

Plexus: Taming Billion-edge Graphs with 3D Parallel Full-graph GNN Training

Aditya K. Ranjan

Department of Computer Science
University of Maryland
College Park, Maryland, USA
aranjan2@umd.edu

Cunyang Wei

Department of Computer Science
University of Maryland
College Park, Maryland, USA
cunyang@umd.edu

Siddharth Singh

Department of Computer Science
University of Maryland
College Park, Maryland, USA
ssingh37@umd.edu

Abhinav Bhatele

Department of Computer Science
University of Maryland
College Park, Maryland, USA
bhatele@cs.umd.edu

Abstract

Graph neural networks (GNNs) leverage the connectivity and structure of real-world graphs to learn intricate properties and relationships between nodes. Many real-world graphs exceed the memory capacity of a GPU due to their sheer size, and training GNNs on such graphs requires techniques such as mini-batch sampling to scale. The alternative approach of distributed full-graph training suffers from high communication overheads and load imbalance due to the irregular structure of graphs. We propose a three-dimensional (3D) parallel approach for full-graph training that tackles these issues and scales to billion-edge graphs. In addition, we introduce optimizations such as a double permutation scheme for load balancing, and a performance model to predict the optimal 3D configuration of our parallel implementation – Plexus. We evaluate Plexus on six different graph datasets and show scaling results on up to 2048 GPUs of Perlmutter, and 1024 GPUs of Frontier. Plexus achieves unprecedented speedups of 2.3–12.5× over prior state of the art, and a reduction in time-to-solution by 5.2–8.7× on Perlmutter and 7.0–54.2× on Frontier.

CCS Concepts

- Computing methodologies → Distributed artificial intelligence; Massively parallel algorithms.

Keywords

graph neural networks, training, social networks, GPGPUs, SpMM

ACM Reference Format:

Aditya K. Ranjan, Siddharth Singh, Cunyang Wei, and Abhinav Bhatele. 2025. Plexus: Taming Billion-edge Graphs with 3D Parallel Full-graph GNN Training. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3712285.3759890>



This work is licensed under a Creative Commons Attribution 4.0 International License.

SC '25, St Louis, MO, USA

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1466-5/2025/11

<https://doi.org/10.1145/3712285.3759890>

1 Motivation

Graphs are used to represent irregular structures and connections that are ubiquitous in the real-world, such as molecular structures, social networks, and financial transaction networks. In recent years, graph neural networks (GNNs) have emerged as a powerful class of neural networks capable of leveraging the inherent expressiveness of graphs to learn complex properties and relationships within them. Among GNNs, the Graph Convolutional Network (GCN) [19] is the most popular and widely adopted, and serves as the foundation for numerous extensions, including the Graph Attention Network (GAT) [39] and the Graph Isomorphism Network (GIN) [46]. Unlike traditional convolutional neural networks [21], which operate on fixed-size neighborhoods, GCNs exploit the irregular structure and connectivity of graphs.

Real-world graphs are often extremely large, and datasets representing them frequently exceed the memory capacity of a single GPU. Kipf et al. [19] recognize this limitation of their seminal work and suggest mini-batch training for scaling to larger graphs, where a small subset of nodes is used in each iteration to update the model. Since efficient and scalable full-graph based approaches are missing, most modern frameworks such as PyTorch Geometric [13] and DGL [43] use mini-batch training as their default.

In mini-batch training, in a single GCN layer, nodes in the mini-batch first collect information from their immediate neighbors. By aggregating feature embeddings from a node's neighborhood and applying a feed-forward transformation, GCNs can address tasks such as node-level, link-level, and graph-level predictions. For a model with K such GCN layers, a node aggregates features from its K -hop neighborhood. However, even for small values of K , this can quickly result in a phenomenon known as neighborhood explosion, accessing large portions of the graph and undermining the efficiency of mini-batch training [9]. To mitigate this issue, sampling algorithms such as GraphSAGE [15] and FastGCN [8] are typically applied alongside mini-batch training to reduce the number of neighbors considered, thereby lowering memory consumption.

While sampling is widely used, it comes with inherent limitations. Most notably, sampling introduces approximations that can lead to degradation in accuracy [17]. Further, CPU-GPU data transfers in sampling often dominate training time and add unnecessary

complexity [49]. Full-graph training, on the other hand, can achieve competitive performance without these trade-offs in many scenarios as shown by Jia et al.'s ROC framework [17]. Full-graph training makes no approximations in the training process and avoids the complexity of choosing an appropriate sampling strategy with suitable hyperparameters. For these reasons, in this work, we focus on the full-graph training paradigm, avoiding any approximations.

Graphs are typically represented as adjacency matrices with a non-zero entry for each edge. The non-zero entries are sparsely and unevenly distributed across the matrix. Among the six graphs we use for evaluation in this work, the fraction of zeros in the adjacency matrix ranges from 99.79% to 99.99%. The largest of these graphs has ~111 million vertices and ~1.6 billion directed edges. These characteristics of graphs introduce several challenges in parallelizing the training. First, high memory requirements necessitate distributing the graph and its features, and the associated computation across multiple GPUs. This incurs high communication overheads due to the need to synchronize large intermediate activations and gradients between GPUs. Consequently, parallel GNN training quickly becomes communication-bound, making it difficult to scale efficiently to a large number of GPUs. Second, the aggregation phase involves Sparse Matrix-Matrix Multiplication (SpMM), which dominates the computational time and suffers from poor performance on GPUs due to irregular memory access patterns and low data reuse. Third, unevenly distributed sparsity patterns in the adjacency matrix can lead to significant computational load imbalance across different GPUs, which can ripple through an epoch, and impact communication times as well.

In order to address the challenges mentioned above, we propose a three-dimensional (3D) parallel algorithm that enables scaling to large graphs by distributing all matrices efficiently across multiple GPUs, and parallelizes all matrix multiplication computations involved in training. Our approach draws inspiration from Agarwal et al.'s 3D parallel matrix multiplication algorithm [3], which has been used in several distributed deep learning frameworks, including Colossal-AI's unified deep learning system [24], AxoNN [34, 35], and Eleuther AI's framework OSLO [1]. We introduce several optimizations in our baseline implementation to improve performance further. One optimization is a double permutation scheme, which ensures a near-perfect even distribution of non-zeros across distributed matrices, which helps eliminate load imbalance. We also develop a performance model that helps users select an optimal configuration for mapping computation to a 3D virtual GPU grid. This eliminates the need for exhaustive testing of different 3D configurations while ensuring robust performance outcomes.

Our key contributions are summarized as follows:

- We present Plexus¹, an open-source 3D parallel framework for full-graph GNN training that scales to massive graphs and large GPU-based supercomputers.
- A performance model to identify the optimal configuration for arranging GPUs within a 3D virtual grid.
- Performance optimizations, including a double permutation scheme to mitigate load imbalance, and blocked aggregation to reduce performance variability.

¹<https://github.com/hpcgroup/plexus>

- Unprecedented scaling to 1024 GPUs on Frontier at OLCF and 2048 GPUs on Perlmutter at NERSC – the largest-scale full-graph GNN training reported to date.
- Significant speedups, achieving 2.3–12.5× faster training than state-of-the-art frameworks, and cutting time-to-solution by 5.2–8.7× on Perlmutter and 7.0–54.2× on Frontier.

2 Background and Related Work

In this section, we provide an overview of how GNNs work, different training paradigms for GNNs, as well as challenges associated with distributed full-graph GNN training. We also present existing GNN frameworks and their limitations, motivating the need for our work.

2.1 Mathematical Formulation of a GCN layer

Similar to other ML models, GCNs can have different downstream tasks depending on the application. They can be used for predicting whether an edge exists between two nodes, predicting a holistic property of the whole graph, predicting classes for individual nodes, etc. In this work, we focus on the node-level classification task. However, we note that our method can be easily be adapted to other downstream tasks as well. The primary goal of a GNN in this setting is not only to learn a function that maps nodes to their target outputs but also to learn high-quality, low-dimensional node embeddings that place similar nodes close together in the embedding space. In this section, we will show how this task is formulated using a GCN.

The edges in a graph are represented by a sparse adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, where N is the number of nodes in the graph. Prior to training, self-loops are added to \mathbf{A} so that each node's learned representation includes its own features. \mathbf{A} is then normalized by scaling each edge $A_{u,v}$ by $\frac{1}{\sqrt{d_u d_v}}$ where d_u and d_v are the degrees of nodes u and v respectively. This is common practice to mitigate numerical instabilities such as exploding/vanishing gradients [19].

The forward pass of a Graph Convolutional Network (GCN) layer i consists of three key steps:

(1) Aggregation: Each node has a low-dimensional feature vector associated with it. These feature vectors are stored in the features matrix $\mathbf{F}^{Li} \in \mathbb{R}^{N \times D^{Li}}$ where D^{Li} is the features dimension at layer i . In the first step of the forward pass, every node aggregates the features from its immediate neighbors using an aggregation operator like sum and captures the local graph structure. This is achieved by performing an SpMM - multiplying the adjacency matrix \mathbf{A} with the features matrix $\mathbf{F}^{Li} \in \mathbb{R}^{N \times D^{Li}}$. This results in an intermediate matrix $\mathbf{H}^{Li} \in \mathbb{R}^{N \times D^{Li}}$

$$\mathbf{H}^{Li} = \text{SpMM} \left(\mathbf{A}, \mathbf{F}^{Li} \right) \quad (2.1)$$

↑ aggregation output
↑ adjacency matrix
↑ features matrix

Without loss of generality, this is shown for the undirected case. For directed graphs, the adjacency matrix can be transposed for aggregation of features from incoming neighbors.

(2) Combination: The aggregated features are transformed into a new low-dimensional space using a weight matrix $\mathbf{W}^{Li} \in \mathbb{R}^{D^{Li} \times D^{Li+1}}$, resulting in an intermediate matrix $\mathbf{Q}^{Li} \in \mathbb{R}^{N \times D^{Li+1}}$.

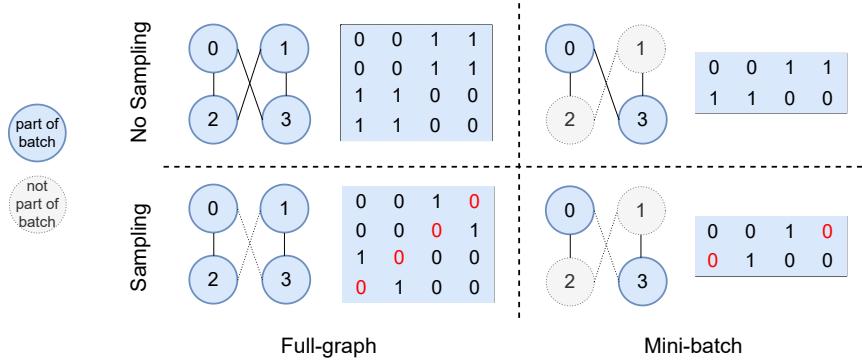


Figure 1: Different paradigms of GNN training that can be combined together, shown in four quadrants. Each quadrant shows a sample graph and its adjacency matrix. Blue nodes are part of the batch and grey nodes are not. Solid lines indicate edges considered during aggregation, and dashed line represent edges that are not considered. Red values in the adjacency matrix indicate that an entry has been modified.

$$\mathbf{Q}^{L_i} = \text{SGEMM} \left(\mathbf{H}^{L_i}, \mathbf{W}^{L_i} \right) \quad (2.2)$$

combination output
↓
↑ aggregation output
weight matrix

(3) Activation: A non-linear activation function σ (e.g. ReLU) is then applied to \mathbf{Q}^{L_i} , yielding the output matrix for the current layer $\mathbf{F}^{L_{i+1}} \in \mathbb{R}^{N \times D^{L_{i+1}}}$. This will be used as the input to the next layer.

$$\mathbf{F}^{L_{i+1}} = \sigma \left(\mathbf{Q}^{L_i} \right) \quad (2.3)$$

output matrix
↓
activation function
↑ combination output

The corresponding backward pass for layer i involves computing gradients as follows:

(1) Compute the gradient of the loss \mathcal{L} with respect to \mathbf{Q}^{L_i} :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Q}^{L_i}} = \frac{\partial \mathcal{L}}{\partial \mathbf{F}^{L_{i+1}}} \odot \sigma' \left(\mathbf{Q}^{L_i} \right) \quad (2.4)$$

element-wise multiplication

(2) Compute the gradient of the loss with respect to the weight matrix \mathbf{W}^{L_i} :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{L_i}} = \text{SGEMM} \left(\left(\mathbf{H}^{L_i} \right)^T, \frac{\partial \mathcal{L}}{\partial \mathbf{Q}^{L_i}} \right) \quad (2.5)$$

(3) Compute the gradient of the loss with respect to \mathbf{H}^{L_i} :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{H}^{L_i}} = \text{SGEMM} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{Q}^{L_i}}, \left(\mathbf{W}^{L_i} \right)^T \right) \quad (2.6)$$

(4) Compute the gradient of the loss with respect to \mathbf{F}^{L_i} :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{F}^{L_i}} = \text{SpMM} \left(\mathbf{A}^T, \frac{\partial \mathcal{L}}{\partial \mathbf{H}^{L_i}} \right) \quad (2.7)$$

The gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{F}^{L_0}}$ at the first layer is then used to update the input features and learn meaningful node embeddings.

2.2 Different Paradigms of GNN Training

Four main GNN training paradigms exist (see Figure 1). *Full-graph* training (upper-left) uses the entire graph in each iteration, updating all node features and requiring the entire graph in memory. This makes no approximations but is memory-intensive. *Mini-batch* training (upper-right) updates only a small subset of nodes per iteration (e.g., Nodes 0 and 3), but suffers from neighborhood explosion in deeper GNNs [9]. To address this, *Mini-batch sampling* (bottom-right), the most common paradigm, combines mini-batching with neighbor sampling at each layer and only uses some edges for aggregation. Finally, *Full-graph sampling* (bottom-left) uses the entire graph as a batch but samples edges, which is less common.

While there are some sampling algorithms that have fairly successful adoption, they still lack a community standard. Graph-SAGE [15] samples a fixed number of neighbors per node, while FastGCN [8] samples per layer. LADIES enhances FastGCN by considering inter-layer dependencies [52]. Cluster-GCN [11] samples within dense subgraphs. Recent work explores adaptive sampling (GRAPES [48]) and handling homophilic/heterophilic graphs (AGS-GNN [12]). However, sampling introduces a trade-off between accuracy and efficiency, causing bias and variance [25] in training. The limited scale of graphs used in these studies (max of 2.5 million nodes) raises concerns about information loss on larger, real-world datasets with different structural properties. Consequently, the effectiveness of sampling remains inconclusive, motivating our focus on distributed full-graph training.

2.3 Distributed Full-graph GNN Training

Early distributed full-graph GNN training frameworks include ROC [17], which partitions graphs using online linear regression and balances CPU-GPU transfer with GPU memory. CAGNET [38] uses tensor-parallel algorithms (1D, 1.5D, 2D, 3D) for SpMM. While the 2D and 3D algorithms offer asymptotic communication reduction, the 1D and 1.5D algorithms scale better due to lower constants. A sparsity-aware version of CAGNET's 1D/1.5D algorithms [26] improves performance by communicating only necessary features.

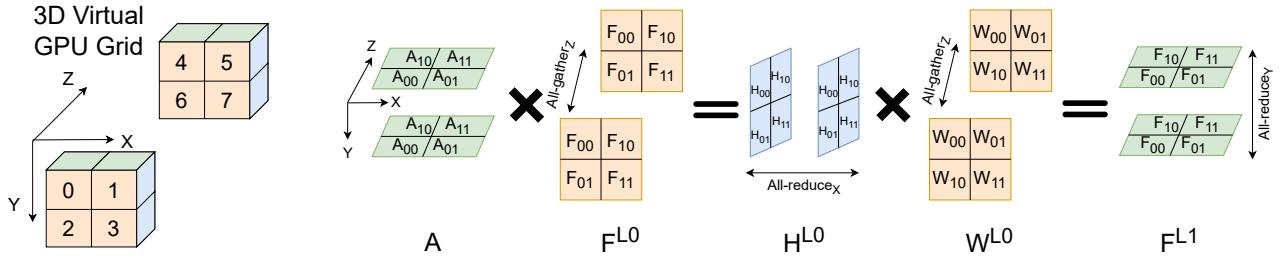


Figure 2: An overview of the 3D tensor parallel algorithm for GNN training. Eight GPUs are arranged in a 3D grid ($X=Y=Z=2$) and matrices in layer 0 of the network are distributed across different planes (shown in different colors).

MG-GCN [6] optimizes CAGNET with communication-computation overlap. RDM [20] builds on CAGNET with near communication-free training by replicating one of the matrices.

Other full-graph frameworks introduce approximations for scalability. BNS-GCN [41] partitions with METIS and samples boundary nodes, but its convergence on diverse datasets needs further validation. PipeGCN [42] pipelines communication and computation, potentially causing stale features/gradients, with sensitivity varying across graphs. DGCL [7] minimizes communication using graph characteristics and cluster topology. NeutronTP [4] uses tensor parallelism by only distributing the features to avoid load imbalance.

Table 1: Summary of state of the art in distributed full-graph GNN training. The number of nodes and edges of the graph datasets, and number of GPUs are the largest values reported in each paper.

Name	Year	# Nodes	# Edges	# GPUs
AdaQP [40]	2023	2.5M	114M	8
RDM [20]	2023	3M	117M	8
MG-GCN [6]	2022	111M	1.6B	8
Sancus [30]	2022	111M	1.6B	8
MGG [45]	2023	111M	1.6B	8
DGCL [7]	2021	3M	117M	16
ROC [17]	2020	9.5M	232M	16
NeutronStar [44]	2022	42M	1.5B	16
GraNNDis [36]	2024	111M	1.6B	16
NeutronTP [4]	2024	244M	1.7B	16
CDFGNN [50]	2024	111M	1.8B	16
PipeGCN [42]	2022	111M	1.6B	32
CAGNET [38]	2020	14.2M	231M	125
BNS-GCN [41]	2022	111M	1.6B	192
SA+GVB [26]	2024	111M	1.6B	256
Plexus (this work)	2025	111M	1.6B	2048

Table 1 shows limited scaling across many GPUs in existing full-graph works, with a handful using more than 16 GPUs. Many focus on 1D SpMM variants, lacking a practical scalable 3D algorithm despite its theoretical communication advantages. This motivates Plexus, our framework aiming for approximation-free, scalable 3D full-graph training for large graphs and high GPU counts.

3 A Three-dimensional Tensor Parallel Approach to Full-graph GNN Training

We now describe our approach to parallelizing a GCN layer and the entire network, and our adaptation of Agarwal’s 3D parallel matrix multiply algorithm for GNN training in Plexus.

3.1 Parallelizing a Single GCN Layer

Tensor parallelism is a popular strategy for parallelizing GNN training. While previous works have experimented with 1D to 3D tensor parallel approaches, in this work, we focus on 3D tensor parallelism. We take inspiration from Agarwal et al.’s three-dimensional (3D) parallel matrix multiplication algorithm [3] for distributing matrices and parallelizing matrix multiplication kernels across multiple GPUs. Below, we describe how we adapt this 3D matrix multiplication approach to parallelize GNN training and Sparse Matrix-Matrix Multiplication (SpMM) computations.

Given a number of GPUs, G , in a job allocation, we first arrange the GPUs into a 3D virtual grid. We refer to the number of GPUs along each dimension as G_x , G_y , and G_z respectively, such that $G = G_x \times G_y \times G_z$. Each GPU creates process groups that allow it to communicate with its neighbors in each of the three dimensions of the grid. The matrices in a layer are then distributed across this grid. Here, we describe how this is done for the first layer of the GCN, and this can be applied similarly to the other layers.

First, we shard (divide and map to different GPUs) the sparse adjacency matrix, A , across the ZX -plane and replicate it across the Y -parallel process group (see Figure 2). Then we shard the input features matrix, F^{L0} , across the XY -plane and further shard it across the Z -parallel process group. The reason that F^{L0} is sharded and not replicated across the third process group is to save memory. Since the input features are made trainable to learn node embeddings, they have gradients and optimizer states associated with them which are additional memory requirements. Finally, we shard the weights across the YX -plane and also further across the Z -parallel process group due to the additional memory requirements of the gradients and optimizer states. Figure 3 shows the shapes of the matrix shards (sub-blocks or sub-matrices) for layer 0.

Pseudo code for the forward and backward pass of layer 0 is shown in Algorithm 1 and 2 respectively. Before describing the algorithm, note that when we refer to any matrix, it is a shard of that matrix on a given GPU. Lines 3-5 show the aggregation step, in which the input features matrix shard F is all-gathered

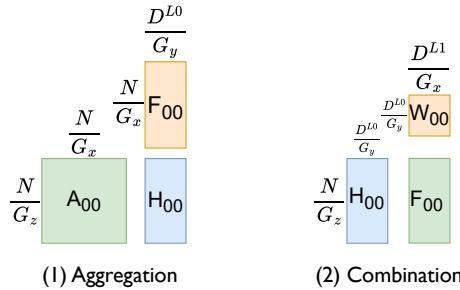


Figure 3: Shapes of the matrix shards (sub-blocks) in the first layer on a single GPU, showing two key matrix multiplications in the forward pass.

across the Z -parallel process group since it is additionally sharded across this dimension of the grid. The adjacency matrix shard A is then multiplied with F to get the aggregation output H . Since this results in a partial output, an all-reduce is performed on H across the X -parallel process group.

Algorithm 1 Forward Pass of Layer 0

```

1: function FORWARD( $A, F, W$ )
2:   // Step 1: Aggregation
3:   All-gather  $F$  across  $Z$ -parallel group
4:    $H = \text{SpMM}(A, F)$ 
5:   All-reduce  $H$  across  $X$ -parallel group

6:   // Step 2: Combination
7:   All-gather  $W$  across  $Z$ -parallel group
8:    $Q = \text{SGEMM}(H, W)$ 
9:   All-reduce  $Q$  across  $Y$ -parallel group

10:  // Step 3: Non-linear Activation
11:   $F = \sigma(Q)$ 
12:  Return  $F$ 
13: end function

```

Lines 7-9 show the next combination step. First, the weights matrix shard W is all-gathered across the Z -parallel process group since it is additionally sharded across this dimension of the grid. The intermediate output from the aggregation is then multiplied by the weights matrix. This again results in a partial output Q , which is all-reduced across the Y -parallel process group. Finally, we apply a non-linear activation on this and return it to be used in the next layer (lines 11-12). These series of matrix multiplications and all-reduce steps are also demonstrated visually in Figure 2. The backward pass for the first layer is shown in Algorithm 2.

3.2 Parallelizing All Layers in the Network

The parallelization of other layers in the GNN is similar to the first layer but we need to address a subtle but important detail first. As can be seen in Figure 2, the output of the first layer F^{L1} is sharded across the ZX-plane. However, this will also be the input to the next layer, which becomes an issue since the adjacency

Algorithm 2 Backward Pass of Layer 0

```

1: function BACKWARD( $\frac{\partial \mathcal{L}}{\partial Q}$ )
2:    $\frac{\partial \mathcal{L}}{\partial W} = \text{SGEMM}(H^T, \frac{\partial \mathcal{L}}{\partial Q})$ 
3:   Reduce-scatter  $\frac{\partial \mathcal{L}}{\partial W}$  across Z-parallel group

4:   All-gather W across Z-parallel group
5:    $\frac{\partial \mathcal{L}}{\partial H} = \text{SGEMM}(\frac{\partial \mathcal{L}}{\partial Q}, W^T)$ 
6:   All-reduce  $\frac{\partial \mathcal{L}}{\partial H}$  across X-parallel group

7:    $\frac{\partial \mathcal{L}}{\partial F} = \text{SpMM}(A^T, \frac{\partial \mathcal{L}}{\partial H})$ 
8:   Reduce-scatter  $\frac{\partial \mathcal{L}}{\partial F}$  across Z-parallel group

9:   Return  $\frac{\partial \mathcal{L}}{\partial F}, \frac{\partial \mathcal{L}}{\partial W}$ 
10: end function

```

matrix A of the next layer is also sharded across the ZX -plane, and so the dimensions of the two matrices are incompatible. To resolve this, we either need to communicate F^{L1} to the XY -plane or communicate A to the YZ -plane. Unfortunately, these solutions would add increased communication complexity and are non-trivial to implement efficiently.

To address this problem, we store a separate shard of the adjacency matrix A^{L1} that is sharded across the YZ -plane for the next layer $L1$. Similarly, for the third layer $L2$, we store a shard of the adjacency matrix A^{L2} that is sharded across the XY -plane. This ensures that the dimensions of the matrices are compatible for local computations. This scheme is shown in Figure 4, where we can see how the three adjacency matrix shards allow for the output of one layer to be used as the input for the next layer. Importantly, this does not result in needing more than three unique shards of the adjacency matrix. The output of the third layer F^{L3} is sharded across the XY -plane, which is the same plane that F^{L0} is sharded across. So for the fourth layer $L3$, we can now reuse A^{L0} and then repeat using the same adjacency matrix shards for subsequent layers.

This process of cycling through three different adjacency shards for different layers also changes a few communication steps in Algorithm 1. For subsequent layers after the first one, the features matrix F will only be sharded across two dimensions of the grid since it does not have optimizer states like the input features. This means that the first all-gather in the forward pass (line 2) will not take place. Likewise, the last reduce-scatter (line 8) in the backward pass is changed to an all-reduce since the gradients are replicated across the third process group. Using different shards of the adjacency matrix is the main change to parallelize all the layers of the model and the core idea remains the same.

4 Performance Model

Next, we describe the performance model we have developed to identify near-optimal 3D configurations of the virtual GPU grid. We model both the SpMM computation and communication times.

4.1 Modeling Computation

Plexus shards matrices such that local matrix operations should take the same amount of time across different 3D configurations,

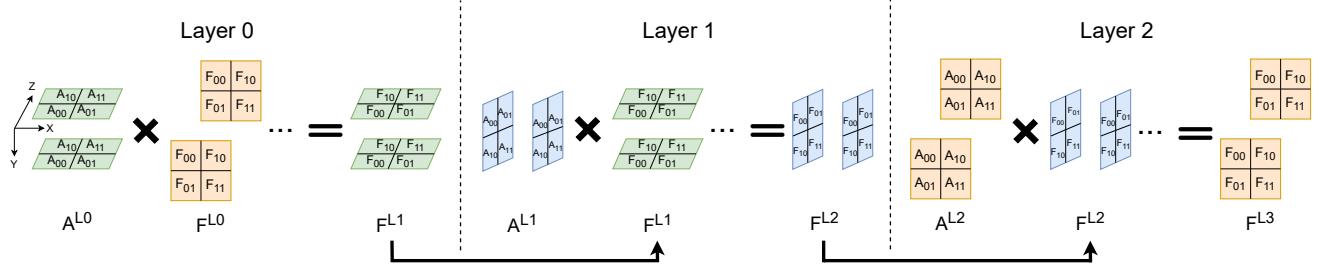


Figure 4: Applying the 3D tensor parallel algorithm to all layers of a 3-layer GCN, connecting the output of one layer to the input of the next using unique shards of the adjacency matrix.

assuming a uniform distribution of nonzeros. We show this in the derivation below, where we see that the total number of FLOPs needed to calculate the aggregation output H is a term that is constant across all configurations for $G = G_x \times G_y \times G_z$ GPUs.

Given the number of nodes in the graph N and the input features dimension D^{L0} , the number of elements in H in the first layer is:

$$\frac{\text{number of nodes}}{G_z} \times \frac{\text{input features dimension}}{G_y} = N \times \frac{D^{L0}}{G_y} \quad (4.1)$$

Given the number of nonzeros in the adjacency matrix NNZ , the number of floating point operations per element is:

$$O\left(\frac{2 \times NNZ}{N \times G_x}\right) \quad (4.2)$$

Hence, the total number of floating point operations to calculate the aggregation output H is a result of multiplying the expressions in equations (4.1) and (4.2) together:

$$O\left(\frac{2 \times NNZ \times D^{L0}}{G}\right) \quad (4.3)$$

Despite expecting similar computation times for different 3D configurations, in practice, we observe that SpMM times vary across configurations. We hypothesize that shorter-fatter dense matrices lead to more efficient SpMMs. This is consistent with the literature optimizing tall-skinny dense SpMM. Yang et al. [47] propose row-splitting for coalesced memory access, which they note is more efficient with fewer nonzeros per row. This is achieved by configurations in our algorithm reducing the common dimension of local multiplications. Selvitopi et al. [32] show non-ideal scaling of SpMM time with the number of processors and that the algorithm choice can impact scaling.

To test our hypothesis, we took the adjacency and feature matrices from ogbn-products and multiplied them under two different configurations for 64 GPUs. In config U, $G_x = 64$ and the common dimension is sharded by 64, reducing the number of nonzeros per row. In config V, $G_y = 64$ and the columns of the dense matrix

are sharded by 64, making it skinny. Both of these have the same workload in terms of the number of FLOPs. However, we observed that V was $\sim 8\times$ slower. After profiling with Nsight Compute [28] (metrics in Table 2), we noticed that it launched ~ 64 times more blocks, which is proportional to its $64\times$ larger common dimension size. This means less work per block and a higher number of smaller memory requests. Consequently, V's L2 Cache and DRAM throughput were drastically lower, and uncoalesced global memory accesses were much higher, indicating poor memory access patterns and suboptimal memory utilization in the tall-skinny dense SpMM regime.

Table 2: Nsight Compute metrics for SpMM(A, H) on a single GPU for two configurations of Plexus – U ($G_z = 1, G_x = 64, G_y = 1$) and V ($G_z = 1, G_x = 1, G_y = 64$).

Metric	U	V
Grid Size	20,223	1,313,241
Uncoalesced Global Memory Access Sectors	84,960	3,939,912
L2 Cache Throughput	61.31	12.65
DRAM Throughput	72.83	8.24

In Plexus, we introduce a computational model to predict which configurations result in more efficient SpMMs. The model is shown for the first layer using the equations below:

$$\begin{aligned} \text{flops_cost} &= NNZ \times D^{L0} \\ \text{fwd_penalty} &= \frac{N}{G_x} \times \frac{G_y}{D^{L0}} \\ \text{bwd_penalty} &= \frac{N}{G_z} \times \frac{G_y}{D^{L0}} \\ \text{comp_cost} &= \sqrt{\text{flops_cost}} \times (1 + \text{fwd_penalty} + \text{bwd_penalty}) \end{aligned} \quad (4.4)$$

The first term `flops_cost` is proportional to the total FLOPs, which is the number of nonzeros NNZ in the sparse matrix A multiplied by the number of columns D^{L0} in the dense matrix F . The second term `fwd_penalty` ranks certain configurations as better than others based on the matrix shape. This term is first weighted proportional to the size of the matrix F 's first dimension: N/G_x (the common dimension). It is then weighted inversely proportional to the size of

the second dimension of F : D^{L0}/G_y . This penalizes configurations causing tall-skinny dense matrices. A similar calculation is done for the backward pass SpMM.

The final computational cost is calculated as the square root of flops_cost (to reduce outlier impact of larger matrices), multiplied by penalty terms to account for poor matrix shapes, and summed across all layers. To convert this to time, we performed runs on Perlmutter across various datasets, configurations, and GPU counts (including all ogbn-products configurations on 64 GPUs). We then used scikit-learn [29] to fit a linear regression model to these 67 data points, determining coefficients for our three terms to predict SpMM time for any configuration.

To validate our model, we used a random train-test split of 70-30 for 1000 independent iterations. We recorded an average R^2 of 0.89 and RMSE of 16.8 ms for the train splits, and an average R^2 value of 0.79 and RMSE of 20.1 ms for the test splits, indicating that the model is able to predict the SpMM time with a relatively high degree of accuracy and can generalize fairly well. The learned coefficients for the three terms are approximately 7.8×10^{-4} , 7.8×10^{-10} , and -2.6×10^{-10} .

4.2 Modeling Communication

Different 3D grid configurations significantly impact communication time and overall performance, especially at scale. Optimal configuration selection is non-trivial. Several works model communication time for distributed deep learning, such as ATP [10], Alpa [51], AxoNN [34, 35], Oases [23], and DGCL [7]. Plexus adapts AxoNN's communication model [35], which uses ring algorithm equations from Thakur et al. [37] and Rabenseifner [31]. The latency term is omitted since the messages are large and bandwidth-bound. The all-reduce time for a buffer of size M across G GPUs with bandwidth β can be modeled as:

$$T_{\text{all-reduce}} = \frac{\text{time}}{\beta} \times \left(\frac{\text{number of GPUs}}{G} - 1 \right) \times M \quad (4.5)$$

Plexus extends this across layers by using the appropriate matrix dimensions and process group sizes for each layer, as described in Section 3.2. The model considers GPU topology, prioritizing Y , X , and then Z parallelism within a node. If a process group is within a node, it can utilize intra-node bandwidth β_{intra} . Otherwise, it is bound by inter-node bandwidth β_{inter} , which can potentially be lower due to link contention. We show how this is calculated for β_z , bandwidth along the Z -parallel group, in the following equation:

$$\beta_z = \begin{cases} \beta_{\text{intra}} & \text{if } G_x \times G_y \times G_z \leq G_{\text{node}} \\ \frac{\beta_{\text{inter}}}{\min(G_{\text{node}}, G_x \times G_y)} & \text{otherwise} \end{cases} \quad (4.6)$$

where G_{node} is the number of GPUs within a node.

After the effective bandwidths are similarly calculated for β_x and β_y , we can plug them in to the equations for each collective and calculate the predicted communication times for each configuration.

4.3 Unified Performance Model

We combine predicted SpMM time and communication time to estimate total epoch time for each configuration, neglecting smaller

dense computation and loss calculation. Figure 5 shows results for ogbn-products on 64 Perlmutter GPUs, indicating better performance for 3D configurations over 2D and 1D. The three-layer GCN favors symmetric configurations for balanced communication and SpMM efficiency. As we can observe, a strong correlation exists between predicted and observed epoch times, accurately predicting top configurations.

Predicted vs. observed time for ogbn-products on 64 GPUs (Perlmutter)

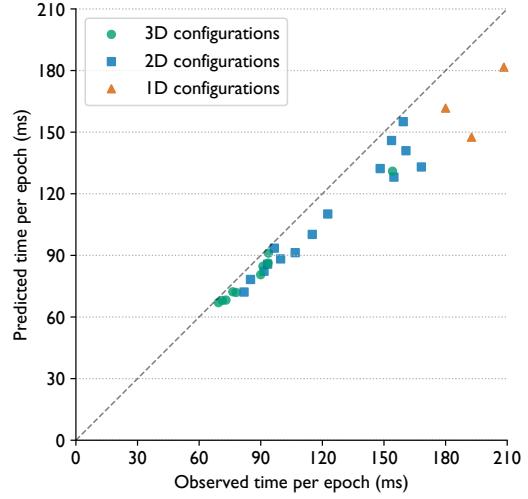


Figure 5: Validating the performance model for the ogbn-products dataset on 64 GPUs of Perlmutter.

5 Performance Optimizations

Parallelizing graph neural networks can pose unique challenges in the form of load imbalance caused by uneven sparsity patterns and high communication overheads arising due to the extremely large sizes of graphs. We address some of these issues in Plexus by introducing several optimizations that improve the performance of our framework.

5.1 Double Permutation for Load Balancing

The sparse and uneven distribution of nonzeros in the adjacency matrix can cause load imbalance among matrix shards assigned to different GPUs, leading to computational stragglers and slower training. Graph partitioners such as METIS [18] can be used to partition graphs to minimize edge cuts and balance vertices, which is beneficial for fine-grained communication. However, the all-reduce in Plexus is performed on dense aggregation outputs and does not require graph structure awareness for communication. While graph partitioners distribute rows/nodes, our 2D matrix decomposition requires even nonzeros to be evenly distributed across 2D shards.

Node permutation offers a simple solution without complex optimization or graph structure knowledge. Unlike graph partitioning, which requires re-partitioning for different GPU counts, permutation is a one-time preprocessing step for each graph dataset. The naïve permutation scheme uses a permutation matrix P to map original node indices to permuted indices.

$$F^{L1} = \sigma \left(\left(P \ A \ P^T \right) \left(P \ F^{L0} \right) W^{L0} \right) \quad (5.1)$$

adjacency matrix
permutation matrix input features weight matrix

$$F^{Li} = \sigma \left(\left(PAP^T \right) F^{Li-1} W^{Li-1} \right) \quad (5.2)$$

Equation (5.1) is used for the first layer and (5.2) is used for all subsequent layers. The same permutation is applied to adjacency matrix columns and input features rows to maintain output consistency for subsequent layers. This preprocessing step significantly reduces load imbalance. However, due to dense graph clusters, a single permutation is insufficient, as nonzeros remain concentrated around diagonal blocks. To further disrupt community coupling, we apply distinct permutations (P_r for rows, P_c for columns) to the adjacency matrix, repeating this two-permutation scheme every two layers for more effective nonzero redistribution. This requires storing two adjacency matrix versions.

$$F^{L1} = \sigma \left(\left(P_r \ A \ P_c^T \right) \left(P_c F^{L0} \right) W^{L0} \right) \quad (5.3)$$

permutation matrix for rows permutation matrix for columns

$$F^{Li} = \sigma \left(\left(P_c A P_r^T \right) F^{Li-1} W^{Li-1} \right) \quad (5.4)$$

Alternating between two permutations (P_r and P_c) further balances computation by disrupting tightly coupled communities. Table 3 shows near-perfect load balance on the europe_osm dataset (8x8 shards) with double permutation, outperforming the naïve single permutation.

Table 3: Comparison of different permutation methods, showing the ratio of the maximum number of non-zeros to the mean across 8x8 shards of the adjacency matrix for the europe_osm dataset.

Method	Max/Mean
Original	7.70
Single permutation	3.24
Double permutation (this work)	1.001

Adopting this optimization increases the memory required to store each adjacency matrix shard by a factor of two. Since the number of such shards is $\min(3, L)$ for L GCN layers, the memory overhead of storing the shards after applying this optimization then becomes $\min(6, L)$. Given that the number of GCN layers is typically small (two to four) to avoid oversmoothing [22], this is a reasonable trade-off for improved load balance and performance.

5.2 Blocked Aggregation

While our double permutation achieves near-perfect adjacency matrix load balance, we observed performance variability in the forward pass SpMM across epochs on larger datasets (Isolate-3-8M, products-14M) for a modest number of GPUs (8-32). This leads to load imbalance in the subsequent all-reduce and increased average epoch time. To address this, we optimized the aggregation by blocking the sparse adjacency matrix into smaller row-blocks, since we did not observe this for smaller matrices. After each block's SpMM,

an all-reduce is performed on it, and blocks are concatenated at the end. This mitigated performance variability in the SpMM, and significantly reduced communication also as a side effect, as shown in Figure 6 (left).

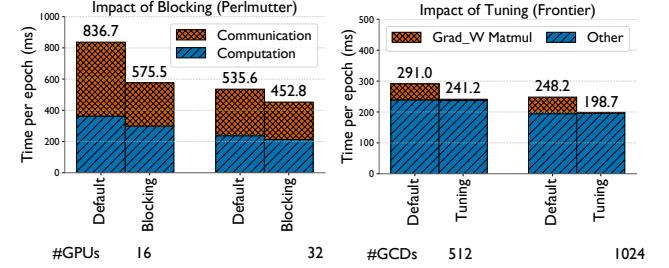


Figure 6: Impact of blocked aggregation on performance for Isolate-3-8M on 16 and 32 GPUs of Perlmutter (left). Impact of dense matrix multiplication tuning on performance for products-14M on 512 and 1024 GCDs of Frontier (right).

5.3 Dense Matrix Multiplication Tuning

Despite dense matrix multiplication taking a small amount of time in our workloads, we observed scaling issues on Frontier at high GCD counts (≥ 512 GCDs) with large datasets (such as Isolate-3-8M and products-14M) for the $\frac{\partial \mathcal{L}}{\partial W}$ calculation, where the first matrix was transposed. GEMM BLAS kernels have NN, NT, TN, TT modes with varying performance (e.g., NT and TN can be slower [33]). We optimized these dense kernels by reversing the multiplication order: $\frac{\partial \mathcal{L}}{\partial W} = \left(\text{SGEMM} \left(\frac{\partial \mathcal{L}}{\partial Q}^T, H \right) \right)^T$. Figure 6 (right) shows a significant time reduction for this GEMM on Isolate-3-8M (from ~ 50 ms to negligible), enabling Plexus to scale to 1024 GCDs on Frontier.

5.4 Parallel Data Loading

Many GNN frameworks load entire datasets into CPU memory before transferring shards to the GPU, which is unsustainable for large graphs. Plexus implements a parallel data loader to avoid this. It shards processed data into 2D files offline (e.g., 8x8), and the data loader for each GPU only loads, merges, and extracts the shards it needs. This significantly reduces CPU memory usage and data loading time. For ogbn-papers100M on 64 GPUs, CPU memory requirements decreased from 146 GB to 9 GB (16x16 shards), and loading time from 139s to 7s with parallel data loading.

6 Experimental Setup

Below, we provide details of the experimental setup used to evaluate Plexus, including the supercomputer platforms and datasets used, model details, and other state-of-the-art (SOTA) frameworks we compare against.

6.1 Details of Supercomputer Platforms

Our experiments were conducted on Perlmutter at NERSC, Lawrence Berkeley National Laboratory, and Frontier at OLCF, Oak Ridge National Laboratory. The GPU partition of Perlmutter is connected by the HPE Slingshot 11 network, and has two kinds of compute

nodes. 1,536 nodes have four NVIDIA A100 GPUs each with 40 GB of HBM2 memory per GPU. 256 additional nodes have four A100 GPUs with 80 GB of HBM2 memory per node. We use the 80 GB nodes for runs on 64 and 128 GPUs for the largest dataset. Frontier is also a Slingshot 11 supercomputer with 9,856 compute nodes. Each node on Frontier has four AMD Instinct MI250X GPUs, each with 128 GB of HBM2E memory. Each MI250X GPU is partitioned into two Graphic Compute Dies (GCDs) and each GCD appears as a separate device for launching GPU kernels. The A100 GPU has a peak of 19.5 FP32 Tflop/s, and the MI250X GPU has a peak of 47.9 FP32 Tflop/s. There are four NICs per node on both systems with an injection bandwidth of 25 GB/s. We use PyTorch Geometric 2.6.1, and PyTorch 2.6.0 with CUDA 12.4 on Perlmutter, and ROCm 6.2.4 on Frontier.

6.2 Description of Graph Datasets and the GNN

We conduct experiments using graph datasets of varying sizes, as shown in Table 4. The Reddit dataset is available through PyTorch Geometric, and contains post data from September 2014, with individual posts as nodes and edges connecting two posts if the same user commented on both [15]. The ogbn-products dataset is part of the Open Graph Benchmark (OGB) [16], and depicts Amazon’s product co-purchasing network, where nodes are products sold and edges indicate that the products are purchased together. The ogbn-papers100M dataset, also part of OGB, represents the Microsoft Academic Graph (MAG), where nodes are papers and edges indicate citation relationships. For the Reddit, ogbn-products, and ogbn-papers-100M datasets, we used the input features and labels that were provided with the datasets.

The products-14M datasets is a larger Amazon products network [27]. The Isolate-3-8M dataset is a subgraph of a protein similarity network in HipMCL’s data repository [5]. The europe_osm dataset, part of the 10th DIMACS Implementation Challenge [14], represents OpenStreetMap data for Europe, where nodes correspond to geographical locations, and edges represent roads connecting these points. For the Isolate-3-8M, products-14M, and europe_osm datasets, we randomly generated input features with a size of 128, and generated labels with 32 classes based on the distribution of node degrees.

Table 4: Details of graph datasets used for experiments.

Dataset	# Nodes	# Edges	# Non-zeros	# Features	# Classes
Reddit	232,965	57,307,946	114,848,857	602	41
ogbn-products	2,449,029	61,859,140	126,167,053	100	47
Isolate-3-8M	8,745,542	654,620,251	1,317,986,044	128	32
products-14M	14,249,639	115,394,635	245,036,907	128	32
europe_osm	50,912,018	54,054,660	159,021,338	128	32
ogbn-papers100M	111,059,956	1,615,685,872	1,726,745,828	100	172

For all the experiments, we create a GNN with three GCN layers and a hidden dimension of 128, as increasing the model size beyond that has diminishing returns on the model’s generalization capabilities as shown in Jia et al. [17]. We train for ten epochs in each trial, and take the average performance of the last eight epochs to account for initial fluctuations. For each experiment, we run three independent trials and report the average epoch time over

three trials. We validated Plexus against PyTorch Geometric for correctness as shown in Figure 7.

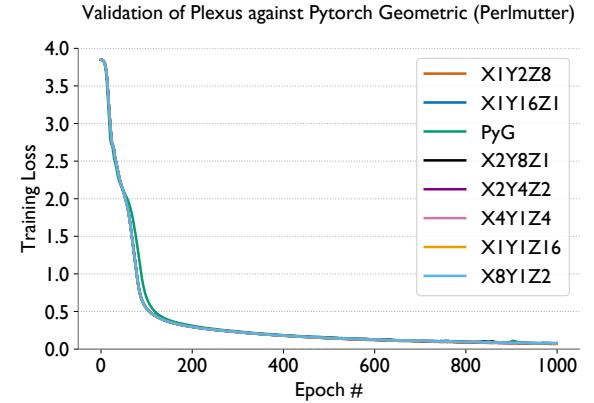


Figure 7: Validating Plexus against a serial PyTorch Geometric baseline on 16 GPUs of Perlmutter with ogbn-products.

6.3 Comparison with Other Frameworks

We compare the performance of Plexus with that of SA, a sparsity-aware implementation of CAGNET [26], and BNS-GCN [41], two SOTA frameworks for distributed full-graph GNN training that have previously been run on hundreds of GPUs as seen in Table 1. We also compare with a variant of SA that uses datasets partitioned using GVB [2], a graph partitioner used by the authors to improve performance (SA+GVB). We contacted the authors to confirm that SA is the most recent and best performing implementation of CAGNET.

For BNS-GCN, we use a boundary sampling rate of 1.0 since Plexus makes no approximations and we are interested in comparing with similar settings. This is akin to vanilla partition parallelism with METIS. We made a small modification to the BNS-GCN code that resolved a bug that led to crashes during training when the boundary size was 0. We also contacted the authors of BNS-GCN regarding unexpectedly high runtimes with METIS, but did not receive a response in time to compare against it. We only compare with these frameworks on Perlmutter as we encountered frequent stability issues and memory errors on Frontier, preventing us from running experiments reliably.

7 Scaling Results

Finally, we present the results of our scaling experiments across six graph datasets on both Perlmutter and Frontier, and compare Plexus with SA, SA+GVB, and BNS-GCN.

7.1 Comparison with SOTA Frameworks

We compare Plexus to the other frameworks only on the Reddit, Isolate-3-8M, and products-14M datasets. ogbn-papers100M results were limited due to partitioning timeouts after 5 hours (BNS-GCN) and out-of-memory issues (SA, SA+GVB). Figure 8 shows these comparative evaluation results. For Reddit, SA performs better at 4 GPUs, but does not scale beyond that. SA+GVB demonstrates

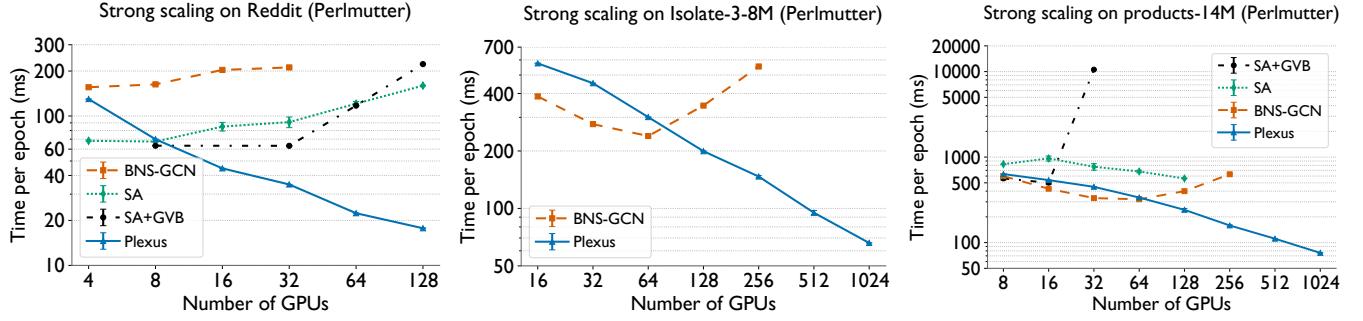


Figure 8: Comparison of strong scaling performance of Plexus, SA, SA+GVB, and BNS-GCN for several datasets on Perlmutter.

somewhat better performance than SA upto 64 GPUs, but also with poor scaling. BNS-GCN scales similarly to SA but is slower in terms of absolute time. Plexus is the only framework that achieves good strong scaling up to 128 GPUs, and a 6x speedup over BNS-GCN on 32 GPUs and 9x over SA on 128 GPUs.

On Isolate-3-8M, both SA and SA+GVB failed to run due to out-of-memory issues. BNS-GCN scales well to 64 GPUs, but quickly degrades in performance beyond this point. Plexus achieves a 3.8x speedup over BNS-GCN at 256 GPUs, and scaling further to 1024 GPUs. BNS-GCN's fine-grained communication is good at a small scale, but has two key issues at larger scales. First, the partitioner starts to divide denser subgraphs, resulting in a larger number of boundary nodes. Second, BNS-GCN utilizes the all-to-all collective during communication. Compared to ring-based collectives used in Plexus where GPUs only communicate with their neighbors, all-to-all send more long-distance messages, which leads to higher latency. Without sampling boundary nodes, METIS is insufficient for BNS-GCN to achieve comparable performance at scale.

For the products-14M dataset, we observe a similar pattern to Isolate-3-8M, where BNS-GCN scales well till 64 GPUs, but then the performance drops sharply following that. SA, on the other hand, starts off with a higher absolute time but is able to scale comparatively better up to 128 GPUs. We tried running it on 256 GPUs, but the job timed out at 20 minutes. SA+GVB performs better than SA for 8 and 16 GPUs, but has a drastic increase in time after that. We observe that Plexus scales up to 1024 GPUs and performs better than both frameworks. It achieves a 2.3x speedup over SA on 128 GPUs and a 4x speedup over BNS-GCN on 256 GPUs.

In order to understand the inflection point between BNS-GCN and Plexus at 64 GPUs further, we look at the breakdown of epoch times in Figure 9. At 32 GPUs, BNS-GCN completes an epoch faster than Plexus primarily due to having a lower communication time, which can be attributed to the fine-grained communication pattern of partition parallelism. In Plexus, on the other hand, the communication time is higher since the collectives are performed on the full dense outputs, and Plexus does not have sparsity-aware modifications like SA. The inefficiency of the all-to-all communication pattern employed by BNS-GCN becomes evident at 64 GPUs.

Another interesting observation is that the computation scaling for the two frameworks also differs. While Plexus shows notable improvements in the computation time from 32 GPUs to 256 GPUs, BNS-GCN's computation time increases with the number of GPUs.

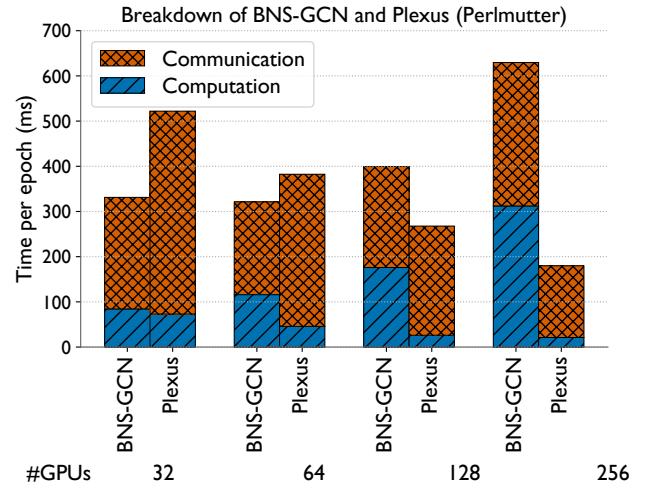


Figure 9: Breakdown of epoch times for BNS-GCN and Plexus on 32-256 GPUs of Perlmutter with products-14M.

After further investigation, we found that the total number of nodes across partitions, including boundary nodes, increased from 18M to 22M for BNS-GCN when going from 32 to 256 GPUs. This explains why the local computation of a partition also increases in addition to the poor scaling of communication.

Overall, Plexus outperforms BNS-GCN, SA, and SA+GVB across the three datasets. While BNS-GCN and SA are more efficient at small scales due to sparsity-aware communication, they struggle at larger scales. Plexus scales well to 1024 GPUs with the lowest absolute epoch times. Its scaling is also more consistent across datasets, even performing competitively at small scales. All of this is achieved without a graph partitioner. Unlike METIS, which timed out for some datasets, and GVB, which ran out of memory on obgn-papers100M at 32 GPUs (as noted by SA's authors in [26]), Plexus' double permutation scheme mitigates load imbalance scalably with minimal overheads.

7.2 Strong Scaling of Plexus

In addition to the three datasets discussed above, we also ran Plexus on three other datasets to demonstrate its strong scaling capabilities

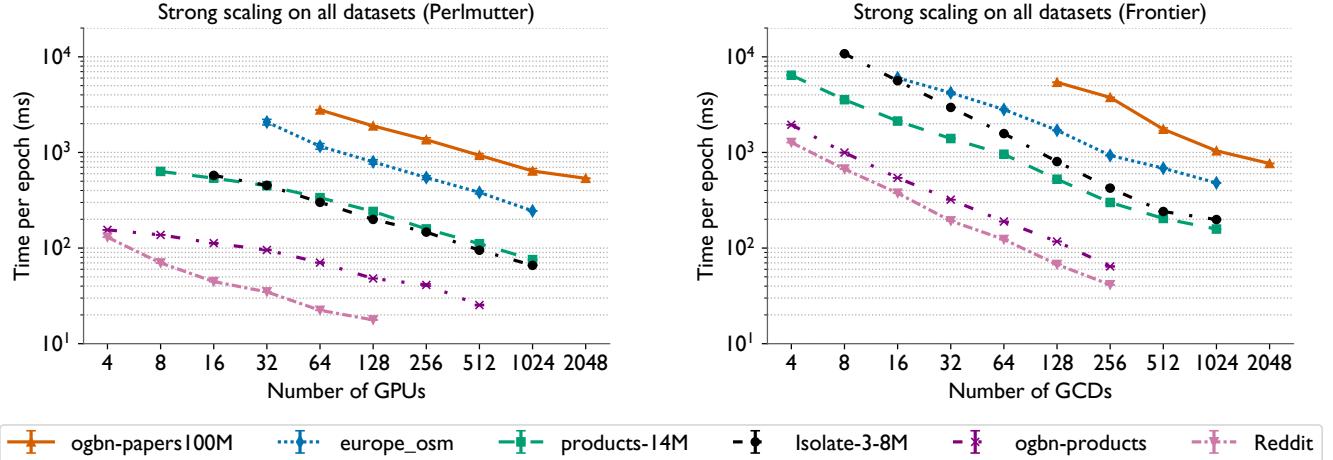


Figure 10: Strong scaling performance of Plexus for six graph datasets of different sizes (Table 4) on both Perlmutter (left) and Frontier (right). Note that the x-axis shows GPUs for Perlmutter and GCDs for Frontier.

on both Perlmutter and Frontier (Figure 10). The sparsity level of a graph determines the communication to computation ratio in Plexus. As a result, Plexus scales better with Reddit, a denser graph compared to ogbn-products on Perlmutter (left plot). When training with ogbn-products, Plexus becomes communication-dominated quicker than Reddit, explaining the increasing gap between the performance for the two datasets (on Perlmutter). This effect can similarly be seen with Isolate-3-8M and products-14M. Even though products-14M has more nodes than Isolate-3-8M, the latter is denser. This explains why Plexus is slower with Isolate-3-8M at 16 GPUs where the computation cost is significant, but for products-14M, which is more communication dominated, eventually Plexus takes longer beyond 64 GPUs. We also show results for europe_osm on 1024 GPUs and ogbn-papers100M on 2048 GPUs of Perlmutter. We observe that the scaling with ogbn-papers100M starts to slow down at 2048 GPUs, at which point the computation cost is marginal. This is, to the best of our knowledge, the largest number of GPUs that have been used for parallel full-graph GNN training to date.

On Frontier (right plot), we notice generally better trends with all datasets when compared to those on Perlmutter. This is because the SpMM times on AMD GPUs were an order of magnitude higher than on NVIDIA GPUs, allowing Plexus to scale better. The trends observed on Perlmutter for Reddit and ogbn-products do not hold here, and we do not observe a growing gap between the two datasets. Similarly, Plexus is consistently slower with Isolate-3-8M than with products-14M since the former has a higher number of edges. We also observe that Plexus demonstrates poorer scaling with europe_osm, a sparser graph than both products-14M and Isolate-3-8M. Finally, we observe that Plexus demonstrates impressive scaling for ogbn-papers100M, which is the largest graph dataset we ran with, on up to 2048 GCDs.

8 Conclusion

GNN training has often relied on approximations such as mini-batch sampling due to the high memory requirements of large graphs.

In the absence of efficient and scalable full-graph alternatives, this approach has become the default in many modern frameworks. In this work, we present Plexus, a three-dimensional parallel framework for full-graph GNN training that adapts Agarwal et al.’s 3D parallel matrix multiplication algorithm [3] to scale training with billion-edge graphs to thousands of GPUs. Plexus includes a performance model that selects an optimal 3D configuration based on communication and computation costs, and incorporates several optimizations to further enhance performance. These include a double permutation scheme to reduce load imbalance, and blocked aggregation to minimize variability. Plexus also offers an easy-to-use API, eliminating the need for a graph partitioner and featuring a parallel data loading utility that reduces CPU memory usage. Overall, this work marks a significant step forward in making full-graph GNN training, a notoriously challenging problem to scale, both practical and efficient.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation (NSF) under Grant No. 2047120. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research also used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 using NERSC award NERSC DDR-ERCAP0034262.

References

- [1] 2021. OSLO: Open Source for Large-scale Optimization. <https://github.com/EleutherAI/oslo>.
- [2] Seher Acer, Oğuz Selvitopi, and Cevdet Aykanat. 2016. Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems. *Parallel Comput.* 59, C (Nov. 2016), 71–96. doi:10.1016/j.parco.2016.10.001
- [3] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of*

- Research and Development* 39, 5 (1995), 575–582. doi:10.1147/rd.395.0575
- [4] Xin Ai, Hao Yuan, Zeyu Ling, Qiange Wang, Yanfeng Zhang, Zhenbo Fu, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. NeutronTP: Load-Balanced Distributed Full-Graph GNN Training with Tensor Parallelism. arXiv:2412.20379 [cs.DC] <https://arxiv.org/abs/2412.20379>
- [5] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyriakis, and Aydin Buluc. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic Acids Research* 46, 6 (01 2018), e33–e33. arXiv:<https://academic.oup.com/nar/article-pdf/46/6/e33/24525991/gkx1313.pdf> doi:10.1093/nar/gkx1313
- [6] Muhammed Fatih Balin, Kaan Sancak, and Ümit V. Çatalyürek. 2021. MG-GCN: Scalable Multi-GPU GCN Training Framework. arXiv:2110.08688 [cs.LG] <https://arxiv.org/abs/2110.08688>
- [7] 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* (Online Event, United Kingdom) (*EuroSys '21*). Association for Computing Machinery, New York, NY, USA, 130–144. doi:10.1145/3447786.3456233
- [8] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. arXiv:1801.10247 [cs.LG] <https://arxiv.org/abs/1801.10247>
- [9] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. arXiv:1710.10568 [stat.ML] <https://arxiv.org/abs/1710.10568>
- [10] Shenggan Cheng, Ziming Liu, Jiangsu Du, and Yang You. 2023. ATP: Adaptive Tensor Parallelism for Foundation Models. *arXiv preprint arXiv:2301.08658* (2023).
- [11] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (KDD '19). ACM. doi:10.1145/3292500.3330925
- [12] Siddhartha Shankar Das, S M Ferdous, Mahantesh M Halappanavar, Edoardo Serra, and Alex Pothen. 2024. AGS-GNN: Attribute-guided Sampling for Graph Neural Networks. arXiv:2405.15218 [cs.LG] <https://arxiv.org/abs/2405.15218>
- [13] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. arXiv:1903.02428 [cs.LG] <https://arxiv.org/abs/1903.02428>
- [14] Geofabrik GmbH. 2010. DIMACS10/europe_osm. SuiteSparse Matrix Collection. https://sparse.tamu.edu/DIMACS10/europe_osm
- [15] William L. Hamilton, Rex Ying, and Jure Leskovec. 2018. Inductive Representation Learning on Large Graphs. arXiv:1706.02216 [cs.SI] <https://arxiv.org/abs/1706.02216>
- [16] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2021. Open Graph Benchmark: Datasets for Machine Learning on Graphs. arXiv:2005.00687 [cs.LG] <https://arxiv.org/abs/2005.00687>
- [17] 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_files/paper/2020/file/91fc23cecc664ebb0cf4257e1ba9c51-Paper.pdf
- [18] George Karypis and Vipin Kumar. 1999. Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20(1), 359–392. *Siam Journal on Scientific Computing* 20 (01 1999). doi:10.1137/S1064827595287997
- [19] 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>
- [20] Süreyya Emre Kurt, Jinghua Yan, Aravind Sukumaran-Rajam, Prashant Pandey, and P. Sadayappan. 2023. Communication Optimization for Distributed Execution of Graph Neural Networks. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 512–523. doi:10.1109/IPDPS54959.2023.00058
- [21] Yann LeCun, Bernhard Boser, John Denker, Donnie Henderson, R. Howard, Wayne Hubbard, and Lawrence Jackel. 1990. Handwritten Digit Recognition with a Back-Propagation Network. In *Advances in Neural Information Processing Systems*, D. Touretzky (Ed.), Vol. 2. Morgan-Kaufmann, 396–404. <https://proceedings.neurips.cc/paper/1989/file/53c3bce6643be4f209556518c2fc54-Paper.pdf>
- [22] Qimai Li, Zhichao Han, and Xiao-Ming Wu. 2018. Deeper insights into graph convolutional networks for semi-supervised learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence* (New Orleans, Louisiana, USA) (AAAI'18/IAAI'18/EAII'18). AAAI Press, Article 433, 8 pages.
- [23] Shengwei Li, Zhiqian Lai, Yanqi Hao, Weijie Liu, Keshi Ge, Xiaoge Deng, Dongsheng Li, and Kai Lu. 2023. Automated Tensor Model Parallelism with Overlapped Communication for Efficient Foundation Model Training. *arXiv preprint arXiv:2305.16121* (2023).
- [24] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. 2023. Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training. In *Proceedings of the 52nd International Conference on Parallel Processing* (Salt Lake City, UT, USA) (ICPP '23). Association for Computing Machinery, New York, NY, USA, 766–775. doi:10.1145/3605573.3605613
- [25] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, and Dongrui Fan. 2021. Sampling methods for efficient training of graph convolutional networks: A survey. arXiv:2103.05872 [cs.LG] <https://arxiv.org/abs/2103.05872>
- [26] Ujjaini Mukhopadhyay, Alok Tripathy, Oguz Selvitopi, Katherine Yelick, and Aydin Buluc. 2024. Sparsity-Aware Communication for Distributed Graph Neural Network Training. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) (ICPP '24). Association for Computing Machinery, New York, NY, USA, 117–126. doi:10.1145/3673038.3673152
- [27] Jianmo Ni, Jiacheng Li, and Julian McAuley. 2019. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 188–197. doi:10.18653/v1/D19-1018
- [28] NVIDIA. [n. d.]. NVIDIA Nsight Compute. <https://developer.nvidia.com/nsight-compute>.
- [29] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
- [30] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jianlong Cao. 2022. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proc. VLDB Endow.* 15, 9 (May 2022), 1937–1950. doi:10.14778/3538598.3538614
- [31] Rolf Rabenseifner. 2004. Optimization of Collective Reduction Operations. In *Computational Science - ICCS 2004*, Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–9.
- [32] Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydin Buluc. 2021. Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication. In *Proceedings of the 35th ACM International Conference on Supercomputing* (Virtual Event, USA) (ICS '21). Association for Computing Machinery, New York, NY, USA, 431–442. doi:10.1145/3447818.3461472
- [33] Shaohuai Shi, Pengfei Xu, and Xiaowen Chu. 2017. Supervised Learning Based Algorithm Selection for Deep Neural Networks. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. 344–351. doi:10.1109/ICPADS.2017.00053
- [34] Siddharth Singh and Abhinav Bhatele. 2022. AxoNN: An asynchronous, message-driven parallel framework for extreme-scale deep learning. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS '22)*. IEEE Computer Society.
- [35] Siddharth Singh, Prajwal Singhania, Aditya Ranjan, John Kirchenbauer, Jonas Geiping, Yuxin Wen, Neel Jain, Abhimanyu Hans, Manli Shu, Aditya Tomar, Tom Goldstein, and Abhinav Bhatele. 2024. Democratizing AI: Open-source Scalable LLM Training on GPU-based Supercomputers. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '24)*.
- [36] Jaeyong Song, Hongsun Jang, Jaewon Jung, Youngsok Kim, and Jinho Lee. 2024. GraNNDis: Efficient Unified Distributed Training Framework for Deep GNNs on Large Clusters. arXiv:2311.06837 [cs.LG] <https://arxiv.org/abs/2311.06837>
- [37] Rajeev Thakur and William D. Gropp. 2003. Improving the Performance of Collective Operations in MPICh. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Jack Dongarra, Domenico Laforenza, and Salvatore Orlando (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–267.
- [38] Alok Tripathy, Katherine Yelick, and Aydin Buluc. 2020. Reducing communication in graph neural network training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) (SC '20). IEEE Press, Article 70, 17 pages.
- [39] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. arXiv:1710.10903 [stat.ML] <https://arxiv.org/abs/1710.10903>
- [40] Borui Wan, Juntao Zhao, and Chuan Wu. 2023. Adaptive Message Quantization and Parallelization for Distributed Full-graph GNN Training. arXiv:2306.01381 [cs.LG] <https://arxiv.org/abs/2306.01381>
- [41] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. 2022. BNS-GCN: Efficient Full-Graph Training of Graph Convolutional Networks with Partition-Parallelism and Random Boundary Node Sampling. arXiv:2203.10983 [cs.LG] <https://arxiv.org/abs/2203.10983>
- [42] Cheng Wan, Youjie Li, Cameron R. Wolfe, Anastasios Kyriadidis, Nam Sung Kim, and Yingyan Lin. 2022. PipeGCN: Efficient Full-Graph Training of Graph Convolutional Networks with Pipelined Feature Communication. arXiv:2203.10428 [cs.LG] <https://arxiv.org/abs/2203.10428>

- [43] Minji 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. 2020. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. arXiv:1909.01315 [cs.LG] <https://arxiv.org/abs/1909.01315>
- [44] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. NeutronStar: Distributed GNN Training with Hybrid Dependency Management. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) (SIGMOD '22). Association for Computing Machinery, New York, NY, USA, 1301–1315. doi:10.1145/3514221.3526134
- [45] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. 2023. MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation* (OSDI'23). USENIX Association, Boston, MA, 779–795. <https://www.usenix.org/conference/osdi23/presentation/wang-yuke>
- [46] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks? arXiv:1810.00826 [cs.LG] <https://arxiv.org/abs/1810.00826>
- [47] Carl Yang, Aydin Buluc, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. doi:10.48550/ARXIV.1803.08601
- [48] Taraneh Younesian, Daniel Daza, Emile van Krieken, Thivyan Thanapalasingam, and Peter Bloem. 2024. GRAPES: Learning to Sample Graphs for Scalable Graph Neural Networks. arXiv:2310.03399 [cs.LG] <https://arxiv.org/abs/2310.03399>
- [49] Hao Yuan, Yajiong Liu, Yanfeng Zhang, Xin Ai, Qiange Wang, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. Comprehensive Evaluation of GNN Training Systems: A Data Management Perspective. arXiv:2311.13279 [cs.LG] <https://arxiv.org/abs/2311.13279>
- [50] Shuai Zhang, Zite Jiang, and Haihang You. 2024. CDFGNN: a Systematic Design of Cache-based Distributed Full-Batch Graph Neural Network Training with Communication Reduction. arXiv:2408.00232 [cs.DC] <https://arxiv.org/abs/2408.00232>
- [51] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. *CoRR* abs/2201.12023 (2022). arXiv:2201.12023
- [52] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-Dependent Importance Sampling for Training Deep and Large Graph Convolutional Networks. arXiv:1911.07323 [cs.LG] <https://arxiv.org/abs/1911.07323>