Accelerating Large Scale Real-Time GNN Inference using Channel Pruning

Hongkuan Zhou
Ajitesh Srivastava
Hanqing Zeng
University of Southern California
Los Angeles, USA
{hongkuaz,ajiteshs,zengh}@usc.edu

Rajgopal Kannan
US Army Research Lab
Los Angeles, USA
rajgopal.kannan.civ@mail.mil

Viktor Prasanna
University of Southern California
Los Angeles, USA
prasanna@usc.edu

ABSTRACT

Graph Neural Networks (GNNs) are proven to be powerful models to generate node embedding for downstream applications. However, due to the high computation complexity of GNN inference, it is hard to deploy GNNs for large-scale or real-time applications. In this paper, we propose to accelerate GNN inference by pruning the dimensions in each layer with negligible accuracy loss. Our pruning framework uses a novel LASSO regression formulation for GNNs to identify feature dimensions (channels) that have high influence on the output activation. We identify two inference scenarios and design pruning schemes based on their computation and memory usage for each. To further reduce the inference complexity, we effectively store and reuse hidden features of visited nodes, which significantly reduces the number of supporting nodes needed to compute the target embedding. We evaluate the proposed method with the node classification problem on five popular datasets and a real-time spam detection application. We demonstrate that the pruned GNN models greatly reduce computation and memory usage with little accuracy loss. For full inference, the proposed method achieves an average of 3.27× speedup with only 0.002 drop in F1-Micro on GPU. For batched inference, the proposed method achieves an average of 6.67× speedup with only 0.003 drop in F1-Micro on CPU. To the best of our knowledge, we are the first to accelerate large scale real-time GNN inference through channel pruning.

1 INTRODUCTION

Recently, Graph Neural Networks (GNNs) have attracted the attention of many AI researchers due to the high expressive power and

generalizability of graphs in many applications. The node embedding generated from GNNs outperforms other graph representation learning methods when fed into downstream applications like node classification, edge prediction, and graph classification. Table 1 shows some popular applications of GNNs on various size graphs with different latency requirements. The knowledge graphs used in few-shot learning could only contain around one hundred of nodes and hundreds of edges, while the social network graphs could have billions of nodes and trillions of edges. Most of these GNN applications are latency sensitive at inference. For example, the applications related to Computer Vision need to perform streaming real-time inference on the data captured by the cameras. The applications related to fraud and spam detection need to identify malicious posts and transactions as fast as possible to avoid the property loss of the victim users. In addition to latency, some vision applications that utilize GNNs [8] need to perform inference on edge devices with limited computing power and memory, such as self-driving cars with 3D-cameras and radars.

Compared with traditional graph analytics algorithms, GNNs have high computation cost as one node needs to gather and aggregate feature vectors from all the neighbors in its receptive field to compute a forward pass. To accelerate the training of GNNs, many works [4, 5, 13, 42] adopt stochastic node sampling techniques to reduce the number of supporting neighbors. GraphNorm [3] normalizes the node attributes to speedup the convergence. As a result, GNNs training scales well with graph size. It only takes seconds to minutes to train on a graph with millions of nodes. However, many GNN applications struggle at inference when deployed to
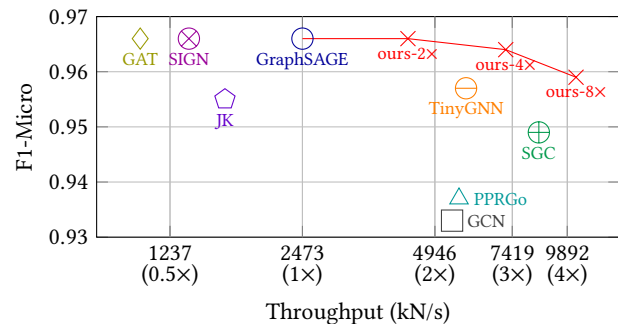

Figure 1: Accuracy and throughput of full inference on the Reddit dataset on GPU.

Table 1: GNN Applications with their conventional graph sizes (in number of nodes) and latency requirement.

Applications	Nodes	Lat.
Knowledge Graph		
Few-shot image classification [10, 23]	$10^2 - 10^3$	ms
Relation extraction and reasoning [24]	$10^3 - 10^6$	ms-s
Image Graph		
Point cloud segmentation [6, 30]	$10^3 - 10^6$	ms
Spatio-Temporal Graph		
Traffic prediction [11]	$10^3 - 10^6$	s
Action recognition [7, 18]	$10^2 - 10^3$	ms
Social Network Graph		
Recommending system [9, 39, 44]	$10^6 - 10^9$	ms
Spam detection [17, 19, 29]	$10^6 - 10^9$	ms

production environment. Performing the full forward pass with all the neighbors at inference leads to high memory usage and latency. The node sampling techniques, when applied to inference, struggle to maintain high accuracy on every sample. In consequence, GNN applications either turn to traditional graph analytics algorithms with lower complexity, or rely on obsolete (not updated recently) embedding. For example, Youtube [12] turns to label propagation to detect abusive videos. Pinterest [39] has to use obsolete embedding generated with the MapReduce framework in an offline process. Taobao [19] runs the GNN based malicious account detection daily, instead of immediately after one transaction pops. Even with the compromise of offline inference, GNN inference is still expensive on large graphs. It is reported that a cluster with 378 computing nodes still needs one day to generate embedding for 3 billion nodes [39]. In addition, GraphBERT [45] shows that pre-trained GNN models could be directly (or with light fine-tuning) transferred to address new tasks, which makes accelerating GNN inference more important.

Although it has not caught much attention of researchers, accelerating GNN inference is as important as accelerating GNN training. Based on these GNN applications, we define two inference scenarios – full inference where the target nodes are all the nodes (or a large portion of nodes, i.e., the test set) in the graph, and batched inference where the target nodes are a few nodes. Full inference applies to GNN applications that operate on small to medium size graphs, or perform offline inference on large graphs. Batched inference applies to GNN applications that have strict requirements on latency, or need to be executed on edge devices such as embedded systems and FPGAs. Full inference performs forward propagation on all the nodes in the graph, while batched inference only propagates from the selected supporting nodes of the target nodes. For batched inference, the number of supporting nodes grows exponentially with the number of GNN layers, which is referred to as the "neighbor explosion" problem. In this work, we propose to accelerate GNN inference by reducing the input feature dimensions in each GNN layer and reusing the hidden features for visited nodes.

Our pruning framework works on most GNN architectures and can significantly improve their inference throughput with little or no loss in accuracy. The main contributions of this work are

- We develop a novel LASSO regression formulation to prune input channels for GNN layers, which outperforms random and greedy pruning methods.
- We design different pruning schemes for full inference and batched inference addressing their computation complexity and memory usage.
- We develop a novel technique to store and reuse the hidden features of visited nodes for batched inference, which mitigates the "neighbor explosion" problem.
- We evaluate the performance of the pruned models on five popular datasets and a real-time spam detection application. The pruned GNN models greatly reduce the complexity and memory usage with negligible accuracy loss. For full/batched inference, the pruned models reduce the computation to 0.19×/0.10× and memory requirements to 0.43×/0.18× with only 0.002/0.003 F1-Micro drop on average. The pruned models achieve an average of 3.27×/6.67× speedup for full/batched inference on GPU/CPU.

2 BACKGROUND

2.1 Graph Neural Networks

For a graph $G(V, E)$ where each node $v \in V$ has node attributes $\boldsymbol{h}(v) \in \mathbb{R}^f$, GNNs iteratively gather and aggregate information from neighbors to compute node embedding. Denote the matrix of all the output features $\boldsymbol{h}^{(i)}(v)$ stacked horizontally in layer-i by $\boldsymbol{h}^{(i)}$. Let $\tilde{A}$ be the normalized adjacency matrix. In general, the output features $\boldsymbol{h}^{(i)}$ of layer-i is computed by

$$\boldsymbol{h}^{(i)} = \sigma \left(\overset{K}{\underset{k=K'}{\|}} \tilde{A}^k \boldsymbol{h}^{(i-1)} \boldsymbol{W}_k^{(i)} \right) \tag{1}$$

where $\|$ denotes the horizontal concatenation operation. $\boldsymbol{W}_k^{(i)}$ is the learnable weight matrix of order k in layer-i. And $\sigma(\cdot)$ denotes the ReLU activation. We stack multiple layers and let the input of the first layer $\boldsymbol{h}^{(0)} = \boldsymbol{h}$ to compute the node embedding. For $K' = K = 1$, Equation 1 shows the forward propagation of vanilla Graph Convolutional Network [16]. For $K' = 0, K = 1$, Equation 1 is the GraphSAGE [13] architecture. For $K' = 0, K > 1$, Equation 1 is the MixHop [1] architecture. For other variants of GNNs [28, 33, 34], Equation 1 could be adapted by adding residue connections or alternating the normalized adjacency matrix.

2.2 Case Study: GraphSAGE Inference

We perform a case study to analyze the complexity and memory usage for both inference scenarios on the widely used GraphSAGE architecture. We choose to analyze the GraphSAGE architecture as it achieves top tier accuracy with relatively high throughput (see Figure 1). For the GraphSAGE architecture, $K' = 0, K = 1$ and the adjacency matrix A is normalized by $\tilde{A} = D^{-1}A$ where D is the diagonal degree matrix.
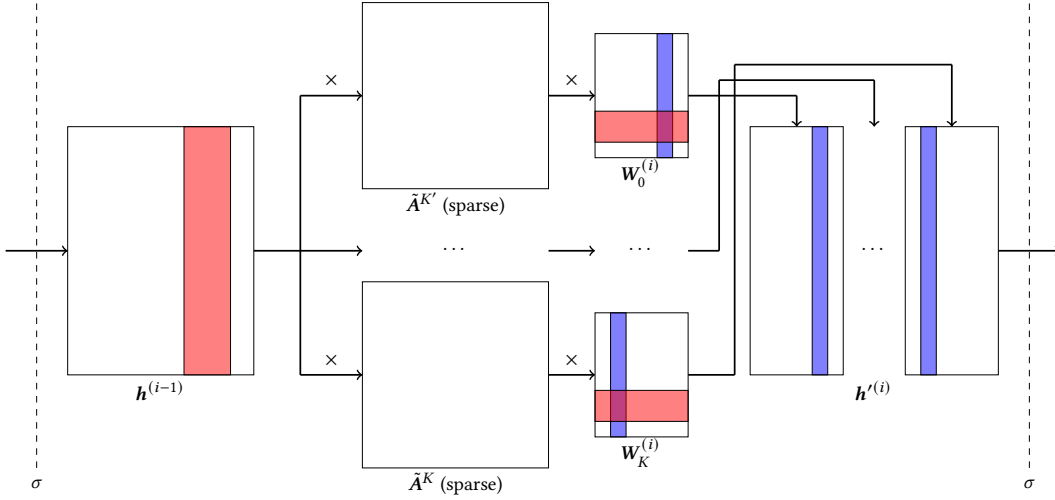
Figure 2: Illustration of one pruned GNN layer. The $\times$ operator denotes sparse or dense matrix multiplication while the shaded areas denote the pruned channels. The blue areas in the weight matrices $W_k^{(i)}$ and the output features $h'^{(i)}$ before activation show the pruned channels in the next GNN layer. The red areas in weight matrices $W_k^{(i)}$ and the input features $h^{(i-1)}$ show the pruned channels in this GNN layer.

2.2.1 Full Inference. To perform full inference that computes node embedding for all the nodes in the graph, we batch the node-wise aggregation and compute sparse-dense matrix multiplication $\tilde{A} \cdot h^{(i-1)}$. Denote the input and output feature dimensions of the weight matrices $W_k^{(i)}$ by $f_k^{\text{in}(i)}$ and $f_k^{\text{out}(i)}$ (all input feature dimensions are equal in each layer). Let $|V|$ be the number of nodes in the graph. Assume the average degree of the whole graph is d. The average complexity per node $C_{\text{full}}^{(i)}$ and total memory consumption $M_{\text{full}}^{(i)}$ of full inference are

$$C_{\text{full}}^{(i)} = O\left(d\min(f_1^{\text{in}(i)}, f_1^{\text{out}(i)}) + \sum_{k=0}^{1} f_k^{\text{in}(i)} f_k^{\text{out}(i)}\right)$$

$$M_{\text{full}}^{(i)} = |V|\left(f_0^{\text{in}(i)} + f_0^{\text{out}(i)} + f_1^{\text{out}(i)}\right) + \sum_{k=0}^{1} f_k^{\text{in}(i)} f_k^{\text{out}(i)} \tag{2}$$

As the output features of all nodes are computed in every layer, the computation and memory consumption distribute evenly in each layer. Each branch in one layer also contributes to a non-negligible portion of the computation and memory usage.

2.2.2 Batched Inference. For batched inference, the GraphSAGE architecture aggregates from L-hop neighbors. Denote the set of target nodes to infer by V_t. In layer-i, the average number of supporting nodes is $|V_t| \sum_{l=0}^{L-i+1} d^l$, which leads to the average complexity per node dominated by the complexity in the last layer

$$C_{\text{batched}} = \sum_{i=1}^{L} C_{\text{batched}}^{(i)} = \sum_{i=1}^{L} \sum_{l=0}^{L-i} d^l C_{\text{full}}^{(i)} = O(d^{L-1} C_{\text{full}}^{(1)}) \tag{3}$$

Similarly, the memory consumption also peaks in the first layer with the most supporting neighbors.

2.3 Related Work

There are many existing works on channel pruning in Deep Neural Networks. The works [14, 31] prune the channels in the convolution layer by applying penalized regression on the input channels. ThiNet [20] prunes the channels based on statistics from the next layer. The work [38] forces some channels to freeze during the training and remove them at inference. Unlike performing inference on texts or images where each instance is independent with the others, inference of nodes depends on the graph structure and attributes of other supporting nodes. The computation pattern is also different for different inference scenarios. These two challenges make it hard to directly apply the existing channel pruning techniques on GNNs. Recently, several works [22, 32] have tried to accelerate training and inference of GNNs by removing the nonlinearity of internal layers and pre-computing the feature aggregation ($A^k h$). PPRGo [2] accelerates inference by performing less aggregation as in training. These methods require pre-processing on either the node attributes or the adjacency matrix, which do not apply to evolving graphs. TinyGNN [36] speeds up inference by training a shallow student GNN supervised by a teacher GNN. Recently, several works have tried to accelerate the full batch propagation in Equation 1 through matrix partitioning [41], node re-ordering [43], and runtime scheduling [26]. Others have developed hardware accelerators [37, 40] and in-memory processors [25]. These hardware-specific optimization techniques do not address the basic problem – high computation complexity of GNNs. On the other hand, although not aiming at rapid inference, some works [35, 46] propose to prune the edges to reduce the noise aggregated from neighbors. However, they are limited to knowledge graphs with specific inference queries.

In contrast, we propose a general method to reduce the inference complexity by directly pruning the input channels. Our method

works for all types of graphs and most GNN architectures. Combined with edge pruning methods and architecture simplification methods, our method has the potential to further speed up the inference. In addition, our pruning method does not incur extra sparse operations. The dimensions of the matrix operations are also lower, which makes it easier to design hardware accelerators and in-memory processors.

3 APPROACH

In GNNs, channels refer to column vectors in the hidden features matrix $h^{(i)}$. We propose to solve the channel pruning problem by applying LASSO regression [27] directly on the input channels. For a pre-trained GNN model, we prune the channels reversely from the output layer to the input layer. Figure 2 shows one pruned GNN layer with multiple branches. In this section, we first introduce the formulation and optimization of channel pruning in a single layer. Then, we discuss the pruning schemes for full inference and batched inference. For batched inference, we further propose a novel technique that stores the hidden features for visited nodes and aggregates directly from them during inference.

3.1 Single Branch Pruning

To prune the channels, we aim to generate the same output features in a branch before activation $h'^{(i)}$ with fewer input dimensions. We focus on $h'^{(i)}$ instead of $h^{(i)}$ to keep all operations linear. For branch k ($K' \leq k \leq K$) in layer-i, let $c_k^{(i)} = f_k^{\text{in}(i)}$ be the number of channels in the original GNN. We formulate the channel pruning problem with budget $\eta_k^{(i)}$ as the following optimization problem

$$\underset{\hat{\beta}_k^{(i)}, \widehat{W}_k^{(i)}}{\arg\min} \left\| Y_k^{(i)} - \tilde{A}^k h^{(i-1)} \odot \hat{\beta}_k^{(i)} \widehat{W}_k^{(i)} \right\|_2^2$$
$$\text{subject to } \left\| \hat{\beta}_k^{(i)} \right\|_0 \leq \eta_k^{(i)} c_k^{(i)} \tag{4}$$

where $Y_k^{(i)} = \tilde{A}^k h^{(i-1)} W_k^{(i)}$ is the target output features. $\|\cdot\|_2$ is the L2-norm and $\|\cdot\|_0$ is the L0-norm measuring the number of non-zero elements. $\hat{\beta}_k^{(i)} \in \mathbb{R}^{c_k^{(i)}}$ is the coefficient vector acting as masks for each channel. $\odot$ denotes element-wise multiplication on each row of the matrix. If $\hat{\beta}_k^{(i)}(j) = 0$, then the jth channel of the input features $h^{(i-1)}$ can be removed. To solve the optimization problem, we first relax the L0-norm to L1-norm and add a penalty term with penalty factor λ.

$$\underset{\hat{\beta}_k^{(i)}, \widehat{W}_k^{(i)}}{\arg\min} \left\| Y_k^{(i)} - \tilde{A}^k h^{(i-1)} \odot \hat{\beta}_k^{(i)} \widehat{W}_k^{(i)} \right\|_2^2 + \lambda \left\| \hat{\beta}_k^{(i)} \right\|_1 \tag{5}$$

We separate the optimization of $\hat{\beta}_k^{(i)}$ and $\widehat{W}_k^{(i)}$ into two sub-problems and optimize on the sub-problems iteratively to find the global minimum. Initially, $\widehat{W}_k^{(i)} = W_k^{(i)}$ and $\hat{\beta}_k^{(i)} = \mathbb{1}$. We optimize both sub-problems on the hidden features of the training nodes. We use the training graph as the normalized adjacency matrix during optimization to avoid information leak.

3.1.1 Optimization on $\hat{\beta}_k^{(i)}$. To optimize $\hat{\beta}_k^{(i)}$, $\widehat{W}_k^{(i)}$ is fixed. The problem becomes a classic LASSO regression problem with "large n,

small p". We solve the LASSO regression with Stochastic Gradient Descent (SGD). Due to the L1-norm term in the constraint, some mask values in the solution of $\hat{\beta}_k^{(i)}$ would shrink to zero, leading to the removal of the corresponding channels.

$$\underset{\hat{\beta}_k^{(i)}}{\arg\min} \left\| Y_k^{(i)} - Z_k^{(i)} \odot \hat{\beta}_k^{(i)} \right\|_2^2 + \lambda \left\| \hat{\beta}_k^{(i)} \right\|_1 \tag{6}$$

where $Z_k^{(i)} = \tilde{A}^k h^{(i-1)} \widehat{W}_k^{(i)}$.

3.1.2 Optimization on $\widehat{W}_k^{(i)}$. To optimize $\widehat{W}_k^{(i)}$, $\hat{\beta}_k^{(i)}$ is fixed. The problem becomes a quadratic programming problem

$$\underset{\widehat{W}_k^{(i)}}{\arg\min} \left\| Y_k^{(i)} - X_k^{(i)} \widehat{W}_k^{(i)} \right\|_2^2 \tag{7}$$

where $X_k^{(i)} = \tilde{A}^k h^{(i-1)} \odot \hat{\beta}_k^{(i)}$. The closed-form least square solution is given by $\widehat{W}_k^{(i)} = (X_k^{(i)\top} X_k^{(i)})^{-1} X_k^{(i)\top} Y_k^{(i)}$.

3.2 Single Layer Pruning

To prune the input channels for one layer with multiple branches, we need to ensure each branch shares the same pruned channels in $h^{(i-1)}$ so that these channels could be removed in the output of the previous layer. We enforce that the same channels are pruned in each branch by applying a shared coefficient vector $\hat{\beta}^{(i)}$ and jointly optimizing on all the branches.

$$\underset{\hat{\beta}^{(i)}, \widehat{W}_k^{(i)}}{\arg\min} \sum_{k=K'}^{K} \left\| Y_k^{(i)} - \tilde{A}^k h^{(i-1)} \odot \hat{\beta}^{(i)} \widehat{W}_k^{(i)} \right\|_2^2 + \lambda \left\| \hat{\beta}^{(i)} \right\|_1 \tag{8}$$

The sub-problem of $\hat{\beta}^{(i)}$ forms a generalized LASSO optimization problem.

$$\underset{\hat{\beta}^{(i)}}{\arg\min} \left\| Y^{(i)} - g\left(h^{(i-1)} \odot \hat{\beta}^{(i)} \right) \right\|_2^2 + \lambda \left\| \hat{\beta}^{(i)} \right\|_1 \tag{9}$$

where $Y^{(i)} = \|_{k=K'}^{K} Y_k^{(i)}$ stacked horizontally. And the generalized function $g(h'^{(i-1)}) = \|_{k=K'}^{K} \tilde{A}^k h'^{(i-1)} \widehat{W}_k^{(i)}$ stacked horizontally. However, as $Y^{(i)}$ is also concatenated horizontally, we can rewrite the LASSO optimization problem by substituting the original observations $h^{(i-1)}$ with $\tilde{A}^k h^{(i-1)}$ concatenated vertically. Then, the sub-problem of $\hat{\beta}^{(i)}$ falls back to a classic LASSO regression with $(K - K')$ times the observations in the single branch pruning and the same number of predictors.

Our pruning method also works for other GNN architectures. For GNN architectures with averaging instead of concatenating the output features in each branch, the generalized function becomes $g(h^{(i-1)}) = \sum_{k=K'}^{K} \tilde{A}^k h^{(i-1)} \widehat{W}_k^{(i)} / (K - K' + 1)$, which can be optimized by concatenating the observations horizontally. For multi-head attention based architecture, we can prune the layers by treating each attention head as a branch.

3.3 End-to-End GNN Pruning

We design the pruning schemes for different inference scenarios based on the computation complexity and memory consumption
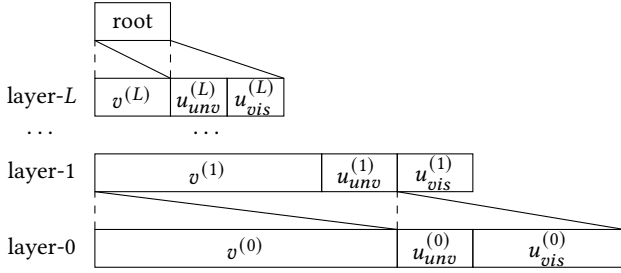
Figure 3: Illustration of forward propagation on $L + 1$-layer GraphSAGE architecture with stored hidden features. $v^{(i)}$ denotes the supporting nodes for the branch $k = 0$. $u_{unv}^{(i)}, u_{vis}^{(i)}$ denote the unvisited and visited supporting nodes for the branch $k = 1$. The hidden features of the visited nodes $u_{vis}^{(i)}$ are obtained directly from the stored hidden features.

in the case study in Section 2.2. The channel pruning in layer-i not only ignores some input channels in $h^{(i-1)}$, but also leads to the reduction of output channels in the weight matrices $W_k^{(i)}$ of the previous layer. Thus, when pruning the whole network, we prune reversely from the output layer to the input layer. Dense layers are treated as GNN layers with $K' = K = 0$.

3.3.1 Pruned Full Inference. For full inference, we simply prune each layer with a constant budget η except the input layer as the computation and memory distribute evenly in each layer. We do not remove any dimensions of the raw node attributes in layer-0. The complexity per node and memory usage of the pruned models range in (η^2, η) and $(\eta, 1)$, compared with the original model. For other GNN architectures that follow similar forward propagation in Equation 1 like JK [34] and SIGN [22], our pruning method could be directly applied with constant budget.

3.3.2 Pruned Batched Inference. The major challenge in batched inference is the "neighbor explosion" problem where the number of supporting nodes grows exponentially as the network goes deeper. We need to visit the node attributes for an exponential amount of nodes to compute the embedding for one target node. Therefore, we focus on reducing the computation and memory usage in the first layer by reducing the channels in the first layer and the second layer. In the first layer, we focus on the branches that have more neighbors than others. For the GraphSAGE architecture with pruning budget η, we prune the $k = 1$ branch in layer-1 and the whole layer-2 with budget η, which reduces the dominant terms in the computation and memory usage by η.

In addition to channel pruning, we store the hidden features $h^{(i)}$ of visited nodes in the middle layers. Their neighbors, when aggregating from them, directly aggregate from the stored hidden features, instead of iteratively looking at farther neighbors. Figure 3 shows the supporting nodes in each layer with stored hidden features. Ideally, if we store the hidden features for all visited nodes, the batched inference would have exactly the same complexity as full inference (i.e., $d = 1$ in Equation 3). However, indexing and storing the hidden features incur extra data transfer which increases the latency. On evolving graphs, out-dated hidden features also

Table 2: Dataset statistics. The Attr. column shows the dimension of the node attributes. (s) in Classes denotes multi-class single-label classification problem while (m) denotes multi-class multi-label classification problem. The Test% column shows the percentage of test nodes.

Dataset	Nodes	Edges	Attr.	Classes	Test%
Flickr	89,250	899,756	500	7(s)	25%
Arxiv	169,343	1,166,243	128	40(s)	29%
Reddit	232,965	11,606,919	602	41(s)	24%
Yelp	716,847	6,977,410	300	100(m)	10%
Products	2,449,029	61,859,140	100	47(s)	88 %
YelpCHI	67,395	287,619	769	2(s)	23%

affect accuracy. The portion of hidden features to store in each batch could be dynamically determined by the task-specific target latency and accuracy. Applications with high latency tolerance could potentially save more hidden features to increase throughput. For out-dated hidden features, we can set a threshold and discard them when the accuracy drop reaches the threshold. In practice, we find storing the hidden features for the root nodes at inference is a good balance point for the datasets we use. In addition, the root nodes usually have the most up-to-date hidden features in batched inference.

3.3.3 Detailed Optimization Procedure. In the experiment, we perform one iteration on each sub-problem instead of multiple iterations [14]. For the sub-problem of $\widehat{W}$, instead of the least square solution, we also apply SGD as the size of X could be large. We partition the matrix $\tilde{A}^k h^{(i-1)}$ and $X_k^{(i)}$ row-wise to form mini-batches. Define one epoch as performing SGD on the whole matrix once. To optimize the whole problem, we first optimize several epochs on the sub-problem of $\hat{\beta}$. At the end of each epoch, we slightly increase the penalty factor λ until pruning budget is met or over-penalized (all values in β are decreasing). Note that as the mask values converge to zero, some mask values may be exactly zero while the others are close to zero. We clip the masks with small values to zero according to the pruning budget to make sure the corresponding channels are completely removed. Then, we optimize the sub-problem for $\widehat{W}_k^{(i)}$ until converge. The final weights of the pruned layer are obtained by applying the mask $\hat{\beta}_k^{(i)}$ to the weights $\widehat{W}_k^{(i)}$.
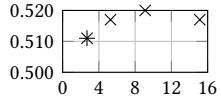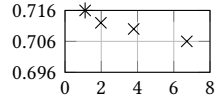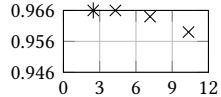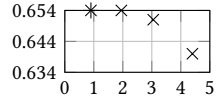
4 EXPERIMENTS

We evaluate the performance of the proposed method with the node classification problem on five popular datasets: 1. **Flickr** [42] classifying the types of user-uploaded images, 2. **Arxiv** [15] classifying subject areas of Arxiv CS papers, 3. **Reddit** [13] classifying communities of Reddit posts, 4. **Yelp** [42] classifying types of businesses on Yelp, 5. **Products** [15] classifying categories of products on Amazon. For batched inference, we also evaluate with a real world spam detection application on the YelpCHI[21] dataset that identifies spam reviews on Yelp. We adopt supervised and inductive settings on all datasets.

For the models to prune, we use the 2-layer GraphSAGE [13] architecture (Equation 1 with $K' = 0, K = 1$) with the common

Table 3: Pruned full inference results on GPU. The ∗ nodes in the plots denote the results of the reference models (no pruning).

	Flickr				Arxiv				Reddit				Yelp			
Budget	-	2×	4×	8×	-	2×	4×	8×	-	2×	4×	8×	-	2×	4×	8×
F1-Micro	0.511	0.517	0.520	0.517	0.716	0.712	0.710	0.706	0.966	0.966	0.964	0.959	0.654	0.654	0.651	0.640
#kMACs/node	545	211	94	48	1242	360	115	40	317	172	112	85	1490	485	180	77
Mem. (MB)	531	269	221	199	1997	1002	505	257	852	738	681	652	8459	4256	2155	1225
Thpt. (mN/s)	2.69	5.28	9.11	15.13	1.11	1.98	3.79	6.72	2.47	4.30	7.17	10.35	0.90	1.95	3.05	4.40
Thpt. Impr.	-	1.96×	3.39×	5.63×	-	1.79×	3.42×	6.07×	-	1.74×	2.90×	4.19×	-	2.16×	3.38×	4.82×
F1mic-Thpt																

hidden feature size 256, 512, 128, 512, 512 on the five node classification datasets, respectively. On the YelpCHI dataset, we use 128 as the hidden feature size. We use the standard single floating point precision for both the original models and the pruned models. To obtain trained models to prune, we adopt the sub-graph based training technique from GraphSAINT [42] with the random walk sampler. For each dataset, we prune with three global budgets $\eta = 0.5, 0.25, 0.125$ and obtain three pruned models (2×, 4×, 8×) in different sizes. We choose 1024 as the batch size and use the ADAM optimizer for SGD in the two sub-problems. After pruning, we re-train the pruned models until convergence.

To test the speedup of the pruned models, we measure the throughput and latency of full inference on the first four datasets with GPU, and batched inference on all five datasets with CPU and GPU. For batched inference, we form batches randomly from the nodes in the test set until all the nodes in the test set are covered. All accuracy (F1-Micro) results are for the test nodes only. The pruning framework is implemented using PyTorch and Python3. We run all experiments on a machine with 64-core ThreadRipper 2990WX CPU with 256GB of DDR4 RAM, and a single NVIDIA RTX A6000 GPU with 48GB of GDDR6 RAM. All the accuracy results are averages of three runs. For batched inference, we limit the number of hop-2 neighbors to be 32.

4.1 Performance of Single Layer Pruning

We compare the proposed pruning method (LASSO) with pruning the channels with small L1-norm in the corresponding weight matrix (Max Res.) and randomly pruning the channels (Random). Figure 4 shows the loss and F1-Micro curves under different numbers of pruned channels in both branches of layer-2 on the Reddit dataset. We apply layer-wise re-training for all three pruning methods. The proposed pruning method clearly outperforms other pruning methods, especially when the number of pruned channels is more than 30%.

4.2 Full Inference

Table 3 shows the results for full inference on GPU. The computation complexity is measured in thousands of Multiplication-and-ACcumulation operations per node (#kMACs/node). The throughput is measured in thousand of target nodes computed per second

(kN/s) or million of target nodes computed per second (mN/s). For memory usage, we adopt in-place point-wise operations without storing the intermediate values as we only need to compute forward propagation at inference. The latency is the GPU execution time of a complete forward propagation. The throughput and memory usage are calculated for all the nodes in the graphs. In the F1mic-Thpt. row, the x and y axes are the throughput (in mN/s) and F1-micro. On the Flickr dataset, the pruned models achieve higher F1-Micro than the original models, possibly due to better convergence of smaller models. The reduction in computation and memory usage depends on the dimension of the input node attributes. We achieve close to η^2 reduction in computation complexity and η reduction in memory usage on the Arxiv and Yelp dataset with small input node attributes dimensions. We achieve an average of 3.27× speedup on GPU with less than 0.006 drop in F1-Micro for all datasets with 4× pruned models. On the Flickr, Arxiv and Reddit datasets, the 8× pruned models still achieve similar accuracy as the original models. The pruned models for full inference reduce the latency on small datasets to meet the requirements for real-time applications and increase the throughput for large datasets. The pruned models also make it possible to run full batch inference of small graphs on edge devices with limited memory. We observe consistent GPU utilization of around 50% for models with different pruning budgets.
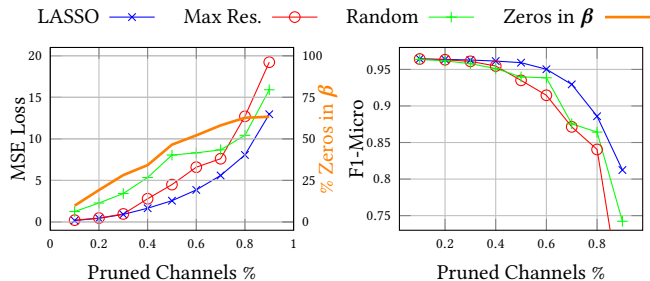

Figure 4: Loss and F1-Micro curves under different numbers of pruned channels in layer-2 on the Reddit dataset. We also show the percentage of β that shrinks to zero for the LASSO pruning method in the left figure.

Table 4: Pruned batched inferences results on CPU (batch size=512). The second rows in each metric show the results with stored hidden features. The ∗ nodes in the plots denote the results of the reference models (no pruning, w/o store).

	Arxiv				Reddit				Yelp				Products			
Budget	-	2×	4×	8×	-	2×	4×	8×	-	2×	4×	8×	-	2×	4×	8×
F1-Micro	0.714	0.710	0.709	0.707	0.966	0.966	0.964	0.955	0.654	0.654	0.652	0.646	0.792	0.791	0.785	0.764
w/ store	0.714	0.710	0.709	0.707	0.966	0.966	0.964	0.954	0.653	0.653	0.652	0.646	0.792	0.792	0.786	0.766
#kMACs/node	3135	1620	846	395	17665	7409	3288	1052	7870	3696	1650	840	3952	2044	1090	520
w/ store	2118	1096	577	286	6225	2627	1171	381	3908	1888	895	485	1590	827	446	240
Mem. (MB)	85	49	30	14	3086	1551	790	409	225	122	53	26	96	65	49	28
w/store	72	42	23	10	1431	568	288	147	165	92	39	19	70	37	21	10
Lat. (ms)	27	20	17	13	411	217	128	85	56	38	25	16	120	58	48	35
w/ store	15	8	6	5	101	56	34	24	22	14	10	7	47	30	27	20
Lat. Impr.	-	1.33×	1.54×	2.12×	-	1.90×	3.22×	4.86×	-	1.46×	2.20×	3.59×	-	2.09×	2.50×	3.42×
w/ store	1.82×	3.24×	4.18×	5.29×	4.09×	7.35×	12.26×	17.04×	2.50×	3.84×	5.84×	8.08×	2.56×	3.99×	4.40×	6.02×
F1mic-Lat. w/o store w/ store																

On the Flickr, Arxiv, Reddit, and Yelp dataset, our pruning method takes 2.35, 4.34, 6.35, and 32.15 seconds in pruning and 1.36, 6.38, 10.02, and 346.21 seconds in re-training. Due to the small number of parameters in the pruned models, the re-training of the pruned models takes less time than training the original models.

4.3 Batched Inference

Table 4 shows the results for batched inference on CPU. We calculate the memory usage by the amount of memory needed to compute the forward path of one batch. The attributes and stored hidden features of the supporting nodes in each batch are fed into CPU from DDR4 (peak bandwidth 68GB/s) and GPU from GDDR6 (peak bandwidth 768GB/s) memory. In the F1mic-Thpt. row, the x and y axes are the throughput (in kN/s) and F1-micro. We achieve η reduction in computation complexity and memory usage on all datasets for batched inference without stored hidden features. We store the hidden features of training and validation nodes, and the root nodes in each batch of inference. The storing of hidden features further reduces an average of 33% of supporting nodes in layer-1. We reduce the memory usage from 85-3086MB to 10-147MB, which makes it possible to perform inference on edge devices like mobiles. The memory usage also reflects an upper bound of the amount of input node attributes needed to perform one batch of forward propagation. On all five datasets, the pruned models with stored hidden features achieve less than 30ms (up to 17× improvement) latency on CPU with less than 0.012 F1-Micro drop. The pruned models with stored hidden feature meet the requirements of most real-time applications on CPU. On GPU, our 4× models achieve 4, 16, 4, 8ms latency without stored hidden features and 4, 6, 3, 6ms latency with stored hidden features on the Arxiv, Reddit, Yelp, and Products datasets, respectively. Compared with full inference, batched inference requires less memory and computation for a small number of target nodes. For latency-sensitive or large scale applications, batched inference provides a light-weight and low-latency solution. We observe 100% CPU utilization on all models,

and 20% to 50% GPU utilization on GPU depending on the model size.

Figure 5.a shows the latency under different batch sizes on the Reddit dataset on CPU. The latency grows linearly with the batch size, which shows that our pruning method accommodates applications with different inference batch sizes. Figure 5.b shows the trade-off between storing hidden features and extra latency and drop in F1-Micro. Note that the extra latency is mostly caused by the storing of the hidden features, which can be done offline.

4.3.1 Spam Detection Application. To evaluate the performance of the proposed pruning technique on real-time applications, we over-sample the YelpCHI [21] dataset 400 times to create a graph with 27 million nodes which has similar scale as the Yelp website. The nodes in the graph represent reviews for restaurants and hotels in Chicago and are attached with timestamps. The task is to identify spam reviews from the posted reviews between October 2011 and
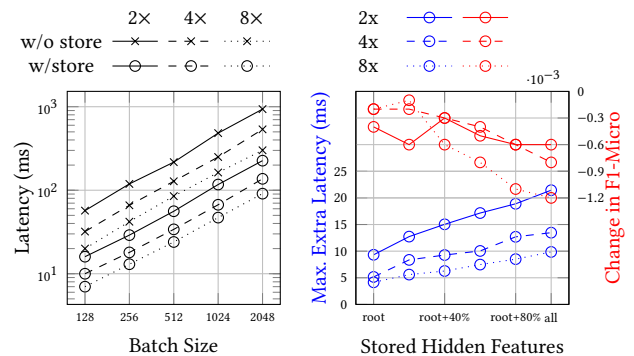

Figure 5: (a). Latency curves under different batch sizes on the Reddit dataset on CPU. (b). Maximum extra latency and accuracy drop curves with different percentages of stored hidden features.
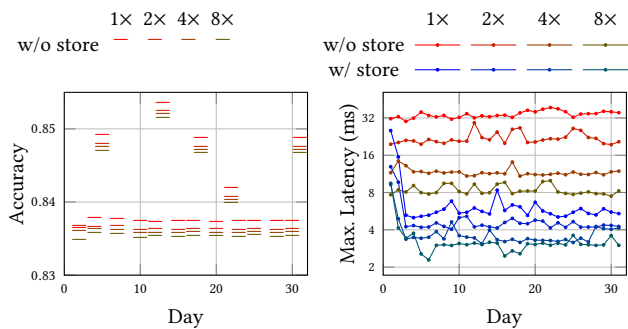
Figure 6: Accuracy and maximum latency of each day in the first month on the YelpCHI dataset. We only show the accuracy without stored hidden features because the accuracy with stored hidden features is very close.

		Pre-Proc.	F1-micro	#kMACs/node
Full Inf.	SGC	-	0.949	146
		✓		25
	SIGN(2,0,0)	-	0.966	978
		✓		858
	PPRGo	-	0.937	148
	TinyGNN	-	0.957	273
	ours-4×	-	0.964	112
Batched Inf.	MLP-2	-	0.702	120
	ours-4× w/o	-	0.964	3288
	ours-4× w/	-	0.964	1171

Table 5: Comparison of accuracy and per node computation for full inference and batched inference on the Reddit dataset. The w/ and w/o in batched inference denotes with and without stored hidden features.

October 2012. We adopt the strategy to perform inference on the emerging reviews every 30 minutes and re-train the model every month. The 1×(reference), 2×, 4×, and 8× models achieve 0.873, 0.871, 0.866, and 0.865 accuracy on the test set. Figure 6 shows the accuracy and maximum latency of each day in the first month. For inference with stored hidden features, the first few days have higher latency due to the indexing and storing of the hidden features. However, the latency is still lower than inference without stored hidden features, even in the first few days.

4.4 Comparison with Other GNNs

We compare the throughput and accuracy of full inference with GAT [28], SIGN [22], Jumping Knowledge Network (JK) [34], GraphSAGE [13], PPRGo [2] [1], GCN [16], SGC [32], and TinyGNN [36]. We use a similar two-layer (or equivalent of two-hop propagation) architecture on all baselines except a one-layer student network supervised by a two-layer teacher network for TinyGNN. Figure 1 shows the inference throughput and accuracy of various GNN architectures on the Reddit dataset on GPU. Our 4× model achieves top-tier accuracy, comparable with GAT, SIGN and GraphSAGE, but with significant improvement in throughput (6.96×, 4.74×, 2.59× with GAT, SIGN, and GraphSAGE, respectively).

4.4.1 Computation Comparison with Simplified GNNs. We compare the accuracy and per node computation on the Reddit dataset of our 4× pruning models with SGC, SIGN with $(r, s, t) = (2, 0, 0)$, PPRGo with two-pass inference, TinyGNN with a 1-layer PAM student network supervised by a 2-layer teacher network, and 2-layer MLP with 128 hidden features (MLP-2). Table 5 shows the result of the comparison with other simplified GNNs. The pre-processing for both SGC and SIGN is to twice compute feature propagation $(\tilde{A}^2 \cdot h^{(0)})$ for 120 kMACs/node. If any graph structure or node attributes change, the pre-processing needs to be re-computed. For SGC, if the input node features are pre-processed, there is only one MLP layer transforming the aggregate features to class probabilities, leading to the lowest computation. SIGN has the highest per node computation as the numbers of hidden units in the feedforward

layers are high (460 for GNN layers and 675 for the classification layer). For full inference, our pruned model achieves higher accuracy than SGC and TinyGNN, and comparable accuracy to SIGN with less computation. For batched inference, our pruned models achieve remarkably higher accuracy than MLP.

5 CONCLUSION

We presented a novel method of pruning the input channels to accelerate large scale and real-time GNN inference. We formulated the GNN pruning problem as a LASSO optimization problem to select from the input channels to approximate the output. We developed different pruning schemes according to the computation complexity and memory usage in different inference scenarios. We designed a unique technique for batched inference to further reduce computation by storing and reusing the hidden features. We conducted experiments on real-world datasets to demonstrate that the pruned models greatly reduce computation and memory usage while still maintaining high accuracy. We showed the improvement on latency and throughput of using the pruned models on CPU and GPU. The light-weight pruned models are attractive to energy-efficient devices like mobile processors and FPGA, as well as applications like real-time recommendation and fraud detection on social networks.

REFERENCES

[1] Sami Abu-El-Haija, Bryan Perozzi, Amol Kapoor, Nazanin Alipourfard, Kristina Lerman, Hrayr Harutyunyan, Greg Ver Steeg, and Aram Galstyan. 2019. MixHop: Higher-Order Graph Convolutional Architectures via Sparsified Neighborhood Mixing *(Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. 21–29.
[2] Aleksandar Bojchevski, Johannes Klicpera, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. 2020.

---

[1] We tune the parameters of PPRGo to fit the supervised learning setting.

Scaling Graph Neural Networks with Approximate PageRank. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2464–2473.

[3] Tianle Cai, Shengjie Luo, Keyulu Xu, Di He, Tie yan Liu, and Liwei Wang. 2020. GraphNorm: A Principled Approach to Accelerating Graph Neural Network Training. arXiv:2009.03294 [cs.LG]

[4] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations (ICLR)*.

[5] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction.. In *ICML*. 941–949.

[6] Charles R C. Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. 2017. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 652–660.

[7] Siyuan S. Qi, Wenguan Wang, Baoxiong Jia, Jianbing Shen, and Song-Chun Zhu. 2018. Learning human-object interactions by graph parsing neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 401–417.

[8] Xiaojuan X. Qi, Renjie Liao, Jiaya Jia, Sanja Fidler, and Raquel Urtasun. 2017. 3d graph neural networks for rgbd semantic segmentation. In *Proceedings of the IEEE International Conference on Computer Vision*. 5199–5208.

[9] Shaohua Fan, Junxiong Zhu, Xiaotian Han, Chuan Shi, Linmei Hu, Biyu Ma, and Yongliang Li. 2019. Metapath-Guided Heterogeneous Graph Neural Network for Intent Recommendation. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '19)*. Association for Computing Machinery, New York, NY, USA, 2478–2486.

[10] Spyros Gidaris and Nikos Komodakis. 2019. Generating classification weights with gnn denoising autoencoders for few-shot learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 21–30.

[11] Shengnan Guo, Youfang Lin, Ning Feng, Chao Song, and Huaiyu Wan. 2019. Attention based spatial-temporal graph convolutional networks for traffic flow forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 922–929.

[12] Jonathan Halcrow, Alexandru Moşoi, Sam Ruth, and Bryan Perozzi. 2020. Grale: Designing Networks for Graph Learning. In *SIGKDD Conference on Knowledge Discovery and Data Mining*.

[13] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30*. 1024–1034.

[14] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*. 1389–1397.

[15] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. arXiv:2005.00687 [cs.LG]

[16] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

[17] Ao Li, Zhou Qin, Runshi Liu, Yiqun Yang, and Dong Li. 2019. Spam Review Detection with Graph Convolutional Networks. *Proceedings of the 28th ACM International Conference on Information and Knowledge Management* (2019).

[18] Yong Li, Zihang He, Xiang Ye, Zuguo He, and Kangrong Han. 2019. Spatial temporal graph convolutional networks for skeleton-based dynamic hand gesture recognition. *EURASIP Journal on Image and Video Processing* 2019, 1 (13 Sep 2019), 78.

[19] Ziqi Liu, Chaochao Chen, Xinxing Yang, Jun Zhou, Xiaolong Li, and Le Song. 2018. Heterogeneous Graph Neural Networks for Malicious Account Detection. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management (CIKM '18)*. New York, NY, USA, 2077–2085.

[20] J. Luo, H. Zhang, H. Zhou, C. Xie, J. Wu, and W. Lin. 2019. ThiNet: Pruning CNN Filters for a Thinner Net. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41, 10 (2019), 2525–2538.

[21] Shebuti Rayana and Leman Akoglu. 2015. Collective opinion spam detection: Bridging review networks and metadata. In *Proceedings of the 21th acm sigkdd international conference on knowledge discovery and data mining*. 985–994.

[22] Emanuele Rossi, Fabrizio Frasca, Ben Chamberlain, Davide Eynard, Michael M. Bronstein, and Federico Monti. 2020. SIGN: Scalable Inception Graph Neural Networks. *CoRR* (2020).

[23] Victor Garcia Satorras and Joan Bruna Estrach. 2018. Few-Shot Learning with Graph Neural Networks. In *International Conference on Learning Representations*.

[24] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*. Springer, 593–607.

[25] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen. 2018. GraphR: Accelerating Graph Processing Using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 531–543.

[26] C. Tian, L. Ma, Z. Yang, and Y. Dai. 2020. PCGCN: Partition-Centric Processing for Accelerating Graph Convolutional Network. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 936–945.

[27] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.

[28] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. *International Conference on Learning Representations* (2018).

[29] Daixin Wang, Jianbin Lin, Peng Cui, Quanhui Jia, Zhen Wang, Yanming Fang, Quan Yu, Jun Zhou, Shuang Yang, and Yuan Qi. 2019. A Semi-supervised Graph Attentive Network for Financial Fraud Detection. In *2019 IEEE International Conference on Data Mining (ICDM)*. IEEE, 598–607.

[30] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. 2019. Dynamic graph cnn for learning on point clouds. *Acm Transactions On Graphics (tog)* 38, 5 (2019), 1–12.

[31] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 2082–2090.

[32] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying Graph Convolutional Networks. In *Proceedings of the 36th International Conference on Machine Learning*. PMLR, 6861–6871.

[33] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*.

[34] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation Learning on Graphs with Jumping Knowledge Networks. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research)*, Jennifer G. Dy and Andreas Krause (Eds.), Vol. 80. PMLR, 5449–5458.

[35] Xiaoran Xu, Wei Feng, Yunsheng Jiang, Xiaohui Xie, Zhiqing Sun, and Zhi-Hong Deng. 2020. Dynamically Pruned Message Passing Networks for Large-scale Knowledge Graph Reasoning. In *International Conference on Learning Representations*.

[36] Bencheng Yan, Chaokun Wang, Gaoyang Guo, and Yunkai Lou. 2020. TinyGNN: Learning Efficient Graph Neural Networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1848–1856.

[37] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. 2020. HyGCN: A GCN Accelerator with Hybrid Architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 15–29.

[38] Jianbo Ye, Xin Lu, Zhe Lin, and James Z. Wang. 2018. Rethinking the Smaller-Norm-Less-Informative Assumption in Channel Pruning of Convolution Layers. In *International Conference on Learning Representations*.

[39] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2018).

[40] Hanqing Zeng and Viktor Prasanna. 2020. GraphACT. *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Feb 2020).

[41] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Accurate, Efficient and Scalable Graph Embedding. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[42] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations*.

[43] B. Zhang, H. Zeng, and V. Prasanna. 2020. Hardware Acceleration of Large Scale GCN Inference. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 61–68.

[44] Jiani Zhang, Xingjian Shi, Shenglin Zhao, and Irwin King. 2019. STAR-GCN: Stacked and Reconstructed Graph Convolutional Networks for Recommender Systems. In *The 28th International Joint Conference on Artificial Intelligence*. 4264–4270.

[45] Jiawei Zhang, Haopeng Zhang, Congying Xia, and Li Sun. 2020. Graph-Bert: Only Attention is Needed for Learning Graph Representations. *arXiv preprint arXiv:2001.05140* (2020).

[46] Yuhao Zhang, Peng Qi, and Christopher D. Manning. 2018. Graph Convolution over Pruned Dependency Trees Improves Relation Extraction. In *Empirical Methods in Natural Language Processing (EMNLP)*.