MG-GCN: A Scalable multi-GPU GCN Training Framework

Muhammed Fatih Balın*
Kaan Sancak*
balin@gatech.edu
kaan@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Ümit V. Çatalyürek†
Georgia Institute of Technology
Atlanta, Georgia, USA
umit@gatech.edu

ABSTRACT

Full batch training of Graph Convolutional Network (GCN) models is not feasible on a single GPU for large graphs containing tens of millions of vertices or more. Recent work has shown that, for the graphs used in the machine learning community, communication becomes a bottleneck, and scaling is blocked outside of the single machine regime. Thus, we propose MG-GCN, a multi-GPU GCN training framework taking advantage of the high-speed communication links between the GPUs present in multi-GPU systems. MG-GCN employs multiple High-Performance Computing optimizations, including efficient re-use of memory buffers to reduce the memory footprint of training GNN models, as well as communication and computation overlap. These optimizations enable execution on larger datasets, that generally do not fit into the memory of a single GPU in state-of-the-art implementations. Furthermore, they contribute to achieving superior speedup compared to the state-of-the-art. For example, MG-GCN achieves super-linear speedup with respect to DGL, on the Reddit graph on both DGX-1 (V100) and DGX-A100.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Muhammed Fatih Balın, Kaan Sancak, and Ümit V. Çatalyürek. 2022. MG-GCN: A Scalable multi-GPU GCN Training Framework. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545082

*Both authors contributed equally to this research.
†Also with Amazon Web Services. This publication describes work performed at the Georgia Institute of Technology and is not associated with Amazon.

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08.
https://doi.org/10.1145/3545008.3545082

1 INTRODUCTION

Graphs are essential data-structures that can represent a variety of information, therefore they surface in many different contexts and disciplines. The Graph Convolutional Network (GCN) model is a type of Graph Neural Network (GNN) which is a very powerful graph embedding method for semi-supervised learning to solve graph representation learning problems [24, 33]. GNNs take advantage of the connectivity information presented in the graph, thus they provide flexibility and greater applicability compared to CNN models where the neighborhood structure of nodes is fixed, hence the model is more restricted. The common use cases of GNN models include *node prediction* [24] which predicts the properties of certain vertices, *graph prediction* [43] which predicts the properties of the whole graph, and *link prediction* [42] which predicts whether there is an edge exists between two nodes. In this work, we will focus on node prediction, but our methods are extendable to graph and link prediction as well.

While training GNNs, the memory requirement for large graphs can exceed the memory capacity of a single accelerator. *Mini-batch* training is a common technique to overcome this problem to reduce the working set by creating a mini-batch of vertex samples to train the model. Consequently, it reduces the memory requirement during training. However, mini-batch training might lead to important problems. First, starting from the mini-batch nodes, it is possible to reach almost every single node in the graph in just a few hops, also known as the neighborhood explosion phenomenon, which increases the work performed during a single epoch exponentially. Second, it has been shown that mini-batch training can lead to lower accuracy compared to full-batch training [20]. In this work, we focus on full-batch training on multi-GPU systems.

A major challenge to full-batch GCN training is its parallelization and scalability. The challenge stems mainly from the irregular structure of the graph which leads to load imbalance and communication cost when training on multiple GPUs. GCN has many underlying kernels, however, one of the most time-consuming part is the Sparse Matrix-Dense Matrix Multiplications (SpMM). Alternative solutions are proposed to improve the performance of SpMM, such as reordering and better-suited graph storage schemes and computation kernels [21].

Most of the existing systems, such as Deep Graph Learning Library (DGL), lack the support for multi-GPU training [38]. One needs to implement the parallelism manually while using DGL. DistDGL is an extension of DGL that enables multi-GPU training, however, it does not provide full-batch training, rather it uses mini-batch training [44]. Recently, ROC [20] has been proposed and it supports automatic multi-GPU GCN full-batch training on a single

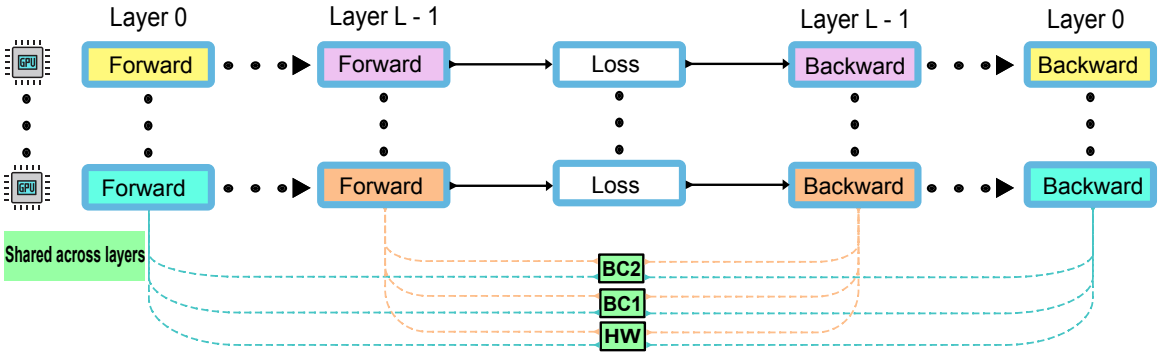


Figure 1: Computation diagram of an L-layer GCN model with shared buffers across layers. $BC1$: broadcast buffer, $BC2$: broadcast buffer for overlapping, HW: temporary result buffer between SpMM and GeMM.

machine, and scales up to multiple machines. CAGNET [35] builds on top of ROC by providing distributed algorithms with different communication patterns. In their work, authors investigate different partitioning strategies to reduce the communication cost and attempt to scale up to hundreds of GPUs. However, their results show that none of the proposed algorithms can achieve speedup beyond a single node (4 GPUs), primarily due to the restricted bandwidth of the available interconnect between nodes in the cluster.

In this work, we provide a framework for training GCNs on multiple GPUs that takes advantage of the high-speed communication links present in today's multi-GPU systems [28]. We address the load imbalance problem by using a simple random permutation strategy and hide the communication by overlapping it with computation. Moreover, we carefully examine the dependency scheme of the buffers used during training and investigate ways to reduce memory requirements for GCN models to fit larger datasets into our target machines. Our optimization techniques are generalizable and can be applied to other frameworks but for reproducibility, we also open-source customizable implementation of MG-GCN.

2 BACKGROUND

The inputs for a GCN $f_A(X)$ are the feature matrix $X \in \mathbb{R}^{n \times d}$ and the adjacency matrix $A \in \mathbb{R}^{n \times n}$, when there are n input instances and each input instance has d dimensional feature vectors. GCNs are useful when the input instances come equipped with a relation, which is represented in matrix format as A. Typically, input instances have features along with them that make up X. To do learning on such a dataset, one can ignore A and fit a model that treats each input instance independently using a multi-layer perceptron. In contrast, GCNs utilize A, and instead of processing each input instance separately, it processes an input instance together with its k-hop neighborhood. Having access to an instance's neighborhood increases the expressiveness of the model, hence aiding performance immensely. As an example, consider guessing which movies an individual would like to watch. It might prove to be a hard task if we have access to a single individual. However, if we consider a group of individuals that are related to the person of interest, then the prediction task becomes much simpler as individual variance vanishes whereas group difference becomes more visible when one looks at whole groups at once. This is why GCN oftens perform much better compared to simple multi-layer perceptron models that do not take into account the relations of instances [24].

The simplest variant of a GCN $f_A(H)$ with a single layer can be represented as

$$f_A(H) = \sigma(\hat{A}^T HW) \tag{1}$$

$$\hat{A}_{uv} = \frac{A_{uv}}{\sum_{w \in \mathcal{N}_i(v)} A_{wv}} \tag{2}$$

where $\mathcal{N}_i(v)$ is the set of in-edges for vertex v and σ is an element-wise nonlinear activation function, ReLU [31] in our case. Using f_A, we can construct deeper GCNs as follows for any number of layers L:

$$H^{(0)} = X \tag{3}$$

$$\vdots$$

$$H^{(L)} = f_A(H^{(L-1)}) \tag{4}$$

As depicted in Figure 1, L-layer GCN model training is composed of L forward passes followed by L backward passes. More specifically, given input matrix H, the operations in the forward pass of a single GCN layer can be broken down as follows:

$$HW = H * W \tag{5}$$

$$AHW = \hat{A}^T * HW \tag{6}$$

$$H' = \sigma(AHW) \tag{7}$$

where $*$ denotes the matrix multiplication operation. Similarly given the gradient from the next layer H'_G, the backward layer can be broken down as follows:

$$AHW_G = \sigma'(H'_G, AHW) \tag{8}$$

$$HW_G = \hat{A} * AHW_G \tag{9}$$

$$W_G = HW_G^T * H \tag{10}$$

$$H_G = HW_G * W^T \tag{11}$$

where we use subscript G in U_G, to denote the derivative of U with respect to the loss function.

As we will experimentally verify in Section 6.1, at the core of these GCN computations, there are two operations that are computationally the most expensive: 1) Sparse Matrix-dense Matrix multiplications (SpMM) in $\hat{A}^T * HW$ and $\hat{A} * AHW_G$, and 2) (General) dense Matrix-Matrix multiplication (GeMM) operations, in

HW, $HW_G * W^T$ and $HW_G^T * H$. For efficient parallel and distributed execution, one needs to pay attention to these two operations.

3 RELATED WORK

The growing size and scale of data encouraged many researchers to develop parallel/distributed algorithms and systems for Deep Neural Networks [2, 27]. Broadly, DNN parallelism can be generalized under 3 categories: data parallelism, model parallelism, and pipelining. Data parallelism can be further divided into 2 categories. Mini-batch parallelism creates batches from the dataset by using sampling methods, and then partitions the batches among computing resources [15, 17], while Coarse- and Fine-Grained or full-batch parallelism divides the dataset among the compute resources [41, 46]. On the other hand, model parallelism divides the model itself, and partitions the work depending on the neurons in each layer [8, 11]. Alternatively, pipelining can be achieved in two ways. Either overlapping the computations between consecutive layers, or partitioning the model according to its depth and dividing layers among processors [1, 6, 10, 32]. Also, there have been hybrid approaches that combine multiple parallelism schemes [26].

While alternative methods exist, most of the research on GNN parallelism is focused on data parallelism, since the models are relatively simple compared to the traditional DNN models. Similar to DNNs, data parallelism can be achieved in two ways. Mini-batch, or sampling, based approaches create batches via neighborhood sampling [5, 7]. After batches are created, they are assigned to CPUs or GPUs. However, in the case of graphs, mini-batching might result in neighborhood explosion in just a few hops, increasing the work performed in an epoch exponentially. Alternatively, to avoid the computation waste, one can apply full-batch parallelism where the parallelism achieved by distributing the workload among CPUs/GPUs while keeping execution order of the layers identical to the sequential method [30, 35]. In full-batch training, the model takes the whole graph and the corresponding features as input, and to achieve any parallelism one has to apply ideas similar to the model parallelism in general DNNs since the work of individual layers has to be partitioned. In this work, we focus on this aspect of GNN model training.

Most of the CPU-based systems are focused on mini-batching based methods. AliGraph is a comprehensive distributed GNN training framework that provides aggregators and operators for various GNN models [45]. AliGraph enables 4 different partitioning algorithms: METIS, Vertex cut & Edge Cut, 2D partitioning, and Streaming-style partitioning. However, it neither provides many details on the subject nor includes any scaling experiments. DistDGL [44] is a framework based on DGL that uses METIS partitioning [22]. It keeps vertex and edge features in a distributed key-value store, which can be queried during the training. DistDGL shows scaling results on the largest available benchmark datasets. However, none of these frameworks provides support for training on the full-graph. DistGNN [30] is a scalable distributed training framework for large-scale GNNs that is an extension of DGL. Unlike other frameworks, DistGNN trains the models on the full graph. It uses a vertex-cut partitioning called Libra [40], and shows substantial scaling on the largest available benchmark datasets. However, as we will show in the later sections, by applying extensive memory optimizations, we are able to fit some of the largest datasets using only 1 to 8 GPUs, while achieving 12.5x faster runtimes than DistGNN's best performance which is achieved with up to 128 sockets.

There has been various frameworks and algorithms proposed for training GNNs on GPUs. Deep Graph Library (DGL) [38] is a well-known library for implementing general GNN models. DGL provides the API for sparse matrix operations and sampling functions to implement various GNN models efficiently. Moreover, it can use Tensorflow [1], PyTorch [32] or MXNet [6] as backends for wide adoption. GNNAdvisor is a runtime system to accelerate GNN workloads in GPU systems, however it works on single GPU platforms and multi-GPU is left as future work [39]. NeuGraph [29] is a single node multi-GPU mini-batch GNN training framework. NeuGraph introduces a programming model for GNN computations that is similar to vertex-centric programming model [16]. ROC [20] is a distributed multi-GPU GNN training framework utilizing graph partitioning via an online regression model and it proposes memory management optimizations for transfers of data between the CPU and GPU. ROC shows scalability on some of the available benchmark datasets such as Reddit and Amazon, and also it can do full-batch training of more complex models and achieve higher accuracy compared to sampling approaches. However, we are not able to compare with ROC, since the available code do not work as expected. CAGNET [35], inspired by the SUMMA algorithm [36], implements 1D, 1.5D, 2D, and 3D partitioning strategies for full-batch training to reduce the communication cost. Additionally, the authors provide a complexity analysis for each strategy. However, CAGNET fails to scale beyond a single node (4 GPUs) in terms of runtime performance due to the available bandwidth and the intra/inter-communication topologies. Moreover, CAGNET does not have an effort to reuse memory buffers, and it relies on PyTorch and PyTorch Geometric libraries [12]. DGCL is a distributed graph communication library for training GNNs on multiple GPUs [3]. P^3 is another framework for distributed GNN training on multi-GPU systems that uses hash partitioning to distribute the graph and the features independently, it also takes advantage of intra-layer parallelism in the first layer and data parallelism in the following layers to pipeline compute and communication [14].

As we will show in the later sections, by adapting extensive memory optimizations, we can fit much larger graphs into our target machines.

4 MG-GCN

Looking at a single layer of a GCN model particularly, we can express it via the following:

$$H^{(l+1)} = f_A(H^{(l)}) = \sigma(\hat{A}^T H^{(l)} W^{(l)}) \tag{12}$$

where matrices A, H, and W are defined in Section 2. That is, one layer of GCN consists of two main operations. For the forward propagation, first, we need to perform a Generalized Matrix Matrix Multiplication (GeMM) between the dense matrices H and W, then we need to perform a Sparse Matrix-Dense Matrix Multiplication (SpMM) between the transpose sparse matrix A, and the resulting matrix of GeMM. Later the result of SpMM is fed into a nonlinear activation function. In the backward pass, the same operations are performed with the non-transposed normalized adjacency matrix

$\hat{A}$. In the rest of this section, we will focus on the forward pass and we refer to $\hat{A}^T$ simply as A.

In addition to our analytical analysis, we have experimentally identified the most computationally expensive kernels in GCN computation. As we will demonstrate in Section 6.1, we have profiled our single GPU GCN training with nvprof to analyze the runtime of our kernels and pinpoint the bottleneck kernels. We have observed that up to 94% of the runtime was spent during the execution of the forward and backward SpMM kernels. Therefore, we have first focused on efficient parallelization of SpMM kernel on multi-GPU setting. Moving into multi-GPU from a single GPU, one needs to find ways to distribute the data into multiple GPUs, and adapt algorithms to perform parallel SpMM.

4.1 Partitioning

Given a matrix A, we can define the 2D tiling (partitioning) of the matrix using two partition vectors p and q, such that p represents the partition vector of the first dimension, and q represents the partition vector of the second dimension. A partition vector p with P parts is defined as:

$$p \in \mathbb{N}^{P+1}, 0 = p(0) \leq \cdots \leq p(i) \leq \cdots \leq p(P) = n \tag{13}$$

Then, let us define A^{ij} as the (i, j)-th tile of the matrix.

$$p(i) \leq u < p(i+1), q(j) \leq v < q(j+1) \tag{14}$$

$$A^{ij}(u, v) = A(u + p(i), v + q(j)), u, v \in \mathbb{N} \tag{15}$$

One way to partition A, H and W is to apply symmetric partitioning, so that $p = q$, to the sparse matrix A, and then assign the tiles of A to GPUs using 1D or 2D distribution. Let's start with 1D column distribution where j-th tile column of A, A^{ij}, is assigned to j-th GPU. Moreover, we will partition dense matrix H, using 1D partitioning by rows, with the same partition vector p, and assign H^i to i-th GPU. Likewise, the resulting dense matrix will be conformally partitioned by its rows. After partitioning, SpMM can be performed in multiple stages. In each stage, one set of rows of the result matrix can be filled, thus taking the algorithm P steps to perform, where P is the number of GPUs. Each GPU performs an SpMM with its local portions in the sparse and dense matrices. That is, at stage i, A^{ij} will be multiplied by H^j by j-th GPU, then partial results will be reduced at GPU i.

$$C^i = \sum_j A^{ij} X^j$$

The only communication needed for this operation is the reduction at the end. In this scheme, W is replicated across GPUs and is reduced at the end of every epoch of training. The reduction of W however is much faster than the communication done for the feature matrix H because of their size difference $O(d^2)$ vs $O(nd)$.

Alternatively, one can do 1D row distribution and assign i-th tile row of A, A^{ij}, to i-th GPU, see Figure 2. Then, at stage i, i-th GPU broadcasts H^i, then A^{ij} will be multiplied by H^j on j-th GPU. The only communication needed for this operation is the broadcast at the beginning, see Figure 3.

$$C^i = C^i + A^{ij} H^j$$

Both of the above approaches partition H by its rows, so one might consider how it would work if H was partitioned by its

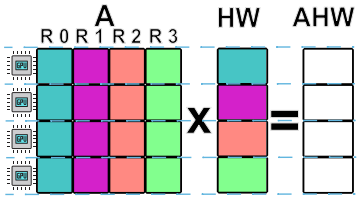
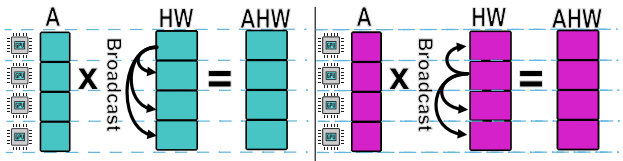


Figure 2: Partitioning of the sparse and dense matrices used in SpMM. Colors represent stages, rows represent GPUs.



(a) Example 1st stage **(b) Example 2nd stage**

Figure 3: Example two stages of SpMM

columns, into $1 \times P$ tiles. For this case, let us use a partition vector p with P parts and a partition vector q with only a single part to partition A. Then, we can assign A^{i1} to i-th GPU, H^{1j} to j-th GPU. Likewise, this operation can be performed in multiple stages. At stage i, i-th GPU broadcasts A^{i1}, then A^{i1} will be multiplied by H^{1j} at j-th GPU. The results are kept at the j-th GPU. The only communication needed for this operation is the broadcast of the sparse matrix at the beginning.

$$C^{ij} = A^{i1} H^{1j}$$

However, for this particular partitioning strategy, there is more communication involved during the GeMM kernel. In particular since H is 1D column-partitioned, $C^{ij} \times W^{jk}$ requires a reduction over j. This means not only A is communicated, but also the dense matrix C is communicated which makes this solution undesirable. Compared to the first solution, solution 2 provides better load balance regardless of the matrix ordering, since each GPU is using the same set of rows broadcasted at each stage, the sparsity pattern of the sparse matrix will be identical across the GPUs. Nevertheless, since communication is the main bottleneck, we decide to use the broadcast variant of solution 1, as in Figure 2. Note that in our system only the model weights are replicated, any other data such as the adjacency matrix A, and the input & intermediate feature matrices H are fully partitioned.

We don't discuss more complicated partitioning strategies such as 1.5D, 2D or 3D as we will explain the reasoning in Section 5.1. Furthermore, note that the GeMM computations on the row-partitioned feature matrices do not require any synchronization as each GPU can compute $H^i W$ in (5) independently. The element-wise activation function is also fully independent, each GPU computes it for their portions. More detailed discussion on different partitioning strategies can be found in [4].

4.2 Memory Optimizations

To reduce memory requirements, we reuse memory buffers in the forward and backward passes, as much as possible.

In the forward and backward computations in eqs. (5) to (7), we will have a temporary result buffer called HW_B and a result buffer

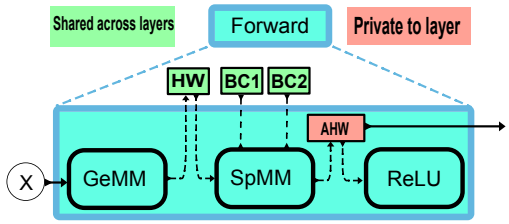
called AHW_B with the following mapping:

$$HW \rightarrow HW_B \tag{16}$$
$$AHW \rightarrow AHW_B \tag{17}$$
$$H' \rightarrow AHW_B \tag{18}$$

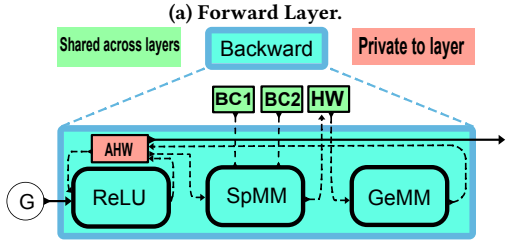
And in the backward computations in eqs. (8) to (11):

$$AHW_G \rightarrow AHW_B \tag{19}$$
$$HW_G \rightarrow HW_B \tag{20}$$
$$H_G \rightarrow AHW_B \tag{21}$$

Figure 4a shows the mappings of the buffers for the forward computation and Figure 4b for the backward propagation.



(a) Forward Layer.



(b) Backward Layer.

Figure 4: Forward and backward layers. Buffers colors indicate whether they layer is shared or private. $BC1$**,** $BC2$**, and** HW **are explained in Figure 1.** AHW**: Buffer for the result.**

Notice that, each layer only requires *a single buffer* to store their output. They also use a temporary buffer that is shared across layers. Hence, each layer only increases the memory used by a single buffer, compared to *4x or 6x* in other deep learning frameworks such as DGL and CAGNET that allocate buffers for the output of SpMM, GeMM and the activation functions. Considering the backward pass adds up to 6 buffers per layer in total, as shown in Figure 1. For L-layer GCN, the total number of buffers is $L + 3$, whose sizes on average are $n \times d$.

4.3 Overlapping Computation and Communication

Each round of our multi-round SpMM is composed of a broadcast of a tile of H and an SpMM computation with a tile of A with the received tile of H. Notice that, there is an opportunity to overlap communication and computation in such a multi-round scheme. After the broadcast of the first tile of H, we overlap communication of the next (remaining) tile(s) of H with the SpMM computations. In order to do that, we need an extra communication buffer for the *next* H tile. Since each GPU *keeps* its own H tile, and receives the H^i in the i-th round, each GPU needs one more extra buffer for the broadcast primitive. In total, overlapping communication and computation would require two additional buffers. In order to fully utilize communication computation overlap, we use two GPU streams: one for communication (stream 1) and one computation (stream 0). We launch all communication and computation kernels asynchronously on those two streams and wait for i-th broadcast to finish on stream 1 before we start on i-th SpMM computation on stream 0 and the $i + 1$-th broadcast waits for the $i - 1$-th SpMM to finish not to overwrite its input when it is ongoing.

4.4 Order of Computation and Saving one SpMM

For computing AHW, we change the order of SpMM and GeMM operations depending on the feature dimension of the current layer $d^{(l)}$ and the next layer $d^{(l+1)}$ as allowed by associativity. If $d^{(l)} < d^{(l+1)}$, then doing SpMM, otherwise running GeMM is faster.

If the gradients all the way back to the input features are not required, then it is possible to skip the SpMM in the first layer during the backward pass. The reason is that SpMM scales each feature dimension independently so it is possible to replace it with a diagonal feature scaling matrix in the first layer's backward pass. In our case, each node takes the average of their neighbors, thus the identity matrix is the scaling matrix, making it a no-operation. Thus we avoid the SpMM of the first layer in the backward pass.

5 DESIGN DECISIONS

5.1 Choice of the Partitioning Strategy

The communication topology of the system directly affects the observed bandwidth of different communication patterns. This is clearly not an issue for systems like DGX-A100 where 8 GPU of the system are connected shared NVlink switch with 12 links, and could achieve full communication bandwidth between any pair of GPUs. Whereas, in DGX-1 there are only 6 links, and connections between GPUs are asymmetric. Such asymmetry will make some theoretically optimum algorithms perform poorly on that system since the underlying communication assumptions are not valid there. For example, the 1.5D algorithm presented in [35] halves the theoretical communication volume, by using more memory with replication factor $c = 2$. If we group the GPUs into two groups as per the replication factor, each group has 4 links available. Then the broadcast can be faster by a factor of $\frac{6}{4\times2}$ in the 1.5D case. However, the last reduction among the two groups has access to only 2 links. Then, if we sum up the time required for communication for the 1.5D case, which necessitates two rounds of broadcast followed by a concurrent reduction (see [35] for the details of the algorithm) we get: $2\frac{nd}{4\times4l} + \frac{nd}{4\times2l} = \frac{nd}{4l}$, where l is the single NVlink bandwidth. In comparison, the 1D algorithm only takes $8\frac{nd}{8\times6l} = \frac{nd}{6l}$ time. On the other hand, in DGX-A100 all broadcasts and reductions can utilize all of the 12 links. Hence, summing up time required for the 1.5D algorithm, we get: $2\frac{nd}{4\times12l} + \frac{nd}{4\times12l} = \frac{nd}{16l}$. In comparison, the 1D algorithm takes $8\frac{nd}{8\times12l} = \frac{nd}{12l}$ time.

According to the above analysis, the 1.5D algorithm is slower on DGX-1 by a factor of $\frac{2}{3}$ but it is faster on DGX A100 by $\frac{4}{3}$, but also requires twice as much memory. Since GNN training is usually bound by the GPU memory, we only implement the 1D version.

5.2 Permutation

In order to balance the number of nonzeros in each part A^{ij} in the uniformly partitioned sparse matrices, we randomly permute their vertices. This has a significant effect on load balance compared to using the original orderings of the sparse matrices which can have highly imbalanced parts. Later in Section 6.2, we show how this permutation improves the execution time with better load balancing, especially with a larger number of GPUs.

6 EXPERIMENTAL EVALUATION

Hardware and Software: We perform our experiments on two machines: NVIDIA DGX-1, also referred to as DGX-V100, and NVIDIA DGX-A100. DGX-V100 has 8 Nvidia V100 GPUs, each equipped with 32GB memory with a 900 GB/s memory bandwidth. Each V100 has 6 NVLink connections, each consisting of 2 sub-links that send data in one direction, and has a 25GB/s bandwidth. That is, each link is capable of 50GB/s bidirectional bandwidth, and theoretically, the aggregate system bandwidth is 300 GB/s. The DGX-1 is equipped with a dual 20-core Intel Xeon E5-2698 CPU with 512 GB RAM. NVIDIA DGX-A100 has 8 NVIDIA A100 GPUs, each equipped with 80GB memory with a 2 TB/s memory bandwidth. Each A100 has 12 NVLink connections, thus twice as much the bandwidth of V100. Unlike the V100, each A100 is connected to an NWSwitch, enabling a full peer-to-peer bidirectional bandwidth of 600 GB/s between any two GPUs. DGX-A100 is equipped with a dual 64-core AMD Rome 7742 CPU with 2 TB RAM. Both machines run Ubuntu 20.04.

We implemented MG-GCN using C++ standard 20 and compiled it with GCC 9.3.0 and CUDA 11.4. We used CUDA's cuSPARSE for SpMM calls with the Compressed Sparse Row format for the sparse matrices, and cuBLAS for GeMM with the Row Major format for the dense matrices. PIGO [13] is used for IO purposes. We use DGL 0.7.1 which is currently the latest available version [38]. We follow the instructions for compiling CAGNET [35] on its repository. For MG-GCN, we use NCCL (Nvidia Collective Communication Library) 2.11.4 and for CAGNET, we use NCCL 2.4.8 for compatibility reasons. The code for MG-GCN is available at https://github.com/GT-TDAlab/MG-GCN. We verified the correctness of our implementation by comparing the train accuracy curve with DGL's.

Datasets: We use two types of datasets in our experiment. The first category is GNN Benchmark datasets which are popular datasets used in GNN research, see Table 1. The Reddit dataset is a graph from Reddit posts that are posted in September, 2014 [18]. The node labels represent the communities (subreddits). Products (OGBN-Products) is a graph from Amazon co-purchase network. Nodes represents the products, and link represent products that are bought together. Proteins (OGBN-Proteins) is a biological network graph dataset where nodes represent proteins and edges represent associations between proteins. Arxiv (OGBN-Arxiv) and Cora are citation networks where each node represents a paper and directed edges represent citation direction [19, 34].

We also used synthetic datasets generated with BTER [25] to evaluate the scalability of our method under varying density. BTER requires a degree distribution and clustering coefficient by degree as input and generates synthetic graphs matching those properties. We first profile the degree distribution of the Arxiv dataset, then by increasing the average degree and fixing the number of vertices, we generate 8 synthetic datasets. We name these datasets as $1x, \ldots, 128x$. As the name suggests, the number represents the scaling factor of the number of edges from the original graph. We generate the features and assign class labels randomly. Each synthetic dataset has a feature vector of size 512, and there are 40 classes. Since the graphs generated by BTER are not deterministic, we generate 10 of each scale and take the median while reporting the results.

Table 1: Benchmark Datasets. n: #vertices, m: #edges, $d^{(0)}$: #features, $d^{(L)}$: #classes, k: average degree.

Dataset	n	m	$d^{(0)}$	$d^{(L)}$	k
Cora	3.3K	9.2K	3.7K	6	3
Arxiv	169K	1.16M	128	40	7
Papers	111M	1.61B	128	172	15
Products	2.5M	126M	104	47	52
Proteins	8.74M	1.3B	128	256	150
Reddit	233K	115M	602	41	492

Model: While we are able to train more complex models, to make fair comparisons, we use 4 different GCN models. First, to compare with CAGNET and DGL, we use a model with 2 layers, and the hidden layer consists of 512 neurons. Our limitation comes from the fact that the available code for CAGNET does not have the option to change the number of layers. Second, to compare with DistGNN on Reddit, we use a model with 2 layers and a hidden layer consists 16 neurons. To compare with DistGNN on Products, Protein and Papers, we use a model with 3 layers and hidden layers consisting of 256 neurons. Finally, we use a 4th model with 3 layer, each consisting of 208 neurons to run MG-GCN on Papers DGX-A100, since 208 is the largest hidden layer size that can fit into DGX-A100. We have implemented and used the Adam optimizer [23] and the softmax cross-entropy loss [9] in all of our experiments.

The model we used in the comparison against DistGNN on Reddit, was able to achieve a test accuracy of %95.95 in the transductive setting after 466 epochs with eight V100s in only 1 minute, 20 seconds of which is spent on preprocessing, which matches the accuracy the DGL baseline code gets using the same model configuration.

6.1 Runtime Breakdown of GCN Computation

We analyze the breakdown of the execution time of GCN computation to find the computational bottlenecks during training. Figure 5 presents the runtime breakdown of the first GCN model described in Section 6. The activation layer refers to the computation in eq. (7), Adam refers to the update of the model parameters W by the Adam optimizer and the loss layer refers to the computations related to the softmax cross-entropy loss. As it is evident from the figure, for sufficiently large datasets, i.e., Proteins, Products, and Reddit, the main bottleneck is SpMM kernel which takes 60%-94% of the runtime, and the second bottleneck is GeMM kernel 5%-20% of the runtime. On the other hand, for small datasets the main bottleneck becomes GeMM. Therefore, we stress the importance of parallelizing SpMM and GeMM kernels to achieve scalability during any GCN training and focus our attention on parallelizing these kernels.

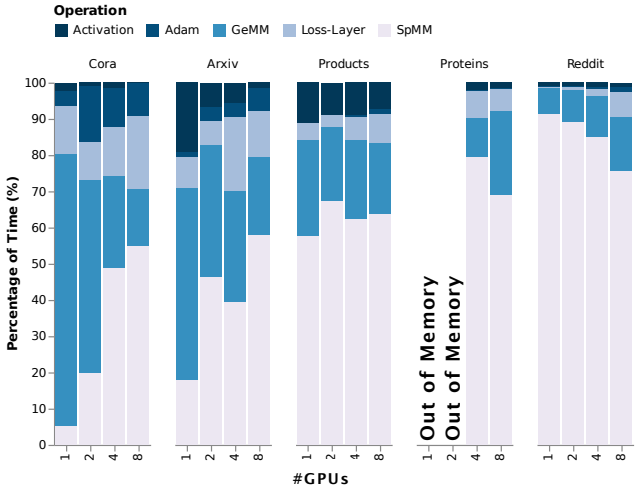


Figure 5: Runtime decomposition of operations involved in forward and backward pass.

6.2 Impact of Permutation

Figure 6 presents the breakdown of execution of SpMM to communication and computation times for each stage for the Product dataset using original and permuted ordering. On the top part of the figure, there is a significant computational imbalance that hampers the efficient parallel execution. To remedy the load imbalance problem we randomly permute the adjacency matrix before the computation. On the bottom part of the figure, permuted ordering achieves better computation load balance and reduces the execution time from 50ms to 38ms. Figure 7 shows normalized runtime improvement of permuted ordering w.r.t. original ordering for each dataset for varying number of GPUs. As seen in the figure, permutation yields slightly slower execution time on a small number of GPUs for some datasets; however, as the number of GPUs increases, the runtime improves significantly with the load balance achieved by permutation. For example, we observed 1.5× runtime improvement on Products and Reddit datasets with 8 GPUs.

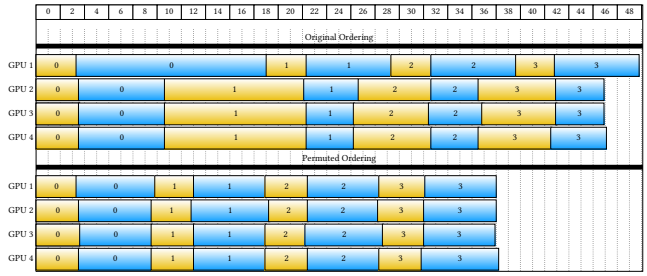


Figure 6: Timeline of the SpMM on the Products dataset using its original and permuted ordering. The numbers on the bars represent stages. For each GPU, computation (blue) and communication (yellow) phases are separately plotted.

6.3 Overlapping Computation and Communication

Figure 8 shows the effect of the communication-computation overlap on Products datasets using 4 GPUs. Notice that overlapping these two operations makes both the computation and the communication slower. This is because of the use of shared resources, in particular the memory bandwidth. Since SpMM is a mostly memory bandwidth-bound operation, it becomes slower when overlapped with the communication kernel that takes up some of the global memory bandwidth. The global memory bandwidth of a V100 GPU is 900 GB/s and the communication bandwidth is 150 GB/s. Assuming the communication happens at full bandwidth, this results in a reduction of the global memory bandwidth for the SpMM operation by a factor of $\frac{1}{6}$. Nevertheless, communication-computation overlap still improves the performance. As seen in the figure, for Products, SpMM time can be reduced to 30ms from 38ms with overlapping communication and computation.

Figure 7 shows normalized runtime improvement of overlapping w.r.t. nonoverlapping for each dataset for varying numbers of GPUs. As seen in the figure, enabling overlapping yields slightly less improvement in runtime on a small number of GPUs for some datasets; however, as the number of GPUs increases, the runtime improves significantly with time saved by hiding communication. For example, we observed an additional 1.15× runtime improvement on Products and Reddit datasets with 8 GPUs via enablement of overlapping. One should also note that the size of the hidden dimension doesn't have an effect on our ability to overlap communication and computation as both of their runtimes scale linearly with the size of the hidden dimension if it is above a certain threshold.

6.4 Impact of Average Degree

The runtime of SpMM can be mainly divided into two parts: computation time and communication time. Since we mostly overlap the two, the runtime can be at best the maximum of those two. Communication time only depends on the dimensions of the matrix, whereas the computation time also depends on the density and sparsity structure of the matrix. Furthermore, computation time starts to dominate the execution time as the average degree increases. To illustrate the effect of this on speedup, we used the synthetic datasets generated by scaling the Arxiv dataset as explained in Section 6. Figure 9 displays the speedups obtained by 2 to 8 GPUs, while we increase the average degree. As seen in the figure, our code starts to achieve super-linear speedup with 2 and 4 GPUs, after 32×, and with 8 GPUs, after 64× scaling. We attribute these super-linear speedup numbers for very dense adjacency matrices because of the blocking effect of partitioning and potentially better use of the cache.

6.5 Comparison on Single Node Systems

Comparison on DGX-V100: In Figures 10 and 11, we compare MG-GCN with DGL and CAGNET using the 2 layer model mentioned in Section 6 on DGX-V100. Note that, CAGNET has different partitioning strategies namely, 1D, 1.5D, 2D, and 3D. We present the best results for CAGNET which are produced by 1D partitioning. In all datasets, we outperform DGL with a single GPU and CAGNET with multiple GPUs. Our single GPU performances are, 2.72× faster on Reddit, 1.42× faster on Products, 1.76× faster on Arxiv and 3.1x faster on Cora than DGL. Our 8 GPU performances are 2.66x faster on Reddit, 8.6× faster on Products, 2.35× faster on Arxiv than CAGNET. Notice that, neither MG-GCN nor CAGNET can

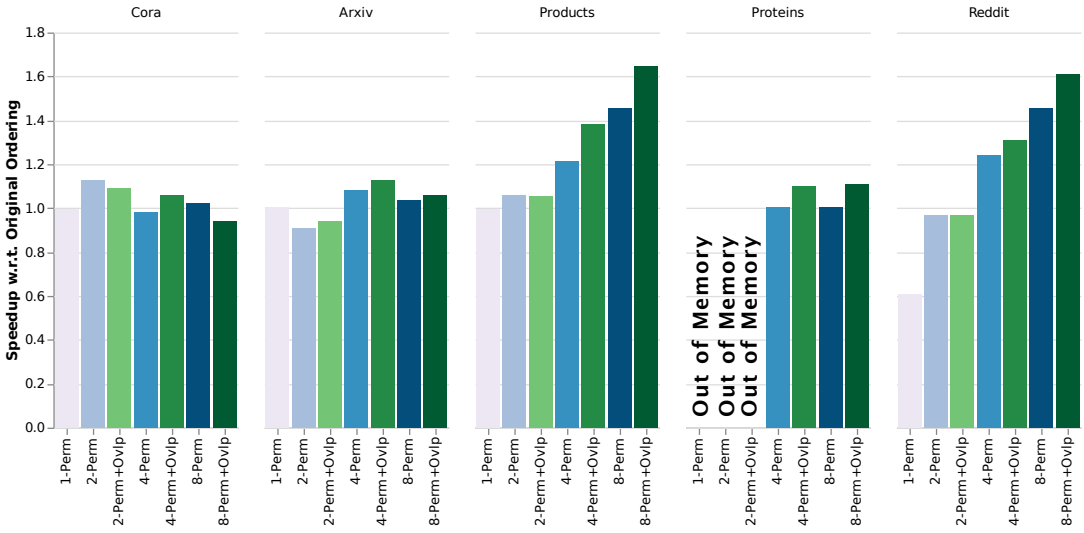


Figure 7: Effect of overlapping of communication with computation and permuting the graph to epoch runtime on DGX-V100. Blue bars show the effect of permutation over original ordering, green bars enable communication-computation overlap in addition to permutation.

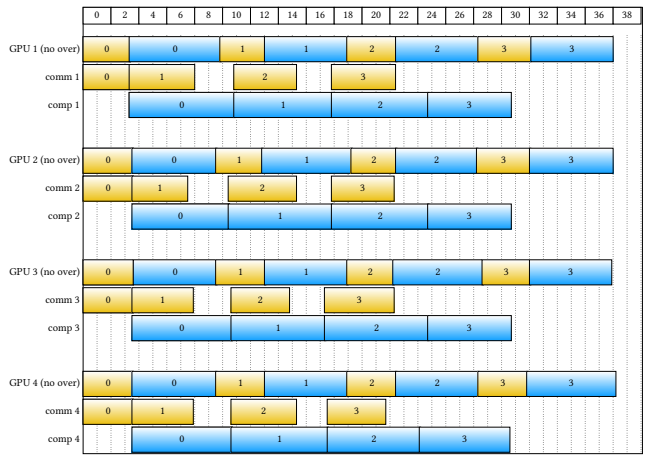


Figure 8: Timeline of the SpMM on the Products dataset using permuted ordering. The numbers on the bars represent stages. Each GPU is represented by 3 lines. First line represents computation without overlapping communication. Next two lines represent computation with overlapping communication. Blue line: computation time. Yellow line: communication time.

get a speedup on Cora dataset, since the graph is very small, and a certain amount of work is expected to achieve any speedup. We are not able to run CAGNET with Proteins dataset using 8 GPUs because of CAGNET's memory requirement; however, MG-GCN is able to fit Proteins dataset into 4 only GPUs. Even though, both CAGNET and MG-GCN use the 1D partitioning strategy, we can fit much larger graphs into our target machines due to extensive memory optimization described in Section 4.2. Also, by overlapping computation and communication, we can achieve substantial speedup compared to CAGNET. Note that, the achieved speedups correlate with the average degree of the graph, as mentioned in Sec. 6.4.

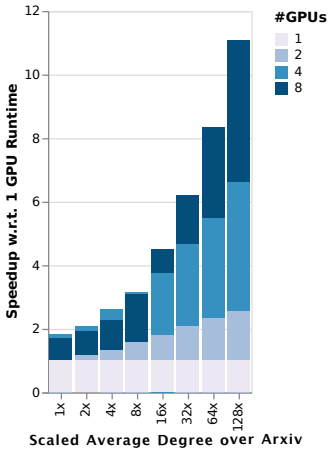


Figure 9: Speedup w.r.t. MG-GCN 1 GPU Runtime

In Figure 12, we compare the memory footprint of MG-GCN with DGL and CAGNET in the single and multi-GPU settings. As seen in the figure, given a GPU memory constraint of 30 GiB, one can fit 20 vs 50 layers using DGL vs MG-GCN in the single GPU setting. In the multi-GPU setting, one can again fit 150 vs 450 layers using CAGNET vs MG-GCN in the 8 GPU setting. One can also observe that the dependency of memory consumption on numbers of layers is linear as expected.

Comparison on DGX-A100: In Figures 13 and 14, we compare MG-GCN with DGL using the 2 layer GCN model mentioned in Section 6 on DGX-A100. We are not able to include CAGNET in this comparison since it is not compatible with CUDA 11. In all the datasets, we outperform DGL with a single GPU. Our single GPU results are 2.2× faster on Cora, 1.8× faster on Arxiv, 1.5× faster on Products and 1.5× faster on Reddit datasets than DGL. On the multi-GPU setting, we achieve 8.5× speedup on Products dataset, and 8.3× speed-up on Reddit dataset using 8 GPUs. MG-GCN can fit Papers dataset, which is the largest available benchmark dataset for GNN

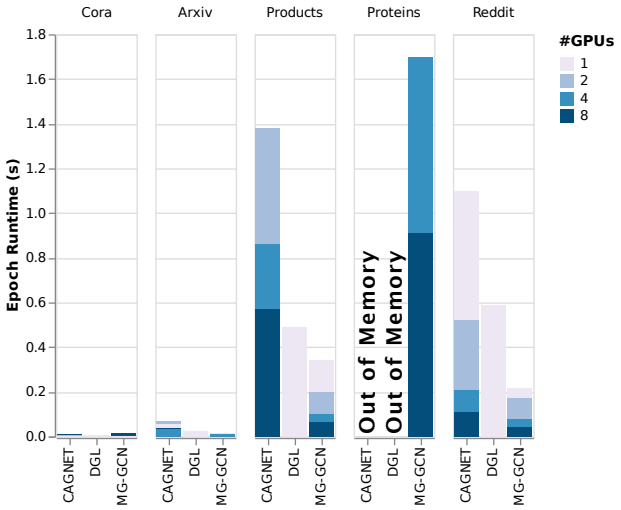


Figure 10: Baseline epoch runtime (seconds) comparison on DGX-V100. On Proteins dataset CAGNET and DGL run out of memory, MG-GCN runs out of memory with 1 and 2 GPUs.

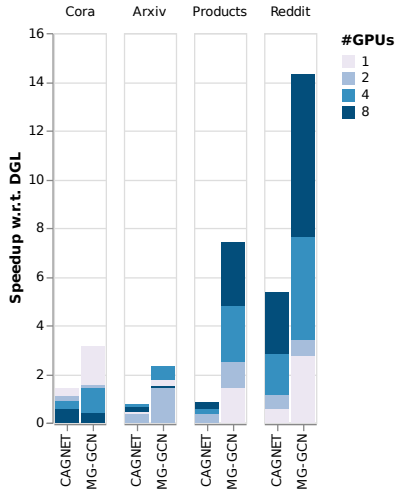


Figure 11: Speedup w.r.t. DGL on DGX-V100.

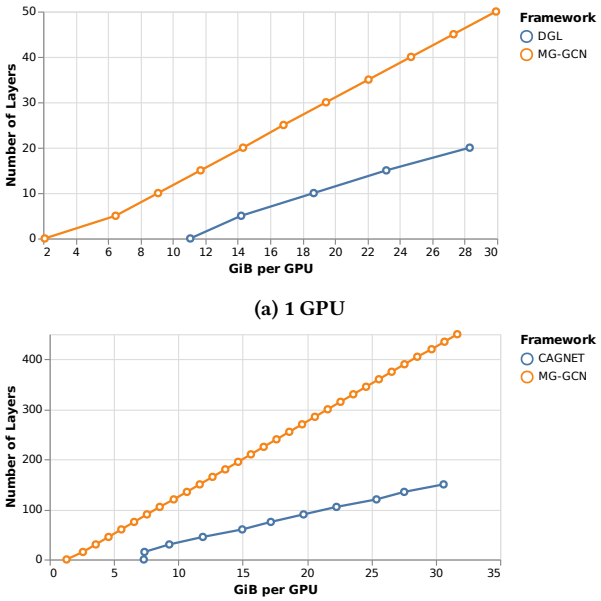


Figure 12: Per GPU memory consumption on the Reddit dataset with hidden layer size 512 varying number of layers.

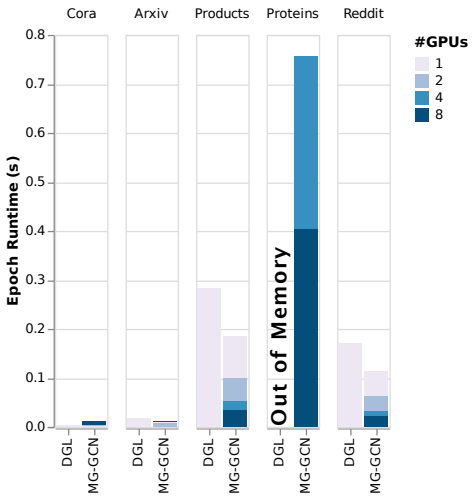


Figure 13: Epoch runtime (s) comparison on DGX-A100.

training, into 8 GPUs with MG-GCN, and achieve 2.89 seconds epoch runtime using the 4th GCN model mentioned in Section 6.

6.6 Single Node vs Distributed Systems

We compare MG-GCN with DistGNN using 2 different GCN models mentioned in section 6. Note that, this is not an exact comparison for two main reasons: First, we are not able to reproduce the results because the source code of DistGNN is not available, so we base our comparison on the numbers reported in the original work. Second, DistGNN is a CPU-based framework, while MG-GCN is designed for GPUs. We believe that comparing the two frameworks will provide important insights on the resource requirements and performance one can get. For the experiments, DistGNN uses a cluster with 64 Intel Xeon 9242 CPU @2.30 GHz with 48 cores per socket in a dual-socket system. The compute nodes consist of 384 GB memory and are connected through Mellanox HDR interconnect with DragonFly topology. In addition, to run Papers on a single socket, they use a single-socket machine with 1.5TB memory.

Table 2 shows the results from DistGNN, while Table 3 shows the performance of MG-GCN on DGX-A100. In Table 2, we only take the single socket and the best socket performances for each dataset from the original work [30]. Also, note that we compare against their baseline version since other variants are not exact computations but approximations. For detailed results, we refer interested readers to [30]. Even though, the authors observe significant speedups in their experiments, MG-GCN outperforms their best performance with a single GPU on all datasets except Proteins. Our 8 GPU performances are 40× faster on Reddit, 12.6× faster on Papers, 12.4× faster on Products, and 1.77× faster on Protein datasets than DistGNN's best performances. Note that, for Reddit

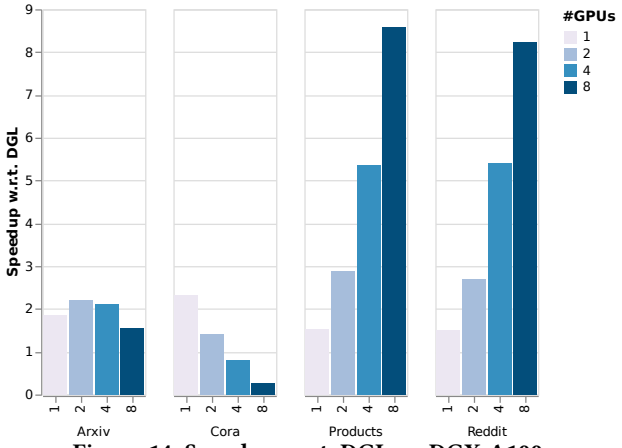


Figure 14: Speedup w.r.t. DGL on DGX-A100.

dataset, since the GCN model is very small, 2 layers with 16 neurons, MG-GCN cannot achieve speedup after 4 GPUs.

Given a single Intel Xeon 9242 CPU has a Thermal Design Power (TDP) of 350W, whereas a single Tesla A100 has 400W. Given that energy consumption is computed by TDP × #of Devices × Time. If we exclude power consumption of the network and the rest of the system's power consumption, a back-of-the-envelope analysis shows that on the Papers dataset, the power consumption ratio equals $\frac{350W \times 128 \times 36.45s}{400W \times 8 \times 2.89s} \frac{208}{256} = 143.46$. We scale the energy consumption by $\frac{208}{256}$ due to the differences of the hidden layer dimensions. We see that using GPUs over CPUs is 2 orders of magnitude more energy-efficient in this case. Note that, we use the exact same GCN models for every comparison except the last comparison on Papers dataset where we reduce the hidden dimension to 208 due to memory constraints. However, MG-GCN's runtime is scaled accordingly in the power efficiency comparison to keep things fair.

Table 2: DistGNN Results: The numbers in the cells are epoch times in second. For each dataset, we take results for 1 Socket and the number of sockets that performs the best from [44]. DS: Dataset, #S: Number of Sockets.

#S \ DS	Reddit	Papers	Products	Protein
1	0.60	1000	11	100
16	0.61	-	-	-
64	-	-	1.74	2.63
128	-	36.45	-	-

7 CONCLUSION

In this paper, we present MG-GCN, a single node multi-GPU GCN training framework that enables efficient distributed training of GCNs over the full graph. MG-GCN adapts a 1D row partitioning strategy. It also adapts extensive memory optimizations by reusing/sharing the allocated buffers across layers and forward/backward phases and enables overlapping communication and computation. We have demonstrated that MG-GCN can achieve significant runtime improvements over the available state-of-the-art frameworks

Table 3: MG-GCN Results on DGX-A100: The values in the cells are epoch times in seconds. Dashed line represents configurations that run out of memory. DS: Dataset, #G: Number of GPUs.

#G \ DS	Reddit	Papers	Products	Protein
1	0.033	-	0.355	4.221
2	0.017	-	0.202	2.272
4	0.012	-	0.110	1.191
8	0.012	2.89	0.067	0.641

on single GPU systems. Moreover, going into the multi-GPU setting, we can fit much larger graphs into the memory of our target machines. In our single GPU experiments, we achieve up to 2.72× speedup compared to DGL on the Reddit dataset, and on multi-GPU experiments, we achieve up to 8.6× speedup on the Products dataset compared to CAGNET on DGX-V100.

In future work, we are aiming to extend our framework to multi-GPU clusters. By doing so, we aim to train larger datasets and enable distributed training of even larger scale GNNs. Another future direction is to accelerate the Sampled Dense Dense Matrix Multiplication (SDDMM) kernel to enable parallel training of several other models such as Graph Attention Networks [37].

ACKNOWLEDGMENTS

We thank Prof. Polo Chau for providing us access to their DGX-A100 for our experiments. This work was partially supported by the NSF grant CCF-1919021.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, et al. 2015. *TensorFlow, Large-scale machine learning on heterogeneous systems.*
[2] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–43.
[3] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: An Efficient Communication Library for Distributed GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21).* Association for Computing Machinery, 130–144.
[4] Ümit V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. 2010. On Two-Dimensional Sparse Matrix Partitioning: Models, Methods, and a Recipe. *SIAM Journal on Scientific Computing (SISC)* 32, 2 (2010), 656–683.
[5] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations.*
[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, et al. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:1512.01274 [cs.DC]
[7] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *International Conference on Knowledge Discovery & Data Mining.* 257–266.
[8] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. 2013. Deep learning with COTS HPC systems. In *International conference on machine learning.* PMLR, 1337–1345.
[9] David R. Cox. 1958. The Regression Analysis of Binary Sequences. *Journal of the Royal Statistical Society. Series B (Methodological)* 20, 2 (1958), 215–242.
[10] Li Deng, Dong Yu, and John Platt. 2012. Scalable stacking and learning for building deep architectures. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* 2133–2136.
[11] Ludvig Ericson and Rendani Mbuvha. 2017. On the performance of network parallel training in artificial neural networks. *arXiv* (2017).
[12] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop.*
[13] Kasimir Gabert and Ümit V. Çatalyürek. 2021. PIGO: A Parallel Graph Input/Output Library. In *IEEE IPDP Workshops.* 276–279.

[14] Swapnil Gandhi and Anand P. Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 551–568.

[15] Boris Ginsburg, Igor Gitman, and Yang You. 2018. Large Batch Training of Convolutional Networks with Layer-wise Adaptive Rate Scaling.

[16] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI'12*. 17–30.

[17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, et al. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv* (2017).

[18] William L. Hamilton, Rex Ying, and Jure Leskovec. 2018. Inductive Representation Learning on Large Graphs. arXiv:1706.02216 [cs.SI]

[19] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, et al. 2021. Open Graph Benchmark: Datasets for Machine Learning on Graphs. arXiv:2005.00687 [cs.LG]

[20] 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 (MLSys)* (2020), 187–198.

[21] Peng Jiang, Changwan Hong, and Gagan Agrawal. 2020. A Novel Data Transformation and Execution Strategy for Accelerating Sparse Matrix Multiplication on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, 376–388.

[22] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20 (1998), 359–392.

[23] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[24] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations*. OpenReview.net.

[25] Tamara G. Kolda, Ali Pinar, Todd Plantenga, and C. Seshadhri. 2014. A Scalable Generative Graph Model with Community Structure. *SIAM Journal on Scientific Computing* 36, 5 (Jan 2014), C424–C452.

[26] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv* (2014).

[27] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[28] Ang Li, Shuaiwen L. Song, Jieyang Chen, Jiajia Li, Xu Liu, et al. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 94–110.

[29] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, et al. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 443–458.

[30] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, et al. 2021. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. *arXiv* (2021).

[31] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *International Conference on International Conference on Machine Learning*. 807–814.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 8024–8035.

[33] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80.

[34] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Magazine* 29, 3 (2008), 93–106.

[35] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing Communication in Graph Neural Network Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Article 70, 17 pages.

[36] Robert A. van de Geijn and Jerrell Watts. 1997. SUMMA: scalable universal matrix multiplication algorithm. *Concurr. Pract. Exp.* 9 (1997), 255–274.

[37] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. arXiv:1710.10903 [stat.ML]

[38] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, et al. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv* (2019).

[39] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, et al. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 515–531.

[40] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed Power-law Graph Computing: Theoretical and Empirical Analysis. In *Advances in Neural Information Processing Systems*, Vol. 27. Curran Associates, Inc.

[41] Kunlei Zhang and Xue-Wen Chen. 2014. Large-Scale Deep Belief Nets With MapReduce. *IEEE Access* 2 (2014), 395–403.

[42] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *NIPS* 31 (2018), 5165–5175.

[43] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An end-to-end deep learning architecture for graph classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[44] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, et al. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 36–44.

[45] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, et al. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2094–2105.

[46] Martin Zinkevich, Markus Weimer, Alexander J Smola, and Lihong Li. 2010. Parallelized stochastic gradient descent.. In *NIPS*, Vol. 4.