GDAŃSK UNIVERSITY OF TECHNOLOGY

TASK Quarterly 28 (3) 2024

Optimizing Graph Learning using a Hierarchical Graph Adjacency Matrix (HGAM)

Robert Benke [1,2,a], **Emmanouil-Ioannis Farsarakis** [2], **Michal Szarmach** [2] **Andrea Zanetti** [2], **Hsien-Hsin S. Lee** [2]

[1] Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology, Narutowicza 11/12, 80-233, Gdańsk, Poland

[2] Intel Corporation, 2200 Mission College Blvd, Santa Clara, USA

https://doi.org/10.34808/tq2024/28.3/b

Abstract

Graph Neural Networks (GNNs) have been increasingly adopted in modern, large-scale applications, such as social network analysis, recommendation systems, drug discovery, and more. However, the training cost of GNNs can be computationally prohibitive, especially when the graph is large and complex, necessitating the use of a mini-batching approach. In this paper, we propose a novel data structure called the Hierarchical Graph Adjacency Matrix (HGAM) to accelerate GNN training by avoiding redundant computations. With HGAM, we can accelerate the training speed of GNNs by up to four times. Additionally, we propose optimizations on top of HGAM to further enhance performance, achieving an overall speedup of up to 7.72 times for training 3-layer deep GNNs. We evaluated our techniques using three benchmark datasets—Reddit, ogbn-products, and ogbn-mag—and demonstrate that the proposed HGAM technique and related optimizations are advantageous for GNN training across modern hardware platforms.

Keywords:

Data structure optimization, graph neural networks, graph learning, sparse data

[a]Email: robbenke@pg.edu.pl

1. Introduction

Deep neural networks and machine learning (ML) have seen a significant increase in capabilities and adoption. Until recently, research in these domains has focused mostly on the processing of Euclidean structured data, while hardware and software have co-evolved to accommodate such data structures. Graph Neural Networks (GNNs) have emerged as a way to simultaneously leverage both advanced learning technologies and the full context and interconnectivity of a graph's underlying structures. The use of GNNs has led to state-of-the-art performance in tasks such as node classification and link prediction, which are employed in application domains including social networks, drug discovery, fraud detection, recommendation systems, and physics simulation [1–5].

Graph Neural Networks Basics

Most GNNs achieve high accuracy on tasks such as node classification, link prediction, and graph classification, in part by creating improved node embeddings from the features of the original graph dataset. Processing a layer in most types of GNNs involves each node collecting and aggregating data from its direct neighbors using an aggregation function specific to the model, commonly sum or mean. Other tasks may involve applying linear and non-linear transformations to the embeddings to increase complexity and add learnable parameters to the model. Each GNN layer, therefore, transfers information to updated embeddings from their direct neighbors (or "1-hop" neighbors). Chaining GNN layers makes it possible to extend the horizon of the resulting node embeddings beyond direct neighbors.

Scaling Challenges

Updating node embeddings in a GNN can be handled in one of two ways: either all nodes are processed together, known as full batch processing, which requires the full dataset to be loaded at once, or nodes are processed in minibatches due to memory constraints. However, in addition to the "seed nodes" that we want to update in a batch, a GNN minibatch must also include additional "support nodes," *e.g.,* the N-hop neighbors of the seed nodes, which will not be updated in this batch but are required to carry out the algorithm. The size of the minibatch subgraph required scales exponentially with the model depth, even for models with a small number of layers.

The memory pressure is further exacerbated by the sheer growth of state-of-the-art (SOTA) GNN model sizes. One current SOTA GNN model on the PCQM4Mv2 dataset [6,7] has more parameters than the 2nd and 3rd best models combined from the Open Graph Benchmark (OGB). The best-performing model for text-attributed graphs has more than 100 times the parameters of any other model in the top five models for ogbn-products and ogbn-arxiv [8].

Previous Solutions

Learning more expressive and generalizable GNN models requires more lightweight minibatch training. One of the early approaches to limit the subgraph size is neighbor sampling (NS) [9]. NS reduces the total number of neighbors at each level by randomly selecting a fixed-size subset of neighbors. This way, we can arbitrarily shrink the subgraph with a predefined upper bound for the number of nodes and edges. More sophisticated techniques for selecting the neighbors have been proposed, based on node importance [10], influence [11], or variance reduction [12]. Those techniques give us control over total memory usage but still might be insufficient due to the exponentially growing neighborhood in networks that require a higher receptive field.

The exponentially growing neighborhood of seed nodes in minibatching is a well-known problem, and many solutions have been proposed. One of the first studies attempting to mitigate this problem was ClusterGCN [13]. Despite the name, this technique can be applied to a wide range of GNN models and reduces the minibatch subgraph by restricting the sampled nodes to a given cluster of the original graph. The authors of SIGN [14] proposed a GNN model that performs all the sparse computation as a preprocessing step and trains the model with no node neighborhood requirements. In GraphSAINT [15], a technique was proposed to mitigate minibatching inefficiency by making use of all nodes that were sampled (support nodes) and decreasing the total number of iterations required for model convergence. All these methods provided improvements in efficiency to various degrees, but they do not generate the same outputs as the original implementations and could be detrimental to model quality.

An alternative method aimed at reducing the total number of computations in message passing is Hierarchically Aggregated Computation Graphs (HAG) [16]. HAG groups together nodes that are all sending information to the same place, reducing the number of nodes to aggregate and speeding up the process. HAG and HGAM operate on different principles but can be employed concurrently for models where HAG can be used. While HGAM focuses on discarding edges and nodes that are no longer needed, HAG is designed to prevent redundant node aggregations throughout the message passing process.

The authors of [9] proposed a technique based on a list of bipartite graphs, one for every GNN layer, to limit the number of unnecessary computations without chang-

ing the final output. Each bipartite graph contains a set of source and destination nodes, where the set of destination nodes is a subset of the source nodes. The paper showed that this technique can eliminate redundant computations from message passing. However, it requires additional memory for storing the extra adjacency matrices, which also increases the data transfer between CPU and GPU, requiring a pre-processing step to be applied to every adjacency matrix separately. Even with the above techniques applied, there is a fundamental problem with optimizing GNNs on currently available hardware due to the random memory access patterns and poor cache reuse in sparse computations [17, 18], which means these workloads are not able to leverage the beneficial architectural features of standard training and inference accelerators. Therefore, techniques such as leveraging a sparse accelerator [19] and heterogeneous systems [20] have been proposed, and promising results have been demonstrated.

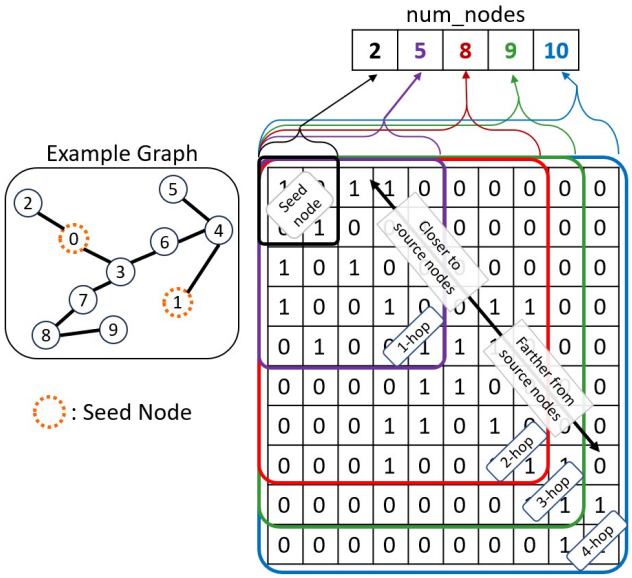

Figure 1: Hierarchical Graph Adjacency Matrix data structure.

Hierarchical Graph Adjacency Matrix

In this paper, we propose **HGAM**: **H**ierarchical **G**raph **A**djacency **M**atrix, a new data structure for graph data that allows us to efficiently limit the computations in deep GNNs. It works by adjusting the minibatch subgraph after each GNN layer to skip any redundant computations in minibatched training or inference. We detail the construction and usage of HGAM in Section 2. Our data structure can be adapted for compression in line with other well-known sparse data structures like coordinate list (COO) and compressed sparse rows (CSR) under the hood, as described in Section 3. This makes this novel data structure easy to adopt, as there is no need to modify existing core computational kernels.

A key concept that underpins our approach is the Message Flow Graph (MFG). MFGs were introduced in the GNN domain to explain how messages propagate through a graph during layer-wise computation. An MFG captures the minimal subgraph required to compute the output for a set of seed nodes, tracing only the edges and nodes along which information actually flows. This enables a more precise understanding of the effective receptive field of each node at a given layer.

The concept of MFGs motivated our work by highlighting that, since a single GNN layer only shares a node's information with its direct neighbors, we can safely prune parts of the graph that are unreachable within the remaining layers. While MFGs define the relevant subgraphs for each computational layer, they do not specify how to construct or use them efficiently. HGAM addresses this gap by implementing MFGs in a way that minimizes both sampling time and memory overhead. It prunes subgraphs in constant time and uses negligible additional memory, setting it apart from existing approaches known to the authors.

Contributions

Our work makes the following contributions to advance the field of GNN optimization:

- We develop a novel data structure that allows us to significantly reduce the minibatching compute overheads with negligible extra memory needed.
- We provide empirical analysis for neighborhood explosion and the computational implications with and without HGAM.
- We study prominent bottlenecks in GNN training and propose additional optimization techniques based on the meta-information stored in HGAM.

Last but not least, we make our HGAM technique publicly available to the broader community so it can be studied and used to accelerate a wider range of workloads beyond GNNs. It is developed and presented in a generic way so that it can be leveraged in any problem involving concepts equivalent to seed nodes and the requirement to easily generate graphs of different horizon widths around them.

2. Hierarchical Graph Adjacency Matrix Data Structure

The Hierarchical Graph Adjacency Matrix (HGAM) data structure is a representation of a graph's adjacency

matrix that allows for efficient extraction of the k-hop neighborhood of a node. The HGAM data structure is a hierarchical matrix, where each level of the hierarchy is a subgraph of the level above. A subset of the nodes has been chosen as "seed nodes," to which a hierarchical neighborhood perspective is referred. The last level of the hierarchy contains all the nodes that are at a finite distance from the seed nodes, and each preceding level is a subset of nodes that are closer to the seed nodes. The subgraphs are constructed by selecting a subset of the nodes and retrieving all edges among the selected nodes. It is usually used in a bottom-up manner, starting from the original graph and ending with the seed nodes only, but a reverse order is also possible.

The HGAM data structure is shown in Figure 1. Nodes in this data structure are partially sorted. Seed nodes are inserted in the matrix first (upper left region of the matrix in Figure 1), followed by support nodes from increasingly distant neighborhoods. In this way, any node with a given index i is always at the same or a greater distance from the seed nodes compared to all nodes with index $j < i$, where distance is measured by the number of neighbor "hops" between them. The order of the nodes within the same distance level can be arbitrary. In our implementation, it is naturally given by the construction process we describe in the next section.

2.1. HGAM Construction

The HGAM data structure is constructed in a breadth-first search (BFS) manner. The first level of the hierarchy is the set of seed nodes. These nodes are selected from the original graph, usually randomly in the case of GNN minibatching. In the next step, we sample (incoming) neighbors for the first seed node and add them to the subgraph, together with the directed incoming edges to the seed node itself. In a dense adjacency matrix representation, this would complete the creation of the first column of the HGAM, since no more incoming edges can be added to the first node in subsequent stages. We move to the next seed node and fill the second column according to the same logic. We repeat this procedure for all seed nodes. Once this first step is completed, our single-level HGAM is ready. Columns that correspond to the 1-hop neighbors are populated with zero values. This is intentional since those support nodes are only a source of information for the seed nodes. Then, we can continue the procedure of neighbor sampling detailed above for level-1 nodes to create the second level of the HGAM. The second level of the hierarchy is the set of support nodes that are connected to the nodes in the first level. However, we could have also sampled one or more nodes that were already in the subgraph: in that case, we would not add them again but instead reuse the ones that have already been added. This does not change the already built structure of the hierarchy and, most importantly, it allows for a single coherent representation of the total graph obtained by multiple sampling procedures. This process continues until the k-hop neighborhood of the seed nodes is constructed for a given k. It is worth noting that to be readily usable, the HGAM construction also records some metadata generated at the sampling time, which keeps track of the number of new support nodes and new edges added at each level of the hierarchy. This metadata is represented by the num_nodes array in Figure 1.

2.2. Minibatch GNN Training with HGAM

Graph Neural Networks can be trained using a full-batch or minibatch approach. Full batch indicates that we are using the whole graph during the forward pass and the gradients are calculated for all training nodes' outputs. By following this method, we make use of all nodes' outputs and obtain stable gradients, but it is a challenge for large graphs. Minibatch training allows us to constrain the memory requirements of training and provide weight updates more frequently. This is a well-known and widely used technique in deep neural network training. There is, however, a major difference when applying it to GNNs because the training nodes (seed nodes) are part of the graph and cannot be considered without their corresponding neighborhood. The neighborhood size is dictated by the model architecture. In most message-passing-based GNNs, the neighborhood required to fully utilize the model capacity includes the L-hop neighbors of the seed nodes, where L is the number of GNN layers. The straightforward implementation would use such a graph through all the layers. However, this is far from optimal as each GNN layer is using only direct neighbors to update each node, and at the end of the forward pass, we need the correct outputs only for the seed nodes. Therefore, a layer with index l requires an $(L-(l-1))$-hop neighborhood of the seed nodes. Moving from one GNN layer to the next in the forward pass, we want to narrow down the neighborhood. On the other hand, we should do the opposite during the backward pass. This is where we use HGAM to allow dynamic resizing of the subgraph. We use the metadata about the sampled nodes and edges to determine how many rows and columns are to be removed from the adjacency matrix after each layer in the forward pass, and how many should be added after each layer in the backward pass.

3. Selecting k-hop Subgraphs

A dense adjacency matrix is not an efficient storage structure for sparse data because it requires $O(N^2)$ mem-

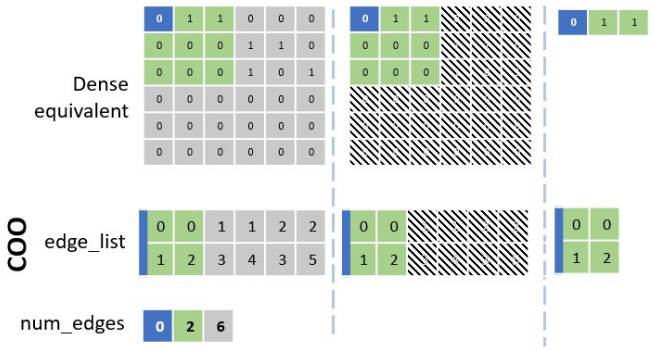

Figure 2: COO pruning.

Figure 3: CSR trimming.

ory, where N is the number of nodes in the graph. In this section, we discuss the use of sparse data formats as a backend for the Hierarchical Graph Adjacency Matrix (HGAM). The two sparse data formats we have integrated with HGAM are the compressed sparse row (CSR) and the coordinate (COO) formats. We will also discuss how to use these data formats to select a k-hop subgraph in constant time, which is essential for making HGAM efficient.

3.1. COO

The coordinate list is a sparse matrix representation where each non-zero element is stored as a triple (src, dst, v), where src and dst are the row and column indices of the element, and v is the value of the element. Trimming a COO matrix is straightforward since both edges and nodes are retrieved in a BFS order; this implies that all the needed nodes for a layer are always at the beginning of the vector generated by the BFS ordering. COO pruning is shown in Figure 2. In the example, we start with a graph that has 2 edges at the first HGAM level and 4 edges at the second level. Restricting the subgraph to a first-level neighborhood is done by cutting off the right part of the edges list. The number of edges that should be left after this operation is stored in the num_edge vector. In this case, it is equal to 2. These are the only edges that connect seed nodes with their direct neighbors. The COO format is efficient for storing a sparse matrix, but it is not efficient for performing matrix operations.

3.2. CSR

The compressed sparse row format is a sparse matrix representation where the non-zero values are stored using row pointers, column indices, and value arrays. Pruning this structure to meet the needs of the k-th layer (counting from the last hierarchical layer) requires several steps. To start with, we need all the (k-1)-hop neighborhood nodes' edges. This includes k-hop node edges, but only those coming into the (k-1)-hop nodes. We know that there will be no edges coming into the nodes in the k-hop neighborhood from the nodes belonging to (k+1)-hops or above, as otherwise, they would, by definition, be k-hop neighbors. We extract the k-hop subgraph by (a) pruning the rowptr to the (k)-hop neighborhood, (b) pruning the cols vector based on the pruned rowptr, and (c) restoring the empty rows for the (k+1)-hop neighborhood by repeating the last rowptr value for all of them, effectively creating what would be rows of zeros in an equivalent dense adjacency matrix. CSR trimming to a 1-hop neighborhood is shown in Figure 3. The first step in the pruning process is to restrict the row pointer array to the number of nodes that will become destination nodes (value at the first position of the num_nodes vector). In the example, that would mean one node. The next step is to prune the column indices and value array to the length given by the last value in the row pointer. That would result in column and value arrays having two elements each. The last part is to extend the row pointer by repeating the last value of this array. The new length of this array is given in the num_nodes vector at the second position.

3.3. Sparse Kernels

Message passing paradigms significantly influence how most GNN models work [21]. The implementation of the message passing algorithm is based mainly on two factors: a graph data structure or format, and a set of appropriate kernels that operate on that format. Two of the most popular combinations are the COO format managed by Scatter kernels and the CSR format used by Sparse Matrix Multiplication (SpMM) kernels. Many researchers focus on optimizing the kernels' implementation by dynamically changing the sparse format [22], improving load balancing [23, 24], or enhancing memory accesses [25]. The optimization of kernels helps to utilize hardware capabilities. Our solution reduces the kernel execution time at a higher level by decreasing the total number of computations that need to be done.

4. GNN Optimizations on Top of HGAM

The additional information about the number of nodes and edges at each level of the hierarchy allows us to further optimize GNNs. In this section, we describe the optimizations that we performed on top of the HGAM data structure. These optimizations, referred to as HGAM++ in the results section, are not universally applicable to every type of GNN layer. Nonetheless, they are suitable for several widely utilized GNN structures. We applied both optimization techniques discussed below to the Graph Convolutional Network (GCN), while the remaining models were optimized using only linear layer pruning.

4.1. Self-connections

Many GNN layers [9,26,27] implement adjacency matrix pre-processing at the beginning of the forward pass. Adding self-connections is one of the steps that can be applied. It is desirable for a node to include knowledge about the previous embedding in the new one. Self-connections ensure that the previous embedding for a node is included in the computations of message passing. Algebraically, this operation is equivalent to setting the diagonal of an adjacency matrix to one. For the furthest nodes in the subgraphs, it would be the only incoming edge for them. They were added to the graph to be a source of information for other nodes and will be removed in the HGAM process of narrowing the graph down. Therefore, these connections are added gratuitously. Having the meta-information about the number of nodes at each level allows us to add only those self-connections that are useful. This saves the time needed to add the extra edges and run the message passing over them.

4.2. Linear Layer Pruning

Layer definitions that follow the Gather-Apply-Scatter and Scatter-Apply-Gather [28] paradigms can contain some type of feature matrix transformation in the *apply* step. Similar to self-connections, we want to transform only those nodes that will not be removed just after the forward pass of the current layer is done. This is possible only if we can switch the order of Apply-Scatter and run the message passing before transforming the node features. Some GNN layers are defined in a way that the order does not influence the results (e.g., GCN [21]), but some are not (e.g., GraphSage-max [9]). The general rule is that this optimization can be used wherever the linear layer occurs after the message passing.

Table 1: Datasets used in the experiments.

Dataset	Nodes	Edges	Features	Classes
products	2,449,029	61,859,140	100	47
Reddit	232,965	114,615,892	602	50
mag	1,939,743	21,111,007	128	349

5. Experiments

In this section, we present the empirical results obtained from the HGAM implementation in PyTorch Geometric [29]. We compare the performance of HGAM, with and without the extra optimizations described in Section 4, to the current PyG implementation. The CPU experiments were performed on a third-generation Intel Xeon Scalable processor (Intel® Xeon® Platinum 8360Y) with 36 cores and 512GB of memory. The GPU data were collected using the same host machine with an Nvidia A100 80GB GPU. We selected the following datasets: *ogbn-products*, *Reddit*, and *ogbn-mag*. The datasets are described in Table 1. We believe these datasets are representative of a large range of graph learning applications. All datasets were downloaded from the OGB repository [8].

For model selection, we aimed to cover a wide range of modern graph neural network models. We chose both isotropic and anisotropic models for homogeneous graphs. The models used in our experiments include GCN [26], GraphSAGE [9], GAT [30], and PNA [31]. Additionally, a heterogeneous HGAM was benchmarked on CPU using RSage [32]. We utilized the original implementation of these models from PyG with default parameters.

5.1. Results

5.1.1 Speedup Analysis

The CPU results for 3-layer deep GNNs with 10 neighbors sampled at each level and CSR sparse data format are shown in Table (2). As shown, native HGAM implementations (the HGAM column) are faster than the current PyG implementations for all datasets and models. The maximum speedup for CSR varies between 1.49x and 3.94x for native HGAM and between 1.9x and 7.72x for HGAM with linear layer optimization (HGAM++).

The best speedup observed for GPU (Table 3) is for PNA on the ogbn-products dataset. Conversely, HGAM and HGAM++ result in a slowdown for the computation of Sage. We will provide a more detailed explanation for this behavior in the next section.

It should be highlighted that CPUs (Table 2) generally perform better with the Compressed Sparse Row (CSR) format, whereas GPUs tend to have faster execution with the Coordinate List (COO) format in the majority of instances.

Table 2: End-to-end CPU performance comparison for CSR - one epoch training time averaged over 5 runs (in seconds).

		CPU time (s)		
Dataset	Model	PyG	HGAM	HGAM++
products	GCN	260.3±3	142.5±0.7	61.6±0.4
products	Sage	276.4±4.8	118±1.1	59.7±0.6
products	GAT	1191±14	616.7±3.8	336.6±1
products	PNA	5440±23	1379±2.8	1068±2
Reddit	GCN	129.2±0.8	86.6±1.1	52.9±0.9
Reddit	Sage	141.7±4	85.2±2.9	55.3±2.2
Reddit	GAT	600±23	345.1±6.5	224±16
Reddit	PNA	4799±13	2887±3	2520±12
mag	RSage	4040±30	1377±9	523±1

Table 3: End-to-end GPU performance comparison for CSR - one epoch training time averaged over 5 runs (in seconds).

		GPU time (s)		
Dataset	Model	PyG	HGAM	HGAM++
products	GCN	58.2±0.4	56.4±0.7	53.6±0.4
products	Sage	52.7±0.7	55±0.6	52.8±0.2
products	GAT	60.9±1	57.7±0.7	57.6±0.5
products	PNA	162±0.5	71.7±0.5	61±0.1
Reddit	GCN	44.5±0.3	43.4±0.5	44.2±0.3
Reddit	Sage	41.7±1	44.1±1	43±0.4
Reddit	GAT	45.9±0.6	46±0.6	44.1±0.1
Reddit	PNA	148.3±0.6	103.6±1	74.9±0.6

This observation underscores the need to implement and optimize both formats.

5.1.2 Runtime Breakdown for Several Use-Cases

Beginning with the GPU, the time distribution for data loading, offloading, and the execution of forward and backward passes is presented in Figure 4. For GCN, GraphSage, and GAT, the data loading phase, which is executed on the CPU, accounts for the bulk of the processing time. Our enhancements do not impact the data loading stage, and as a result, the time spent in this phase remains unchanged across the baseline, HGAM, and HGAM++. The forward and backward computations were performed on the GPU. While we do observe some performance improvements in the forward and backward passes, they constitute a minor portion of the total runtime, so the overall speedup is not substantial. PNA is different, as forward and backward computations are the primary contributors to the total processing time.

The breakdown of CPU runtime is presented in Figure 5. The data indicates that the implementation of HGAM, with or without extra optimizations, positively affects the performance of both forward and backward passes. The time saved during the backward pass results from the linear optimization techniques incorporated in HGAM++. These techniques rearrange the order of message passing and linear transformations. Consequently, in the case of GCN, the message passing phase doesn't require a backward

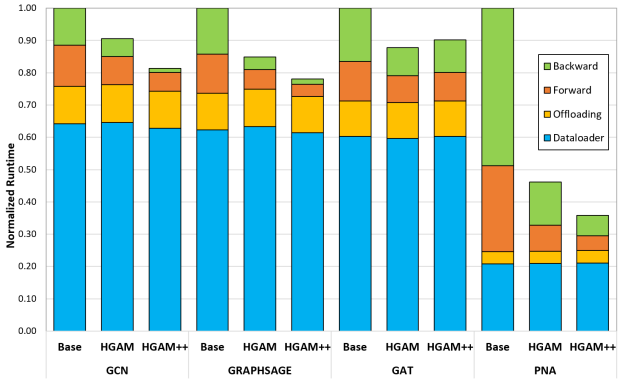

Figure 4: GNN training time breakdown of dataloader, forward, and backward passes for ogbn-products using CSR data format on GPU.

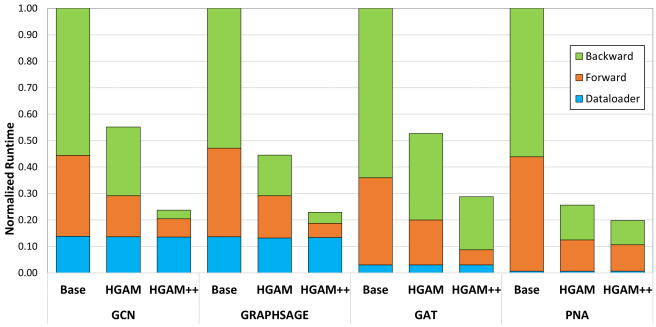

Figure 5: GNN training time breakdown of dataloader, forward, and backward passes for ogbn-products using CSR data format on CPU.

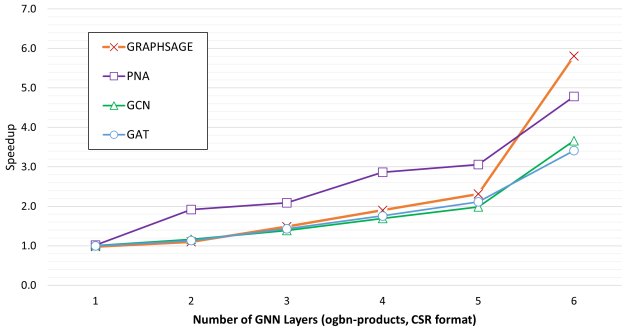

Figure 6: Speedup with HGAM on ogbn-products (CSR format) for $k_l = 2$ with varying numbers of layers.

operation for the first layer because it's the foremost operation and doesn't involve any learnable parameters, which reduces the overall time required for the backward pass.

The (CSR, ogbn-products, PNA-base) configuration achieves a backward pass time reduction of over 3.5 times and a forward pass time reduction of over 2.7 times. Meanwhile, the (CSR, Reddit, GAT) configuration exhibits a 2.6-fold decrease in backward pass time and a 4.7-fold improvement in forward pass time. The effects of these optimizations contrast between CPUs and GPUs, with the most computationally demanding models showing greater improvements on GPUs, whereas CPUs experience varying but significant gains in all cases.

Using HGAM, we decrease the number of nodes pro-

cessed in the next layer exponentially, since this is the speed in which the subgraph grows. Only the first layer needs to be computed with the whole subgraph which makes all the others computationally less important. Figure 6 shows that the speedup increase with the number of layers. This trend is expected because the runtime for both the baseline and the optimized versions starts off comparably at the first layer and then decreases exponentially with each subsequent layer for HGAM and HGAM++ but stays at the same level for the baseline.

5.1.3 The impact of additional optimizations

Applying HGAM shifts the GNN training bottleneck to the first layer, which can be observed in Figure 7. These four charts present the forward time for each layer in the network separately. The additional optimization we introduced with hgam++ works mostly for all layers but our intent was to improve the performance of the first layer with the extra knowledge that comes from the HGAM structure. An counter-intuitive results can be observed for GCN on Reddit. In this case hgam++ gives us significant slowdown. This is expected, since Reddit comes with huge input features of 602 dimensions. Calculating message passing before linear transformation forces the model to perform all the reductions in the input feature space. It is more expensive than it would be if we first project the feature to a lower dimensional space using linear transformation. Despite the extra cost, we achieve better end-to-end performance because we greatly reduce the backward propagation which is not covered by this visualization, but can be observed in Table 2.

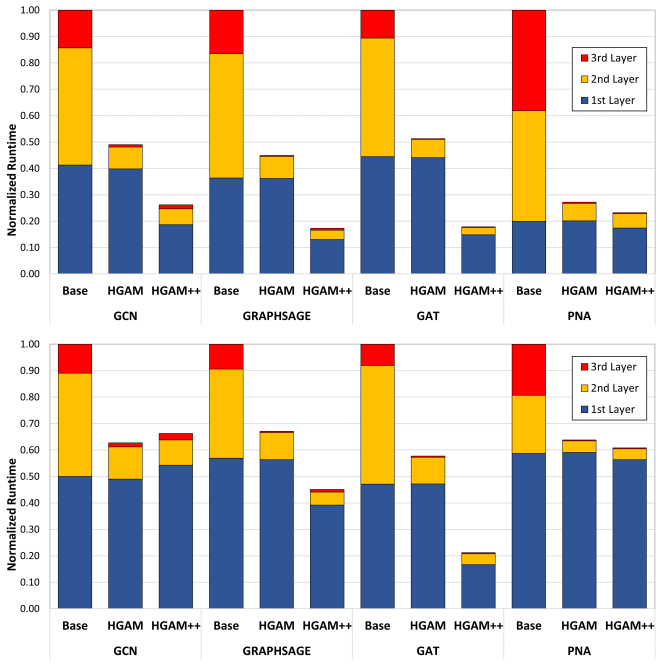

Figure 7: Forward pass split by layers in 3-layer deep graph neural networks on ogbn-products (top) and Reddit (bottom) dataset.

6. Known Limitations

Our Hierarchical Graph Adjacency Matrix (HGAM) was proposed to improve performance for all mini-batched message-passing-based GNN workloads. We did not encounter any problems using it for GNN models built with one homogeneous type of neural network layer (e.g., GCN layers) and an appropriate, directed subgraph for the mini-batch. However, more specialized architectures and sampling strategies may not fully benefit from HGAM. For these special circumstances, additional work may be required to make them compatible with HGAM. We split the known limitations into two categories and describe them in the following sections.

6.1. Incompatible Architectures

HGAM stores the number of nodes and edges at each subgraph level as additional information regarding the graph. There is a hidden assumption that this information does not change after the creation of minibatches. This assumption could be violated if we apply some topological operations on the graph that change the total number of edges [33]. For example, if we remove some paths from the graph or add a new connection, the number of edges in the subgraph will be different, and the meta-information of the graph will be incorrect. Therefore, such operations need to be done prior to the creation of the minibatch if possible, or special treatment is needed to ensure that the support nodes and edges will be correctly pruned. A common operation that can raise problems is the addition of self-connections to the graph to make every node aware of its own embedding. Although this operation needs special treatment, we found that it can be implemented much more efficiently with HGAM. The details were described in Section 4.

Another family of operations that can cause issues are those that change the number of nodes in the graph. Reducing the number of nodes in a graph is often used in graph prediction tasks [34] but can also be used to reduce the problem size [35]. We call these operations local pooling operations, and their goal is to reduce the number of nodes in the graph by merging groups of them into a single node. After the pooling operation, the number of nodes and edges in the graph is reduced. Therefore, an HGAM restoration is needed to ensure that the meta-information of the graph is correct. Alternatively, global nodes or global means over all nodes are often used for graph prediction or classification tasks. Adding an extra node to the graph and connecting it to all other nodes is not compatible with HGAM because the new node is a seed node and needs to receive messages from all nodes in the subgraph at each layer. Calculating the final embedding for graph classifica-

tion or regression as the mean of all nodes is also infeasible for HGAM, since all nodes in that scenario are treated as seed nodes. Thus, it is not possible to prune the adjacency matrix at any stage of the computations.

6.2. Incompatible Sampling Strategies

The sampling strategy is a crucial part of the minibatch creation process. It is responsible for selecting the seed nodes and their neighborhood nodes that will be included in the minibatch. In all of our experiments, we used k-hop neighborhoods for k-layer deep GNNs and restricted our sampling strategy to directed subgraphs. However, there are some other sampling strategies that could limit the performance benefits from using HGAM.

More Layers than Neighborhood Levels

Adding more layers to the model is often motivated by the desire to increase model capacity. The downside of a greater number of GNN layers, which increases the receptive field exponentially, is that more neighbors must be added to the minibatch to ensure that the model can fully utilize the new potential. The additional memory and computational requirements can render the hardware impractical. Therefore, some machine learning practitioners [36] choose to increase the model depth while setting an upper limit for the number of neighbors sampled. In that case, HGAM can still provide computational benefits in performance, but such benefits will be curtailed as the first layers will use the full neighborhood sampled, and only a subset of the last layers will use increasingly smaller parts of it.

Undirected Minibatch Graph

A different challenge can arise with an undirected minibatch graph. The undirected minibatch graph is created by a bidirectional connection when a new node is sampled. This only makes sense in the settings described above when the number of GNN layers is greater than the number of subgraph levels. In that case, the undirected minibatch graph can be used to train more appropriate node embeddings without adding additional nodes to the minibatch. The problem with an undirected minibatch graph is that the BFS-based approach does not ensure that the support nodes added at level l are at distance l from the seed nodes. Therefore, using an undirected minibatch graph would require an extra reordering of the nodes after the creation of minibatches.

7. Discussion and Conclusions

Training large neural networks is a time-consuming task that requires significant computational resources. Graph neural networks (GNNs) are a special class of neural networks that need specialized and novel techniques to improve training efficiency for large graphs. Our work demonstrates that the minibatch training of GNNs can be improved by using a novel data structure called the Hierarchical Graph Adjacency Matrix (HGAM). We showed that HGAM can be used to reduce the training time by up to 7.72x on 3-layer GNNs with no loss in accuracy.

Performance improvements were achieved by removing all unnecessary nodes and edges from the graph at every stage during training. We also proposed how the additional structural information stored in HGAM can be leveraged to further improve performance by adding self-connections, message passing, and linear transformation in GNN layers. The cumulative effect of applying all these techniques is a significant reduction in the computational resources required for training GNNs by exponentially improving the computational efficiency of all additional layers after the first. The exponential reduction of nodes and edges makes the first layer responsible for most of the forward and backward time. Boosting the performance of the first layer is our primary goal for future work.

Another expensive operation in minibatch training pertains to the sparse data format employed. Currently, HGAM supports CSR and COO sparse data formats as the underlying data structures. We found CSR to be much more efficient for SpMM computation on CPU, but it comes with a high cost in transposition, which is needed for backward computation. We are actively looking for a sparse data format that will be efficient for both SpMM and transposition. Similarly, we hope others will be able to build on this work to propose new formats for HGAM and HGAM-derived data structures, which will bring further performance and efficiency improvements.

The scope of the research presented in the paper did not extend to assessing performance across multiple GPUs or machines. The technique we discussed, HGAM, was not designed to reduce the amount of data that needs to be offloaded, since the first layer requires all the nodes to be present. However, it is important to note that HGAM can still enhance the efficiency of both the forward and backward processes on each individual machine, just as it does within a single machine setup. Additionally, it can decrease the amount of data transferred between machines when model parallelism is used.

Finally, HGAM has been developed to be as domain-agnostic as possible. The same principles for computational efficiency can be applied to any problem that in-

volves processing graph datasets with concepts equivalent to the seed nodes and support nodes of GNNs, as well as the requirement to iterate between varying widths of neighborhoods during processing. It is our intent to share HGAM with the open-source community to not only benefit graph learning practitioners but also have it adapted to new, unexplored domains beyond GNNs.

References

[1] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[2] J. Wang, S. Zhang, Y. Xiao, and R. Song, "A review on graph neural network methods in financial applications," 2022.

[3] Y. Wang, Z. Li, and A. B. Farimani, "Graph neural networks for molecules," in *Challenges and Advances in Computational Chemistry and Physics*, pp. 21–66, Springer International Publishing, 2023.

[4] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui, "Graph neural networks in recommender systems: A survey," *ACM Comput. Surv.*, vol. 55, dec 2022.

[5] W. Liao, B. Bak-Jensen, J. R. Pillai, Y. Wang, and Y. Wang, "A review of graph neural networks and their applications in power systems," 2021.

[6] A. Cattaneo, D. Justus, H. Mellor, D. Orr, J. Maloberti, Z. Liu, T. Farnsworth, A. Fitzgibbon, B. Banaszewski, and C. Luschi, "Bess: Balanced entity sampling and sharing for large-scale knowledge graph completion," 2022.

[7] W. Hu, M. Fey, H. Ren, M. Nakata, Y. Dong, and J. Leskovec, "Ogb-lsc: A large-scale challenge for machine learning on graphs," 2021.

[8] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," 2021.

[9] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2018.

[10] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," in *International Conference on Learning Representations*, 2018.

[11] J. Gasteiger, C. Qian, and S. Günnemann, "Influence-based mini-batching for graph neural networks," 2022.

[12] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," 2018.

[13] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, ACM, jul 2019.

[14] F. Frasca, E. Rossi, D. Eynard, B. Chamberlain, M. Bronstein, and F. Monti, "Sign: Scalable inception graph neural networks," 2020.

[15] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graphsaint: Graph sampling based inductive learning method," 2020.

[16] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken, "Redundancy-free computation graphs for graph neural networks," 2019.

[17] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen, "Understanding and bridging the gaps in current gnn performance optimizations," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, (New York, NY, USA), p. 119–132, Association for Computing Machinery, 2021.

[18] Z. Gong, H. Ji, Y. Yao, C. W. Fletcher, C. J. Hughes, and J. Torrellas, "Graphite: Optimizing graph neural networks on cpus through cooperative software-hardware techniques," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, (New York, NY, USA), p. 916–931, Association for Computing Machinery, 2022.

[19] M. J. Adiletta, J. J. Tithi, E.-I. Farsarakis, G. Gerogiannis, R. Adolf, R. Benke, S. Kashyap, S. Hsia, K. Lakhotia, F. Petrini, G.-Y. Wei, and D. Brooks, "Characterizing the scalability of graph convolutional networks on intel® piuma," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 168–177, 2023.

[20] B. Zhang, S. R. Kuppannagari, R. Kannan, and V. Prasanna, "Efficient neighbor-sampling-based gnn training on cpu-fpga heterogeneous platform," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2021.

[21] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," 2017.

[22] S. Qiu, Y. Liang, and Z. Wang, "Optimizing sparse matrix multiplications for graph neural networks," 2021.

[23] M. Guo, Y. Wang, J. Huang, Q. Wang, Y. Zhang, M. Xu, and F. Lu, "Rgs-spmm: Accelerate sparse matrix-matrix multiplication by row group splitting strategy on the gpu," in *Network and Parallel Computing* (S. Liu and X. Wei, eds.), (Cham), pp. 61–66, Springer Nature Switzerland, 2022.

[24] M. Shan, D. Gurevin, J. Nye, C. Ding, and O. Khan, "Mergepath-spmm: Parallel sparse matrix-matrix algorithm for graph neural network acceleration," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, (Los Alamitos, CA, USA), pp. 145–156, IEEE Computer Society, apr 2023.

[25] G. Huang, G. Dai, Y. Wang, and H. Yang, "Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks," 2020.

[26] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017.

[27] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," in *International Conference on Learning Representations*, 2019.

[28] Z. Zhang, J. Leng, L. Ma, Y. Miao, C. Li, and M. Guo, "Architectural implication of graph neural networks," *IEEE Computer Architecture Letters*, pp. 1–1, 2020.

[29] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," 2019.

[30] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[31] G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Veličković, "Principal neighbourhood aggregation for graph nets," in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), vol. 33, pp. 13260–13271, Curran Associates, Inc., 2020.

[32] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," 2017.

[33] E. Dai, W. Jin, H. Liu, and S. Wang, "Towards robust graph neural networks for noisy graphs with sparse labels," 2022.

[34] C. Liu, Y. Zhan, J. Wu, C. Li, B. Du, W. Hu, T. Liu, and D. Tao, "Graph pooling for graph neural networks: Progress, challenges, and opportunities," 2023.

[35] D. Grattarola, D. Zambon, F. M. Bianchi, and C. Alippi, "Understanding pooling in graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–11, 2022.

[36] G. Li, M. Müller, B. Ghanem, and V. Koltun, "Training graph neural networks with 1000 layers," 2022.