ABSTRACT

Title of Thesis: Scaling Parallel Full-graph GNN Training to Thousands of GPUs

Aditya Kishore Ranjan

Thesis Directed by: Associate Professor Abhinav Bhatele
Department of Computer Science

Graph neural networks have emerged as a potent class of neural networks capable of leveraging 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 using GNNs on them requires techniques such as mini-batch sampling to scale. However, this can lead to reduced accuracy in some cases, and sampling and data transfer from the CPU to the GPU can also slow down training. On the other hand, distributed full-graph training suffers from high communication overhead and load imbalance due to the irregular structure of graphs.

In this thesis, we propose Plexus, a three-dimensional (3D) parallel approach for full-graph training that tackles these issues and scales to billion-edge graphs. Additionally, we introduce performance optimizations such as a permutation scheme for load balancing, and a performance model to predict the optimal 3D configuration. Plexus is evaluated on several graph datasets and scaling results are shown for up to 2048 A100 GPUs on Perlmutter, which is 33% of the

supercomputer, and 1024 MI250X GPUs on Frontier. Plexus achieves unprecedented speedups of 2.3x-12.5x over existing methods and a reduction in the time to solution by 5.2-8.7x on Perlmutter and 7-54.2x on Frontier.

SCALING PARALLEL FULL-GRAPH GNN TRAINING TO THOUSANDS OF GPUS

by

Aditya Kishore Ranjan

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2025

Advisory Committee:
Associate Professor Abhinav Bhatele, Chair
Assistant Professor Laxman Dhulipala
Professor Ramani Duraiswami

श्री श्री दुर्गा सहाय
श्री श्री दुर्गा सहाय
श्री श्री दुर्गा सहाय
श्री श्री दुर्गा सहाय
श्री श्री दुर्गा सहाय
श्री श्री दुर्गा सहाय
श्री श्री दुर्गा सहाय
श्री श्री दुर्गा सहाय
श्री श्री दुर्गा सहाय
श्री श्री दुर्गा सहाय
श्री श्री दुर्गा सहाय

Acknowledgments

I would first like to express my gratitude to Professor Abhinav Bhatele for giving me the opportunity to work with him over the past four years during my BS/MS studies. His guidance on exciting and cutting-edge research projects has had a lasting impact on me, solidifying my passion for computer science and problem solving. I am especially thankful to Siddharth, who has been a valuable mentor throughout the development of this thesis. His thoughtful insights and meticulous approach to research have taught me how to conduct thorough and precise work. I am also grateful to all of my labmates and friends for their constant support and for fostering an encouraging and collaborative environment. I would like to thank the professors I've had the pleasure of learning from, many of whom went above and beyond to ensure their students received the best possible education and opportunities for growth.

I owe my deepest thanks to my family—Mom, Dad, and Arya—for their unwavering love, support, and encouragement. Their belief in me has always motivated me to stay curious and work hard. I am also thankful to my grandparents—Baba, Dadi, Nana, and Nani—whose early influence helped instill in me a deep respect for learning and education.

Lastly, I am grateful for the resources provided by many high-performance computing centers. In particular, I acknowledge the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory and the National Energy Research Scientific Computing Center, which provided access to the Frontier and Perlmutter supercomputers, respectively.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

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) [1] is the most popular and widely adopted, and serves as the foundation for numerous extensions, including the Graph Attention Network (GAT) [2] and the Graph Isomorphism Network (GIN) [3]. Unlike traditional convolutional neural networks [4], 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. [1] recognized 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. Consequently, due to a lack of efficient and scalable full-graph alternatives , most modern frameworks such as PyTorch Geometric [5] and DGL [6] now adopt this approach as the default.

In a single GCN layer, nodes in the mini-batch first collect information from their immediate neighbors. By aggregating the feature embeddings from nodes' neighborhoods 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 aggregate features from its K-hop neighborhood. However, even for small values of K, this can quickly result in a phenomenon known as neighborhood explosion, covering large portions of the graph and undermining the efficiency of mini-batch training [7]. To mitigate this issue, sampling algorithms like GraphSAGE [8] and FastGCN [9] are typically applied alongside mini-batch training to reduce the number of neighbors considered, thereby lowering memory consumption.

While sampling is widely used for training GNNs on large graphs, it comes with inherent limitations. Most notably, sampling introduces approximations that can lead to accuracy degradation [10]. Further, CPU-GPU data transfers in sampling often dominate training time and add unnecessary complexity [11]. In many cases, full-graph training can achieve competitive performance without these trade-offs as shown by Jia et al.'s ROC framework [10]. 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, this work focuses on the full-graph training paradigm, avoiding any approximations.

Graphs are typically represented as adjacency matrices with a non-zero entry for each edge, stored using sparse formats such as COO and CSR. However, these are sparsely and unevenly distributed across the matrix. Among the 6 graphs we evaluate our work on, the sparsity levels (fraction of zeros in the adjacency matrix) range 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 two key issues. (1) Varying sparsity patterns in the adjacency matrix can lead to significant load imbalance in computation, which can ripple through an epoch and affect communication times as well. Additionally, the aggregation phase involves Sparse Matrix-Matrix

Multiplication (SpMM), which dominates the computational time and suffers from poor performance on both CPUs and GPUs due to irregular memory access patterns and low data reuse. (2) High memory requirements necessitate distributing the graph, its features, and the associated computation across multiple GPUs. This incurs high communication overhead due to the need to synchronize large intermediate activations and gradients between GPUs. Consequently, GNN training quickly becomes communication-bound, making it difficult to scale efficiently to a large number of GPUs. These challenges severely limit the scalability of distributed GNN training for large graphs – a core focus of this thesis.

To address these challenges, we propose Plexus, a framework that implements a three-dimensional (3D) parallel algorithm that parallelizes all matrix multiplications involved in training and enables scaling to large graphs. Plexus is designed to support a wide range of graph types—including undirected, directed, weighted, and unweighted graphs. Our approach draws inspiration from Agarwal et al.'s 3D parallel matrix multiplication algorithm [12], which has been used in several distributed deep learning frameworks, including Colossal-AI's unified deep learning system [13], AxoNN [14], Eleuther AI's framework OSLO [15], etc. On top of our initial implementation, we introduce several optimizations to improve performance. One such optimization is a permutation scheme which ensures a near-perfect even distribution of nonzeros across matrices and helps alleviate load imbalance. We also propose a performance model that helps users select an optimal configuration from the 3D virtual GPU grid. This eliminates the need for exhaustive testing across configurations while ensuring robust performance outcomes.

The key contributions are summarized as follows:

- Plexus: An open-source 3D parallel framework for distributed GNN training that scales to

massive graphs and GPU clusters.

- A performance model to identify the optimal configuration within the 3D virtual GPU grid.
- Performance optimizations, including a permutation scheme to mitigate load imbalance and blocked aggregation to reduce variability.
- Unprecedented scaling to 2048 GCDs/GPUs on Frontier and Perlmutter —- the largest-scale full-graph GNN training reported to date.
- Significant speedups, achieving 2.3-12.5x faster training than state-of-the-art frameworks, and cutting time-to-solution by 5.2-8.7x on Perlmutter and 7-54.2x on Frontier.

Chapter 2: Background and Related Work

In this chapter, we will provide an overview of GNNs and their training paradigms, as well as the challenges associated with distributed full-graph GNN training. We will also discuss existing 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 issues such as exploding/vanishing gradients and numerical instabilities [1].

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 $\mathbf{H}^{Li} \in \mathbb{R}^{N \times D^{Li}}$.

$$\mathbf{H}^{Li} = SpMM(\mathbf{A}, \mathbf{F}^{Li}). \tag{2.1}$$

   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 $\mathbf{Q}^{Li} \in \mathbb{R}^{N \times D^{Li+1}}$.

$$\mathbf{Q}^{Li} = SGEMM(\mathbf{H}^{Li}, \mathbf{W}^{Li}). \tag{2.2}$$

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

$$\mathbf{F}^{Li+1} = \sigma(\mathbf{Q}^{Li}). \tag{2.3}$$

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

1. Compute the gradient of the loss $\mathcal{L}$ with respect to $\mathbf{Q}^{Li}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Q}^{Li}} = \frac{\partial \mathcal{L}}{\partial \mathbf{F}^{Li+1}} \odot \sigma'(\mathbf{Q}^{Li}), \tag{2.4}$$

   where $\odot$ denotes element-wise multiplication.

2. Compute the gradient of the loss with respect to the weight matrix $\mathbf{W}^{Li}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{Li}} = SGEMM((\mathbf{H}^{Li})^\top, \frac{\partial \mathcal{L}}{\partial \mathbf{Q}^{Li}}). \tag{2.5}$$

3. Compute the gradient of the loss with respect to $\mathbf{H}^{Li}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{H}^{Li}} = SGEMM(\frac{\partial \mathcal{L}}{\partial \mathbf{Q}^{Li}}, (\mathbf{W}^{Li})^\top). \tag{2.6}$$

4. Compute the gradient of the loss with respect to $\mathbf{F}^{Li}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{F}^{Li}} = SpMM(\mathbf{A}^\top, \frac{\partial \mathcal{L}}{\partial \mathbf{H}^{Li}}). \tag{2.7}$$

   The gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{F}^{L0}}$ at the first layer is then used to update the input features and learn meaningful node embeddings.

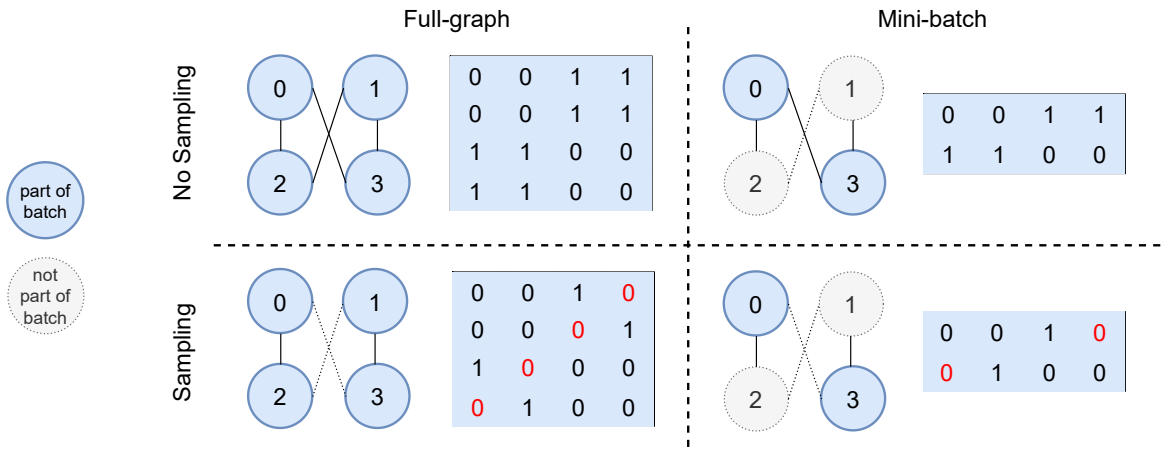

Figure 2.1: Different paradigms of GNN training that can be combined together, shown in four quadrants. Each quadrant shows a sample graph and its adjacency matrix. Blue nodes are part of the batch and grey nodes are not. Solid lines indicate the edges considered for aggregation and dashed line edges are not considered. Red values in the adjacency matrix indicate that an entry has been modified.

2.2 Paradigms of GNN Training

There are a four main paradigms of GNN training that can be used in practice. The simplest way of doing GNN training is the *Full-graph* paradigm, where the entire graph is used in each iteration. All nodes' features are updated every iteration and the entire graph is loaded into GPU memory. This is shown in the upper-left quadrant of Figure 2.1. This is the most accurate paradigm since it avoids approximations entirely, but is also the most memory-intensive. Due to a lack of efficient full-graph training frameworks, ML practitioners often utilize a technique called *Mini-batch* training, which is the second paradigm. This method only selects a small subset of nodes to update in each iteration. As can be seen in the upper-right quadrant of Figure 2.1, only Nodes 0 and 3's features will be updated in the iteration after aggregating features from their full neighborhood. Unfortunately, this leads to neighborhood explosion with multiple layers, where the neighborhood size grows exponentially and makes this impractical for large-scale

graphs [7]. To mitigate this, another technique called *Mini-batch sampling* is employed. At each layer, nodes only aggregate features from a subset of their neighbors. This is the most typical training paradigm for GNNs as of today since it reduces the memory consumption drastically. As shown in the bottom-right quadrant, only some edges are used in aggregation. Finally, the last paradigm is *Full-graph sampling*, but this isn't as common since the entire graph is the batch and brought into memory. Shown in the bottom-left quadrant, we see that all nodes are part of the batch, but only some edges are sampled.

As GNNs are an emerging field, there is no standard sampling algorithm that is agreed upon by the community. However, there are some which have proven to be fairly successful with adoption. GraphSAGE [8] samples a fixed number of neighbors for each node. FastGCN [9] samples a fixed number of neighbors for each layer. LADIES extends FastGCN's method by accounting for inter-layer dependencies more effectively. Cluster-GCN [16] uses a clustering algorithm to limits the sampling to dense subgraphs. There is still ongoing and more recent research on how to improve sampling algorithms further. GRAPES [17] proposes a solution to adaptively change the sampling probabilities using a second GNN. AGS-GNN [18] focuses on improving sampling for homophilic and heterophilic graphs simultaneously. However, there is always a trade-off between accuracy and efficiency when using sampling, which inevitably introduces bias and variance into the training process [19]. The largest graph examined in these studies contains approximately 2.5 million nodes, which is small compared to the scale of many real-world datasets including the ones that we evaluate our work on (Table 6.1). Especially for larger graphs, the loss of information from sampling may be more detrimental for graphs with different structural properties (e.g. large diameter), as well as tasks that have long-range dependencies. Furthermore, the decision of which sampling algorithm to choose and what hyperparameters to set for each is a non-obvious

one. Sampling is also often bottlenecked by data transfer of the features from the CPU to the GPU. The outlook on sampling is therefore largely inconclusive. Given these considerations, we concentrate on the full-graph paradigm in this work and focus our discussion on distributed full-graph training algorithms relevant to our approach.

2.3 Prior Work on Distributed Full-graph GNN Training

One of the very first frameworks for distributed full-graph GNN training is ROC [10], partitions a graph using an online linear regression based partitioner that learns from previously processed graphs. ROC requires the data to fit in CPU memory and introduces an algorithm to determine how to balance the cost of CPU-GPU transfer with GPU memory utilization.

Following ROC, CAGNET [20] introduced a series of parallel algorithms used for full-graph training that can be considered tensor-parallel approaches. They provide 1D, 1.5D, 2D, and 3D algorithms which parallelize the SpMM. The 1D algorithm divides matrices along the row dimension and broadcasts features in steps to perform local computations. The 1.5D algorithm is similar, but introduces a replication factor that trades memory usage for communication costs. The 2D algorithm divides the matrix across both dimensions and is based on the SUMMA algorithm [21]. The 3D algorithm executes the 2D algorithm independently at each layer with a reduction at the end. While the 2D and 3D algorithms provide asymptotic reductions in communication, these don't scale as well as the 1D and 1.5D algorithms due to higher constants.

In a newer paper [22], the same authors propose a sparsity-aware version of their 1D and 1.5D algorithms, which outperforms the original algorithms. The key insight in this work is to only communicate the features that are needed based on the sparsity pattern of the adjacency

matrix.

MG-GCN [23] adds optimizations on top of CAGNET like overlapping the broadcast with the local computation. RDM [24] implements an algorithm on top of CAGNET, but replicates one of the matrices across GPUs, making the training process nearly communication-free.

There are other frameworks which are full-graph, but make some approximations for increased scalability. BNS-GCN [25] first partitions the graph using METIS, and then samples boundary nodes, only storing and communicating features for the sampled ones. They provide an analysis and some experiments to confirm that their sampling method converges. However, it isn't clear if this would hold for other datasets and more rigorous hyperparameter tuning would be needed. The same authors develop PipeGCN [26] which pipelines the communication and computation, but results in stale boundary features/gradients. While they attempt to show that this doesn't impact the accuracy, different graphs with varying properties may be more sensitive to this staleness.

Apart from these, some works focus on reducing the communication costs in other ways. DGCL [27] provides an algorithm to minimize the communication time using the graph characteristics and cluster topology. NeutronTP [28] adapts tensor parallelism for GNN training and only shards the features since distributing the graph can lead to load imbalance.

Other works are included in Table 2.1, where we see the largest dataset sizes and number of GPUs used in each work. While there has been significant progress in making distributed full-graph training more efficient, very few works show scaling behavior across a wide range of GPU counts. Only 4 of the 15 works in the table use more than 16 GPUs and these are from 2 author groups. A lot of full-graph work has been done with approximations made in the form of sampling, staleness, etc. Most works that focus on algorithms for distributed SpMM focus on

variants of the block-row 1D algorithm, and there is no scalable 3D algorithm in practice. This is despite Agarwal's 3D algorithm having $P^{1/6}$ less communication than the 2D algorithms. This is precisely the gap we aim to fill with Plexus, providing a framework that makes no approximations and implements a 3D approach that can scale to large graphs and high GPU counts efficiently.

Table 2.1: Summary of state of the art in distributed full-graph GNN training. The number of nodes and edges for the graph datasets, and number of GPUs are the largest values reported in each paper.

Name	Year	# Nodes	# Edges	# GPUs
AdaQP [29]	2023	2.5M	114M	8
RDM [24]	2023	3M	117M	8
MG-GCN [23]	2022	111M	1.6B	8
Sancus [30]	2022	111M	1.6B	8
MGG [31]	2023	111M	1.6B	8
DGCL [27]	2021	3M	117M	16
ROC [10]	2020	9.5M	232M	16
NeutronStar [32]	2022	42M	1.5B	16
GraNNDis [33]	2024	111M	1.6B	16
NeutronTP [28]	2024	244M	1.7B	16
CDFGNN [34]	2024	111M	1.8B	16
PipeGCN [26]	2022	111M	1.6B	32
CAGNET [20]	2020	14.2M	231M	125
BNS-GCN [25]	2022	111M	1.6B	192
SA+GVB [22]	2024	111M	1.6B	256
Plexus (this work)	2025	111M	1.6B	2048

Chapter 3: Methodology

We now describe how we parallelize the sequential formulation of a GCN layer described in Section 2.1, and how we adapt Agarwal's 3D parallel matrix multiply algorithm for GNN training in Plexus.

3.1 Parallelizing a Single GCN Layer

As seen in Section 2.3, tensor parallelism is a common strategy for parallelizing neural network training. Tensor parallelism distributes matrices to multiple processes or GPUs. It implements parallel matrix multiplication algorithms that compute on these distributed matrices, and then perform collective operations to get the results on the right sets of processes/GPUs. In this work, we take inspiration from Agarwal et al.'s three-dimensional parallel matrix multiplication algorithm [12]. Similar three-dimensional approaches have been used in several parallel training frameworks, including Colossal-AI's unified deep learning system [13], Eleuther AI's framework OSLO [15], AxoNN [14], Megatron-Deepspeed [35] CAGNET [20], etc. Below, we describe how we adapt this 3D matrix multiplication approach to parallelize GNN training and Sparse Matrix-Matrix Multiplications (SpMM).

Given a number of GPUs, G, in a job allocation, we first arrange the GPUs into a 3D virtual grid with dimensions X, Y, and Z. We refer to the number of GPUs along each dimension as

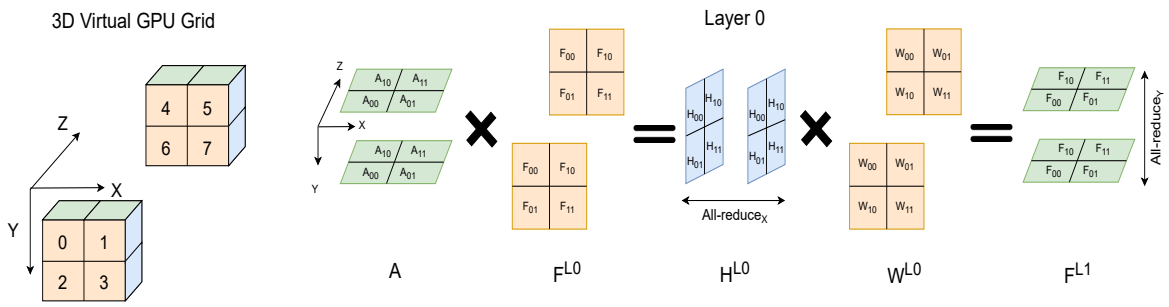

Figure 3.1: An overview of the 3D tensor parallel algorithm, showing how 8 GPUs are arranged in a 3D grid (X=Y=Z=2) and how the matrices in a layer are distributed across different planes. Each plane of the grid is colored differently.

G_x, G_y, and G_z respectively, such that $G = G_x \times G_y \times G_z$. Each GPU will create process groups that allow it to communicate with its neighbors in the three dimensions of the grid. The matrices in a layer are then distributed across this grid. We now describe how this is done for the first layer of the GCN, and this can be applied similarly to the other layers.

The sparse adjacency matrix A is sharded across the ZX-plane and replicated across the Y-parallel process group. The input features matrix F^{L0} is sharded across the XY-plane and then further sharded across the Z-parallel process group. The reason that it 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. The weights matrix W^{L0} is sharded across the YX-plane and also further sharded across the Z-parallel process group due to the additional memory requirements of the gradients and optimizer states. The distribution of these matrices across the 3D grid are shown in Figure 3.1, and the shapes of the matrix shards are shown in Figure 3.2.

Psuedocode for the complete algorithm for the first layer is shown in Algorithm 1. Before describing the algorithm, note that when we refer to any matrix, it is a shard of that matrix on a given GPU. First, we describe the steps for the aggregation (line 2-4). The input features matrix

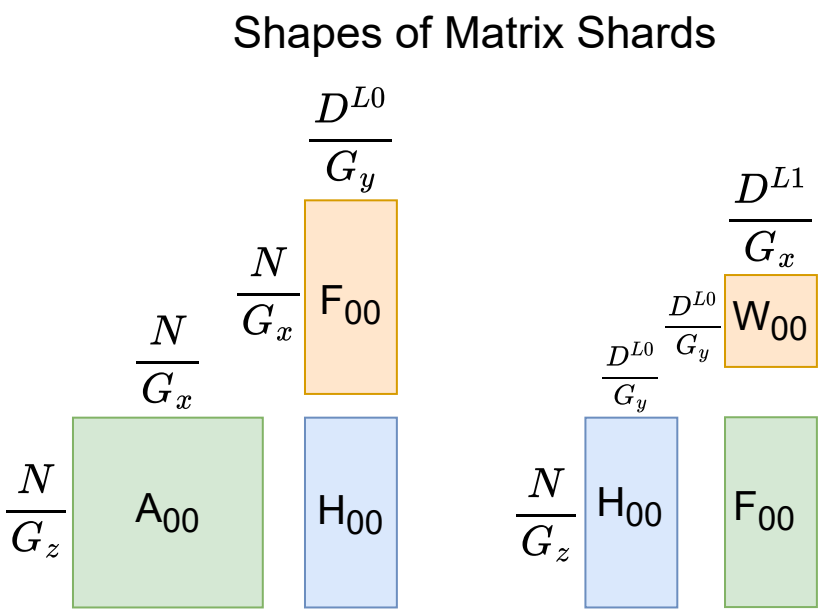

Figure 3.2: Shapes of the matrix shards in the first layer, showing how the two key matrix multiplications are performed on a GPU.

shard F has to be all-gathered across the Z-parallel process group since it additionally sharded across this dimension of the grid. The adjacency matrix shard A is then multiplied by this to get the aggregation output H. Since this results in a partial output, an all-reduce is performed on this across the X-parallel process group. Next, we describe the steps for the combination (lines 5-7). 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. This again results in a partial output Q, so it 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 8-9). These series of matrix multiplications and the all-reduce steps are also demonstrated visually in Figure 3.1. Note that the sharding across the third process group for F and W as well as the all-gather steps aren't shown in this figure for the sake of simplicity. The backward pass for the first layer is also shown in Algorithm 1, where we

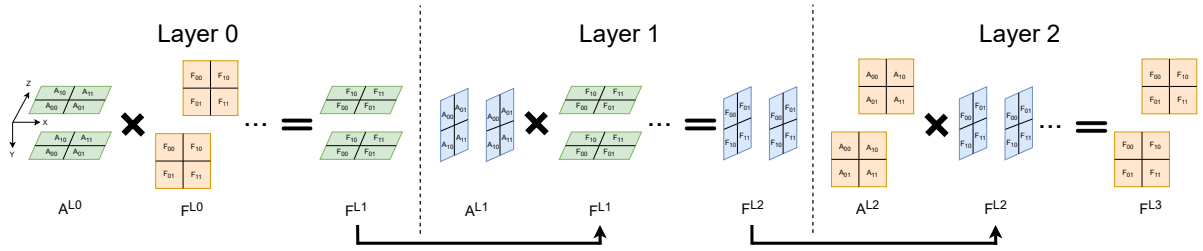

Figure 3.3: A visual representation of how the 3D algorithm is applied to multiple layers of the GCN, connecting the output of one layer to the input of the next using the unique shards of the adjacency matrix.

receive the gradient of the loss with respect to the output Q from the next layer (line 11). The first step is to calculate the gradient of the loss with respect to W (lines 12-13). This is done by first multiplying H^T by $\frac{\partial \mathcal{L}}{\partial Q}$. The output of this then needs to be reduce-scattered across the Z-parallel process group since the weights are sharded across this dimension of the grid. Next, we calculate by the gradient of the loss with respect to H (lines 14-16). First, W has to be all-gathered across the Z-parallel process group. After this, $\frac{\partial \mathcal{L}}{\partial Q}$ is multiplied by W^T and then the result is all-reduced across the X-parallel process group so that the partial outputs are summed up. The final step is to calculate the gradient of the loss with respect to F (lines 17-18). This is done by performing a SpMM with A^T and $\frac{\partial \mathcal{L}}{\partial H}$ and then doing a reduce-scatter on this across the Z-parallel process group since the input features are sharded across this dimension of the grid. After these steps, we return both $\frac{\partial \mathcal{L}}{\partial F}$ and $\frac{\partial \mathcal{L}}{\partial W}$ (line 19).

3.2 Parallelizing the Entire GNN

So far, we have only described the parallelization of the first layer of the GCN due to a subtle but important detail that needs to be addressed. As can be seen in Figure 3.1, the output of the first layer F^{L1} is sharded across the ZX-plane. However, this will also be the input to

Algorithm 1 Tensor Parallel Forward and Backward Pass

function FORWARD($\mathbf{A}, \mathbf{F}, \mathbf{W}$)
  All-gather $\mathbf{F}$ across $\mathbf{Z}$
  $\mathbf{H} = \mathbf{SpMM}(\mathbf{A}, \mathbf{F})$
  All-reduce $\mathbf{H}$ across $\mathbf{X}$

  All-gather $\mathbf{W}$ across $\mathbf{Z}$
  $\mathbf{Q} = \mathbf{SGEMM}(\mathbf{H}, \mathbf{W})$
  All-reduce $\mathbf{Q}$ across $\mathbf{Y}$

  $\mathbf{F} = \boldsymbol{\sigma}(\mathbf{Q})$
  Return $\mathbf{F}$
end function

function BACKWARD($\frac{\partial \mathcal{L}}{\partial Q}$)
  $\frac{\partial \mathcal{L}}{\partial W} = \mathbf{SGEMM}(H^T, \frac{\partial \mathcal{L}}{\partial Q})$
  Reduce-scatter $\frac{\partial \mathcal{L}}{\partial W}$ across $\mathbf{Z}$

  All-gather $\mathbf{W}$ across $\mathbf{Z}$
  $\frac{\partial \mathcal{L}}{\partial H} = \mathbf{SGEMM}(\frac{\partial \mathcal{L}}{\partial Q}, W^T)$
  All-reduce $\frac{\partial \mathcal{L}}{\partial H}$ across $\mathbf{X}$

  $\frac{\partial \mathcal{L}}{\partial F} = \mathbf{SpMM}(A^T, \frac{\partial \mathcal{L}}{\partial H})$
  Reduce-scatter $\frac{\partial \mathcal{L}}{\partial F}$ across $\mathbf{Z}$

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

the next layer, which is a problem since the adjacency matrix A 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, especially considering that the 3D grid is not necessarily a cube. 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 the local computations. This scheme is shown in Figure 3.3, 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 doesn't result in needing more than 3 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 layers after the first, the features matrix F will only be sharded across two dimensions of the grid since it doesn't have optimizer states like the input features. This means that the first all-gather in the forward pass (line 2) won't take place. In correspondence with this, the last reduce-scatter (line 18) 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 the entire model and the core idea remains the same.

Chapter 4: Performance Model

We develop a performance model to help us in selecting near-optimal 3D configurations for Plexus by modeling both the SpMM computation and communication times.

4.1 Modeling Computation

Plexus shards the matrices in a way such that the local computation should be the same in theory for all configurations, assuming an even distribution of nonzeros throughout the adjacency matrix. We can see this with a simple derivation for the aggregation SpMM. Let's take the example of the first layer. The aggregation output H will be sharded across the ZY-plane and has size $\frac{N}{G_z} \times \frac{D^{L0}}{G_y}$. Each element of this matrix will be the result of a dot product between a row of the sparse adjacency matrix A and a column of the dense adjacency matrix F. The number of floating point operations needed per element of H is equal to $2 \cdot NNZ$, the number of nonzeros in the original matrix, divided by $N \cdot G_x$. As can be seen below, the total FLOPs for the SpMM results in a term that is constant across all configurations for G GPUs.

$$\frac{N}{G_z} \times \frac{D^{L0}}{G_y}$$

Number of floating point operations per element is:

$$O\left(\frac{2 \cdot \text{NNZ}}{N \cdot G_x}\right)$$

Hence, the total number of floating point operations to calculate the aggregation output H is:

$$O\left(\frac{2 \cdot \text{NNZ} \cdot D^{L0}}{G}\right), \quad \text{where } G = G_x G_y G_z$$

However, this is not the case, and we do observe an increase in SpMM times for certain configurations. We hypothesize that the SpMMs perform more efficiently for configurations where the sparse matrix is taller and skinnier and the dense matrix is shorter and fatter. This is also documented in the literature, with many works specifically focusing on the optimization of tall-skinny dense matrix SpMM. Yang et al. [36] provide a row-splitting algorithm where each row of the sparse matrix is assigned to a warp since it gives them coalesced memory accesses into the dense matrix. They also address that this method is more efficient with fewer nonzeros per row since it lends itself to lower load imbalance across the warps. This depends on the sparsity patterns of different matrices, but we see later that this is also achieved by certain configurations in our algorithm that decrease the size of the common dimension of the local multiplications. Selvitopi et al. [37] in their work on distributed memory SpMM algorithms show that the total computation time doesn't scale ideally with the number of processors. The choice of the algorithm can also impact the scaling behavior, showing that their 1.5D algorithm performs better than the 2D algorithm.

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 floating point operations. We then ran SpMM for both of these configs for 25 trials. We observed that V was $\approx$ 8x slower. After profiling with Nsight Compute [38] (metrics in Table 4.1), we noticed that it launched ≈ 64 times more blocks, which is proportional to its 64x larger common dimension size. This means less work and data assigned per block, and a higher number of smaller requests to memory. This can lead to higher latency and poor saturation of the memory bandwidth. 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 4.1: Studying the performance of $SpMM(A, H)$ on a single GPU for two configs 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:

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

The first term $\mathrm{flops\_cost}$ is constant across all configurations for G GPUs and is proportional to the total FLOPs in the SpMM, which is the total 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 $\mathrm{fwd\_penalty}$ ranks certain configurations as better than others based on our hypothesis of ideal matrix shapes. The primary idea is to first weight the penalty cost of the SpMM proportional to the size of the matrix F's first dimension: N/G_{x}. We then weight the penalty inversely proportional to the size of the second dimension of F: D^{L0}/G_{y}, which is the common dimension of the multiplication between A and F. This penalizes configurations where the dense matrix is taller and skinnier. We don't need to consider the first dimension of the adjacency matrix A as it can be inferred from the other two values. A similar calculation is done for the backward pass SpMM.

For the final computational cost, we take the square root of $flops\_cost$ to reduce the effect of outlier times, which we especially observed for larger matrices. We then multiply this by the penalty terms to weight the effect of poor matrix shapes by the total amount of work that is done. These values are then summed up across all layers of with the appropriate matrix dimensions.

However, this doesn't calculate the computational cost as a time. To do this, we conducted several runs on Perlmutter across different datasets, configurations, and GPU counts. This also included all configurations of the ogbn-products dataset on 64 GPUs to study the effect of the penalty term more closely. We then took these 67 data points and fit a linear regression model using scikit-learn [39] to fit coefficients to the three terms in our model and predict the SpMM time for a given 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}. Interestingly, the last coefficient is negative, which is not what we expected. This indicates that there might be some other factors at play that we are not accounting for. One possible explanation is that the number of memory reads/writes needs to taken into account as well due to the low computational intensity of the SpMM. However, more data points would be needed to confirm this, as the model was fit on a limited number of configurations with some observed outliers and noise. We leave this for future work.

4.2 Modeling Communication

Different configurations of the 3D grid can have a significant impact on the communication times and thereby the overall performance of the model, especially at larger scales where the communication overhead dominates the training process. However, which configuration to select optimally is a non-obvious choice. There have been several works which have provided

methods to model the communication time for distributed deep learning, such as [40], Alpa [41], AxoNN [14], [42], DGCL [27], etc.

In Plexus, we adapt the communication model from AxoNN [14] to predict the communication time for different configurations. This model adapts equations for the ring algorithm based collectives from Thakur et al. [43] and Rabenseifner [44]. Because the message sizes are in the 100s of MB range and the communication is bandwidth-bound, the latency term is omitted. For a buffer with size M, a process group with G GPUs, and bandwidth β, the communication time for the ring-based all-reduce collective used in Plexus can be calculated as follows:

$$T_{All-reduce} = \frac{2}{\beta} \times \left(\frac{G-1}{G} \right) \times M$$

Similar formulas are used for the all-gather and reduce-scatter collectives. We then apply these equations to the communication steps in Algorithm 1, taking the sizes of the matrix shards and the number of GPUs in the process groups into account. We adapt this communication model to work for multiple layers with Plexus by using the appropriate process groups based on how the sharding of the matrices changes across layers, as described in Section 3.2.

AxoNN's communication model also takes the mapping of the GPUs onto the underlying topology into account to calculate more accurate bandwidth terms. It assumes a hierarchy where certain process groups are placed within a node first. In Plexus, this hierarchy is Y, X, and then Z. The Y-parallel process groups are placed within a node first, if possible, and then the X-parallel and Z-parallel process groups. If the number of GPUs in a process group fits within a node, then it simply is assigned the intra-node bandwidth β_{intra}. However, if a process group

crosses node boundaries, it is limited by the inter-node bandwidth β_{inter}. Additionally, if there are many such inter-node collectives happening, the effective bandwidth is further reduced due to link contention. This reduction is proportional to the product of the process group sizes in the preceding preceding levels of the Y, X, Z hierarchy. We show how this is calculated for β_Z in the following equation:

$$\beta_Z = \begin{cases} \beta_{\text{intra}} & \text{if } G_{\text{X}} \cdot G_{\text{Y}} \cdot G_{\text{Z}} \leq G_{\text{node}} \\ \frac{\beta_{\text{inter}}}{\min(G_{\text{node}}, G_{\text{X}} \cdot G_{\text{Y}})} & \text{otherwise} \end{cases}$$

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

Finally, we combine the predicted SpMM time from the computation model and the predicted communication time from the communication model to calculate the predicted total epoch time for each configuration. While this doesn't take the dense computation and the loss calculation into account, these are smaller parts of the overall runtime and can safely be ignored. The result of this unified performance model is shown in Figure 4.1 for all configurations with the ogbn-products dataset on 64 GPUs of Perlmutter. We see that the 3D configurations perform better overall than the 2D and 1D configurations. The GNN the model is evaluated on has three GCN layers, where all three shards of the adjacency matrix are used. This encourages more symmetric configurations because it helps in balancing the communication load across all three

layers as well as having more favorable matrix shapes for the SpMMs. Overall, we see a strong linear relationship between the observed epoch time and the predicted epoch time, predicting the top configurations especially well.

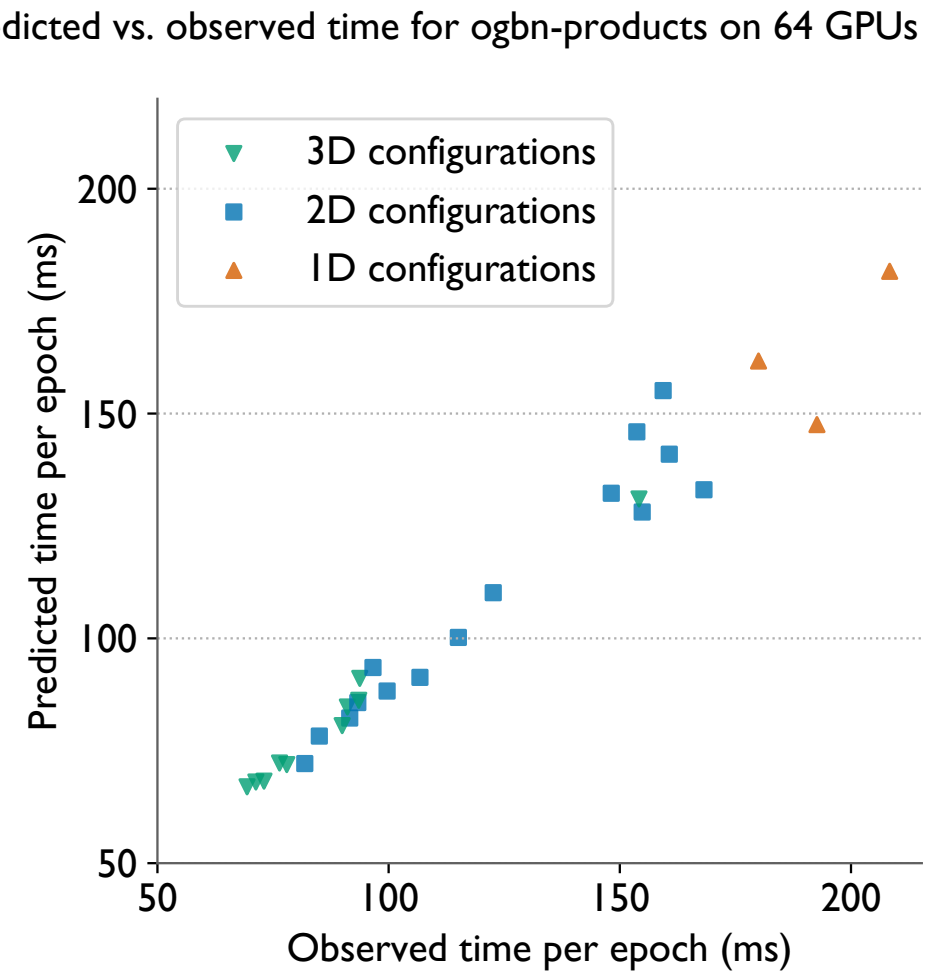

Figure 4.1: Observed time versus time predicted by the performance model for the ogbn-products dataset on 64 GPUs of Perlmutter.

Chapter 5: Performance Optimizations

Graph neural networks can pose unique challenges in the form of load imbalance caused by concentrated sparsity patterns and high communication overhead due to a large number of nodes. We address some of these issues in Plexus by introducing a few optimizations that improve the performance of our framework in such scenarios.

5.1 Double Permutation for Load Balancing

Since the adjacency matrix has a sparse and uneven distribution of nonzeros, distributing it across GPUs can cause load imbalance by creating some shards with more nonzeros than others. This can lead to computational stragglers, which ripple through the training process and significantly slow down the epoch time.

There are some typical methods employed to tackle this problem. Graph partitioners like METIS [45] are often used to partition the graph into subgraphs that are then distributed across the GPUs. METIS in particular primarily optimizes for two metrics - minimizing the edge cut and balancing the number of vertices in each partition. These are useful for many distributed graph processing frameworks with fine-grained communication where nodes explicitly have to communicate their features to neighbors in other partitions. However, in our case, we perform an all-reduce on the dense outputs of the aggregation, so there is no awareness of the graph structure

needed for the communication. A graph partitioner essentially provides a way to distribute the rows of the adjacency matrix or the nodes in the graph. However, since we are doing a 2D decomposition of the matrix, what is needed is a method to evenly distribute the nonzeros throughout all of the 2D shards.

One common technique that can alleviate this issue without the complexity of graph partitioning is to permute the nodes of the graph. This scheme doesn't have to optimize for any complex combination of metrics and doesn't require any knowledge of the graph's structure. Additionally, graph partitioning has to be done every time the GPU count changes, whereas a permutation is a preprocessing step that only has to be applied once before training, making it much simpler to use. The naive permutation scheme is shown below, where P is a permutation matrix mapping each node's original index (row) to its permuted index (column).

$$F_1 = \sigma((PAP^T)(PF_0)W_0)$$

$$F_2 = \sigma((PAP^T)F_1W_1)$$

The same permutation is also applied to the adjacency matrix's columns and the input features. This preserves correctness by ensuring that the output is permuted in a consistent manner for the next layer. This simple permutation requires preprocessing the dataset once before training. While this method provides a significant reduction in the observed load imbalance, we noticed that since graphs often have dense clusters and communities, applying a single permutation alone is insufficient. Since the same permutation is applied to both the rows (nodes) and columns (edges), we observed that some nonzeros remain concentrated around the diagonal blocks of the matrix. Applying the permutation repeatedly doesn't help either. To further disrupt

this tight coupling within communities, we apply a different permutation (P_c) to the columns of the adjacency matrix than the one that is applied to the rows (P_r), redistributing the nonzeros of the adjacency matrix more effectively. This permutation scheme needs two different versions of the adjacency matrix, which repeats every two layers.

$$F_1 = \sigma((P_r A P_c^T)(P_c F_0)W_0)$$

$$F_2 = \sigma((P_c A P_r^T)F_1 W_1)$$

By alternating between these two permutations (P_r and P_c), we reduce the effects of tightly coupled node communities and further balance the computation betweem GPUs. We show the result of our method on the europe_osm graph (see Table 6.1 for graph characteristics) in Table 5.1. We distribute the adjacency matrix into 8x8 2D shards and record the ratios of of the max number of nonzeros in a shard to the mean. We see that while the single permutation scheme helps balance the nonzeros considerably compared to the original graph, our double permutation scheme achieves near-perfect load balance.

Table 5.1: 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 (ours)	1.001

One factor to consider is the additional memory requirements of having two differently

permuted versions of the adjacency matrix. Adopting this optimization increases the memory required to store each adjacency matrix shard by 2. 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 $\min(6, L)$. However, GCNs typically have a small (2-4) number of layers to prevent oversmoothing, a phenomenon where node representations become indistinguishable from each other [46]. Given this, the overhead of storing these additional shards is a reasonable tradeoff to make for improved load balance and performance.

5.2 Blocked Aggregation

While our double permutation scheme achieves near-perfect load balance when sharding the adjacency matrix, we observed variability in training where some epochs had higher load imbalance for the forward pass SpMM than others. This caused even greater load imbalance in the all-reduce immediately following it, increasing the average epoch time. We only observed this behavior for larger datasets like Isolate-3-8M and products-14M (Table 6.1) when run on smaller GPU counts like 8-32. In this regime, the matrix size of the sparse adjacency matrix is on the order of millions of rows and columns. Since we didn't notice this for smaller matrix sizes, we optimized the aggregation step to be more efficient by blocking the sparse adjacency matrix into smaller row-blocks. Once an SpMM for a block is completed, the all-reduce on that block is performed, and then all of the blocks are concatenated together at the end. The advantage of this optimization is that the all-reduce for a block can be overlapped with the SpMM for the next block. Blocking the aggregation mitigated the variable load imbalance observed and also signifantly reduced the communication time as a result, which can be seen in Figure 5.1 (note

that these times are without overlap enabled).

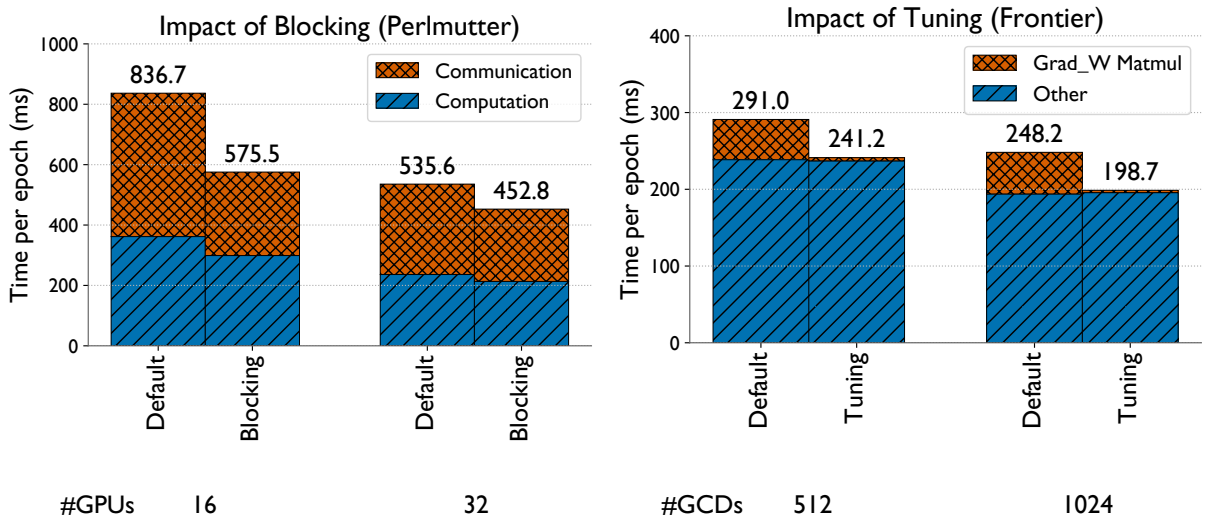

Figure 5.1: Impact of blocked aggregation on performance for Isolate-3-8M on 16 and 32 GPUs of Perlmutter (left). Impact of dense matrix multiplication tuning on performance for products-14M on 512 and 1024 GCDs of Frontier (right).

5.3 Dense Matrix Multiplication Tuning

BLAS kernels for GEMM can be called in four different modes which specify if the matrices are transposed or not: NN, NT, TN, TT. Some prior work has established that the kernels for some of these modes like NT and TN are not as performant [47]. Despite dense matrix multiplications forming a very small part of the total runtime in our workloads, we observed this behavior at high GPU counts on Frontier. This suggests that the BLAS kernels for some modes and matrix shapes are not tuned on AMD GPUs. Specifically, at GCD (1 MI250x GPU = 2 GCDs) counts equal to 512 or higher with large datasets like Isolate-3-8M and products-14M, we noticed that the matrix multiplication for calculating $\frac{\partial \mathcal{L}}{\partial W}$ stopped scaling and the time it took increased significantly. Interestingly, this matrix multiplication, as seen in Algorithm 1, has the first matrix transposed. We optimized this by reversing the multiplication order, so the calcula-

tion then becomes $\frac{\partial \mathcal{L}}{\partial W} = (SGEMM(\frac{\partial \mathcal{L}}{\partial Q}^T, H))^T$. The improvements of this optimization can be seen in Figure 5.1, where the time for the single GEMM to calculate $\frac{\partial \mathcal{L}}{\partial W}$ for the Isolate-3-8M dataset drops significantly, from approximately 50 milliseconds to a negligible value. This allows Plexus to scale more efficiently on Frontier and cross the 1024 GCD mark. While this optimization specifically helped at a large scale on Frontier, it can be extended to tuning other matrix multiplication as well on other architectures, trying all the different modes in the first few epochs and picking the best performing one.

5.4 Parallel Data Loading

Many frameworks load the entire dataset into CPU memory, get the relevant shards of data, and then copy these to the GPU. This is not sustainable as it can exceed the memory capacity of a GPU for larger graphs and feature sizes. Additionally, time to load the data from the file system can slown down the initialization. To avoid this, we implement a parallel data loader. Plexus first statically shards the processed data into 2D shards and stores each shard as a separate file. We use 8x8 2D shards for most datasets, but this number can be modified based on the dataset size and number of GPUs. Plexus' data loader then only loads the individual files needed, merges them, and extracts the relevant shards. This reduces the memory consumption by avoiding the need to load the entire dataset into memory and also reduces the time to load the data from the file system. The sharding process is also concurrent, making it scalable for larger graphs. For the ogbn-papers100M (Table 6.1) dataset with its best configuration on 64 GPUs, the CPU memory requirements for loading the data reduces from 146 GB to 9 GB when sharding the data into 16x16 files. The time to load the data from the file system also dropped from 139 seconds to 7

seconds.

Chapter 6: Experimental Setup

The following chapter outlines the experimental setup used to evaluate Plexus, including the details of the supercomputer platforms we ran on, datasets used, model details, and other SOTA frameworks we compared against.

6.1 Details of Supercomputer Platforms

Our experiments were conducted on the Perlmutter and Frontier supercomputers. Each node on Perlmutter has 4 NVIDIA A100 GPUs each with 40GB of HBM2 memory. We use the 80GB nodes for runs on 64 and 128 GPUs for the largest dataset. Each node on Frontier has 4 AMD Instinct MI250X GPUs each with 128GB of HBM2E memory. Each MI250X GPU is partitioned into two Graphic Compute Dies (GCDs) and each GCD appears as a separate device to high level frameworks. Whereas the A100 has a peak of 19.5 FP32 TFLOPS, the MI250X has a peak of 47.9 FP32 TFLOPS. The nodes on both systems have 4 HPE Slingshot 11 NICs each 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 Model and Datasets

We conduct experiments on a wide range of graph datasets, shown in Table 6.1. The Reddit dataset is available through PyTorch Geometric and contains post data from September 2014, with nodes as individual posts and edges connecting posts if the same user commented on both [8]. The ogbn-products dataset is part of the Open Graph Benchmark (OGB) [48] and depicts Amazon's product co-purchasing network, where nodes are products sold and edges indicate that the products are purchased together. The Isolate-3-8M is a subgraph of a protein similarity network in HipMCL's data repository [49]. The products-14M datasets is a larger Amazon products network [50]. The europe_osm dataset, part of the 10th DIMACS Implementation Challenge [51], represents OpenStreetMap data for Europe, where nodes correspond to geographical locations, and edges represent roads connecting these points. The ogbn-papers100M dataset, part of OGB, represents the Microsoft Academic Graph (MAG), where nodes are papers and edges indicate citation relationships. For the Reddit, ogbn-products, and ogn-papers-100M datasets, we used the input features and labels that were provided with the datasets. For the Isolate-3-8M, products-14M, and europe_osm datasets, we randomly generated input features with a size of 128 and generate labels with 32 classes based on the distribution of node degrees. Throughout our experiments, we use 3 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. [10].

Table 6.1: 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

6.3 Other State-of-the-art Frameworks

We compare with SA, a sparsity-aware implementation of CAGNET [22], and BNS-GCN [25], two SOTA frameworks for distributed full-graph GNN training that evaluate their performance on hundreds of GPUs as seen in Table 2.1. For SA, we additionally use partitioned files created with GVB [52], a graph partitioner used by the authors to improve performance. 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 only compare with these baselines on Perlmutter as we encountered frequent stability issues and memory errors on Frontier, preventing us from running experiments as reliably. For BNS-GCN, we made a small modificaton to the code which resolved a bug that crashed training when the boundary size was 0. We also contacted the authors of BNS-GCN regarding unexpectedly high runtimes with METIS, but didn't receive a response in time to compare against it for this work. We train for 10 epochs in each trial and take the average of the last 8 epochs to account for initial fluctuations. For each experiment, we run 3 independent trials to account for potential performance variability on the supercomputers and report the average epoch time. We

also provide proof of our framework's correctness in Figure 6.1.

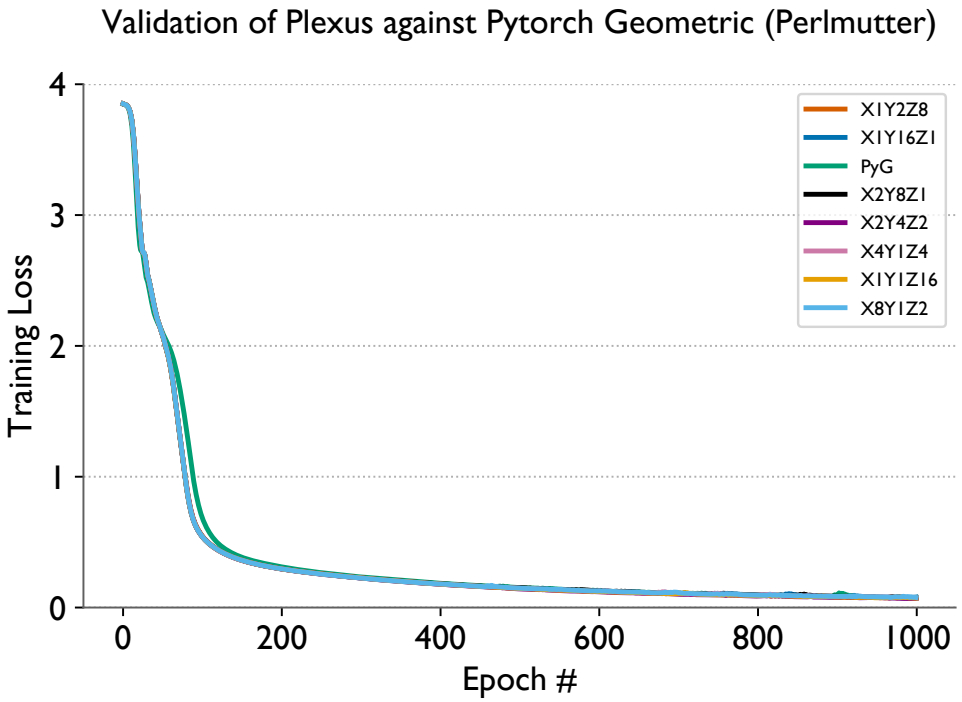

Figure 6.1: Validating Plexus against a serial Pytorch Geometric baseline on 16 GPUs of Perlmutter with ogbn-products.

Chapter 7: Results

In this chapter, we present the results of our experiments across all datasets on both Perlmutter and Frontier and compare Plexus against SA, SA+GVB, and BNS-GCN.

7.1 Comparison with SOTA Frameworks

We compare Plexus with the baselines we chose on three datasets - Reddit, Isolate-3-8M, and products-14M, which have varying sizes. We were planning to show comparison results for the ogbn-papers100M dataset as well, but ran into some issues. Partitioning ogbn-papers100M for BNS-GCN either timed out or the job was killed, even when running for 5 hours. SA and SA+GVB ran out of CPU memory since it instantiates the entire dataset. The results of these experiments comparing with BNS-GCN, SA, and SA+GVB are shown in Figure 7.3.

On Reddit, we see that SA achieves better performance than Plexus at 4 GPUs, but doesn't scale beyond that. SA+GVB has slightly better performance than SA up to 64 GPUs, but also has poor scaling. Note that we don't collect data on 4 and 16 GPUs for SA+GVB as we were unable to receive those partitioned files in time from the authors. BNS-GCN achieves better performance than both BNS-GCN scales similarly to SA, but is slower in terms of absolute time taken. We were unable to create 64 partitions for BNS-GCN as it timed out after 30 minutes despite the small size of the graph, but we observe a clear pattern where both frameworks are unable to scale

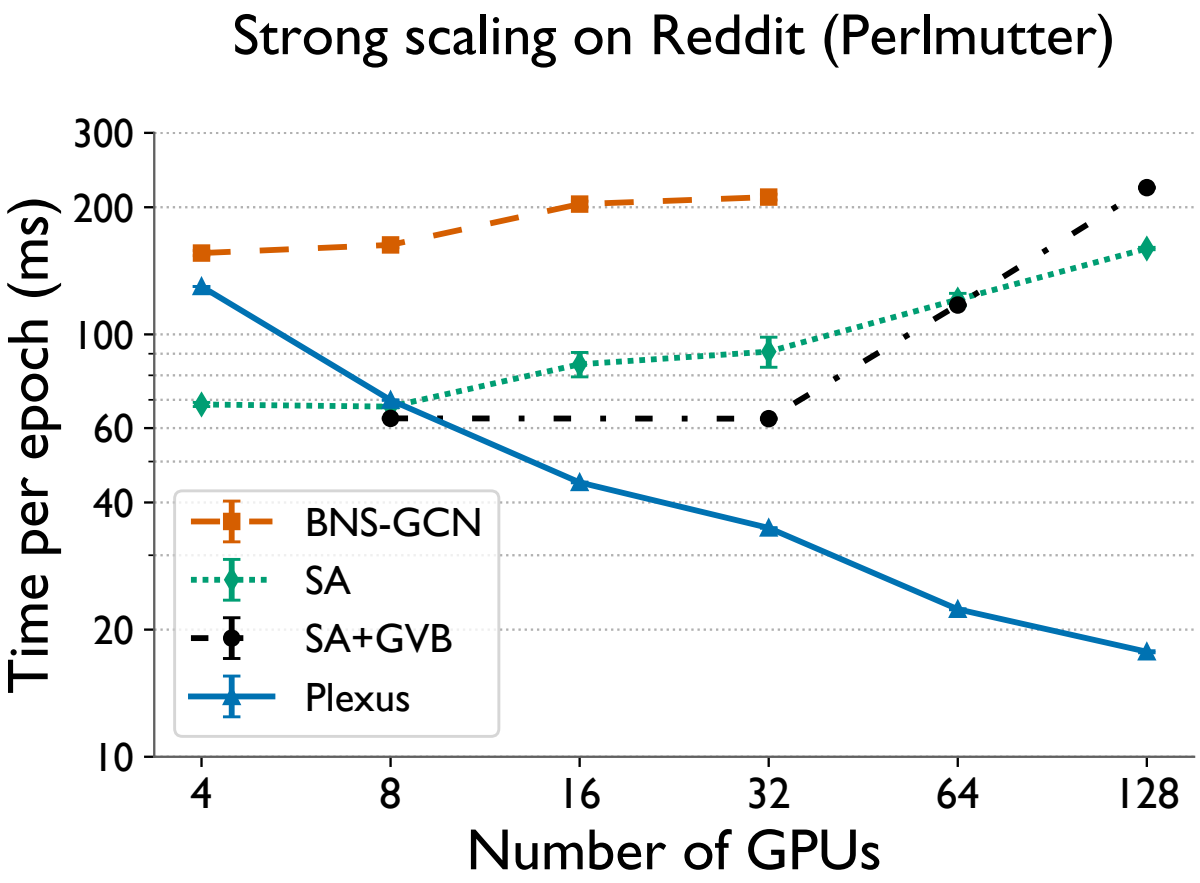

Figure 7.1: Comparison of strong scaling performance of Plexus, SA, SA+GVB, and BNS-GCN for Reddit on Perlmutter.

on Reddit. Plexus achieves a 9x speedup over SA on 128 GPUs, 6x speedup over BNS-GCN on 32 GPUs, and a 12.5x speedup over SA+GVB on 128 GPUs Compared to the other frameworks, Plexus scales constistently well up to 128 GPUs.

On Isolate-3-8M, we were unable to run SA or SA+GVB, as they ran out of CPU memory. Interestingly, we observe that BNS-GCN scales till 64 GPUs and achieves better performance than Plexus, but rapidly degrades beyond this. Plexus achieves a 3.8x speedup over BNS-GCN on 256 GPUs and continues to scale well up to 1024 GPUs. BNS-GCN has more fine-grained communication, where only the boundary nodes communicate, whereas Plexus is more coarse-

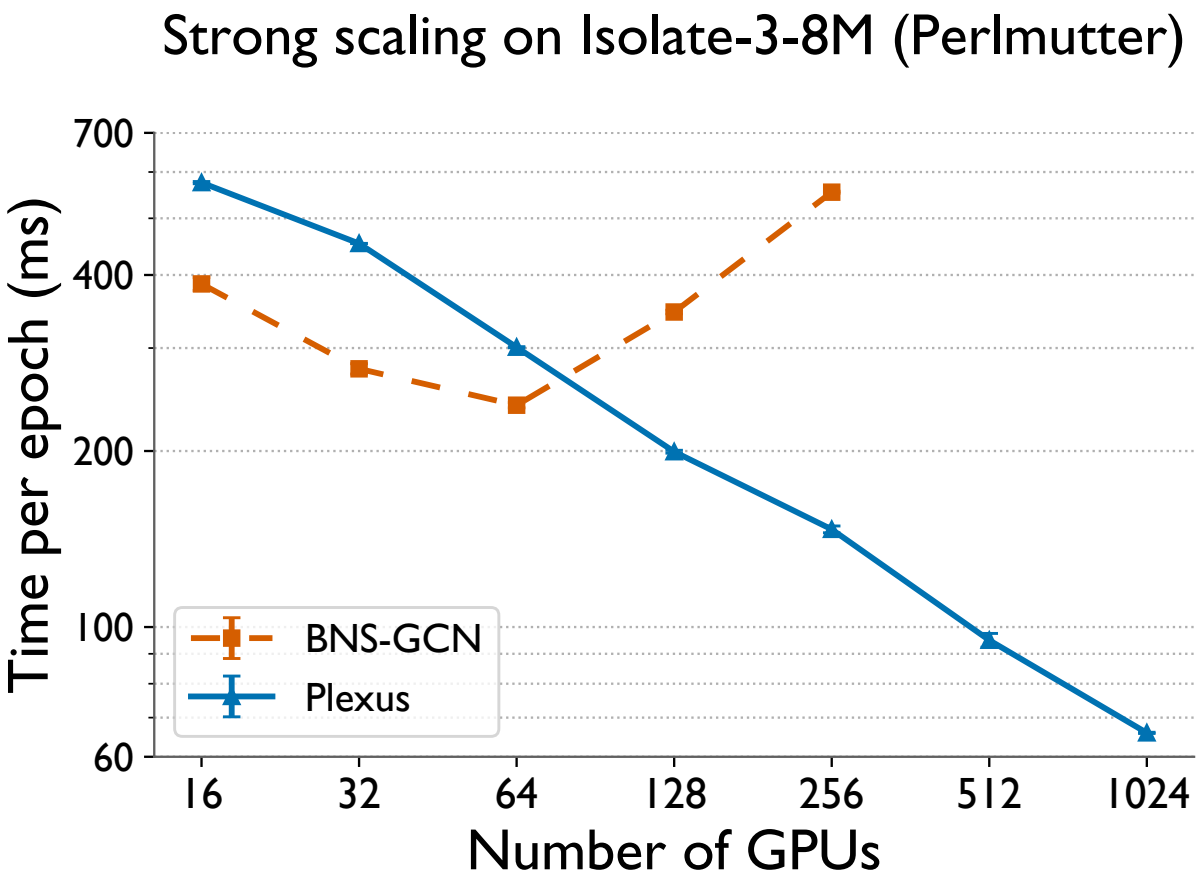

Figure 7.2: Comparison of strong scaling performance of Plexus, SA, SA+GVB, and BNS-GCN for Isolate-3-8M on Perlmutter.

grained and communicates on the dense outputs. At a smaller scale, BNS-GCN can achieve good performance due to there being few boundary nodes after METIS partitioning. However, at a larger scale, there are two issues. Firstly, the partitioner starts to divide denser subgraphs, resulting in a larger number of boundary nodes. Secondly, BNS-GCN utilizes the all-to-all collective, which is naively implemented with pairwise P2P messages. Compared to ring based collectives used in Plexus where GPUs only communicate with their neighbors, the all-to-all sends more messages across longer distances, which leads to higher latency. Without sampling boundary nodes, METIS alone is insufficient for BNS-GCN to achieve comparable performance at scale.

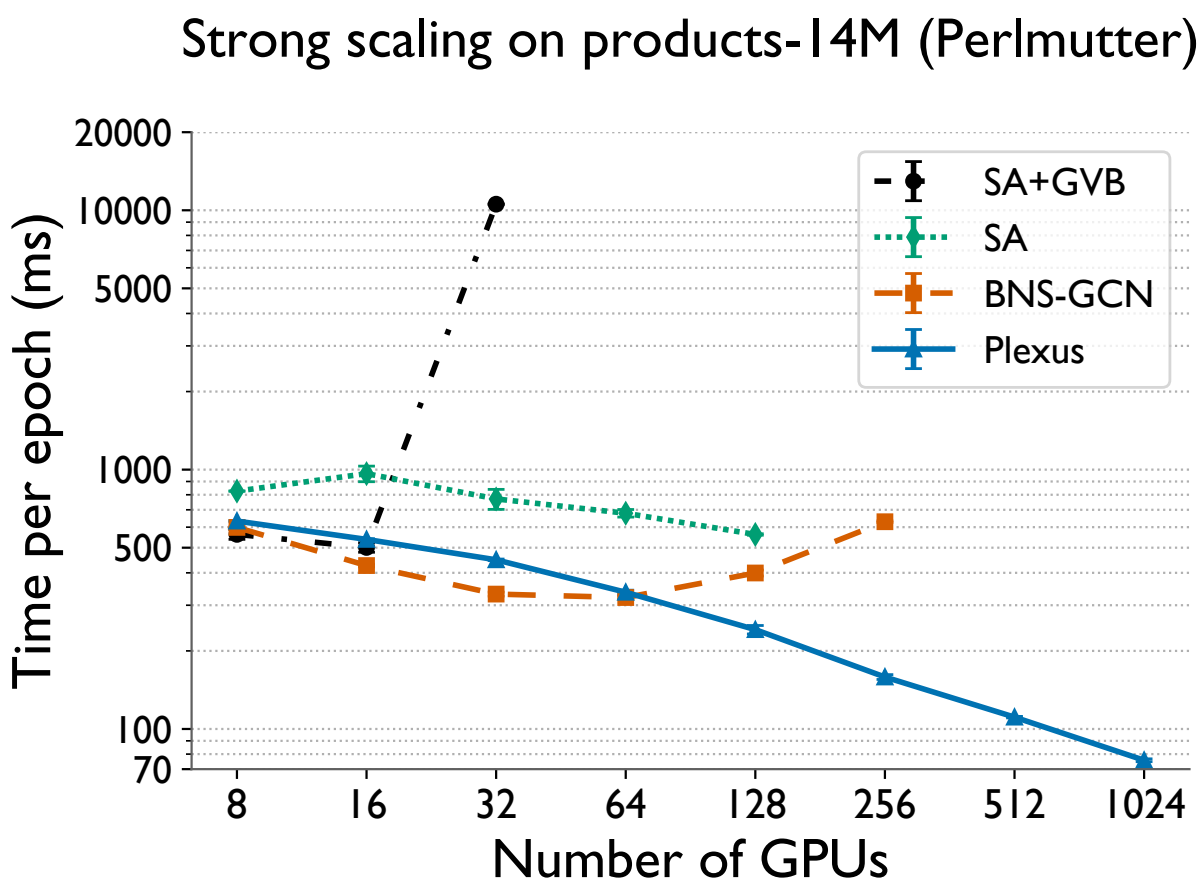

Figure 7.3: Comparison of strong scaling performance of Plexus, SA, SA+GVB, and BNS-GCN for products-14M on Perlmutter.

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.

To understand more about the inflection point between BNS-GCN and Plexus at 64 GPUs, we look at the time breakdown in Figure 7.4. 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 doesn't have sparsity-aware modifications like SA. The inefficiency of the all-to-all communication pattern that BNS-GCN employs starts to become evident at 64 GPUs. Another interesting observation is that the computation scaling for both also differ significantly. 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. After further investigation, we found that the total number of nodes across partitions, including boundary nodes, increased from 18M to 22M from 32 to 256 GPUs. This explains why the local computation of a partition also increases considerably in addition to the poor scaling of communication.

Overall, we see that Plexus is able to achieve better performance than both BNS-GCN, SA, and SA+GVB across a variety of datasets. While BNS-GCN and SA have communication patterns that are more efficient at smaller scales as they take advantage of the sparsity of the graph, they are unable to perform well at a larger scale. Plexus, on the other hand, is able to scale well to 1024 GPUs for these datasets and achieves the lowest absolute epoch time when compared to what the other frameworks achieve. Additionally, the performance of Plexus is more consistent across different datasets, whereas we see that BNS-GCN and SA's performance can vary significantly depending on the characteristics of the dataset, such as worse performance on Reddit which is a denser graph. Even at a small scale, we see that Plexus is competitive with other frameworks. This is especially noteworthy given that it requires no graph partitioner. BNS-GCN uses METIS, which can take on the order of hours for some datasets, and SA uses

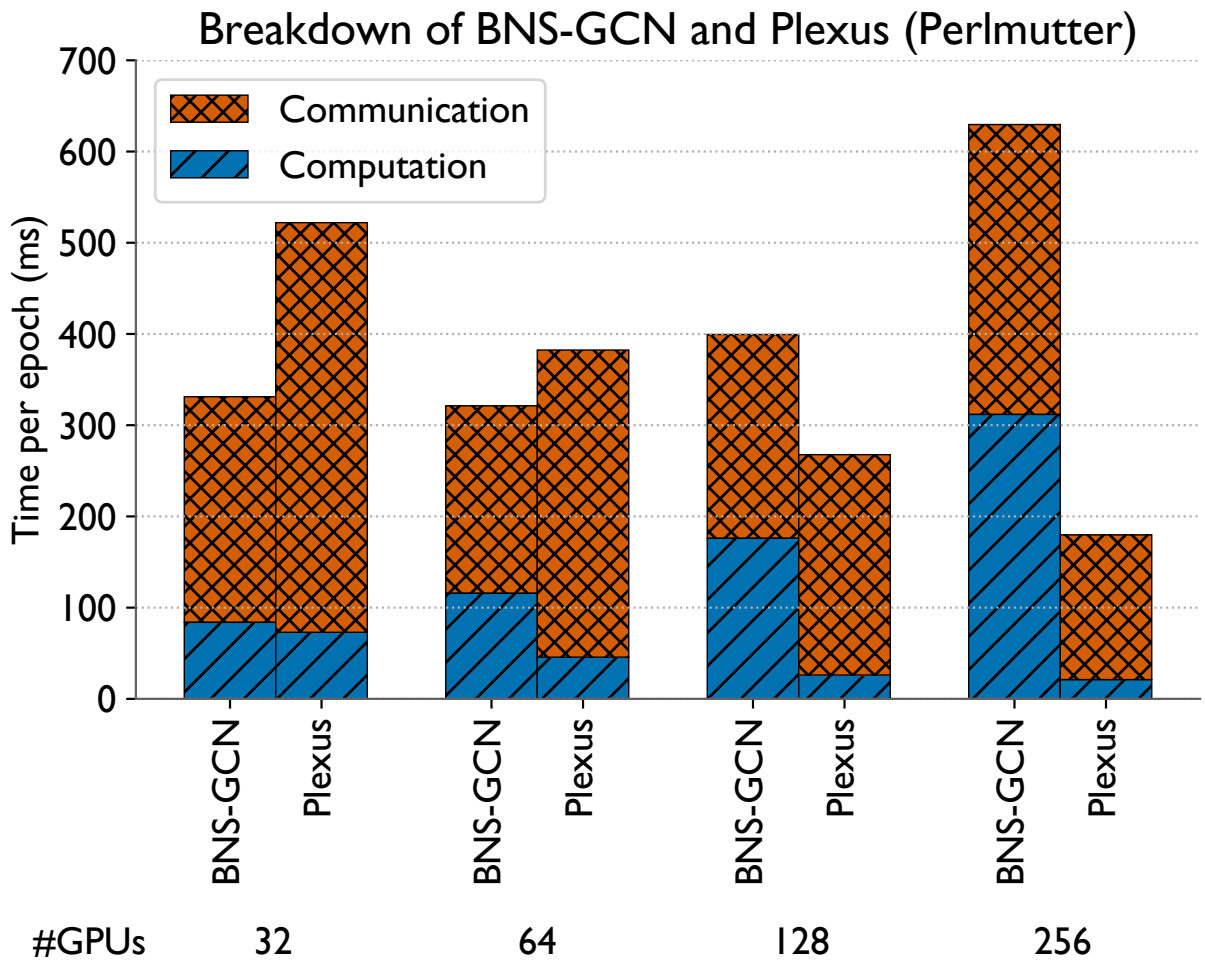

Figure 7.4: Breakdown of epoch times for BNS-GCN and Plexus on 32-128 GPUs of Perlmutter with products-14M.

GVB, which they mention goes out of memory when trying to create 32 partitions for ogbn-papers100M. On the other hand, Plexus is able to mitigate most of the load-imbalance caused by the graph's structure with its simple double permutation scheme, which takes a few minutes at most and doesn't consume excessive amounts of memory.

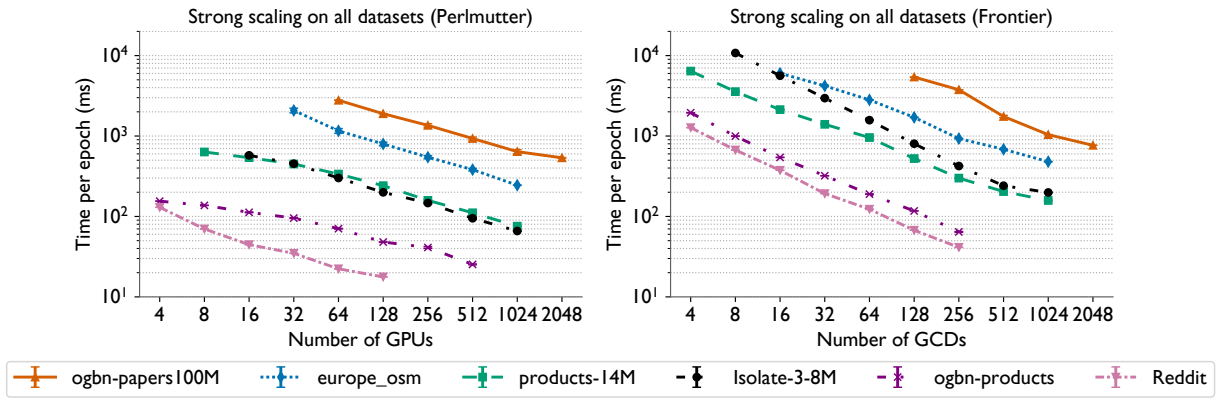

Figure 7.5: Strong scaling performance of Plexus for six different datasets of different sizes (Table 6.1) on both Perlmutter (left) and Frontier (right). Note that x-axis shows GPUs for perlmutter and GCDs for Frontier.

7.2 Strong Scaling of Plexus

Apart from the datasets discussed above, we also ran Plexus on three other datasets to demonstrate its strong scaling capabilities on both Perlmutter and Frontier (Figure 7.5). The sparsity level of a graph determines the communication to computation ratio in Plexus. Reddit, a denser graph compared to ogbn-products, scales better. This translates to ogbn-products becoming communication-dominated quicker than Reddit, explaining the increasing gap between the two datasets. 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 Isolate-3-8M starts out with a higher time at 16 GPUs when the computation cost is significant, but products-14M, which is more communication dominated, eventually takes longer after 64 GPUs. We also show results for europe_osm on 1024 GPUs and ogbn-papers100M on 2048 GPUs, and see that ogbn-papers100M's scaling starts to slow down at 2048 GPUs, at which point the computation cost is marginal. This is to our knowledge the largest number of GPUs that have been used for parallel full-graph GNN training to date.

On Frontier, we notice generally better trends for all datasets when compared to Perlmutter. This is because the SpMM times on AMD GPUs were an order of magnitude higher than on NVIDIA GPUs, and so training remains computation dominated for longer and Plexus is able to scale better. The past comparison we made between Reddit and ogbn-products doesn't hold here, as we don't see a growing gap between the two datasets. Similarly, we see that Isolate-3-8M consistently takes more time than products-14M because of its higher number of edges, and both only start to get close at 512 GCDs. We see that europe_osm, which is a sparser graph than both products-14M and Isolate-3-8M, has slower scaling than these two and the gap between them starts to grow because of its higher communication-to-computation ratio. Finally, we see that Plexus demonstrates impressive scaling for ogbn-papers100M, which is the largest graph we ran with, for up to 2048 GCDs.

Chapter 8: Future Directions and Conclusion

We discuss a promising future direction to make Plexus performant across a wide range of GPU counts, and also summarize the contributions of this thesis in making parallel full-graph GNN training efficient and practical.

8.1 Sparsity-Aware All-reduce

While Plexus performs demonstrates impressive performance on larger scales, where the benefits of 3D parallelism outweigh the practical limitations of current sparsity-aware approaches, it could be further improved at a smaller scale. The current implementation of Plexus uses a regular all-reduce operation, where the entire matrix is communicated. However, taking advantage of the graph's sparsity could be beneficial on a lower number of GPUs, as seen earlier in Section 7.1. Specifically, in Plexus, this could be implemented by each GPU only receiving the rows of the matrix that are needed for its computation in the next layer. This could be implemented by keeping track of the adjacency shard's nonzero columns and only communicating the corresponding rows in the previous layer. The all-reduce operation would then effectively become an all-to-all operation, where each GPU communicates the rows needed by all other GPUs in the process group. Further work could be done to optimize the all-to-all operation, which we observed scales poorly due to naive implementations. These efforts could be combined with the

existing framework to create a hybrid approach that performs efficiently across all GPU counts.

8.2 Summary

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. We present Plexus, a three-dimensional parallel framework for distributed full-graph GNN training that adapts Agarwal et al.'s 3D parallel matrix multiplication algorithm [12] to scale to thousands of GPUs for billion-edge graphs. 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, blocked aggregation to minimize variability, and others. 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, which is a notoriously challenging problem to scale, both efficient and practical.

Bibliography

[1] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[2] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[3] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019.

[4] Yann LeCun, Bernhard Boser, John Denker, Donnie Henderson, R. Howard, Wayne Hubbard, and Lawrence Jackel. Handwritten digit recognition with a back-propagation network. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 396–404. Morgan-Kaufmann, 1990.

[5] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric, 2019.

[6] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks, 2020.

[7] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction, 2018.

[8] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[9] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: Fast learning with graph convolutional networks via importance sampling, 2018.

[10] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 187–198, 2020.

[11] Hao Yuan, Yajiong Liu, Yanfeng Zhang, Xin Ai, Qiange Wang, Chaoyi Chen, Yu Gu, and Ge Yu. Comprehensive evaluation of gnn training systems: A data management perspective, 2024.

[12] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[13] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-AI: a unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 766–775, New York, NY, USA, 2023. Association for Computing Machinery.

[14] 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. 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, November 2024.

[15] Oslo: Open source for large-scale optimization. `https://github.com/EleutherAI/oslo`, 2021.

[16] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 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, July 2019.

[17] Taraneh Younesian, Daniel Daza, Emile van Krieken, Thiviyan Thanapalasingam, and Peter Bloem. Grapes: Learning to sample graphs for scalable graph neural networks, 2024.

[18] Siddhartha Shankar Das, S M Ferdous, Mahantesh M Halappanavar, Edoardo Serra, and Alex Pothen. Ags-gnn: Attribute-guided sampling for graph neural networks, 2024.

[19] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, and Dongrui Fan. Sampling methods for efficient training of graph convolutional networks: A survey, 2021.

[20] Alok Tripathy, Katherine Yelick, and Aydın Buluç. Reducing communication in graph neural network training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[21] Robert A. van de Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. Technical report, USA, 1995.

[22] Ujjaini Mukhopadhyay, Alok Tripathy, Oguz Selvitopi, Katherine Yelick, and Aydin Buluc. Sparsity-aware communication for distributed graph neural network training. In *Proceedings of the 53rd International Conference on Parallel Processing*, ICPP '24, page 117–126, New York, NY, USA, 2024. Association for Computing Machinery.

[23] Muhammed Fatih Balın, Kaan Sancak, and Ümit V. Çatalyürek. Mg-gcn: Scalable multi-gpu gcn training framework, 2021.

[24] Süreyya Emre Kurt, Jinghua Yan, Aravind Sukumaran-Rajam, Prashant Pandey, and P. Sadayappan. Communication optimization for distributed execution of graph neural networks. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 512–523, 2023.

[25] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling, 2022.

[26] Cheng Wan, Youjie Li, Cameron R. Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication, 2022.

[27] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. Dgcl: an efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 130–144, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Xin Ai, Hao Yuan, Zeyu Ling, Qiange Wang, Yanfeng Zhang, Zhenbo Fu, Chaoyi Chen, Yu Gu, and Ge Yu. Neutrontp: Load-balanced distributed full-graph gnn training with tensor parallelism, 2024.

[29] Borui Wan, Juntao Zhao, and Chuan Wu. Adaptive message quantization and parallelization for distributed full-graph gnn training, 2023.

[30] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proc. VLDB Endow.*, 15(9):1937–1950, May 2022.

[31] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. 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)*, pages 779–795, Boston, MA, July 2023. USENIX Association.

[32] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. Neutronstar: Distributed gnn training with hybrid dependency management. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1301–1315, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Jaeyong Song, Hongsun Jang, Jaewon Jung, Youngsok Kim, and Jinho Lee. Granndis: Efficient unified distributed training framework for deep gnns on large clusters, 2024.

[34] Shuai Zhang, Zite Jiang, and Haihang You. Cdfgnn: a systematic design of cache-based distributed full-batch graph neural network training with communication reduction, 2024.

[35] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti,

Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. Technical report, 2022.

[36] Carl Yang, Aydin Buluc, and John D. Owens. Design principles for sparse matrix multiplication on the gpu, 2018.

[37] Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydın Buluç. Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication. In *Proceedings of the 35th ACM International Conference on Supercomputing*, ICS '21, page 431–442, New York, NY, USA, 2021. Association for Computing Machinery.

[38] NVIDIA. Nvidia nsight compute. `https://developer.nvidia.com/nsight-compute`.

[39] 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. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[40] Shenggan Cheng, Ziming Liu, Jiangsu Du, and Yang You. Atp: Adaptive tensor parallelism for foundation models. *arXiv preprint arXiv:2301.08658*, 2023.

[41] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. *CoRR*, abs/2201.12023, 2022.

[42] Shengwei Li, Zhiquan Lai, Yanqi Hao, Weijie Liu, Keshi Ge, Xiaoge Deng, Dongsheng Li, and Kai Lu. Automated tensor model parallelism with overlapped communication for efficient foundation model training. *arXiv preprint arXiv:2305.16121*, 2023.

[43] Rajeev Thakur and William D. Gropp. Improving the performance of collective operations in mpich. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 257–267, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[44] Rolf Rabenseifner. Optimization of collective reduction operations. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, pages 1–9, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[45] George Karypis and Vipin Kumar. 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.

[46] Qimai Li, Zhichao Han, and Xiao-Ming Wu. 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*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[47] Shaohuai Shi, Pengfei Xu, and Xiaowen Chu. Supervised learning based algorithm selection for deep neural networks. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 344–351, 2017.

[48] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs, 2021.

[49] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. Hipmcl: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks. *Nucleic Acids Research*, 46(6):e33–e33, 01 2018.

[50] Jianmo Ni, Jiacheng Li, and Julian McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 188–197, Hong Kong, China, November 2019. Association for Computational Linguistics.

[51] Geofabrik GmbH. Dimacs10/europe_osm. SuiteSparse Matrix Collection, 2010.

[52] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems. *Parallel Comput.*, 59(C):71–96, November 2016.