Graph Neural Network Training with Data Tiering

Seung Won Min
University of Illinois at Urbana-Champaign
min16@illinois.edu

Kun Wu
University of Illinois at Urbana-Champaign
kunwu2@illinois.edu

Mert Hidayetoğlu
University of Illinois at Urbana-Champaign
hidayet2@illinois.edu

Jinjun Xiong
University at Buffalo
jinjun@buffalo.edu

Xiang Song
AWS AI Research and Education
xiangsx@amazon.com

Wen-mei Hwu
University of Illinois at Urbana-Champaign / NVIDIA
whwu@nvidia.com

ABSTRACT

Graph Neural Networks (GNNs) have shown success in learning from graph-structured data, with applications to fraud detection, recommendation, and knowledge graph reasoning. However, training GNN efficiently is challenging because: 1) GPU memory capacity is limited and can be insufficient for large datasets, and 2) the graph-based data structure causes irregular data access patterns. In this work, we provide a method to statistically analyze and identify more frequently accessed data ahead of GNN training. Our data tiering method not only utilizes the structure of input graph, but also an insight gained from actual GNN training process to achieve a higher prediction result. With our data tiering method, we additionally provide a new data placement and access strategy to further minimize the CPU-GPU communication overhead. We also take into account of multi-GPU GNN training as well and we demonstrate the effectiveness of our strategy in a multi-GPU system. The evaluation results show that our work reduces CPU-GPU traffic by 87–95% and improves the training speed of GNN over the existing solutions by 1.6–2.1× on graphs with hundreds of millions of nodes and billions of edges.

CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**; • **Information systems** → **Data layout**; • **Theory of computation** → **Data structures design and analysis**.

KEYWORDS

graph neural networks, gpu-acceleration, very large data

ACM Reference Format:
Seung Won Min, Kun Wu, Mert Hidayetoğlu, Jinjun Xiong, Xiang Song, and Wen-mei Hwu. 2022. Graph Neural Network Training with Data Tiering. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22), August 14–18, 2022, Washington, DC, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3534678.3539038

KDD '22, August 14–18, 2022, Washington, DC

ACM ISBN 978-1-4503-9385-0/22/08…$15.00
https://doi.org/10.1145/3534678.3539038

1 INTRODUCTION

Graph neural networks (GNNs) have shown promising successes on multiple graph-based machine learning tasks including fraud detection [15], recommendation [5], search and knowledge graph reasoning [3]. With a rapidly growing need to apply GNNs in various domains, there are several efforts from the community to provide open-source GNN-specific machine learning frameworks such as PyTorch Geometric (PyG) [6], Deep Graph Library (DGL) [25], and Spektral [7]. Those GNN frameworks implement several highly optimized message passing operators and graph-specific computation layers which were lacking in the previous DNN frameworks.

Yet, the challenges of GNN training are not limited to the implementations of message passing operators or computational layers. With the successes of using large datasets in machine learning to increase the training accuracy [20, 27], the importance of using larger graphs took a place in GNN training as well [9]. The number of nodes and the edges of these graphs reach millions to billions [23, 28] and the graphs with such scales make the ordinary naïve software/hardware approaches ineffective.

The earlier implementations of GNN were mostly focusing on a small scale graphs [12, 24] and assumed the whole graph fits into a single GPU memory. Therefore, previously, accessing an arbitrary node's feature data was merely a process of indexing the GPU's own memory space. However, for large graphs whose node/edge feature data cannot fit into the GPU memory, at least part of the graph needs to be placed into the CPU memory. One common practice to train GNNs in such scenario is to create a smaller set of problem by performing a mini-batched training. With the mini-batch training, only a subset of nodes are randomly picked along with their neighboring nodes and sent to GPU. This method is very effective when training GNNs on large graphs as it practically reduces the memory footprint of the application. Not only that, several recently introduced GNN models [2, 21] showed that the mini-batched based approaches are superior in achieving high training accuracy as well.

A mini-batch training process that places the all or part of the input graph feature data in the CPU memory needs to frequently transfer mini-batch data from CPU to GPU through a slow PCIe interconnect. Furthermore, the minibatch method amplifies the total amount of data access because the different minibatches can have overlapping nodes. Due to these reasons, training GNN is often throttled by CPU-GPU data transfer time. In many of our measurements, we find the GPU is only about 30-40% utilized during the GNN training when the datasets do not fit in GPU memory.

To remedy this problem, in this work, we introduce *Data Tiering* in GNN, which does not inflict any algorithmic changes on the training models unlike the previous approaches, but yet dramatically reduces CPU-GPU data transfer volume. Our data tiering improves GNN training in two ways. First, it provides a statistical method by using reverse pagerank to effectively predict the importance of each node in the input graphs and identifies which nodes should be located in the GPU memory. Second, it introduces a hardware-friendly data placement and access strategy which minimizes the cost of accessing cold data in CPU memory. Our data placement and access strategy is comprehensive and it also enables more advanced optimization techniques for the multi-GPU systems with high speed GPU-to-GPU interconnects.

We evaluate our work using public frameworks PyTorch and DGL. The demonstration of our work on realistic mini-batched training shows that our approach eliminates PCIe traffic by 87–95% in various datasets by loading only 10% of them into GPU memory. With the data transfer time optimization alone from our data tiering strategy, we find the training speeds of the existing GNN implementations can be improved by 1.6–2.1×. To demonstrate the scalability of our work, we also train a dataset with 350GB of size in a system with four NVIDIA V100 32GB GPUs.

2 BACKGROUND

2.1 GNN and Node Classification

GNNs are a series of multi-layer feedforward neural networks that propagate and transform layer-wise features following a graph structure. Among these models, a graph convolutional network (GCN) [12] architecture is widely employed, which relies on the layer-wise message passing scheme. Formally, the $(l+1)$-th layer of a GNN is defined as:[1]

$$H^{(l+1)} = \sigma(f_w(\mathcal{A}, H^l)), \tag{1}$$

where the function $f_w(\mathcal{A}, H^l)$ is determined by learnable parameters w and $\sigma(\cdot)$ is an optional activation function. $\mathcal{A}$ represents an adjacency matrix of the input graph composed of N nodes and E edges. Additionally, H^l represents the embeddings of the nodes in the l-th layer, and H^0 is initialized with the nodes' input features X. The node input features are stored in a $N \times D$ matrix where N is the total number of nodes of the input graph and D is the dimension of each node feature. The size of node feature varies significantly depending on the dataset, but the typical sizes are in between 512B to 4KB. When the node feature size is multiplied with the total number of nodes in a graph, the node feature matrix (or tensor) can reach tens of gigabytes to hundreds of gigabytes. Therefore, storing the node feature tensors of GNN datasets is the most difficult task with the large graphs.

When l is identical to the last layer of GNN, the corresponding H^l tensor is the output embedding tensor Z. The output embedding tensor Z is used to create predicted labels and to classify unclassified nodes. For training purposes, if the nodes already have ground truth labels, then the predicted labels are compared with them to perform a backpropagation and a model update.

[1]We omit edge features for simplicity.

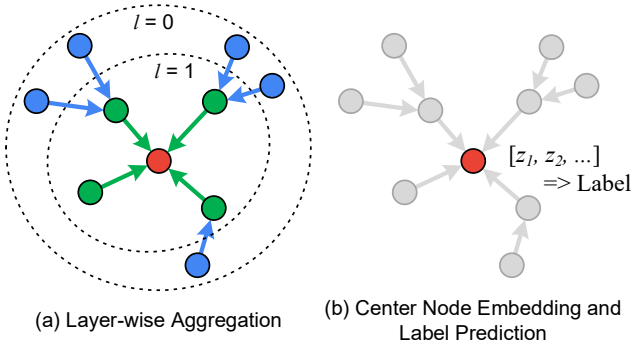

Figure 1: Node feature aggregation and label prediction.

2.2 Neighborhood Sampling

The layer-wise aggregation implementation of GNN provides a promising way of gathering relational information from graphs, but it requires reading all neighboring nodes of each layer. With a large graph, this approach quickly shows a scalability challenge as the number of nodes that we need to read exponentially grows with an increasing number of layers. To alleviate this scalability problem, GraphSAGE [8] introduces neighborhood sampling and aggregating approach. By sampling a fixed number of neighboring nodes per target node instead of demanding the whole adjacency matrix, the neighborhood sampling reduces the computation and memory footprints. With the predefined numbers of sampling per layer, we can also effectively control the size of each mini-batching in both training and inference.

Neighborhood sampling is applied to every neighboring node in every aggregation step. GraphSAGE uses a uniformly random selection process to sample the neighboring nodes to provide an enough randomness to the training process. The commonly used hyperparameters for the neighborhood sampling size S_{layer} are $(S_1, S_2) = (25, 10)$ for a 2-layer sampling approach and $(S_1, S_2, S_3) = (10, 10, 10)$ for a 3-layer approach. It is uncommon to go beyond the three layers of sampling because of the need to limit the size of mini-batch. After the sampling, a sub-graph which only contains the sampled nodes is created so the computation kernel knows how to aggregate the node features of interest. Over different iterations of training, a new sampling is done to increase the learning entropy.

If the node feature tensor is too large to fit into the GPU memory, the features of the sampled nodes must be transferred to GPU from the CPU memory in runtime. Due to the slow PCIe interconnect between CPU and GPU, this data transfer process can be quite time consuming. In this paper, we present an optimization technique called *data tiering* that exploits locality in accessing feature data and minimizes the need for cross-PCIe accesses.

3 DATA TIERING

3.1 Score Function

By definition, the neighborhood sampling process is random and it is difficult to exactly predict which nodes will be sampled during training. Thus, we must statistically approach the problem of identifying and exploiting locality.

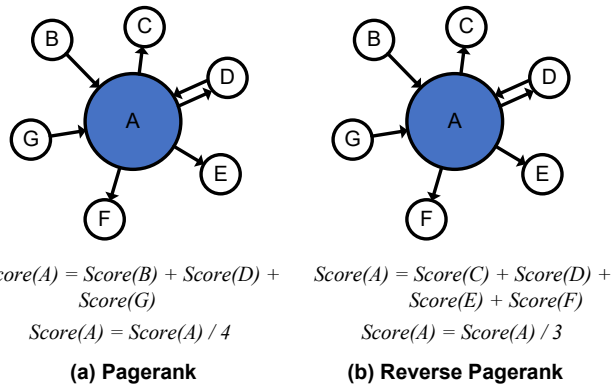

Figure 2: Snapshot of Pagerank vs. Reverse Pagerank. Only a single iteration of algorithms shown. In case of regular pagerank, the score is divided by the out-degree, but in case of reverse pagerank, the score is divided by the in degree.

[Method 1] Node Out-Degree: The first metric we can use is the out-degree of each node in the input graph. With a high out-degree, even if the node is not selected in a specific run of neighborhood sampling, the cumulative chance of the node being selected during the entire training process is higher than the less connected nodes. Considering that we perform quite significant number of sampling per training epoch for the large graphs, this prediction is statically reasonable as we empirically prove it in Section 5.2. In case of ogbn-papers100M, we sample about 130 millions of nodes per training epoch.

[Method 2] Reverse-PageRank: The second option is a reverse pagerank (R-Pagerank) [1]. In the original pagerank, the score of each node is higher if the in degree is higher and the out-degree is lower. For the reverse pagerank, it is the opposite. In Figure 2, we depict the difference between the original pagerank and the reverse pagerank further. For simplicity, we only show a case of node A with single iteration, but this is done for all nodes until the score values converge in the real implementation. In the original pagerank, to calculate the score of the source node, we sum the scores of the nodes which are targeting the source node and divide the summed score by the out-degree of the source node. Now with the reverse pagerank, we sum the scores of the nodes which are targeted by the source node and divide the summed score by the in degree of the source node. The idea behind this mechanism is that if a certain node A has many outgoing edges, it can potentially get a higher score by summing many nodes' scores. Therefore, if there is another node B which is targeting node A, node B also gets a higher score by adding the score of node A.

In the context of neighbor sampling, the scoring mechanism of the reverse pagerank can be understood by referring to Figure 1. The green nodes are sampled while generating the embedding for the red node, referred to a node A, because they have an outgoing edge to A. The blue nodes are sampled because they have an outgoing edge to the green nodes. Therefore, If a node can reach many nodes directly or indirectly through its outgoing edges, it is likely to be picked during the sampling process. Since the probability of node A being picked is high, the other nodes which are targeting this node also has a relatively higher probability of being picked when we are sampling multiple layers. However, if the node A also has a high in-degree, because now there are so many nodes which can be sampled from node A, the other nodes should less expect them to be sampled when the node A was selected. Thus, in this case, we divide the score of node A by the in degree before propagating it to the other nodes so these nodes receive less increase to their estimated probability of being picked during sampling.

Algorithm 1 Weighted Reverse Pagerank

```
1:  Input: graph g, iteration iter, damp d, train_id tid
2:  num_node = num_nodes(g)
3:  num_train = length(tid)
4:  for i = 0 to num_node − 1 do
5:      score[i] = 1/num_node
6:      in_degree[i] = num_in_degree(g, i)
7:  end for
8:  weight = num_node/num_train
9:  for i = 0 to num_train − 1 do
10:     score[tid[i]] = score[tid[i]] ∗ weight
11: end for
12: for i = 0 to iter − 1 do
13:     for j = 0 to num_node − 1 do
14:         score[j] = score[j]/in_degree[j]
15:     end for
16:     pull_from_neighbor(g, score)
17:     for j = 0 to num_node − 1 do
18:         score[j] = (1 − d)/num_node + d ∗ score[j]
19:     end for
20: end for
```

The potential advanatge of using the reverse pagerank over the simple degree method is to capture further multi-layer sampling patterns. For the simple degree method, the information we can capture is limited to a single hop of relationship, while the actual neighbor sampling can extend to multiple hops. On the other hand, in reverse pagerank, the score value of each node is propagated to multiple layers of nodes away until the score converges to a certain limit. Therefore, by using the reverse pagerank, it is possible to capture the subtle pattern of multi-layer sampling in the neighbor sampling in a better way.

[Method 3] Weighted Reverse-PageRank: The third option is to further incorporate the labeling status of the nodes into the reverse pagerank method. As we explained in Section 2.1, the goal of GNN training is to create a model which can predict the labels for the unlabeled nodes. To train such models, we must be able to compare the predicted labels with the ground-truth labels. Therefore, during training, the nodes which we can pick to start the neighbor sampling are reduced to the nodes that come with with labels. This means that, if we can create a method to statistically put further emphasis to those nodes and their surrounding nodes, we can compress the search space.

To do this, we add some tweaks on top of the reverse pagerank algorithm by uniformly applying a weight value to the labeled nodes. The detailed implementation of the weighting process is described in Algorithm 1. First, before we decide how to weight the labeled nodes, we need to decide how much we want to weight

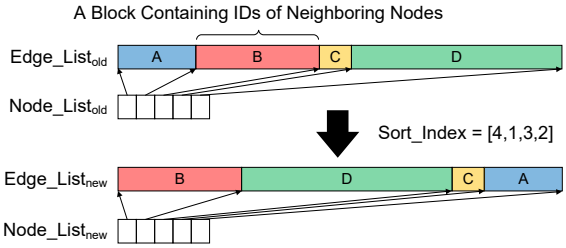

Figure 3: Adjacency Matrix Reordering Overview.

them. The assumption behind the weighting is that by knowing the exact starting locations of the neighbor sampling, we can more focus on those nodes and their surroundings. This means, that if there are few starting nodes available in the graph, the sampling tendency will be more biased toward them and their surrounding nodes. In the opposite, if every node can be selected as a starting node, there is no starting bias and simply the nodes with high out-degree is likely to be selected during the sampling. As a result, the weighting intensity should be high if there are few labeled nodes, and the weighting intensity should be low if there are many labeled nodes. In our algorithm, we reflect this by defining weight = (# of all nodes) / (# of labeled nodes).

Next, the actual weighting is done by multiplying the initial scores of the labeled nodes with the weight value we calculated (Algorithm 1, Line #10). In the original pagerank algorithm, the default initial score of all nodes is 1 divided by the total number of nodes in the graph. In general, by running pagerank long enough, the initial impact of the initial scores wear down and the scores start to converge to certain values. To avoid this, we do not run our weighted reverse pagerank until the scores converge, but only five iterations. The rest of the algorithm is identical to the reverse pagerank algorithm. We call this algorithm as weighted reverse pagerank (Weighted R-Pagerank).

3.2 Adjacency Matrix Reordering

With the score values, now we want to split the node features into a high score group and a low score group. The simplest way to achieve this is to sort the node features in the descending order of the score values and divide them into top X% of hot portion and bottom 100-X% of cold portion. To maintain the correctness of the GNN function, reordering the node feature tensor H in Section 2.1 also requires reordering the adjacency matrix $\mathcal{A}$.

However, unfortunately, the process of reordering the adjacency matrix is less intuitive than reordering the node feature tensor because the adjacency matrix is often represented by a sparse matrix format. Due to it's non-straightforward nature, the current implementation of adjacency matrix reordering in existing frameworks like DGL has a simple sequential implementation, but the sequential approach may consume significant amount of time when the input graphs have hundreds of millions of nodes. Therefore, in this work, we implement our own parallel version of algorithm to accelerate the reordering process.

To better understand our implementation, we first briefly explain the adjacency matrix reordering problem in general (Figure 3). In compressed sparse representation (CSR), the adjacency matrix is divided into an edge list and a node list. The edge list is a collection of many neighbor lists where each contains the IDs of the connected nodes. To reorder an adjacency matrix, we need to perform the following three tasks: First, we need to create a new node list which contains the new starting indices of the neighbor lists. Second, we need to reorder the neighbor lists based on the new starting indices. Third, we need to relabel all node ID values inside all neighbor lists.

The key to parallelize the workloads is to generate a full mapping of old to new IDs in advance so the ID translations in the later processes become simple table lookup processes. The old to new ID mapping table is created based on the score values we generated from Section 3.1. The indices in this mapping table indicates the new placing order of the old neighbor lists. For example, if the table has [4, 1, 3, 2], that means the 1st neighbor list (A in Figure 3) now should be placed in the 4th place, the 2nd neighbor list (B in Figure 3) now should be placed in 1st place, the 3rd neighbor list (C in Figure 3) now should be placed in 3rd place, and so on. Because the sparse matrix $\mathcal{A}$ has varying row sizes unlike the simple 2-D matrix H, the new starting index of each neighbor list should be calculated using cumulative sum.

Once the cumulative sum is done, we simply need to copy & paste the old neighbor lists into the new edge list based on the new starting indices. This process can be easily done in parallel. After relocating the neighbor lists, now we need to update the ID values in each neighbor list as well. The update can be done by simply looking up the previously set mapping table and this process can be also easily done in parallel. When we define n as a number of nodes and e as a number edges, the time complexity of this algorithm is either $O(nlogn)$ due to the score sorting during the mapping table creation, or $O(e)$ when the number of edges is very large. However, thanks to our parallelizable approach, we find the end-to-end adjacency matrix reordering takes only about 31 seconds with ogbn-papers100M dataset (Table 1), which has 111M nodes and 3.2B edges.

4 DATA PLACEMENT AND ACCESS

4.1 Memory Allocation and Indexing Scheme

The overall data placement and access strategy is shown in Figure 4. With the sorted node feature tensor, the *hot data* portion with a high score is placed in the GPU memory and the rest of *cold data* portion with a low score is placed in the CPU memory. From the user application perspective, we provide a single monolithic and contiguous fake view of the two tensors so the user application can use the existing array indexing scheme.

For the cold data access, it is important to maintain a low end-to-end data transfer overhead since crossing over PCIe is already a huge burden. One the of most common mistakes made by programmers during CPU-GPU data communications is that often the programs spend too much time on coordinating the data transfer. Using cross-device data copy engines like DMA is wide spread, but it is only effective when the size of data that we want to transfer is large enough. In the case of node feature sampling and aggregation, the data transfer size is typically between 512B and 4KB, much smaller than the size required to make DMA transfers efficient.

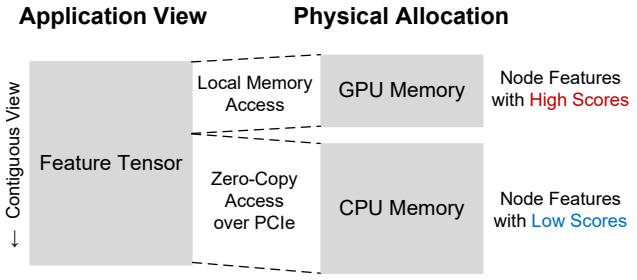

Figure 4: Simple data placement and access method overview.

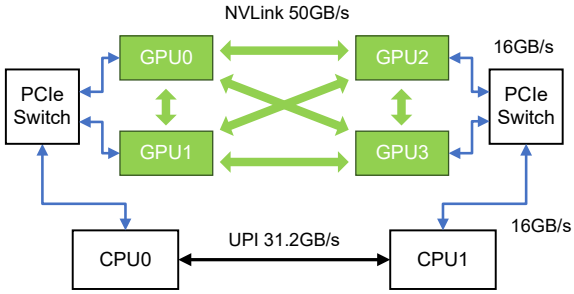

Figure 5: An example system with four NVIDIA V100 GPUs connected over NVLink. Unidirectional bandwidths shown.

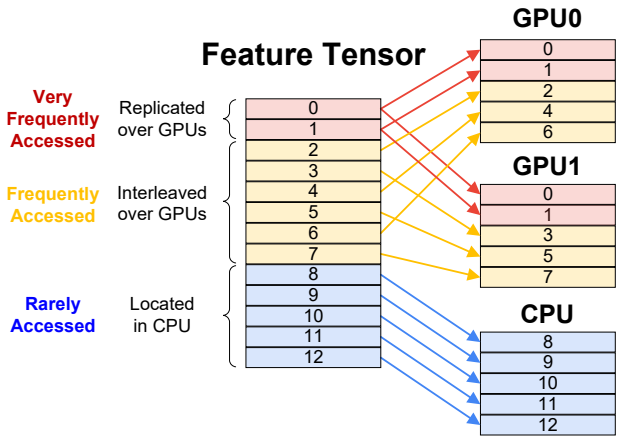

Figure 6: Available permutation of data placement in multi-GPU environment. Red: replicated in multiple GPUs. Yellow: interleaved over multiple GPUs. Blue: located in CPU.

Therefore, in this work, we take a GPU-centric approach to accessing data instead of depending on DMA or CPU. At the hardware level, the GPU-centric method is enabled by using the zero-copy access capability of NVIDIA GPUs. The zero-copy accessible CPU memory space is mapped into the GPU page table and it allows us to directly access the CPU memory space from the GPU kernel code. At the user (application) level, we utilize the DGL's UnifiedTensor [17] implementation to enable this data access mechanism.

With our data placement and access strategy, the overall flow of node feature access in GNN training is as follows. First, the neighbor sampling function traverses the input graph and generates a list of sampled node IDs. Next, the node IDs are sent to GPU(s) and GPU threads start accessing the node features with the node IDs. While reading the node IDs, the GPU threads check if the ID values are within the hot data boundary set by the users. If the ID values are within the boundary, then the threads use a pre-stored GPU memory pointer and take the advantage of fast local memory access. If the node ID is outside the boundary, then the threads use a pre-stored CPU memory pointer and perform the zero-copy access.

One major benefit of our approach is that we do not inflict any changes in the original programming structure. Our implementation of the fake tensor can provide a same experience to the users as if they are accessing a single large tensor like how they accessed previously. The hardware level details can be hidden behind the framework and the user-level understanding requirement is minimal. Unlike the other large GNN training solutions such as Roc [11] and NeuGraph [16] which mandate a new holistic pipeline restructuring for the data transfer optimization, only changing about 2-3 lines of code would be sufficient for our approach.

4.2 Tensor Distribution over Multiple GPUs

Similar to the other neural network training methods, GNN training also extensively utilizes multiple GPUs to further accelerate the training process. With fast GPU-to-GPU interconnects like NVLink, we can create a larger pool of collective GPU memory space (Figure 5) from multi-GPU systems. In Figure 6, we show the complete view of our data tiering strategy in a multi-GPU system. To load the hot data into this collective memory space, instead of using a naïve blocked partition method, we use an interleaved data loading method. Since the node feature tensor is sorted in a descending order of the score, a simple block partitioning scheme can result in unbalanced memory and interconnect bandwidth consumption across GPUs. With the combined GPU memory space, we can hold a larger portion of hot data in a faster tier of memory space. The specific usage of CUDA APIs to enable our data placement strategy in multi-GPU environment is explained in Appendix A.

In our experience, the NVLink bandwidth is fast enough to hide most of the data transfer time of GNN training, but in case the data transfer time is still an issue, we can additionally replicate some hot data over multiple GPUs. In this case, the most frequently accessed data will come from the local GPU memory, the next most frequently accessed data from the peer GPU memory, and the least frequently accessed data from the CPU memory. The generation of the combined tensors is fully automated in our implementation. To generate this combined tensor, users simply need to provide the sizes of local GPU tensor and multi-GPU tensor, and the number of GPUs connected over NVLink. If there are no high speed links between GPUs, the mapping simply falls back to Figure 4. The optimal distribution factor of each GPU's memory capacity between the replicated hot data and the interleaved hot data in the collective memory space may vary depending on the dataset. For our experiments, we simply maximize the multi-GPU tensor and do not utilize the replicated local GPU tensor.

Table 1: Evaluation Datasets.

Name	#Nodes	#Edges	Node Feature Total Size
ogbn-papers100M	111.1M	3.2B	53GB
MAG240M	244.2M	3.5B	350GB
WikiKG90M	87.1M	1.0B	125GB

5 EVALUATION

5.1 Methodology

5.1.1 Application & Dataset. For the evaluation, we implement data tiering on PyTorch and DGL. Since neither of the frameworks support kernel-level direct peer GPU memory access, we modify their tensor implementations to enable it. Currently, the frameworks can only perform peer-to-peer DMAs. We use GraphSAGE implementation of DGL to explore various neighbor sampling strategies. For the dataset, we use the following three from Open Graph Benchmark (OGB) [10]: ogbn-papers100M, MAG240M, and WikiKG90M. WikiKG90M is from a different task domain and does not come with the labels needed for node classification but due to the lack of public large datasets, we repurpose it as a node classification task dataset with synthetic label values. Further details of the datasets are shown in Table 1. To make our experiment realistic, we use the carefully tuned hyper parameters for different datasets which are taken from previous GNN training works with high accuracy models [18, 21, 22]. Based on the previous works, we use (12,12,12) as neighbor sampling fanout parameter for ogbn-papers100M and (25,15) for MAG240M. For WikiKG90M, we use the identical parameters used in ogbn-papers100M training.

5.1.2 Hardware. Throughout the evaluation, we use a machine with two Intel Xeon Gold 6230 CPUs and four NVIDIA V100 32GB GPUs (Figure 5). All NVIDIA V100 GPUs are connected over NVLink with 50GB/s unidirectional bandwidth per connection. Because each GPU is connected to three other peer GPUs, the aggregated NVLink bandwidth is 3×50GB/s = 150GB/s for each GPU.

5.2 Score Function vs. Measured Data Reuse

In this experiment, we try to find if the score functions we discussed in Section 3.1 can correctly predict the data reusability in real GNN training. In Figure 7, we list the nodes of graphs in the X-axis in descending order of scores with the three different scoring functions: node degree, reverse pagerank, and weighted reverse pagerank. In the Y-axis, we show the measured access frequency of each node during GNN training, in a cumulative fashion. In general, we find all functions can provide some benefits when we perform data tiering in GNN training. For example, based on the scores calculated, when we keep top 10% of nodes, we can expect at least 35% of hit during the training regardless of the dataset. By keeping top 25% of nodes, the minimum hit ratio further increases to 56%.

Also, in general, we find it becomes easier to predict which nodes would have high data access counts if a graph has a more extreme power law distribution. For example, WikiKG90M graph has an extremely unbalanced edge connectivity and 80% of edges in the entire graph are connected to only 1% of nodes. With such extremely concentrated connections, we can observe that simply choosing a few nodes with the highest degrees automatically guarantees a very high hit ratio during the neighbor sampling. In cases of ogbn-papers100M and MAG240M, the edge connectivities of the datasets are more balanced and the ratios are 32% and 46%, respectively. However, even though the simple degree method can be effective for certain graphs, we find the weighted reverse pagerank is more preferable in general because it consistently gives the best prediction result.

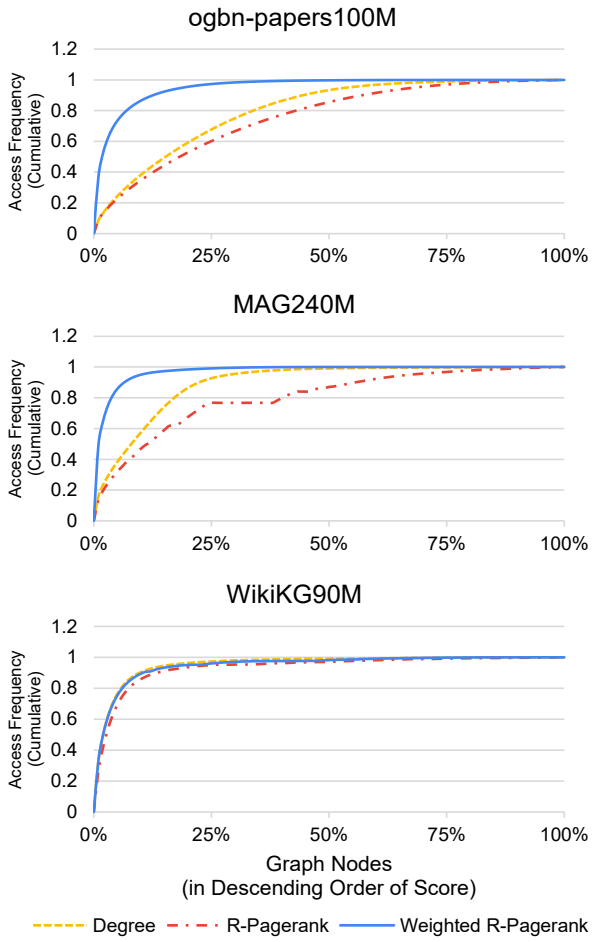

Figure 7: Access frequency distribution comparison on different datasets with different score functions. It is easy to cluster frequently accessed nodes with Weighted R-Pagerank function.

5.3 GNN Training Time (Single GPU)

In this section, we evaluate the actual benefit of our work in GNN training. We compared the performance of our work against the following two existing methods: 1) CPU gathering and 2) zero-copy access. The CPU gathering method relies on CPU to gather node features and then utilize GPU DMA engine to copy the gathered node features into GPU memory. Due to the additional data gathering

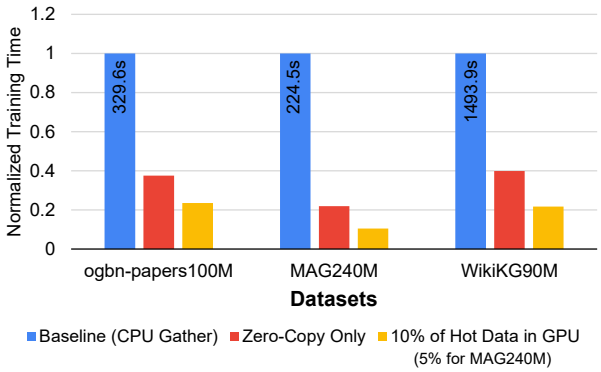

Figure 8: Single epoch training time comparison.

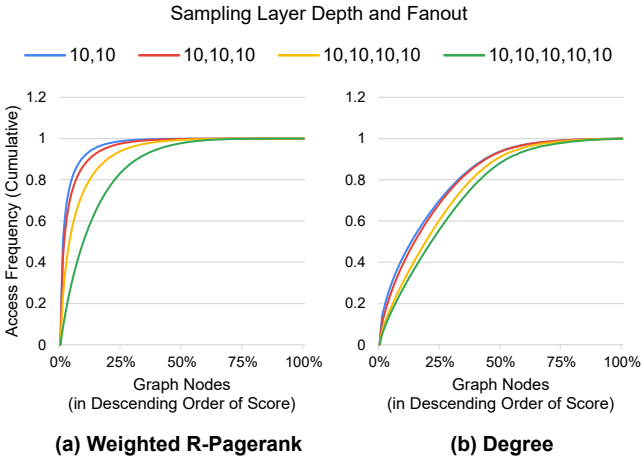

Figure 9: Access frequency distribution comparison of using different neighbor sampling depths in ogbn-papers100M.

process, this method not only wastes the CPU memory bandwidth, but also adds a non-negligible amount of data transfer latency to GPU. This is the only way currently available in PyTorch to transfer scattered data in CPU memory to GPU memory.

The zero-copy access method is a method recently introduced in DGL to overcome the data gathering overhead in the CPU gathering method. With this method, GPU kernels can directly dereference CPU memory pointers and thus we do not need to rely on CPU to gather data for the GPUs. To enable the zero-copy access capability, DGL implements a new class of tensor called UnifiedTensor which transforms the CPU tensor of PyTorch into a zero-copy accessible tensor. In UnifiedTensor, the specific task is done by utilizing `cudaHostRegister()` API from CUDA on top of existing CPU memory allocation. The further technical detail is identical to the process explained in Appendix A.2.

For our work, we first score the nodes with the weighted reverse pagerank function like in Figure 7, and then reorder the node feature tensor and the graph nodes in the datasets. For this experiment, we load 10% of hot data for ogbn-papers100M and WikiKG90M, but only 5% for MAG240M due to the GPU memory limitation.

In Figure 8, we show the overall comparison. From this comparison, we can first observe that relying on CPU to gather data results in seriously increasing the overall training time. In this case, we find that the GPU is only about 10-30% utilized and mostly idling. By adopting the zero-copy access method, the training performance is visibly improved (2.5-4.6×). The zero-copy only method does not leverage any temporal data locality strategies, but simply removing CPU from the data access path significantly increases the overall performance. Finally, with our method, the training performance is further improved by 1.6-2.1× on top of the zero-copy only method. Considering that we have run the entire experiment on top of PyTorch and DGL with python, the benefits that we observe are immediately deliverable to the regular users as well.

5.4 Case Study: ogbn-papers100M

Now, we take ogbn-papers100M as an example for more detailed analysis. First, even though we know that the most of existing GNN works use two to three layers of sampling depths, we still like to know how much increasing the sampling depth can affect the data tiering efficiency. To understand the impact, we use different sampling depths during the GNN training and observe how the node access frequency distribution varies. In Figure 9, we show two access frequency charts similar to Figure 7, but now with the varying sampling parameters of (10,10), (10,10,10), (10,10,10,10), and (10,10,10,10,10). For the score functions, we use the weighted reverse pagerank and the degree count. For both cases, we can observe the accesses are now more spread out with the deeper sampling parameters. This is an expected behavior because with a deeper sampling depth, the graph coverage of each minibatch becomes larger and we start to access secluded nodes more frequently.

However, even with the spread out in the deeper sampling cases, we can still identify a significant portion of accesses are made to a few selected nodes. For example, with the (10,10,10,10,10) sampling parameter, top 10% of the highest score nodes of the weighted reverse pagerank and the degree count functions account for 52% and 28% of the entire accesses, respectively. This experiment result shows that the benefit of data tiering is not immediately nullified with a growing sampling layer depth and it gives some room for the future GNN models which may attempt to sample deeper.

For the second analysis, we would like to more closely: 1) verify the hardware-level benefit of data tiering and 2) observe what is the impact of controlling the portion of data loaded to GPU. To better understand these, we sweep the portion of data loaded in GPU during GNN training and measure the volume of PCIe traffic and the training time (Figure 10). For this experiment, we perform data tiering with the weighted reverse pagerank function on ogbn-papers100M. To measure the PCIe traffic, we use the NVIDIA profiling tool *nvprof*. As we can see, our data placement and access strategy effectively reduces the PCIe traffic with more hot data loaded into the GPU memory. When we compare the cases with no data loading and 25% of data loading, we can achieve about 97% of PCIe traffic reduction. At this point, most of the node feature accesses are resolved within GPU and only very few data accesses need to be directed to the CPU memory over slow PCIe.

The performance gains in GNN training show a similar trend to the PCIe traffic reduction. With the 5% of data loaded, we can

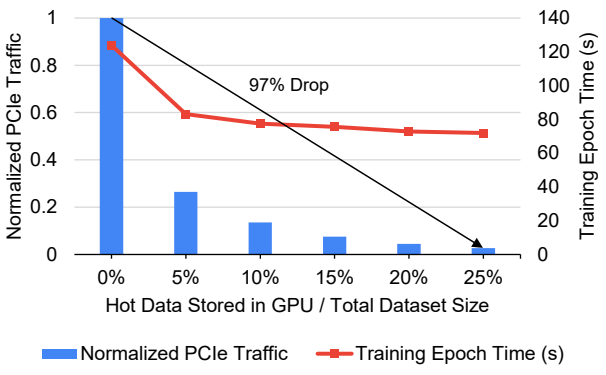

Figure 10: Single GPU PCIe traffic and training time comparison in actual GNN training with increasing hot data portion loaded in GPU. ogbn-papers100M dataset used.

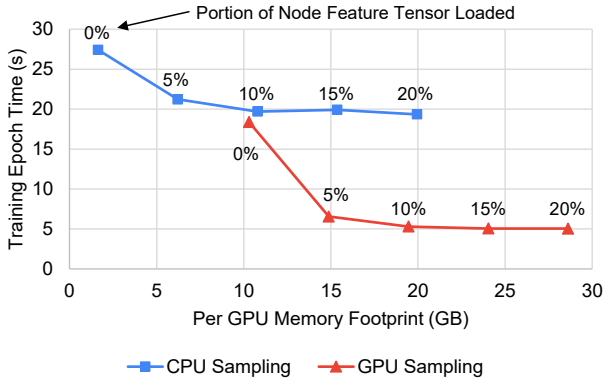

Figure 11: Multi-GPU MAG240M training time comparison while loading different amounts (in %) of node feature tensor into GPU memory. GPU sampling has a higher footprint since it has the adjacency matrix loaded in GPU memory.

already reduce the training time by 33% and with the 25% of data loaded, we can further reduce the training time by 42%. In general, the GPU memory consumed by the training process itself is proportional to the minibatch size, and the minibatch size is exponentially proportional to the sampling depth [26]. Therefore, hypothetically, if the sampling depth is very deep, the GPU memory available for hot data can be limited, but the base space complexity of GNN is relatively low. For example, in case of ogbn-papers100M training with 3-layer sampling, we consume only about 400MB of GPU memory and the rest of the space is left unused. Additionally, considering the trend of increasing capacity of GPU memory (e.g., NVIDIA A100 80GB) and the distributed Multi-GPU tensor solution we discussed in Section 4.2, we believe the actual impact is negligible.

5.5 Multi-GPU Training

In this section, we show the performance benefit of the multi-GPU implementation of our work described in Section 4.2. In this experiment, we use four V100 32GB and therefore we can have toal 128GB of collective GPU memory space. For the training dataset, we use MAG240M which has 350GB of node feature tensor. For data placement, we divide the node feature tensor into two tensors, a multi-GPU tensor and a CPU tensor. We do not allocate any space for the replicated GPU tensor.

In Figure 11, we show the training time evaluation of MAG240M with increasing sizes of hot data loaded in the multi-GPU tensor. Before we go into further detail of the GPU sampling results, we first focus on the CPU sampling results. Similar to the results from Figure 10, we observe a sharp drop of training time with 5% of node feature loaded into GPU memory. Beyond that, we observe only marginal performance improvements. The overall performance improvements in multi-GPU training is underwhelming because the single GPU training of MAG240M in Figure 8 can reach 23.5 seconds already. This means, with four GPUs, we can reduce only about 3 seconds of training time further.

After several profiling, we find that in the multi-GPU training, the neighbor sampling process itself starts to throttle the whole training process and gives us a poor scalability. As we described in Section 2.2, the sampling step traverses the graph structure and generates the node IDs for a minibatch in preparation for the node feature aggregation step. For the neighbor sampling, we have been using CPU since the graph structure is store in the CPU memory. In a single GPU training, CPU was able to sample neighbors fast enough and provide their IDs to GPU in a reasonable amount of time. However, now in a multiple-GPU training, the number of minibatches that we need to generate is multiplied by the number of GPUs and this starts to affect the overall training time. Just to clarify, the implementation of CPU sampling process is already done in a parallel fashion. In short, the amount of parallelism available from CPU is not enough to quickly traverse the graph structure and sample neighbors for multiple GPUs.

This problem can be overcome with the GPU-based sampling method, but only with our multi-GPU data placement strategy. This is because to perform the GPU-based sampling, we now need to consider how to let GPUs to access the graph structure as well. Of course, the simplest way of achieving this is loading the entire graph structure to each GPUs' memories, but the size of the graph structure of MAG240M alone is 30GB and it is too wasteful to load it into every GPU. To resolve this issue, we expand the idea of multi-GPU node feature data placement strategy to the graph structure as well and distribute the graph structure over multiple GPUs. The benefit of the combination of our data placement strategy and the GPU sampling is shown in Figure 11. When we compare the memory footprints of the CPU sampling method and the GPU sampling method, we can find in general the GPU sampling chart has been shifted to right because now the graph structure is consuming some GPU memory space. However, in terms of overall performance, the GPU sampling method removes the CPU sampling bottleneck and notably increases the training speed in the multi-GPU setup.

6 RELATED WORK

GNS [4] samples a global cache of nodes periodically for all mini-batches and stores them in GPUs. It employs a preferential sampling approach in generating mini-batches, which gives priority to neighbors that exist in the GPU cache. This can greatly reduce data copy

between CPU and GPU, but it requires the modification of native node-wise sampling algorithm, which cannot be easily extended to other sampling methods. On the other hand, the design of our data tiering is orthogonal to these sampling methods, which makes it sampling-agnostic. Furthermore, the method in GNS lacks the capability to leverage multi-GPU memory to store the cached data.

LazyGCN [19] periodically samples mega-batches and recycles the sampled nodes within a mega-batch to generate mini-batches. This reduces the overhead of data preparation. Though, LazyGCN is sampling-agnostic, it requires a large mega-batch size, regardless of which sampling method is used, to guarantee the model accuracy. With node-wise neighbor sampling, it can easily run out of GPU memory on large graph such as ogbn-papers100M.

PaGraph [14] and AliGraph [28] provide static node caching scheme in GNN training, but their caching strategies are simply limited to high out degree nodes and their entire cache managements are done by CPU. This approach makes GPUs to always synchronize with CPU to access data.

7 CONCLUSION

In this work, we presented a data tiering technique for GNN training. In general, we find the training time of GNN can be easily improved with well-defined data placement and rearrangement optimizations. Our data tiering strategy is a novel solution that does not affect the algorithm of GNN at all but still maximizes the benefit of multi-tier memory subsystem of modern hardware. We further demonstrate that our approach improves the scalability of multi-GPU training. We demonstrated our work by using existing libraries such as PyTorch and DGL, and our data tiering implementation can be immediately adopted by the end users.

8 ACKNOWLEDGEMENT

This work was partially supported by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR). The authors would also like to thank the anonymous reviewers for their constructive feedback.

REFERENCES

[1] Ziv Bar-Yossef and Li-Tal Mashiach. 2008. Local Approximation of Pagerank and Reverse Pagerank. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management* (Napa Valley, California, USA) *(CIKM '08)*. Association for Computing Machinery, New York, NY, USA, 279–288. https://doi.org/10.1145/1458082.1458122

[2] Michał Daniluk, Jacek Dabrowski, Barbara Rychalska, and Konrad Gołuchowski. 2021. Synersize at KDD Cup 2021: Node Classification in Massive Heterogeneous Graphs.

[3] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. 2018. Convolutional 2d knowledge graph embeddings. In *Thirty-second AAAI conference on artificial intelligence*.

[4] Jialin Dong, Da Zheng, Lin F. Yang, and George Karypis. 2021. Global Neighbor Sampling for Mixed CPU-GPU Training on Giant Graphs. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*. 289–299.

[5] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph Neural Networks for Social Recommendation. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 417–426. https://doi.org/10.1145/3308558.3313488

[6] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[7] Daniele Grattarola and Cesare Alippi. 2020. Graph Neural Networks in TensorFlow and Keras with Spektral. *CoRR* abs/2006.12138 (2020). arXiv:2006.12138 https://arxiv.org/abs/2006.12138

[8] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[9] 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). arXiv:2103.09430 https://arxiv.org/abs/2103.09430

[10] 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. *arXiv preprint arXiv:2005.00687* (2020).

[11] 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*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 187–198. https://proceedings.mlsys.org/paper/2020/file/fe9fc289c3ff0af142b6d3bead98a923-Paper.pdf

[12] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907 (2016). arXiv:1609.02907 http://arxiv.org/abs/1609.02907

[13] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *CoRR* abs/2006.15704 (2020). arXiv:2006.15704 https://arxiv.org/abs/2006.15704

[14] 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* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 401–415. https://doi.org/10.1145/3419111.3421281

[15] Zhiwei Liu, Yingtong Dou, Philip S. Yu, Yutong Deng, and Hao Peng. 2020. Alleviating the Inconsistency Problem of Applying Graph Neural Network to Fraud Detection. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval* (Virtual Event, China) *(SIGIR '20)*. Association for Computing Machinery, New York, NY, USA, 1569–1572. https://doi.org/10.1145/3397271.3401253

[16] 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)*. USENIX Association, Renton, WA, 443–458. https://www.usenix.org/conference/atc19/presentation/ma

[17] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoglu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture. *Proc. VLDB Endow.* 14, 11 (Oct. 2021), 2087–2100.

[18] Mengyang Niu. 2021. ogbn-papers100m-sage. https://github.com/mengyangniu/ogbn-papers100m-sage.

[19] Morteza Ramezani, Weilin Cong, Mehrdad Mahdavi, Anand Sivasubramaniam, and Mahmut T Kandemir. 2020. GCN meets GPU: Decoupling" When to Sample" from" How to Sample".. In *NeurIPS*.

[20] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Li Fei-Fei. 2014. ImageNet Large Scale Visual Recognition Challenge. *CoRR* abs/1409.0575 (2014). arXiv:1409.0575 http://arxiv.org/abs/1409.0575

[21] Yunsheng Shi, Zhengjie Huang, Weibin Li, Weiyue Su, and Shikun Feng. 2021. R-UNIMP: Solution for KDDCup 2021 MAG240M-LSC.

[22] Yunsheng Shi, Zhengjie Huang, Wenjin Wang, Hui Zhong, Shikun Feng, and Yu Sun. 2020. Masked Label Prediction: Unified Massage Passing Model for Semi-Supervised Classification. *CoRR* abs/2009.03509 (2020). arXiv:2009.03509 https://arxiv.org/abs/2009.03509

[23] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. 2011. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503* (2011).

[24] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[25] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[26] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[27] Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. 2018. MoleculeNet: a benchmark for molecular machine learning. *Chemical science* 9, 2 (2018), 513–530.

[28] 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).

A MEMORY ALLOCATION WITH CUDA

In this section, we explain some of the technical challenges of creating shared address space in python programs and describe how to expose certain device's memory space to GPU by using several CUDA APIs.

A.1 Unified Virtual Memory (UVM)

By default, CUDA provides a unified memory addressing scheme for GPU programming with the unified virtual memory (UVM) implementation. This capability is backed by the memory managed unit of GPU which allows GPU to map any system addresses into its virtual address space. Therefore, with this capability, it is possible to weave multiple separate physical addresses into a single contiguous virtual address space. The underlying idea of this virtual addressing in GPU is very similar to the CPU virtual addressing which allows programs to observe physically non-contiguous space like a contiguous space. The actual allocation of this memory space can be done by calling `cudaMallocManaged()`. By default, this API only creates a shared virtual address space of GPUs and the actual physical allocation can be managed by programmers. By using several memory hints with `cudaMemAdvise()`, programmers can control the actual location of data in a GPU page granularity (e.g., 64KB). For example, we can first allocate 1GB space of memory with `cudaMallocManaged()` and assign first 512MB to CPU and the latter 512MB to certain GPU's memory. After the allocation, when we need to access a certain piece of data in the UVM space, GPU automatically generates the necessary type of memory request depending on the page table entry (e.g., local memory access for data in GPU and PCIe memory access for remote access).

However, there is one major drawback of this method with most of the python-based DNN frameworks. Due to the global interpreter lock (GIL) implemented in python, only one thread in a process can actually proceed in python code. To avoid this issue, programmers either need to 1) add a C/C++ module which can run in multithreaded manner or 2) launch multiple processes instead of multiple threads to run python codes in parallel. In case of PyTorch, it takes the second method [13] for multi-GPU training. With the multiprocessing method, the overall workload is as follows. First, a master process initializes multiprocessing environment. Second, the master process spawns multiple child processes. Each child process takes a control of single GPU. When there are 8 GPUs, then we need 8 child processes. Next, each child process trains neural network on its own, and occasionally synchronizes together to update its gradients.

This approach greatly simplifies the distributed training model in python, but multiprocessing also isolates virtual address spaces between different processes. Therefore, in case of CPU memory, different processes cannot simply directly access other processes' memory space, and they must first create a *shared memory* space. Only after that, data placed in the shared memory space can be directly accessed from different processes. The allocation of the shared memory space can be done by the APIs existing in operating systems such as Linux.

Unfortunately, for CUDA, there is no such single universal method which allows all GPUs running on different processes to share a single virtual address space. The scope of memory allocated by `cudaMallocManaged()` is limited to a single process. Therefore, even though the implementation of UVM is convenient, it is not suitable under the python environment. There has been a several years of effort to remove GIL from python for better multithreading implementations, but it is unlikely to be done in any foreseeable future from now. However, hypothetically, if the GIL is removed and if we can freely use multithreading instead of the multiprocessing in python, the implementation of our data tiering can be radically simplified by using UVM.

A.2 User-Managed Virtual Address

Another alternative method for CUDA is to generate mappings for all individual allocations and to manage them manually. For example, in case of Figure 4, we need to keep two pointers: one local GPU pointer and one remote CPU pointer. In case of multi-GPU environment, each GPU needs to keep: its own GPU memory pointer, peer GPUs' memory pointers, and one CPU memory pointer. This can be considered as a software-managed or user-managed virtual addressing.

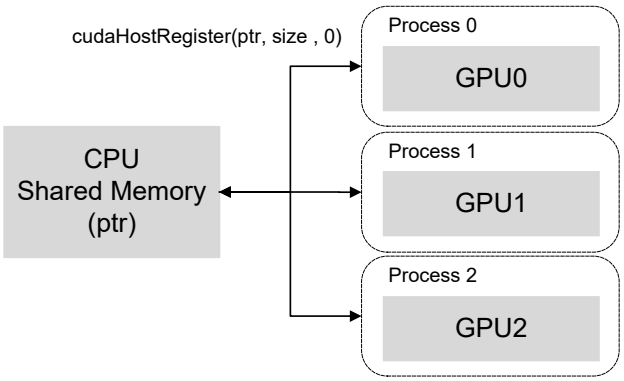

Figure 12: cudaHostRegister() is used to map memory space allocated in CPU into GPU. cudaHostRegister() needs to be called by all processes individually.

To map the CPU memory space into GPU memory space, we use `cudaHostRegister()` API (Figure 12). This API does not newly allocate a space in CPU, but it maps the memory already allocated in CPU into the GPU memory space. To allow multiple GPUs running in multiple different processes to access the same data in CPU, we allocate the shared memory space in CPU and then let all processes to call `cudaHostRegister()` on the shared memory space. A memory space allocated by other memory allocators such as `malloc()` cannot be shared with other processes, and therefore we must use the shared memory space in CPU.

For the GPU memory sharing, we use `cudaIpcGetMemhandle()`, `cudaMalloc()`, and `cudaIpcOpenMemhandl()` APIs. `cudaMalloc()` allocates a memory space in GPU and `cudaIpcGetMemhandle()` creates a memory handle which can be shared with other process to create a virtual mapping of the originally allocated GPU space. In short, this *memory handle* can be understood as a medium to share the virtual mapping between two different processes. In action, `cudaIpcOpenMemhandl()` takes the memory handle created

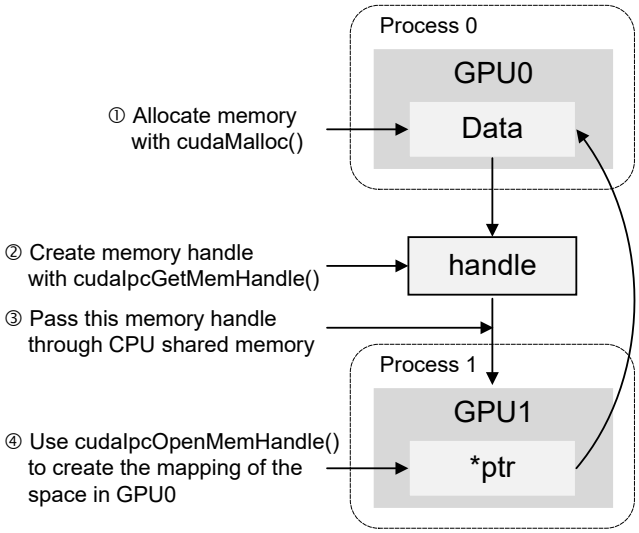

Figure 13: Peer-to-Peer GPU memory sharing mechanism in multiprocessing setup.

by `cudaIpcGetMemhandle()` and maps the other GPU's memory space into the GPU device which belongs to the current process. The overview of this process is shown in Figure 13.

A.3 Combined Tensor

The combined tensor we proposed in Section 4.2, is basically the table which contains the pointers of different memory allocations in different devices (e.g., CPU and peer GPUs). Depending on the index value from the user space, we pick the corresponding pointer and let the GPU kernel to access the node feature. The brief mechanism of this process is explained in Listing 1. The real CUDA kernel implementation of the indexing function is more complicated than the code provided here for several optimization purposes.

```
#define WARP_SIZE 32
void index_feature(int row_id, int **table, int *
    threshold, int feat_len, int gpu_num, int *
    output_feat) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int *row_ptr;
  local_boundary = threshold[0];
  multi_boundary = threshold[1];

  // Data is in local GPU tensor
  if (row_ids[i] < local_boundary) {
    row_ptr = table[0] + feat_len * row_ids[i];
  }
  // Data is in multi-GPU tensor
  else if (row_ids[i] < multi_boundary) {
    int offset = row_ids[i] - local_boundary;
    row_ptr = table[offset % gpu_num + 1] + feat_len *
     offset / gpu_num;
  }
  // Data is in CPU tensor
  else {
    int offset = row_ids[i] - multi_boundary;
    row_ptr = table[gpu_num + 1] + feat_len * offset;
  }

  for (int i = tid; i < feat_len; i+=WARP_SIZE) {
    output_feat[i] = row_ptr[i];
  }
}
```

Listing 1: Indexing the Combined Tensor.

B TRAINING SETUP

The baseline GraphSAGE implementation that we used can be found at `github.com/dmlc/dgl/blob/master/examples/pytorch`. For the single-GPU, `graphsage/training_sample.py`, for the multi-GPU, `graphsage/training_sample_multi_gpu.py`. The specific training parameters that we used are listed in Table 2. The datasets can be found at `https://github.com/snap-stanford/ogb`.

Table 2: Evaluation Parameters.

Parameter	Dataset	Value
`--num-hidden`	All	1024
`--num-layers`	ogbn-papers100M	3
	MAG240M	2
	WikiKG90M	2
`--fan-out`	ogbn-papers100M	12,12,12
	MAG240M	25,10
	WikiKG90M	25,10
`--batch-size`	All	1024