartite graph partitioning problem, and introduce two novel techniques to partition a batch. (1) Betty constructs a redundancy-embedded graph and transforms the redundancy reduction problem to a min-cost flow cut problem. The later can be effectively solved via a generic graph partitioning algorithm. (2) To reduce the maximal memory footprint, Betty introduces a memory-aware partitioning algorithm which partitions a batch based on accurate estimation of the memory usage of GNN batches. Together, Betty effectively reduces the memory footprint while minimizing the redundancy across micro-batches, making it feasible to train GNN with deeper aggregation, larger sampling rate, or bigger GNN networks.

In summary, we make the following contributions.

- We conduct an analysis of the memory bottleneck in GNN training and identify opportunities and challenges of reducing the memory usage of GNN via batch-level partitioning.
- We formulate the batch-level partitioning as a multi-level bipartite graph partitioning problem and provide system support to partition a batch while allowing it to achieve the same accuracy with reduced memory footprint, without requiring any hyperparameter changes.
- We introduce two novel techniques, redundancy-embedded graph and memory-aware partitioning, to reduce the redundancy while effectively mitigating the load imbalance issue from the batch-level partitioning.
- We conduct extensive experiments on different sizes of graphs, including a billion-scale graph to demonstrate how Betty enables large-scale GNN training on single GPU without suffering from out of memory (OOM) and losing accuracy. Compared with a set of other graph partition algorithms (Metis, range, and random), Betty improves computation efficiency by 20.6%, 21.1%, and 22.9% respectively.

2 BACKGROUND AND RELATED WORK

We review background information in this section.

2.1 GNN Preliminaries

GNNs are a family of neural networks performed on an input graph to encode graph information. Nodes in the graph used by GNN represent entities in a learning problem (e.g., a user in a social network), and each node carries a feature vector. Edges in the graph represent the relationship between nodes, which is quantified with edge weights. A computation layer in a GNN model consists of graph operations and neural operations. All computation layers of a GNN builds its computation graph.

- Graph operations: A node (called the *center node*) collects features vectors of its neighbor nodes, performs aggregation operations (e.g., reduction), and then updates its own feature vector. After the graph operations, GNN encodes graph structure and information using the new feature vectors.
- Neural operations are performed either independently among nodes or in center-neighbor patterns according to the graph structure. When using the center-neighbor patterns, the neighborhood relationship is utilized and the neural operations are performed for each center node using neighbors' features. Figure 1 is an example performing neural operations (particularly LSTM) using a center-neighbor pattern.

Equation 1 is an example of a computation layer including the graph and neural operations. This equation computes the hidden features of a center node v on the layer $l+1$. h^l is the hidden features used as an input for the layer l. $u \rightarrow v$ indicates a directed edge from the node u to the node v; e_{uv} is the edge weight, D_v is the in-degree of v. The layer averages the features of neighbor nodes of v, and then computes using weights W^l. In practice, aggregating all the neighbors can lead to huge memory consumption. As a result, practitioners often set a bound called *fanout degree*, where

Table 1: Common computation layers in GNN

Layer type	Formulation
Sum	$SUM_{u\to v}(h_u^l * e_{uv})$
Mean	$SUM_{u\to v}(h_u^l * e_{uv}/D_v)$
Pooling	$MAX_{u\to v}(ACT(W^l \otimes h_u^l \odot e_{uv}))$
LSTM	$LSTM_{u\to v}(h^l)$

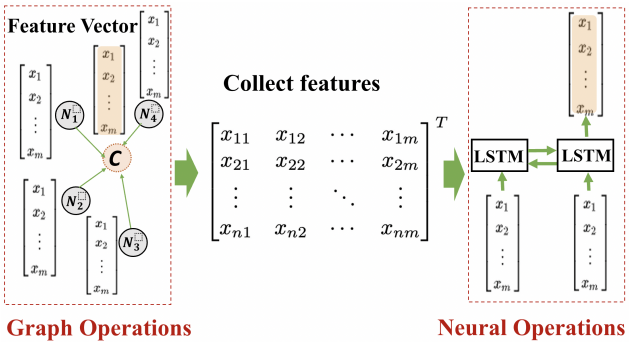

Figure 1: An example of applying neural operations (LSTM).

the maximum number of neighbors used for aggregation should not exceed this bound. The fanout degree bound is often satisfied via graph *sampling*. The hidden features for the next layer h^{l+1} is produced after applying an activation function, *ReLU*. Table 1 lists a couple of common computation layers.

$$h^{l+1} = ReLU((SUM_{u\to v}(e_{uv} \odot h_u^l/D_v)) \otimes W^l) \quad (1)$$

In the rest of the paper, all nodes in the input graph are called *input nodes*, and the center nodes in the last layer of GNN model are named *output nodes*. Like other neural network training, GNN training can either use full batch (e.g., in gradient descent) or mini-batches (e.g., in stochastic gradient descent).

2.2 GNN Framework

There are a handful of GNN frameworks [4, 12, 13, 19, 24, 42–44]. Deep Graph Library (DGL) is a popular GNN framework. Its distributed version, DistDGL [44], uses the Metis partition algorithm [14] to reduce communication between GPUs; AliGraph [42] and DistGNN [26] use the similar strategies. Except the graph partition method, P3 [5] uses pipeline and a caching strategy to speed up distributed GNN training. PyTorch Geometric (PyG) [4] partitions features (not nodes) to implement the distributed GNN learning. [46] use feature decomposition to speedup GNN inference with small input graphs. PCGraph [43] uses the similar method. NeuGraph [24] focuses on a single machine with multi-GPUs and leverages a variant of node-centric parallel graph abstraction (GAS, the gather-apply-scatter) [22] to partition the input graph using Metis. Roc [13] and Lux [12] uses a partition strategy that dynamically adjusts loads between GPU based on a cost model. They pay attention to load balance but not redundancy. Pytorch-BigGraph [19] reduces GPU memory by swapping embeddings of each partition to hard drive.

For graph partitioning, most of the existing efforts pay attentions to reduce communication overhead or balance loads between GPU. PyTorch Geometric aims to save memory but the memory saving is constrained by the feature dimension. In contrast, Betty is the only work that recognizes the big impact of node redundancy on GNN training efficiency and reduces it while balancing loads. Also, Betty is the only effort that use heterogeneous GPU/CPU memory to maximize memory saving.

We implement Betty based on DGL in this paper and use DGL as strong baseline, because DGL is a state-of-the-art GNN library. It has been generally considered as the best performing framework for GNN training due to its many optimizations against single-GPU efficiency such as fused message passing kernels, shared-memory graph store, indegree bucketing, and many more [40]. It has been shown that DGL has superior performance than other alternatives (e.g., DGL performs 2x faster than PyTorch Geometric, another state-of-the-art GNN library. Also, for a 2-layer GCN model with Reddit dataset on a P100 GPU, the training throughput of DGL is 6x better than Roc. Hence, DGL provides a high bar for the study of performance and memory saving.

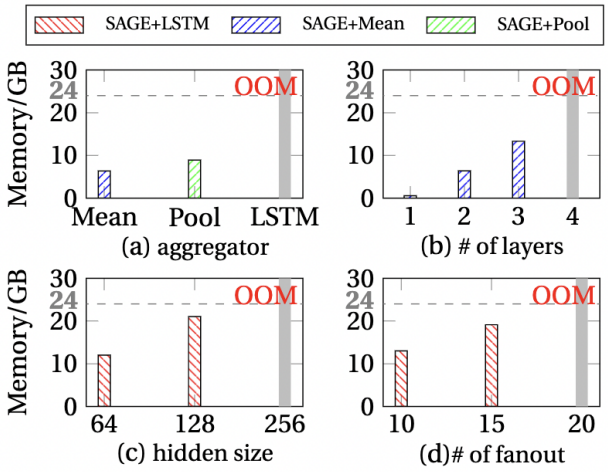

Figure 2: The memory consumption of running GraphSAGE on ogbn-products. (a) Comparison results between neighbor aggregators. The number of SAGE layers is 2, the hidden size is 256, and the fanout degree for the two layers is 10 and 25 respectively. (b) Comparison results varying the number of SAGE layers. The aggregator is Mean and the hidden size 256. For the four layers of SAGE, the fanout degree is 10, 25, 30 and 40 respectively. (c) Comparison varying the hidden size. Similar to the configuration in (b), but varying the hidden dimension sizes from 64 to 256. (d) Comparison results varying fanout degree. The SAGE layer is 1, the hidden size is 256 and the aggregator is LSTM.

2.3 Removing Memory Capacity Wall for DNN Training

There are many recent efforts [7, 7, 11, 17, 27, 29, 31–34, 39] mitigating GPU-side memory space limitation by leveraging larger

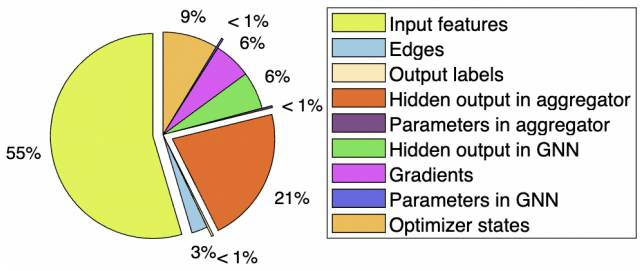

Figure 3: The GPU memory consumption of 1-layer GraphSAGE with the Mean aggregator and the dataset ogbn-products (fanout=10 and hidden size=64).

CPU-side system memory. AutoTM [7] uses linear programming to decide how to partition large neural network models on heterogeneous memory. ZeRO-Offload [33] offloads optimizer states and computation to CPU memory to save GPU memory consumption. ZeRO-Infinity [31] leverages heterogeneous storage architecture (i.e., GPU memory, CPU memory, and NVMe storage) to offload activation, parameters, and optimizer states. Capuchin [29] uses a combination of tensor recomputation and a prefetching mechanism to decide tensor offloading to CPU memory. Sentinel [32] uses a prefetching mechanism and limited domain knowledge on neural work for tensor offloading, based on a tensor-level profiling mechanism. The above efforts focus on partitioning the large neural network model itself, not the model input (i.e., the input graph in the case of GNN), and hence cannot be easily applied to GNN with large graphs as input.

3 GNN WORKLOAD ANALYSIS

In this section, we conduct a preliminary analysis that guides the design in Section 4.

3.1 Memory Capacity Bottleneck

To analyze the scalability bottleneck, we analyze the memory usage of a popular GNN network GraphSAGE [6] with a large dataset obgn-products [9] on an NVIDIA RTX6000 GPU with 24GB memory (Section 6 includes more details about this dataset and our experiment setup). Figure 2 shows that GNN scalability has been severely limited by the GPU memory capacity. While simple aggregators such as Mean and Pool incur <10GB memory consumption, more advanced aggregators such as LSTM are much more memory hungry and easily lead to over 24GB of memory consumption and OOM error, making training with these more advanced aggregators infeasible on larger datasets. Similarly, as we increase the aggregation depth, the memory consumption increases almost exponentially and runs into OOM with deeper GNNs (e.g., running OOM at the 4th-layer as in Figure 2(b)). Furthermore, the limited memory also prevents GNN from using wider hidden size (Figure 2(c)) and larger fanout degree for aggregation (Figure 2(d)). As such, the memory capacity has become a severe bottleneck for data scientists and practitioners to use more advanced GNN training methods. As we show in Section 6, Betty avoids OOM and enables those memory-consuming cases.

3.2 Memory Consumption Analysis

To understand the reason behind the GPU memory bottleneck, we conduct analysis to characterize the memory consumption during GNN training, using GraphSAGE [6] and the Mean aggregator on the dataset obgn-products [9] as an example. Figure 3 shows the results. The output node labels, input node features, and edges in Figure 3 come from the graph. The hidden layer output, activation, optimizer states, and gradients come from GNN. The aggregator also consumes memory because of the hidden layer output and parameters in Mean.

Figure 3 reveals that the input node features take the largest portion (55%) of the total memory consumption. This is different from the traditional neural network models where model parameters, activation, and optimization states dominate the memory consumption [31, 33]. The input features might not be the biggest part of the memory footprint all the time. It's related to the aggregator type. For some aggregators (e.g., LSTM), the input features are smaller than the activation, but still takes the second largest memory consumption with large-scale graphs. Hence, we focus on reducing the peak memory consumption of input node features to enable large-scale GNN training without losing training accuracy.

3.3 Reducing Batch Size to Remove Memory Capacity Bottleneck

One straightforward solution to mitigate the GNN memory capacity bottleneck is to use a smaller mini-batch size, such that each mini-batch corresponds to a smaller subgraph for aggregation. However, this solution has a major drawback that it has a non-trivial impact to the statistical property of GNN model training. Figure 4 shows the difference in training loss and validation accuracy between using full-batch training versus small-batch training. The experiments are done with GraphSAGE and ogbn-products, using the same hyperparameter settings for both cases. While a smaller mini-batch size reduces the total memory consumption, the training and validation curve of these two trainings are quite different.

For example, the training curve of the small batch size has more fluctuations than the full-batch training, and the test accuracy also degrades when training with more epochs. Although the test accuracy climbs more quickly in the first few iterations comparing with full batch training, the training diverges after a few epochs. Such a difference is caused by the change of the effective batch size of the model, which is one of the most critical hyperparameters for GNN training to achieve fast convergence and high model quality. Changing the effective batch size often requires model scientists and practitioners to make additional adjustment to the hyperparameters, such as learning rate schedule and weight decaying to ensure that the model still carries the same convergence quality. Therefore, it is desirable to reduce the memory overhead and improve the scalability of GNN in a way transparent to the GNN users.

4 DESIGN

4.1 Overview

Betty reduces the memory consumption of GNN training via the batch-level partitioning and using both CPU and GPU memory

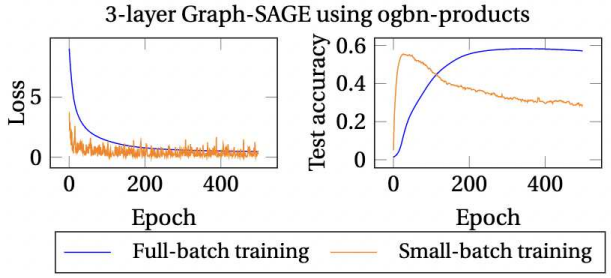

Figure 4: The loss and test accuracy of full-batch vs. small-batch training using ogbn-products. The full batch size is 196,615. The number of small batch is 16, each with 12,289.

to enable training of advanced GNNs on single GPU. We formulate the batch-level partitioning in GNN as a multi-level bipartite graph partitioning problem and study how such partitioning can be done while allowing the GNN training to leverage accumulative gradients of the partitioned micro-batches to achieve the same training results as the original batch without the need of adjusting any hyperparameters from model scientists (Section 4.2). During the generation of micro-batches, duplicated nodes are created. Minimizing the redundancy is critical in the graph partition algorithm in Betty, discussed in Section 4.3.

While the batch-level partitioning reduces the memory footprint, it introduces challenges of high redundancy across partitioned micro-batches and load imbalance. We describe two novel techniques to mitigate the redundancy. In Section 4.3, we describe how we construct a redundancy-embedded graph and transform the redundancy reduction problem as an equivalent min-flow cost cut problem, which we solve via a high performance min-flow cut algorithm. In Section 4.4, we describe how we perform memory-aware graph partitioning via accurate estimation of the memory usage of GNN batches. This method allows Betty to quickly figure out how many partitions a batch should be split to meet a memory capacity constraint. Figure 5 overviews the workflow of Betty.

4.2 Batch-Level Partitioning

4.2.1 Why batch-level partitioning? We propose to use the batch-level partitioning to reduce memory footprint of GNN training. Although it is also possible to reduce the mini-batch size to reduce memory footprint, such a method has a non-trivial impact on the model convergence and generalization, discussed in Section 3.3. The batch-level partitioning partitions a batch into micro-batches and calculates the loss and gradients after each micro-batch. In short, reducing the mini-batches can cause degradation in accuracy and demand changes of hyper-parameters; using micro-batches, there is no such problems.

Figure 6 shows the difference between mini-batch training and micro-batch training training in terms of gradient accumulation. Using the micro-batch training, instead of updating the model parameters after each backpropagation, the model can wait and accumulate the gradients over consecutive micro-batches and ultimately updates the parameters based on the accumulated gradients from the entire batch. The benefit of this simple optimization is that it allows the model to train with a much lower memory footprint while achieving the same convergence without any changes on hyperparameters or optimizers, a system-level optimization that is transparent to the model training process.

4.2.2 Background: Batch in GNN — Multi-level Bipartite. Different from traditional DNN training, where the input to a neural network consists of mini-batches and each batch contains 1:1 mapping of features (e.g., images and texts) and labels, in the simplest case, a GNN mini-batch is actually a bipartite graph. In a bipartite structure, the center node is called *destination node* and one-hop neighbors of the destination node are called *source nodes* in the rest of the paper. The bipartite graph is represented as $G = (U, V, E)$, where U is a set of source nodes, V is a set of destination nodes, and E is a set of edges that records the source-to-destination relationships. V contains the nodes that are labelled and are used for learning GNN, and V is the union of the neighbors of $v \in V$ for the entire V. Since each node v can have multiple neighbors and each node in U can also be a neighbor of multiple nodes in U, they form a N:M mapping of features (e.g., neighbors of nodes) and labels (e.g., labels for nodes). Furthermore, for each $u \in U$, there is a corresponding feature vector X_u, and inputs to GNN may include features from high-order neighbors, e.g., to capture k-hop neighbors' feature information and aggregate neighbors from multiple hops recursively by stacking multiple DNN layers [6]. Therefore, in a GNN model, each mini-batch is a hierarchical bipartite.

Figure 7 gives an example of a two-level bipartite structure in GNN training where the nodes 8 and 5 form a batch. For the destination node 8, the GNN aggregates features of the source nodes from the node 8 (i.e., the nodes 4, 5, 7, and 11); for each source node (e.g., the node 4), the GNN aggregates features of its source nodes one hop-away (e.g., the source nodes 3, 5, and 8 for the node 4), which forms a two-level bipartite structure. The bipartite together with the embedding features of the destination nodes form a batch and are sent as a single object to the GNN model for training.

4.2.3 Partitioning the Multi-level Bipartite for Micro-Batch GNN Training. Based on the bipartite structure of the batch in GNN training, Betty divides each batch into K micro-batches, where each micro-batch is still a hierarchical bipartite that is a subgraph of the original bipartite. In particular, we define the *batch-level partitioning* problem here: We partition a bipartite graph $G = (U, V, E)$ into k sets of bipartite sub-graphs $G'_k = (U_k, V_k, E_k)$ where $k \in [0, K-1]$. The disjoint union of V_k is V, and the union of U_k is U. U_k for different subgraphs can have overlapping nodes.

These partitioned multi-level bipartite then get streamlined from the host memory to the accelerator's on-device memory for aggregation and the forward computation. During the backward pass, gradients for each micro-batch are computed based on the same model parameters used for the forward pass. Furthermore, after each backward propagation, the intermediate results are released and only the gradients are stored in GPU memory. At the end of each batch, gradients from all K micro-batches are accumulated and applied to update the GNN model parameters. Those gradients are equivalent to the gradients calculated using the full-batch training.

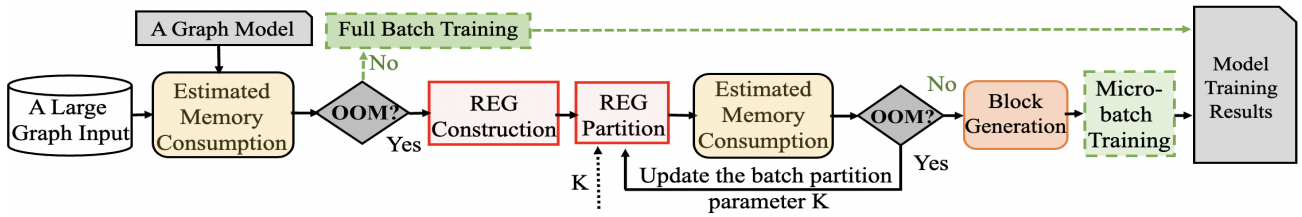

Figure 5: Workflow of Betty.

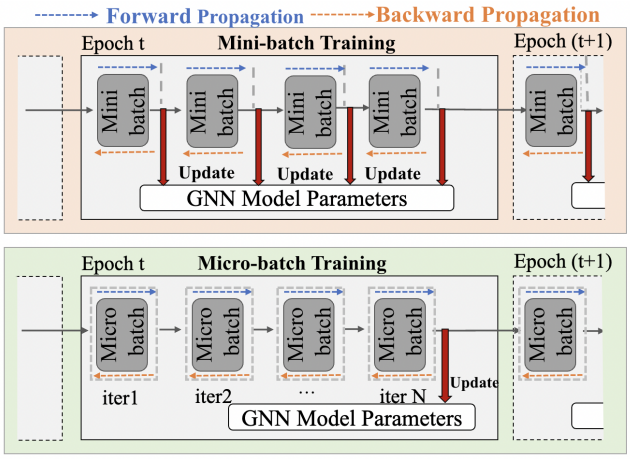

Figure 6: Workflow for mini-batch training and micro-batch training to tell the difference between the two trainings.

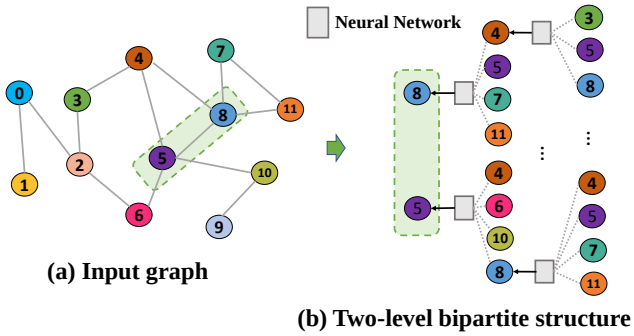

Figure 7: An example of the bipartite structure in GNN.

4.3 Redundancy Reduction

4.3.1 Why redundancy matters? While partitioning the batch (i.e., the multi-level bipartite) reduces the memory consumption because a micro-batch corresponds to a smaller bipartite that consumes less memory, there are two challenges in this partitioning process: (1) Without any control, each micro-batch may consume a variable amount of memory and there is no guarantee that the memory consumption per micro-batch remains constant. Therefore, it is still possible that the GNN model runs OOM when some of the micro-batches have unbalanced loads that exceed the memory capacity; (2) The partitioning can introduce a large amount of redundant nodes that lead to wasted memory and compute. We note that conceptually for some input graphs that have special topology, such as those with isolated subgraphs, there might be zero redundancy after partitioning. When these factors are mixed together with the multi-level bipartite structure in the batch, this problem becomes extremely complicated.

To further explain the redundancy problem, we use Figure 8 as an example. Figure 8 shows a bipartite graph that corresponds to a batch. To simplify the explanation, we let the batch to include only 1-hop neighborhood and the batch is only partitioned into two micro-batches. Each micro-batch corresponds to a bipartite subgraph that contains the partitioned destination node as well as all the source nodes required for aggregation. Figure 8 shows two partition methods ((a) and (b)), and the nodes 1 and 8 are the destination nodes.

We note that although the destination nodes 1 and 8 are disjoint in these two subgraphs, there are source nodes shared by both subgraphs (e.g., nodes 3, 5, 6, and 7 in the partition method (a) are included in both subgraphs, and similarly for nodes 5 and 6 in the partition method (b)). Therefore, the partition method selected affects the number of shared nodes. The downside of having shared nodes included in multiple micro-batches is that the GNN model needs to perform multiple rounds of data loading and computation for those nodes, which not only increases the memory consumption for a micro-batch but also adds additional cost from computation and data movement from CPU to GPU. Figure 8 also shows that a different method would lead to smaller bipartite subgraphs while satisfying the memory constraints. In Section 6, we also empirically show that when using a redundancy-unaware graph partition algorithm, such as Metis [14], to partition a batch into 8 partitions, it can introduce node redundancy by 23.5% in comparison to a redundancy-aware partitioning algorithm (e.g.,Betty). As a result, the memory consumption and training time both get increased by 27.6% and 47.6%, respectively.

4.3.2 Redundancy-Embedded Graph Construction and Partition. Given the importance of redundancy reduction, one obvious challenge is how to partition a GNN batch to minimize the redundancy. It may appear at first that one can apply an existing graph partitioning algorithm, such as [14, 15], to solve this problem. However, the generic graph partitioning algorithms are often designed to solve the min/max-cost flow problem, which minimizes/maximizes the weights of cut edges while balancing the workload across partitions, an objective different from reducing redundancy.

To effectively reduce the redundancy within micro-batches, we transform the redundancy elimination problem to a min-cost flow

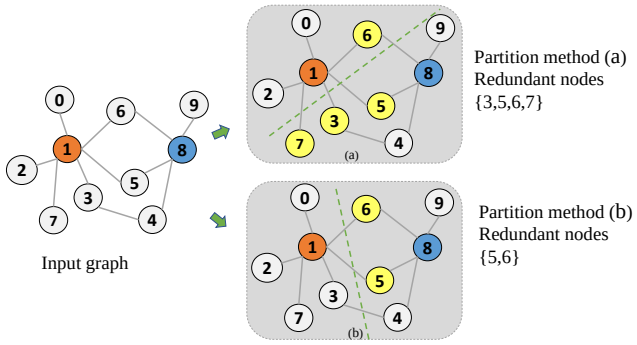

Figure 8: An example to depict the node redundancy problem.

problem by constructing an auxiliary graph called *Redundancy-Embedded Graph* (or REG). In REG, the weight on an edge is the number of neighbors shared by the two nodes connected by the edge. Therefore, the higher weight an edge has in REG, the more neighbors the two connected nodes share, and the more redundancy will be created when splitting those two nodes into two micro-batches. The node set in REG is the same as the original node set. In essence, the weights embed redundancy information, and finding a K-way partitions of REG that minimizes the cut flow is equivalent to minimizing the redundancy from micro-batches. We depict the construction of REG in detail as follows and in Algorithm 1.

The construction of REG happens on CPU and it is based on the adjacency matrix representation of the graph. Assume that the matrix A in Equation 2 is a binary adjacency matrix where the element a_{ij} in A is a binary value indicating if there is any connection from the nodes i to j. The representation of REG, C, is a multiplication of A^T and A. The elements of C are calculated using Equation 3. The element c_{ij} in Equation 4 counts the number of shared neighbors between the nodes i and j, because when $a_{ki}a_{kj}$ = 1, we know that the node k is a shared neighbor between i and j. Lines 1- 4 in Algorithm 1 depicts the above details.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, a_{ij} = \begin{cases} 1, & \text{If node i has} \\ & \text{edge pointing} \\ & \text{to node j} \\ 0, & \text{Otherwise} \end{cases} \quad (2)$$

$$\begin{aligned} C &= A^T \times A \\ &= \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \quad (3) \\ &= \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} \end{aligned}$$

$$c_{ij} = a_{1i}a_{1j} + a_{2i}a_{2j} + \cdots + a_{ni}a_{nj} = \sum_{k=1}^{n} a_{ki}a_{kj} \quad (4)$$

Once C is generated, we further remove nodes that are not output and remove self-loop. The resulting graph is the final REG (Lines 5-7 in Algorithm 1). With REG, we can apply any existing graph partitioning algorithm that minimize the cut flow. In this work, we adopt Metis (Line 8) to partition REG, given that Metis has demonstrated strong scalability on large-scale graph partitioning problems [44]. We collect output nodes to construct a list for partitioning and bipartite generation (Lines 9-12).

Algorithm 1: REG partitioning

input : The bipartite graph (e.g., a DGL Block) of the output (smallest) layer in multi-layer GNN full batch training : $bipartite_{last}$
The number of partitions: K
output: *batched_output_nodes_list*

```
1  u, v = Get_edges(bipartite_last);
2  graph_homo = Graph_init((u, v));
3  A = Get_adjacency_matrix(graph_homo);
4  REG = Matrix_multiplication( Transpose(A), A);
5  non_output_nodes = get_non_output_nodes(REG);
6  REG = Remove_nodes(REG, non_output_nodes);
7  REG = Remove_self_loop(REG);
8  partitions = Metis_partition(REG, K);
9  for part in partitions do
10     nids = Get_output_nodes_ids(part);
11     batched_output_nodes_list.append(nids)
12 return batched_output_nodes_list
```

4.4 Reducing Maximal Memory Footprint

Reducing redundancy reduces the redundant data movement and computation, but it is not sufficient to get the best performance because of partitioning imbalance.

4.4.1 Why looking into imbalanced partitions? The load imbalance can lead to large variance in memory usage when executing micro-batches. The peak memory consumption of GNN training is determined by the largest micro-batch. Table 2 gives two examples of such load imbalance. The two examples use GraphSage with dataset ogbn-arxiv and use the REG-based partitioning. Example 1 partitions the graph into two micro-batches, while Example 2 partitions it into four. In Example 1, the memory consumption of the micro-batches differ by 10.4%, while Example 2, it differs by up to 31.9%. In the two examples, the second and the fourth micro-batches determine the peak memory consumption respectively.

One may think that simple solutions such as random graph partition can mitigate the load imbalance because it does not explicitly control node distribution between micro-batches. However, as we show Section 6.5, random partitioning (applied on output nodes,

Table 2: Two examples with load imbalance.

	2 batches (Example 1)		4 batches (Example 2)			
Batch id	0	1	0	1	2	3
Mem/GB	1.91	2.13	0.97	1.04	0.79	1.17

based on that generate hierarchical bipartite) creates a large number of redundancy, which to an extent barely reduces the memory consumption, especially when a graph topology is complex.

4.4.2 Source of Load Imbalance. To figure out what causes the imbalanced partitions, we perform a deeper analysis on how state-of-the-art GNN frameworks process batches, using DGL as an example. Existing GNN frameworks often employ *in-degree bucketing* algorithms to perform the aggregation operation, i.e., the nodes with the same in-degrees are gathered into the same bucket (called *NodeBatch*) to improve computation efficiency of aggregation [40]. However, such an algorithm creates challenges both in memory management as the graph becomes larger and in the batch-level partitioning.

Indegree-bucketing leads to the bucketing explosion and partition imbalance. Figure 9 (a) shows the distribution of destination nodes in terms of in-degree. During the aggregation stage, GNN would group nodes with the same in-degree into one bucket so that the received messages in a bucket can be processed as a dense tensor. However, this bucketing is often done for the first M degrees, where M is often a small value (e.g., ≤ 10). The nodes whose in-degrees are larger than M are bucketed into the last bucket. Such a method does not create an issue when a graph is small, because the connections (or in-degrees) in small graphs are often limited. However, as the graph size increases, the in-degree often follows a power law distribution and may have a *long tail* [1], where the accumulated nodes that fall into this long tail become intractable.

The in-degree bucketing leads to a so called *explosion of bucketing* problem, where the last bucket becomes much larger than the rest of the buckets that represent lower in-degrees due to accumulated long tail. When performing aggregation one NodeBatch after another, the GPU memory consumption can be bounded by the last bucket where the number of nodes is the largest among all buckets.

The in-degree explosion creates the load imbalance during the batch-level partitioning. Figure 9 (b) shows the results of partitioning a batch into two micro-batches on GraphSAGE (using dataset ogbn-arxiv) using the REG-based partitioning. The results show that the micro-batch 1 has nearly 19% more nodes than the other in the last bucket (the bucket 10). This is because the last bucket has much more nodes than other buckets. The REG partitioning cannot eliminate such imbalance.

4.4.3 Memory-aware Partitioning. One straightforward idea to address the load imbalance problem is to further increase the number of partitions. However, there is a major challenge in this approach: the memory consumption of a batch in GNN does not follow a linear relationship, making it difficult to know ahead of time how many partitions are sufficient to reduce the maximal memory consumption to meet a memory constraint. Although it is possible to figure this out via trial-and-error if a partition strategy does not

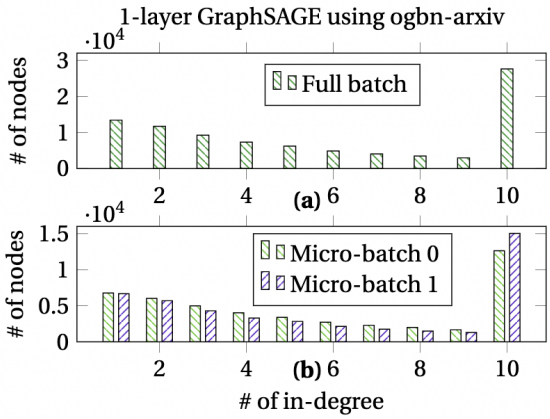

Figure 9: (a) The in-degree distribution of destination nodes; (b) The in-degree distribution for two micro-batches.

prevent OOM, it may largely waste computation in the retrying process. To tackle this challenge, we introduce a memory-aware re-partitioning algorithm that quickly decides the partition count via an accurate estimation of the memory usage of a micro-batch without triggering the expensive training cost. In the next, we describe how we estimate the memory usage of a micro-batch and introduce a memory-aware batch-level partitioning algorithm.

Partition memory estimation. The memory consumption is estimated by counting (1) the number of model parameters (NP_{GNN}), (2) the total dimensions of input features ($N_{in} \times H_{in}$), (3) the number of output node labels (N_{out}), (4) the size of all blocks, (5) the size of hidden layers' output in GNN, (6) the size of intermediate results from the aggregator, (7) the number of gradients, and (8) the number of optimizer states. The definition of all parameters can be found in Table 3.

While counting (1)–(3) are straightforward, the estimation for (4)–(8) requires some explanation. The estimation of (4) is based on a quantification of edges in all blocks (a block represents a bipartite structure). The number of edges in a block is E. Since each edge in a block is represented by two node IDs (the source and destination node of the edge) and a weight (if there is any), hence, the size of a block is $E \times 3$.

The estimation of (5) is $\sum_{i=0}^{n} (N_i \times h_i)$, where n is the number of hidden layers in GNN; N_i and h_i are the number of destination nodes in each hidden layer, and the dimension of hidden feature in each node.

The estimation of (6) depends on the type of the aggregator. If the aggregator is LSTM, then the size of intermediate results for the LSTM aggregator is estimated by Equation 5.

$$\sum_{i=1}^{fanout} L_i \times B_i \times H \times 18 \tag{5}$$

The size of intermediate results of LSTM is dominated by the product of the input sequence length and the input feature size H. In our context, LSTM is applied to the nodes with various in-degrees, but the in-degree is bounded by fanout. Given an in-degree

Table 3: Notation for memory estimation

Parameter	Description
NP_{GNN}	# of model parameters in GNN w/o aggregator
NP_{Agg}	# of model parameters in the aggregator
N_{in}	# of input nodes
H_{in}	Dimension of input feature
N_{out}	# of output nodes
h	Hidden dimension of GNN model
n	# of layers in the GNN model
E	# of edges in a block
L	In-degree of an destination node
B	# of nodes share the same in-degree
K	# of partitions (micro-batches)

L_i, the number of nodes with the same in-degree L_i is B_i. $L_i \times B_i$ gives the number of the nodes fed into LSTM. The constant 18 in Equation 5 is an implementation-dependent. Using PyTorch, the number of intermediate results generated per node is 18 [35]. Hence, Equation 5 quantifies the number of intermediate results for LSTM.

The estimation of (7) is $NP_{GNN} + NP_{Agg}$.

The estimation of (8) depends on the type of optimizer. For the most popular optimizer Adam, the optimizer states are the momentum and variances of the gradients. Hence, the estimation of (8) is $(NP_{GNN} + NP_{Agg}) \times 2$.

The memory allocation for (6) can be freed during the backward propagation where the memory consumption for (7) starts to dominate. Besides (6) and (7), the memory consumption of other tenors remains stable throughout the training steps. Hence, the peak memory consumption comes from the maximum of (6) and (7) plus other tenors.

Batch re-partitioning based on memory estimation. After $K - way$ partitioning of REG, Betty estimates the memory consumption of each micro-batch without execution on GPU. If the micro-batch with the largest memory consumption violates the memory capacity constraint, Betty tries $(K + 1) - way$ partitioning of REG and estimates the memory consumption again. The above process stops until the memory consumption of all micro-batches meets the memory capacity constraint.

5 IMPLEMENTATION

Betty is based on DGL and uses its APIs to implement the batch-level graph partitioning and enable large-scale GNN training. Betty is a Python-based module for DGL.

Graph partition. Betty implements the graph partition algorithm (discussed in Section 4.3.2) as a DGL function, `dgl.betty()`. `dgl.betty()` employs `dgl.adj_product_graph()` to create REG and split it with `dgl.metis_partition()`. `dgl.betty()` uses the existing `DGLgraph.in_edges()` to get the indexes of source nodes and edges for a given destination node, which is needed for graph partition. Those indexes and the raw graph, after the partition, are fed to `dgl.edges_subgraph()` to generate subgraphs in the DGLgraph format. Those subgraphs are then used to generate blocks (representations of bipartite structures in DGL) by using `dgl.to_block()`. The blocks are needed for future graph operations during the GNN training.

Table 4: Training datasets

Dataset	#node features	#nodes	#edges
Cora	1,433	2,708	10,556
Pubmed	500	19,717	44,338
Reddit	602	232,965	114,615,892
ogbn_arxiv	128	169,343	2,315,598
ogbn_products	100	2,449,029	61,859,140

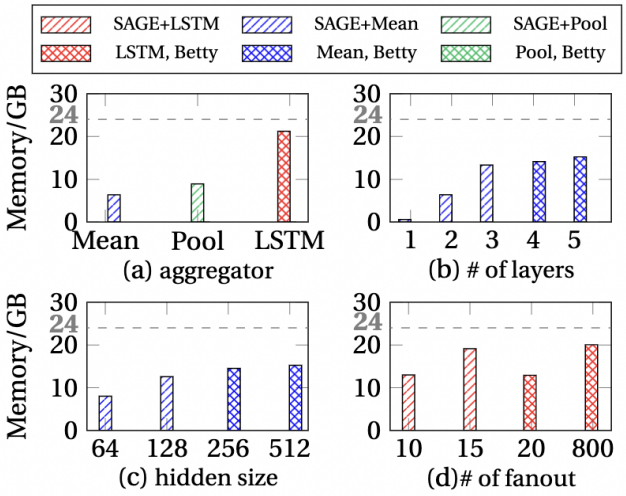

Figure 10: Betty breaks the memory capacity constraint in Figure 2. This figure uses the same configurations as Figure 2.

Index mapping. In DGL, each node (or edge) can have multiple indices, depending on the scope where nodes and edges are. Before sampling, each node/edge can has a global index in the raw graph. After sampling, each node/edge has a global index in the full-batch graph. After the graph partition, each node/edge has a local index in each micro-batch.

Using the global indices in the graph (i.e., the symbols `dgl.NID` and `dgl.EID`) provided by DGL, Betty can retrieve node and edge information when working in the full-batch graph by using `full_batch_block.srcdata[dgl.NID]` and `full_batch_block.edata[dgl.EID]` respectively. Betty can also retrieve node and edge information from the full batch graph when working in a micro-batch i by using `batch_i.srcdata[dgl.NID]` and `batch_i.edata[dgl.EID]` (note that `dgl.NID` and `dgl.EID` refer to the indices in the scope of the full-batch graph). However, there is no way to allow Betty to retrieve node and edge information from the raw graph when working in a micro-batch. To address the problem, Betty introduces a dictionary to bookmark the relationship between the local indices in the micro-batch and the global indices in the graph during the block generation and after the graph partition.

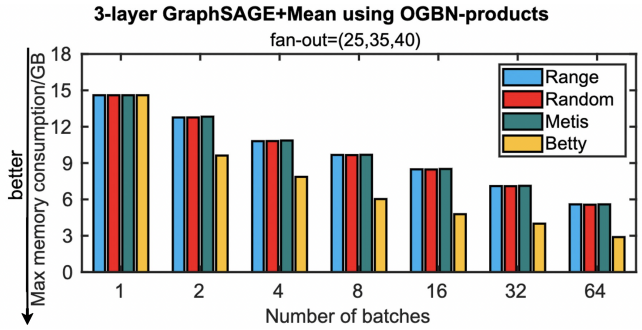

Figure 11: The reduction of max memory consumption of GPU compared with the full-batch training and three graph partition algorithms (i.e., range partition, random partition, and Metis).

6 EVALUATION

Platform. We use a system with two Xeon E5-2698 v4 CPUs (40 cores running at 2.20 GHz) and an NVIDIA Quadro RTX 6000 GPUs. We use CUDA 10.1/cuDNN 7.0 to run GNNs on NVIDIA GPUs. We use Python 3.6, Pytorch [28] v1.7.1 and DGL [40] v0.7.1 for model training. The RTX6000 we use only has 24 GB GPU memory. Though one A100 can provides 80 GB GPU memory, it still cannot meet the memory requirements for very large scale GNN training.
Workloads. We use five datasets (or five input graphs) with various features, which are shown in Table 4. We use two GNN models, GraphSage [6] and GAT [18].

6.1 Reduction of Peak Memory Consumption

Breaking the memory capacity wall. We re-evaluate the cases in Figure 2 but with Betty. Figure 10 shows the results.

With Betty, all OOM cases are addressed. With Betty, we are able to run a sophisticated aggregator LSTM (Figure 10.a) using nine micro-batches; we can run the GNN model with more layers (4 and 5 layers) using 3 and 60 micro-batches respectively; we are able to run the GNN model with larger hidden sizes (256 and 512) using 3 and 32 micro-batches respectively; we are also able to increase fanout to 20 and 800 using 2 and 13 micro-batches respectively.

Comparison with other graph partition algorithms. Besides using Betty, we use three common graph partition algorithms: range partition, random partition, and Metis. The three partition algorithms partition the graph based on the IDs of output nodes. With the range partition, the space of output node IDs is evenly and sequentially partitioned. With the random partition, the space of output node IDs is evenly and randomly partitioned. With the three partition algorithms, in each partition, the output nodes and their one-hop neighbors are co-located in the same partition.

In general, Betty reduces max memory consumption by 16.3%, 10.5%, 27.2%, 30.98%, and 48.3% for datasets Cora, Pubmed, Reddit, ogbn-arxiv, and ogbn-products respectively, compared with other graph partition methods. Figure 11 shows the results for GraphSAGE model with the dataset OGBN-products with different numbers of batches. The figure reveals that compared with other partition algorithms(range, random and Metis), Betty reduces max memory consumption by 48.3% and 37.7% on average. This demonstrates the effectiveness of Betty in reducing node redundancy. This effectiveness comes from the quantification of redundancy and embed it into the graph partition algorithm, which is missing in any other partition algorithm.

Table 5: DGL v.s. Betty training Accuracy

Dataset	Model	DGL / Acc (%)	Betty/ Acc (%)
Cora	SAGE	80.65 ± 0.71	80.28 ± 0.73
	GAT	80.40 ± 1.17	78.24 ± 1.85
Pubmed	SAGE	77.12 ± 0.63	76.22 ± 0.32
	GAT	76.89 ± 0.79	76.15 ± 0.89
Reddit	SAGE	94.80 ± 0.08	94.80 ± 0.08
	GAT	89.73 ± 2.23	89.61 ± 2.79
ogbn-arxiv	SAGE	71.80 ± 0.32	71.80 ± 0.32
	GAT	69.52 ± 0.09	68.62 ± 0.10
ogbn-products	SAGE	75.95 ± 0.43	76.12 ± 0.36

Peak memory consumption vs. training time. As we decrease the batch size, Betty is able to reduce max memory consumption as a cost of increasing training time on GPU (without considering data transfer time). Micro-batches are executed sequentially, hence increasing training time (compared with the full-batch training). The increase of training time comes from the increase of node redundancy shown in Section 6.5 and lower GPU utilization. Compared with the traditional methods of using mini-batch training to break the wall, micro-batch training is 36.4% faster (using 2-layer GraphSAGE with the mean aggregator and OGBN-products) and reduces hardware cost. There is a batch size where there is a good balance between the reduction of peak memory consumption and training time.

Figure 12 shows the trend of the reduction of memory consumption and increase of training time as we increase the number of batches. The configurations of GraphSage (such as the number of layers and aggregator) for each dataset are carefully chosen such that when the number of batches is 1, we can train the GNN without OOM. The figure shows that depending on the GNN model topology, the aggregator type, and dataset, the batch size that achieves the good balance varies, and often in the range of 4-8.

6.2 Training Accuracy and Convergence

Table 5 shows the training accuracy for the original DGL using the full-batch training, and DGL using Betty and micro-batch training. GAT cannot use the ogbn-product dataset. Hence, there is no result for GAT using ogbn-product in Table 5. Compared with the full-batch training, micro-batch training enabled by Betty has ignorable loss in training accuracy, because the micro-batch training is mathematically equivalent to the full-batch training.

To study the model convergence, we train the GNN models with all datasets, and present the convergence curves for GraphSage with ogbn_arxiv, which are shown in Figure 13. The figure show the convergence curves for the full-batch training and micro-batch training with three different numbers of batches and with the same model hyperparameters. The figure shows that the four curves are very close to each other, no matter how we change the number of batches. This result shows the model convergence is not impacted

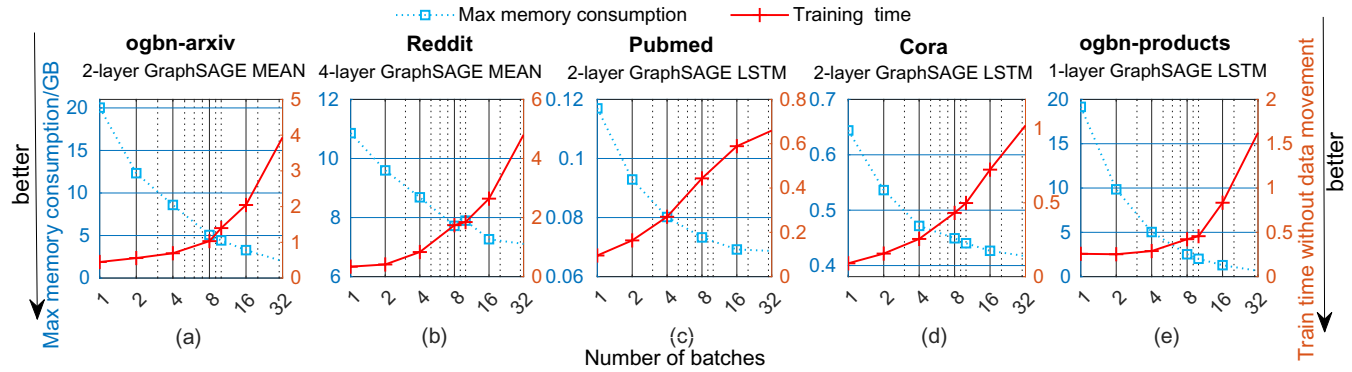

Figure 12: The tendency of peak memory consumption and training time per epoch as the number of micro batches increases.

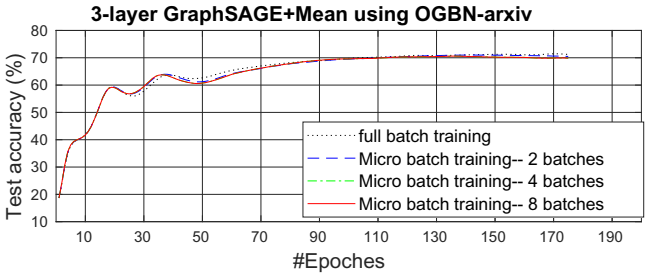

Figure 13: Convergence curves for full-batch training and micro-batch training with three different numbers of batches.

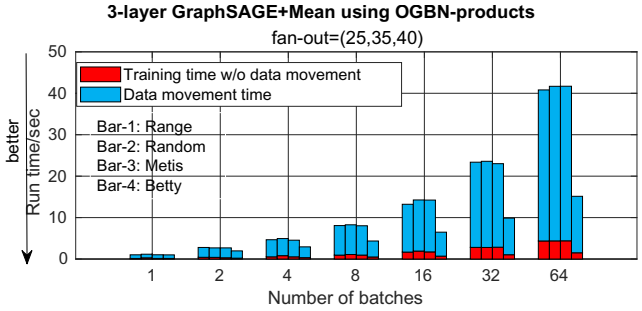

Figure 14: Training time + data movement time variance when we use different numbers of batches. We use GraphSage with the aggregator Mean and dataset ogbn-products.

by Betty and micro-batch training. Using all other datasets, we can see the same results.

6.3 Training Time

We evaluate the training time (including data transfer) with various numbers of batches. Since using Betty, there is no training accuracy loss and the convergence curve remains the same as using the full-batch training, we report the training time in one epoch. Figure 14 shows the results. We choose GraphSage model with an aggregator Mean and dataset ogbn-products. Among all cases that do not cause OOM when the number of batches is one, this case has the largest memory consumption.

We notice that when the number of batches becomes larger, the training time becomes larger. This is because a larger number of batches leads to higher node redundancy. For example, when the number of batches is 64, the node redundancy increases by 440%, compared with the case when the number of batches is 1. Also, the data transfer time increases because of lower GPU bus utilization.

6.4 Computation Efficiency

We evaluate the computation efficiency of Betty. The computation efficiency is defined as the total number of nodes in all micro-batches divided by one epoch time. Using micro-batches, node redundancy is created because of the block structures, although Betty reduces the node redundancy. With the node redundancy, we want to see if the computation efficiency is decreased, compared with using the full-batch training. Figure 15 shows the results. The figure reveals that as we change the number of batches, the computation efficiency of Betty remains the same as that of the full-batch training, although the number of redundant nodes increases. This indicates that Betty does not unproportionally increase the training time, leading to stable computation efficiency.

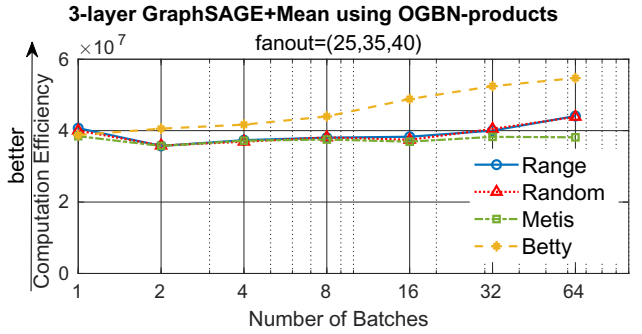

Figure 15: Evaluation of computation efficiency.

Table 6: Time and Memory consumption between Micro-batch and Mini-batch train (mean aggregator)

	Total number of the first layer input		Training time per epoch/(sec)		CUDA Memory consumption/(GB)	
# of batches	Micro-batch	Mini-batch	Micro-batch	Mini-batch	Micro-batch	Mini-batch
1	1,829,066	1,829,275	0.49	0.47	6.38	6.37
2	2,277,172	3,318,923	0.5841	0.6213	3.56	4.91
4	2,964,874	5,810,587	0.7551	1.0331	2.12	3.52
8	4,061,037	9,665,382	0.8535	1.3364	1.27	2.35
16	4,980,601	14,997,172	1.2219	2.4867	0.71	1.47
32	6,335,635	21,424,398	1.6699	3.4034	0.46	0.87
64	7,820,693	27,988,444	2.5511	4.8501	0.29	0.50

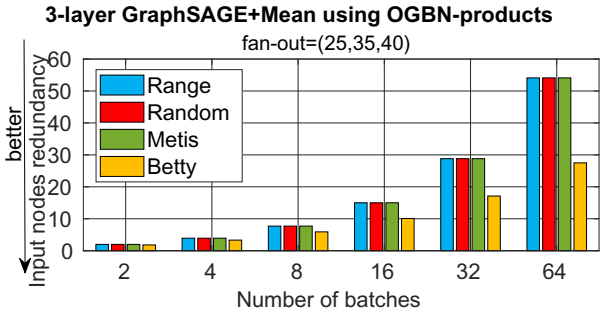

Figure 16: Evaluation of input nodes redundancy

Table 7: Memory Estimation Error for the LSTM aggregator

Dataset	Fanout	# of batches		Error rate(%)	
Cora	10	4	8	6.9	7.4
pubmed	10	4	8	7.6	8.2
Reddit	10	4	8	5.4	6.2
ogbn-arxiv	10	4	8	3.5	4.0
ogbn-products	10	4	8	3.2	3.8

6.5 Evaluation of Removing Node Redundancy

We evaluate the effectiveness of the graph partition in Betty, compared with the range, random, and Metis partition algorithms. Figure 16 shows the results. Using 64 batches, the number of redundant input nodes for the range, random, Metis and Betty is 118,253,165, 118,269,959, 118,281,306 and 5,808,923 respectively, compared with using two batches. Betty reduces the node redundancy by up to 49.2% and 28.4% on average. In all cases, Betty leads to the smallest number of redundant nodes. This benefit is especially pronounced when the number of batches is large. Such large benefit demonstrate the effectiveness of using REG.

6.6 Time and Memory reduction comparing with mini-batch

Table 6 shows an example about the training time and memory consumption between micro-batch and mini-batch. The dataset used is OGBN-products, the model is 2-layer GraphSAGE with mean aggregator. And the size of fanout is 10,25.

We observe that overall the training time (without data preparing) and memory consumption of micro-batch is smaller than mini-batch based training. The difference becomes larger when the number of batches increases. The time and memory reduction primarily comes from the node redundancy reduction. For example, when the number of batches is 64, the input node redundancy of Betty is $7820693/1829066 \approx 4.2$, which is significantly smaller in comparison with that of mini-batch train based training, which is $27988444/1829275 \approx 15.3$.

6.7 Evaluation of Memory Estimation Error

We present the memory estimation error for the LSTM aggregator in Table 7. The LSTM aggregator is memory consuming and estimating its memory consumption is more challenging than other common aggregators (e.g., pooling and Mean). We use the hidden size 256 in LSTM and the LSTM has 1 layer in our evaluation. In general, the error rate of our estimation is low (less than 8% in all cases). Though OOM rarely happens in our evaluation because the error rate of the estimation is low, it can still be triggered in theory. In the future, we plan to incorporate the estimation error into Betty's batch re-partitioning strategy if Betty detects that a micro-batch is getting close to the memory capacity.

7 CONCLUSIONS

GNN has been becoming a powerful tool to learn from graph-structured data. Continuing the success of GNN depends on not only GPU model innovation, but also system software support. In this paper, we target on a specific system problem faced by GNN: the memory capacity problem. We introduce a system support (named Betty) to enable GNN training with large and complex graphs, based on the idea of micro-batches to partition the large graph and fit into GPU memory. The method of graph partitioning in Betty is aligned with multi-level bipartite in GNN training. Recognizing the challenges brought by the bipartite structure for redundancy reduction and load balance, we introduce a new graph partition algorithm and system implementation to support the GNN training based on micro-batches. We show that Betty allows GNN training with billion-scale graphs on single GPU without OOM and losing training accuracy. As for future work, we plan to optimize the REG construction and graph partition to reduce the partitioning overhead. We also plan to extend Betty to multi-GPU training to speed up the training process.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive comments. This work was partially supported by U.S. National Science Foundation (OAC-2104116, CNS-1617967, CCF-1553645 and CCF-1718194) and the Chameleon Cloud.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact includes the source codes and expected experimental data for replicating the evaluations in this paper.

We implement figure 2 to denote the OOM situation of current advanced GNN training, and applied figure 10 to illustrate Betty break the memory wall. We implement memory consumption estimation during the workflow of Betty, shown in figure 5. We use figure 12 to denote the tendency of peak memory consumption and training time per epoch as the number of micro batches increases. And the model convergence is not impacted by Betty and micro-batch training can be proved by the figure 13.

The framework of Betty is developed upon DGL(pytorch backend). The requirements: pytorch >= 1.7, DGL >= 0.7. The other software dependency include sortedcontainers, pyvis, pynvml, tqdm, pymetis, seaborn.

Our experiments result denoted in paper were collected from the machine with a RTX6000 GPU(24 GB memory) and Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz. You can use a different configuration with at least one GPU.

A.2 Artifact Summary

- **Model:** In artifact evaluation, we mainly use GraphSAGE model to show the performance of Betty.
- **Data set:** The datasets used are ogbn-arxiv and ogbn-products, which can be download directly from Open Graph Benchmark(OGB) website.
- **Run-time environment:** Ubuntu18.04, CUDA 11.2 pytorch >= 1.7, DGL >= 0.7. It's also compatible with Ubuntu16.04, CUDA 10.1, the details can be found in github repo README.md 'install requirements'. Here, python 3.6 is the basic configuration in requirements. You also can use other python version, e.g. python3.8, but you need configure the corresponding pytorch and dgl version.
- **Hardware:** At least one GPU.
- **Metrics:** GPU memory usage and execution time.
- **Experiments:** To save time, we choose figure 5, 9, 2&10, 12, 13 to denote that Betty can reduce max memory consumption efficiently without change 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)?:** GNU General Public License v3.0
- **Data licenses (if publicly available)?:** MIT License

A.3 Description

A.3.1 How to access. You can obtain the artifact from here .

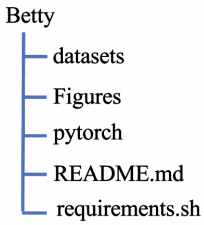

Figure 17: Directory structure of the artifact.

A.3.2 Hardware dependencies. The results denoted in paper were collected from the machine with a single RTX6000 GPU(24 GB memory), when the GPU you use with a different memory limitation, the OOM situation of mini batch train might be different. And to replicate the largest benchmark requires up to 60GB RAM. Except above, there are no other special hardware requirements.

A.3.3 Software dependencies. Ubuntu18.04, CUDA 11.2, python 3.6, pytorch 1.7 or higher, DGL 0.7 or higher. The main softwares will be used includes DGL, pytorch,sortedcontainers, pyvis, pynvml, tqdm, pymetis. The different version of software might have incompatible problems, please take care it when you install software.

A.3.4 Data sets. The datasets (OGBN-arxiv and OGBN-products) we use in artifact can be download directly from Open Graph Benchmark(OGB) Dataset.

A.3.5 Models. The model used in artifact is GraphSAGE with different aggregator, number of layers, hidden size and so on.

A.4 Installation

Obtain the artifact (see Section A.3.1), extract the archive files.

Then, download the benchmarks and generate full batch data into folder `/Betty/dataset/`. You can execute the bash file `gen_data.sh` in folder `/Betty/pytorch/micro_batch_train/` with fanout 10,25, then you can find the folder `/Betty/dataset/fan_out_10,25` contains pickle files of full batch data after sampling.

A.5 Experiment workflow

We display the directory structure of our artifact in Figure 17. The directory `pytorch` contains all necessary files for the micro-batch training and mini-batch training. In folder `micro_batch_train`, `graph_partitioner.py` contains our implementation of redundancy embedded graph partitioning. `block_dataloader.py` is implemented to construct the micro-batch based on the partitioning results of REG. The folder `Figures` contains these important figures for analysis and performance evaluation. In Section A.6, we describe how these scripts to replicate the results shown in these figures to help performance evaluation.

A.6 Evaluation and expected results

As we choose scripts of some figures to replicate the evaluation, hence, the commands need to be executed are mainly located in the folder `Figures`. For example, in `figure12/`, you can execute bash file to get the result of full batch, 2, 4, 8, 16, 32 micro batches and save result to folder `/Betty/Figures/figure12/log/`, then

execute `data_collection.py` to summarize the max memory and time consumption data of different micro batches training. These data are stored into a table shown in README.md in `figure12/`.

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

More details you can find in `README.md` in each figure folder.

REFERENCES

[1] Abraham Addisie, Hiwot Kassa, Opeoluwa Matthews, and Valeria Bertacco. 2018. Heterogeneous memory subsystem for natural graph analytics. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 134–145. https://doi.org/10.1109/IISWC.2018.8573480

[2] Ben Bogin, Matt Gardner, and Jonathan Berant. 2019. Representing schema structure with graph neural networks for text-to-SQL parsing. *arXiv preprint arXiv:1905.06241* (2019). https://doi.org/10.1145/3534678.3539294

[3] DGL. [n. d.]. Deep Graph Library. https://www.dgl.ai/.

[4] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019). https://doi.org/10.48550/arXiv.1903.02428

[5] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed deep graph learning at scale. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 551–568.

[6] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017). https://doi.org/doi/10.5555/3294771.3294869

[7] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 875–890. https://doi.org/10.1145/3373376.3378465

[8] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. *CoRR* abs/2103.09430 (2021). https://doi.org/10.48550/arXiv.2103.09430

[9] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133. https://doi.org/10.48550/arXiv.2005.00687

[10] Yaochen Hu, Amit Levi, Ishaan Kumar, Yingxue Zhang, and Mark Coates. 2020. On Batch-size Selection for Stochastic Training for Graph Neural Networks. (2020).

[11] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1341–1355. https://doi.org/10.1145/3373376.3378530

[12] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A Distributed Multi-GPU System for Fast Graph Processing. *Proc. VLDB Endow.* 11, 3 (2017), 297–310. https://doi.org/10.14778/3157794.3157799

[13] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[14] George Karypis and Vipin Kumar. 1995. METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0. (1995).

[15] Brian W Kernighan and Shen Lin. 1970. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal* 49, 2 (1970), 291–307. https://doi.org/10.1002/j.1538-7305.1970.tb01770.x

[16] Diederik P. Kingma and Jimmy Ba. 2018. Graph Attention Networks. In *International Conference for Learning Representations (ICLR)*. https://doi.org/10.48550/arXiv.1710.10903

[17] Tung D Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. 2018. Tflms: Large model support in tensorflow by graph rewriting. *arXiv preprint arXiv:1807.02037* (2018). https://doi.org/10.48550/arXiv.1807.02037

[18] John Boaz Lee, Ryan Rossi, and Xiangnan Kong. 2018. Graph Classification Using Structural Attention. In *International Conference on Knowledge Discovery and Data Mining*. https://doi.org/doi/pdf/10.1145/3219819.3219980

[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. *Proceedings of Machine Learning and Systems* 1 (2019), 120–131. https://doi.org/10.48550/arXiv.1903.12287

[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). https://doi.org/10.48550/arXiv.2006.07739

[21] 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*. https://doi.org/10.1145/3419111.3421281

[22] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. 2010. GraphLab: A New Framework for Parallel Machine Learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*. https://doi.org/doi/10.5555/3023549.3023589

[23] 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. https://doi.org/10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00056

[24] 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 (USENIX ATC 19)*. 443–458. https://doi.org/doi/10.5555/3358807.3358845

[25] Fragkiskos D Malliaros and Michalis Vazirgiannis. 2013. Clustering and community detection in directed networks: A survey. *Physics reports* 533, 4 (2013), 95–142. https://doi.org/10.1016/j.physrep.2013.08.002

[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. https://doi.org/10.1145/3476483

[27] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. 2017. Training deeper models by GPU memory optimization on TensorFlow. In *Proc. of ML Systems Workshop in NIPS*, Vol. 7.

[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). https://doi.org/doi/10.5555/3454287.3455008

[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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 891–905. https://doi.org/10.1145/3373376.3378505

[30] PyG. [n. d.]. PyTorch Geometric. https://pytorch-geometric.readthedocs.io.

[31] 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. https://doi.org/10.1145/3458817.3476205

[32] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2020. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA51647.2021.00057

[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 (USENIX ATC 21)*. 551–564. https://doi.org/10.48550/arXiv.2101.06840

[34] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13. https://doi.org/doi/10.5555/3195638.3195660

[35] Haşim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *arXiv preprint arXiv:1402.1128* (2014). https://doi.org/10.48550/arXiv.1402.1128

[36] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2017. Modeling Relational Data with Graph Convolutional Networks. *CoRR* abs/1703.06103 (2017). https://doi.org/10.48550/arXiv.1703.06103

[37] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Magazine* 29, 3 (2008), 93–106. https://doi.org/10.1609/aimag.v29i3.2157

[38] 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. https://doi.org/10.48550/arXiv.1902.09130

[39] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory

Management for Training Deep Neural Networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 41–53. https://doi.org/10.1145/3178487.3178491

[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). https://doi.org/10.48550/arXiv.1909.01315

[41] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks? *arXiv preprint arXiv:1810.00826* (2018). https://doi.org/10.48550/arXiv.1810.00826

[42] Hongxia Yang. 2019. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 3165–3166. https://doi.org/10.14778/3352063.3352127

[43] Lizhi Zhang, Zhiquan Lai, Yu Tang, Dongsheng Li, Feng Liu, and Xiaochun Luo. 2021. PCGraph: 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, 279–287. https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00048

[44] 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 (IA3)*. IEEE, 36–44. https://doi.org/10.1109/IA351965.2020.00011

[45] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, Qidong Su, Minjie Wang, Chao Ma, and George Karypis. 2021. Distributed hybrid CPU and GPU training for graph neural networks on billion-scale graphs. *arXiv preprint arXiv:2112.15345* (2021). https://doi.org/10.48550/arXiv.2112.15345

[46] Ao Zhou, Jianlei Yang, Yeqi Gao, Tong Qiao, Yingjie Qi, Xiaoyi Wang, Yunli Chen, Pengcheng Dai, Weisheng Zhao, and Chunming Hu. 2021. Optimizing memory efficiency of graph neural networks on edge computing platforms. *arXiv preprint arXiv:2104.03058* (2021). https://doi.org/10.1109/RTAS52030.2021.00048

Received 2022-07-07; accepted 2022-09-22