GNN Transformation Framework for Improving Efficiency and Scalability

Seiji Maekawa[1] ✉, Yuya Sasaki[1], George Fletcher[2], and Makoto Onizuka[1]

[1] Osaka University, 1–5 Yamadaoka, Suita, Osaka, Japan
{maekawa.seiji,sasaki,onizuka}@ist.osaka-u.ac.jp
[2] Eindhoven University of Technology, P.O. Box 513, MB, Eindhoven, Netherlands
g.h.l.fletcher@tue.nl

Abstract. We propose a framework that automatically transforms non-scalable GNNs into precomputation-based GNNs which are efficient and scalable for large-scale graphs. The advantages of our framework are two-fold; 1) it transforms various non-scalable GNNs to scale well to large-scale graphs by separating local feature aggregation from weight learning in their graph convolution, 2) it efficiently executes precomputation on GPU for large-scale graphs by decomposing their edges into small disjoint and balanced sets. Through extensive experiments with large-scale graphs, we demonstrate that the transformed GNNs run faster in training time than existing GNNs while achieving competitive accuracy to the state-of-the-art GNNs. Consequently, our transformation framework provides simple and efficient baselines for future research on scalable GNNs.

Keywords: Graph neural networks · Large-scale graphs · Classification.

1 Introduction

Graph is a ubiquitous structure that occurs in many domains, such as Web and social networks. As a powerful approach for analyzing graphs, Graph Neural Networks (GNNs) have gained wide research interest [30, 25]. Many GNNs have been proposed for node classification and representation learning including GCN [15], which is the most popular GNN variant. Most existing GNNs adopt graph convolution that performs three tasks; 1) feature aggregation, 2) learnable weight multiplication, and 3) activation function application (e.g., ReLU, a non-linear function). By stacking multiple graph convolutional layers, they propagate node features over the given graph topology. However, these existing GNNs cannot be efficiently trained on large-scale graphs since the GNNs need to perform three tasks in graph convolution every time learnable weights are updated. In addition, large-scale graphs cannot be put on GPU memory for efficient matrix operations. As a result, graph convolution is not efficient and scalable for large-scale graphs.

A major approach to apply GNNs to large-scale graphs is to separate feature aggregation from graph convolution so that GNNs can precompute aggregated features [24, 8, 18]. These methods are called *precomputation-based* GNNs. In

detail, they remove non-linearity, i.e., activation functions, from graph convolution so that feature aggregation is separated from weight learning. Thanks to the independence of feature aggregation and weight learning, precomputation-based GNNs are efficient in learning steps by precomputing feature aggregation before training learnable weights.

Though some existing works tackle the scalability problem of GNNs as discussed above, most widely studied GNNs are not scalable to large-scale graphs for the following two reasons. First, existing studies on precomputation-based GNNs [24, 8, 18] focus on introducing several specific GNN architectures that are manually designed. So, it is laborsome to apply the same precomputation idea to other GNNs. An interesting observation is that they share the common motivation: precomputation of feature aggregation is indispensable for high scalability. To our best knowledge, there are no works that study a general framework that transforms non-scalable GNNs to scalable precomputation-based GNNs. Second, existing precomputation schemes are not scalable because they need to put complete graphs (e.g., graphs with one billion edges [12]) on GPU memory. Since the size of large graphs typically exceeds the memory size of general GPU, existing works precompute feature aggregation on CPU.

To tackle the above issues, we address two research questions: **Q1**: *Can we design a general procedure that transforms non-scalable GNNs to efficient and scalable precomputation-based GNNs while keeping their classification performance?* and **Q2**: *Can we efficiently execute the precomputation on GPU?* There are two technical challenges which must be overcome to answer our questions. First, we need to automatically transform non-scalable GNNs to precomputation-based GNNs. We should develop a common transformation procedure that can be applied to various non-scalable GNNs while preserving their expressive power. Second, we need to decompose large graphs into small groups each of which can be handled efficiently with GPU. Typically, graph decomposition suffers from an imbalance problem since node degree distributions usually follow power law distributions [19]. Hence, we should divide graphs into balanced groups and select an appropriate group size so that precomputation time is optimized.

In this paper, we propose a framework[3] that automatically transforms non-scalable GNNs into precomputation-based GNNs with a scalable precomputation schema. As for the first challenge, we develop a new transformation procedure, called Linear Convolution (LC) transformation, which can be applied to various non-scalable GNNs so that transformed GNNs work efficiently and scale well to large-scale graphs. Our transformation procedure removes non-linear functions from graph convolution, but incorporates non-linear functions into weight learning. This idea is derived from our hypothesis that it is not crucial to incorporate non-linearity into graph convolutional layers but into weight learning for prediction. Since our transformation preserves the major functionality of graph convolution and a similar expressive power to original GNNs, the transformed GNNs can achieve competitive prediction performance to the original ones while improving their scalability. As for the second challenge, we develop a block-

[3] Our codebase is available on (https://github.com/seijimaekawa/LCtransformation).

wise precomputation scheme which optimally decomposes large-scale graphs into small and balanced blocks each of which can fit into GPU memory. We introduce a simple decomposition approach to ensure that blocks are balanced and give minimization formulas that decide the optimal block size under limited GPU memory.

Through extensive experiments, we validate that our transformation procedure and optimized block-wise precomputation scheme are quite effective. First, we show that our LC transformation procedure transforms non-scalable GNNs to efficient and scalable precomputation-based GNNs while keeping their node classification accuracy. Second, we show that our precomputation scheme is more efficient than that of existing precomputation-based GNNs. In summary, our transformation procedure provides simple and efficient baselines for future research on scalable GNNs by shining a spotlight on existing non-scalable methods.

The rest of this paper is organized as follows. We describe notations and fundamental techniques for our method in Section 2. Section 3 proposes our framework. We give the purpose and results of experiments in Section 4. Section 5 describes the details of related work. Finally, we conclude this paper in Section 6.

2 Preliminaries

An *undirected attributed graph with class labels* is a triple $G = (\boldsymbol{A}, \boldsymbol{X}, \boldsymbol{C})$ where $\boldsymbol{A} \in \{0,1\}^{n \times n}$ is an adjacency matrix, $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ is an attribute matrix assigning attributes to nodes, and a class matrix $\boldsymbol{C} \in \{0,1\}^{n \times y}$ contains class information of each node, and n, d, y are the numbers of nodes, attributes and classes, respectively. If there is an edge between nodes i and j, $\boldsymbol{A}_{ij}$ and $\boldsymbol{A}_{ji}$ are set to one. We define the degree matrix $\boldsymbol{D} = \mathrm{diag}(D_1, \ldots, D_n) \in \mathbb{R}^{n \times n}$ as a diagonal matrix, where D_i expresses the degree of node i. We also define an identity matrix $\boldsymbol{I} = \mathrm{diag}(1, \ldots, 1) \in \mathbb{R}^{n \times n}$ and an adjacency matrix extended with self-loops $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$. We define node embeddings $\boldsymbol{H} \in \mathbb{R}^{n \times h}$, where h is the dimension of a hidden layer. We summarize notation and their definitions in Table 1.

2.1 Graph Convolutional Networks

Multi-layer GCN is a standard GCN model which was proposed in [15]. GCNs learn a feature representation for the feature of each node over layers. For the k-th graph convolutional layer, we denote the input node representations of all nodes by the matrix $\boldsymbol{H}^{(k-1)}$ and the output node representations by $\boldsymbol{H}^{(k)}$. The initial node representations are set to the input features, i.e., $\boldsymbol{H}^{(0)} = \boldsymbol{X}$. Let $\boldsymbol{S}$ denote the normalized adjacency matrix

$$\boldsymbol{S} = \tilde{\boldsymbol{D}}^{-\frac{1}{2}} \tilde{\boldsymbol{A}} \tilde{\boldsymbol{D}}^{-\frac{1}{2}}. \tag{1}$$

This normalized adjacency matrix is commonly used as a graph filter for graph convolution. The graph filter is known as a low-pass filter that filters out noise

Table 1: Notation and definitions

n	number of nodes
d	dimension of features
y	number of classes
h	dimension of hidden layer
K	number of hidden layers
$\boldsymbol{A} \in \mathbb{R}^{n \times n}$	adjacency matrix
$\tilde{\boldsymbol{A}} \in \mathbb{R}^{N \times N}$	extended adjacency matrix
$\boldsymbol{S} \in \mathbb{R}^{n \times n}$	normalized adjacency matrix
$\boldsymbol{X} \in \mathbb{R}^{n \times d}$	feature matrix
$\boldsymbol{C} \in \mathbb{R}^{n \times y}$	class matrix
$\boldsymbol{D} \in \mathbb{R}^{n \times n}$	degree matrix
$\boldsymbol{H} \in \mathbb{R}^{n \times h}$	node embeddings
$\boldsymbol{W}_1 \in \mathbb{R}^{d \times h}, \boldsymbol{W}_2, \ldots, \boldsymbol{W}_{K-1} \in \mathbb{R}^{h \times h}, \boldsymbol{W}_K \in \mathbb{R}^{h \times y}$	weight matrices
$\boldsymbol{Y} \in \mathbb{R}^{n \times y}$	predicted label matrix

in node features [15]. For each layer, GCN propagates the embedding of a node to its neighbors as follows:

$$\boldsymbol{H}^{(k)} = \sigma(\boldsymbol{S}\boldsymbol{H}^{(k-1)}\boldsymbol{W}_k), \tag{2}$$

where $\boldsymbol{W}_k$ denotes the weight matrix of the k-th layer and σ denotes a non-linear function, e.g., ReLU. In the output layer, K-layer GCN outputs a predicted label matrix $\boldsymbol{Y} \in \mathbb{R}^{n \times y}$ as:

$$\boldsymbol{Y} = \mathrm{softmax}(\boldsymbol{S}\boldsymbol{H}^{(K-1)}\boldsymbol{W}_K), \tag{3}$$

where $\mathrm{softmax}(\boldsymbol{P})_{ij} = \frac{\exp(\boldsymbol{P}_{ij})}{\sum_{j=1}^{y} \exp(\boldsymbol{P}_{ij})}$ for a matrix $\boldsymbol{P}$. The number of layers is typically set to $K = 2$ [15].

2.2 Precomputation-based GNNs

Several precomputation-based GNNs have been proposed recently [24, 8, 18]. Their fundamental and common idea is to remove non-linear functions between each layer in order to precompute feature aggregation. We explain Simplifying Graph Convolution (SGC for short) [24] which is the simplest precomputation-based GNN. Thanks to the removal, K-layer GCN can be rewritten as follows by unfolding the recursive structure:

$$\boldsymbol{Y} = \mathrm{softmax}(\boldsymbol{S} \ldots \boldsymbol{S}\boldsymbol{X}\boldsymbol{W}_1 \ldots \boldsymbol{W}_K). \tag{4}$$

The repeated multiplication with the normalized adjacency matrix $\boldsymbol{S}$ can be simplified into a K-th power matrix $\boldsymbol{S}^K$ and the multiple weight matrices can be reparameterized into a single matrix $\boldsymbol{W} = \boldsymbol{W}_1 \ldots \boldsymbol{W}_K$. The output becomes

$$\boldsymbol{Y} = \mathrm{softmax}(\boldsymbol{S}^K\boldsymbol{X}\boldsymbol{W}). \tag{5}$$

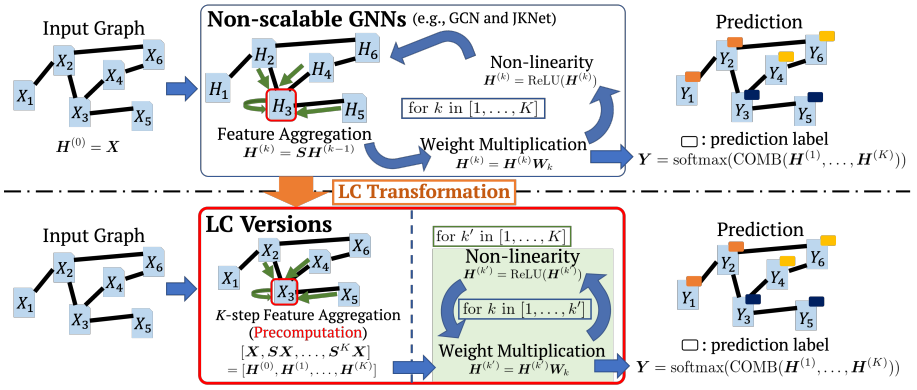

Fig. 1: Example of LC transformation. Upper part: non-scalable GNNs operate K-layer graph convolution combining feature aggregation, weight multiplication, and activation function application (ReLU). This example corresponds to K-layer GCN if COMB outputs only $\boldsymbol{H}^K$. Lower part: LC transformation separates feature aggregation and weight learning while keeping the similar architectures with the original GNNs. LC versions avoid recomputing feature aggregation whenever learnable weights are updated at each learning step.

By separating graph feature aggregation and weight learning, SGC precomputes $\boldsymbol{S}^K\boldsymbol{X}$ before learning $\boldsymbol{W}$. The other methods also follow the same idea: separating feature aggregation and weight learning and precomputing feature aggregation.

3 GNN Transformation Framework

We propose a general framework that automatically transforms non-scalable GNNs to efficient and scalable precomputation-based GNNs and efficiently executes precomputation of feature aggregation on GPU. We first introduce a transformation procedure that automatically rewrites the formulations of non-scalable GNNs so that the transformed GNNs run efficiently and scale well to large-scale graphs (Section 3.1). We also describe a limitation of our transformation, namely, that it does not support GNNs that require dynamical changes of graph filters during weight learning. Our transformation procedure is applicable not only to GCN [15] but also to the state-of-the-art GNNs, such as JKNet [27], H2GCN [32] and GPRGNN [7]. Next, we introduce a block-wise precomputation scheme that efficiently computes feature aggregation for large-scale graphs (Section 3.2). The core idea is to decompose an adjacency matrix and feature matrix into disjoint and balanced blocks each of which can be handled on GPU. Also, we formulate and solve an optimization problem that decides the optimal size of blocks. Note that this scheme is a general approach since it can be applied to existing precomputation-based GNNs [24, 8, 18].

3.1 Linear Convolution Transformation

LC transformation is the first concrete procedure that transforms non-scalable GNNs to efficient and scalable precomputation-based GNNs, which have a similar functionality to the input GNNs. We call the output the *LC version* of the input GNN. LC transformation is motivated by the effectiveness of SGC and Multi-Layer Perceptron (MLP). SGC preserves the major benefit of graph convolution with efficient training by precomputing feature aggregation, but it degrades the accuracy due to the lack of non-linearity [7]. Beside, MLP outperforms linear regression in classification task by using non-linear functions but does not capture the structures of graphs. LC version of GNN leverages both the strengths of SGC and MLP by precomputing feature aggregation and then learning weights with non-linearity.

Figure 1 demonstrates an example of LC transformation by comparing it with non-scalable GNNs. Intuitively, LC transformation separates feature aggregation from graph convolution that performs 1) feature aggregation, 2) weight multiplication, and 3) activation function application (e.g., ReLU, a non-linear function). Notice that a normalized adjacency matrix $\boldsymbol{S}$ is adjacent to the feature matrix $\boldsymbol{X}$ in the formulation of LC versions (see the left part of the red box of the figure). So, we can precompute $\boldsymbol{S}^k\boldsymbol{X}$ in the same way as SGC [24]. Thanks to the separation, LC versions can avoid computing feature aggregation whenever learnable weights are updated at each learning step (see the right part of the red box of the figure). Hence, LC versions efficiently work and scale well to large-scale graphs.

Discussion. We discuss why LC versions work from two aspects, feature aggregation and weight learning. As in the discussion on the spectral analysis [24], feature aggregation acts as a low-pass filter that produces smooth features over the graph, which is the major benefit of graph convolution. In this sense, LC versions are expected to have the same functionality as the input GNNs since LC transformation preserves feature aggregation within multi-hops. As for weight learning, LC versions have a similar learning capability to their original GNNs since they have a similar model architecture of multi-layer neural networks. As a result, LC versions can achieve a similar prediction performance to their original GNNs while scaling to large-scale graphs.

Procedure. Next, we describe the procedure of LC transformation, which removes non-linear functions from graph convolution, but incorporates non-linear functions into weight learning. We first give the definition of LC transformation below:

Definition (LC transformation)**.** *Given a non-scalable GNN algorithm, LC transformation iteratively applies a function f_{LC} to the formulation of the input GNN since non-scalable GNNs have multiple graph convolutional layers. f_{LC} commutes matrix multiplication of $\boldsymbol{S}$ and a non-linear function σ as follows:*

$$f_{LC} : g_2(\boldsymbol{S}\sigma(g_1(\boldsymbol{X}))) \underset{f_{LC}}{\longrightarrow} g_2(\sigma(\boldsymbol{S}g_1(\boldsymbol{X}))), \tag{6}$$

where g_1 and g_2 indicate any functions that input and output matrices. The iteration continues until the formulation does not change. LC transformation outputs a precomputation-based GNN having the transformed formulation, i.e., the LC version of the input GNN.

To intuitively explain the details, we use JKNet [27] as an example, which is a widely used GNN. The formulation of JKNet (GCN-based) is as follows:

$$\boldsymbol{H} = \mathrm{COMB}_{k=1}^{K}(\boldsymbol{S}\sigma(\boldsymbol{S}\sigma(\dots(\boldsymbol{S}\boldsymbol{X}\boldsymbol{W}_1)\dots)\boldsymbol{W}_{k-1})\boldsymbol{W}_k), \tag{7}$$

where COMB expresses a skip connection between different layers, such as concatenation of intermediate representations or max pooling. By applying a softmax function to feature representations $\boldsymbol{H}$, JKNet outputs a prediction result $\boldsymbol{Y}$, i.e., $\boldsymbol{Y} = \mathrm{softmax}(\boldsymbol{H})$. We apply f_{LC} to it in order to transform the formulation of an input GNN. To this end, we assign $g_1(\boldsymbol{X}) = \boldsymbol{S}\sigma(\dots(\boldsymbol{S}\boldsymbol{X}\boldsymbol{W}_1)\dots)\boldsymbol{W}_{k-1}$ and $g_2(\boldsymbol{S}\sigma(g_1(\boldsymbol{X}))) = \mathrm{COMB}_{k=1}^{K}(\boldsymbol{S}\sigma(g_1(\boldsymbol{X}))\boldsymbol{W}_k)$. By utilizing f_{LC}, g_1, and g_2, we transform Eq (7) as follows:

$$\boldsymbol{H} \underset{f_{LC}}{\longrightarrow} \mathrm{COMB}_{k=1}^{K}(\sigma(\boldsymbol{S}^2\sigma(\dots(\boldsymbol{S}\boldsymbol{X}\boldsymbol{W}_1)\dots)\boldsymbol{W}_{k-1})\boldsymbol{W}_k). \tag{8}$$

Then, we iteratively apply f_{LC} to the formulation by appropriately assigning g_1 and g_2 for each iteration. Finally, we obtain the formulation of the LC version of the input GNN, $\boldsymbol{H}^{LC}$, as follows:

$$\boldsymbol{H}^{LC} = \mathrm{COMB}_{k=1}^{K}(\sigma(\sigma(\dots(\boldsymbol{S}^k\boldsymbol{X}\boldsymbol{W}_1)\dots)\boldsymbol{W}_{k-1})\boldsymbol{W}_k). \tag{9}$$

Then, in the same way as the input GNN, the LC version outputs a predicted label matrix $\boldsymbol{Y} = \mathrm{softmax}(\boldsymbol{H}^{LC})$.

The LC transformation procedure is applicable not only to JKNet but also to general non-scalable GNNs including APPNP [16], MixHop [1], H2GCN [32], and GPRGNN [7]. We give another example of applying LC transformation in Appendix A.

Limitation. Precomputation-based GNNs can use multiple graph filters such as an exact 1-hop away adjacency matrix and Personalized PageRank diffusion matrix [16]. Those GNNs do not dynamically control the propagation of features during weight learning, since they use constant graph filters in order to precompute feature aggregation. Since our framework also leverages a precomputation scheme, it cannot support those existing GNNs [22, 26, 21] which dynamically sample edges or modify the importance of edges during weight learning. For example, Dropedge [21] randomly reduces a certain number of edges at each iteration. A possible future research direction is that we simulate random edge reduction by utilizing the deviations of feature aggregation.

3.2 Efficient Precomputation

Existing precomputation-based GNNs need to use CPUs to compute feature aggregations for large-scale graphs since they do not fit on GPU memory. This CPU computation has large cost and a deteriorating effect on efficiency.

To tackle this problem, we propose a simple yet efficient block-wise precomputation scheme and provide a formulation for optimal decomposition for our block-wise precomputation scheme. The core idea is to decompose the edge set of a given graph into disjoint and balanced groups, while existing approaches [31] decompose the node set into groups, i.e., row/column wise decomposition. Our scheme is inspired by edge partitioning [17, 9], which aims to decompose a graph into groups having similar numbers of edges such that communication costs for graph operations are minimized in distributed environments. Our scheme consists of three steps. First, it decomposes an adjacency matrix and feature matrix into small disjoint blocks each of which can be put on GPU memory. Second, the scheme computes block-wise matrix operations for the disjoint blocks on GPU. Third, it aggregates the results of the block-wise matrix operations and obtains the whole matrix operation result.

Precomputation on GPU. There are two matrix operations to be precomputed, adjacency matrix normalization and feature aggregation. First, we describe the computation of adjacency matrix normalization shown by Eq. (1). Since an adjacency matrix is typically sparse, we utilize adjacency list $(i, j) \in \mathcal{E}$, where $\tilde{\boldsymbol{A}}_{ij} = 1$. To obtain small blocks each of which can be loaded on GPU memory, we decompose $\mathcal{E}$ into disjoint sets that include similar numbers of edges, $\mathcal{E}^{(1)} \cup \cdots \cup \mathcal{E}^{(a)}$, where a is a number of sets and $\mathcal{E}^{(p)} \cap \mathcal{E}^{(q)} = \emptyset$ if $p \neq q$. Note that the sizes of the sets $\mathcal{E}^{(1)}, \ldots, \mathcal{E}^{(a)}$ are balanced. Then, we decompose $\tilde{\boldsymbol{A}} = \tilde{\boldsymbol{A}}^{(1)} + \cdots + \tilde{\boldsymbol{A}}^{(a)}$, where $\tilde{\boldsymbol{A}}^{(1)}, \ldots, \tilde{\boldsymbol{A}}^{(a)} \in \mathbb{R}^{n \times n}$ and $\tilde{\boldsymbol{A}}^{(l)}_{ij} = 1$ if $(i, j) \in \mathcal{E}^{(l)}$. Then, we can rewrite Eq. (1) as follows:

$$\boldsymbol{S} = \tilde{\boldsymbol{D}}^{-\frac{1}{2}} \tilde{\boldsymbol{A}} \tilde{\boldsymbol{D}}^{-\frac{1}{2}} = \textstyle\sum_{l=1}^{a} \tilde{\boldsymbol{D}}^{-\frac{1}{2}} \tilde{\boldsymbol{A}}^{(l)} \tilde{\boldsymbol{D}}^{-\frac{1}{2}}. \tag{10}$$

By appropriately selecting the number of blocks a, $\tilde{\boldsymbol{D}}^{-\frac{1}{2}} \tilde{\boldsymbol{A}}^{(l)} \tilde{\boldsymbol{D}}^{-\frac{1}{2}}$ can be executed on GPU. We sum the results of the block-wise matrix computations. This summation can be efficiently computed on CPU by disjoint union of edge lists since $\mathcal{E}^{(l)}$, i.e., $\tilde{\boldsymbol{A}}^{(l)}$, is disjoint each other. Since our decomposition is agnostic on nodes, the decomposed blocks can be easily balanced while row/column(node)-wise decomposition approaches suffer from an imbalance problem. Further discussion on Limitations follows below in this subsection.

Next, we introduce a block-wise computation for feature aggregation on GPU. Algorithm 1 describes the procedure of the computation. To obtain small blocks of a normalized adjacency matrix $\boldsymbol{S}$, we decompose it into $\boldsymbol{S}^{(1)}, \ldots, \boldsymbol{S}^{(b)} \in \mathbb{R}^{n \times n}$ where b is a number of blocks (line 2). Similarly to the decomposition of $\boldsymbol{A}$, each corresponding edge list is disjoint and includes similar numbers of edges. Also, in order to obtain small blocks of a feature matrix $\boldsymbol{X}$, we decompose it into $\boldsymbol{X}^{(1)}, \ldots, \boldsymbol{X}^{(c)}$, where c is a number of blocks (line 5). Since we assume that $\boldsymbol{X}$ is a dense matrix, we adopt column-wise decomposition, i.e., $\boldsymbol{X} = \mathrm{concat}(\boldsymbol{X}^{(1)}, \ldots, \boldsymbol{X}^{(c)})$. Then, we compute matrix multiplication $\boldsymbol{S}^{(j)} \boldsymbol{X}^{(i)}$ for each pair on GPU (line 9). We aggregate $\boldsymbol{S}^{(j)}$ by summation (line 10) and aggregate $\boldsymbol{X}_{tmp}$ by concatenation (lines 11–14). $\boldsymbol{X}_{prev}$ is updated by the aggregated features $\boldsymbol{X}_{conc}$ (line 16). We repeat this aggregation K times (lines 4–16).

Algorithm 1 Block-wise feature aggregation.

Require: normalized adjacency matrix $\boldsymbol{S}$, feature matrix $\boldsymbol{X}$, number of layers K

Ensure: aggregated feature list *SX_list*

SX_list = []
$\boldsymbol{S}^{(1)}, \boldsymbol{S}^{(2)}, \ldots, \boldsymbol{S}^{(b)} = \text{split}(\boldsymbol{S})$ ▷ disjoint edge sets
$\boldsymbol{X}_{prev} = \boldsymbol{X}$
for $k = 1$ to K **do**
  $\boldsymbol{X}^{(1)}, \boldsymbol{X}^{(2)}, \ldots, \boldsymbol{X}^{(c)} = \text{split}(\boldsymbol{X}_{prev})$
  for $i = 1$ to c **do**
    $\boldsymbol{X}_{tmp} = [0]^{n \times \lceil d/c \rceil}$ ▷ same size to $\boldsymbol{X}^{(i)}$
    for $j = 1$ to b **do**
      $\boldsymbol{Z}_{tmp} = \boldsymbol{S}^{(j)} \boldsymbol{X}^{(i)}$ ▷ on GPU
      $\boldsymbol{X}_{tmp} = \boldsymbol{X}_{tmp} + \boldsymbol{Z}_{tmp}$ ▷ on GPU
    if $i == 1$ **then**
      $\boldsymbol{X}_{conc} = \boldsymbol{X}_{tmp}$ ▷ on CPU
    else
      $\boldsymbol{X}_{conc} = \text{concat}(\boldsymbol{X}_{conc}, \boldsymbol{X}_{tmp})$ ▷ on CPU
  SX_list.append($\boldsymbol{X}_{conc}$)
  $\boldsymbol{X}_{prev} = \boldsymbol{X}_{conc}$

Optimal graph decomposition. We discuss an optimal decomposition for our block-wise precomputation scheme. We have two requirements to decompose large matrices into disjoint blocks. First, each matrix operation for disjoint blocks can be executed on GPU. Second, the number of disjoint blocks is as small as possible to reduce the number of block-wise matrix operations. To simplify the discussion, we assume that the running time of a matrix operation on GPU is the same regardless of the matrix size.

As for the block-wise adjacency matrix normalization, we minimize a number of disjoint blocks, a. We formulate the minimization as follows:

$$\min(a), \text{ subject to } \frac{\alpha_{\boldsymbol{A}} B_{\boldsymbol{A}} + \alpha_{\boldsymbol{S}} B_{\boldsymbol{S}}}{a} + \alpha_{\boldsymbol{D}} B_{\boldsymbol{D}} \leq B_{\text{GPU}}, \tag{11}$$

where $\alpha_{\boldsymbol{A}}, \alpha_{\boldsymbol{S}}, \alpha_{\boldsymbol{D}}$ indicate coefficients for executing matrix operations regarding $\boldsymbol{A}, \boldsymbol{S}, \boldsymbol{D}$, respectively, and $B_{\boldsymbol{A}}, B_{\boldsymbol{S}}, B_{\boldsymbol{D}}, B_{\text{GPU}}$ indicate the volume of an adjacency matrix, the volume of a normalized adjacency matrix, the volume of a degree matrix, and the available volume of a GPU, respectively. As for block-wise feature aggregation, we minimize the number of pairs of disjoint blocks, bc. We formulate the minimization as follows:

$$\min_{b,c}(bc), \text{ subject to } \frac{\beta_{\boldsymbol{S}} B_{\boldsymbol{S}}}{b} + \frac{\beta_{\boldsymbol{X}} B_{\boldsymbol{X}}}{c} \leq B_{\text{GPU}}, \tag{12}$$

Table 1: Summary of datasets.

Dataset	Nodes	Edges	Features	Classes
Flickr	89,250	899,756	500	7
Reddit	232,965	11,606,919	602	41
arxiv	169,343	1,166,243	128	40
papers100M	111,059,956	1,615,685,872	128	172

where $\beta_{\boldsymbol{S}}, \beta_{\boldsymbol{X}}$ indicate coefficients for executing matrix operations regarding $\boldsymbol{S}, \boldsymbol{X}$, respectively, and $B_{\boldsymbol{X}}$ indicates the volume of a feature matrix. Note that $\alpha_{\boldsymbol{A}}, \alpha_{\boldsymbol{S}}, \alpha_{\boldsymbol{D}}, \beta_{\boldsymbol{S}}$, and $\beta_{\boldsymbol{X}}$ depend on execution environments[4].

Next, we discuss optimization regarding Eq. (11) and (12). As for Eq. (11), it is trivial to find the minimum number of blocks a since there are no other parameters. As for Eq. (12), an exhaustive search is applicable since the number of combinations of b and c (natural numbers) is not large. Consequently, these optimization problems can be easily solved.

Limitation. Our precomputation scheme focuses on feature aggregation on a whole graph. This indicates that our scheme is not suitable for node-wise operations since it may decompose the edge set of the same node into different groups. However, accelerating feature aggregation on a whole graph is still crucial since many graph neural networks [24, 8, 18, 15] adopt it.

4 Experiments

We design our experiments to answer the following questions; **Q1**: Can our LC transformation improve the efficiency and scalability of GNNs? **Q2**: Can our block-wise precomputation scheme accelerate precomputation?

Dataset. We use four commonly used datasets, `Flickr` [29], `Reddit` [10], `ogbn-arxiv` (arxiv for short), and `ogbn-papers100M` (papers100M for short) [12]. Table 1 provides the summary of the datasets. The sizes of the datasets range from 9K nodes to 110M.

In the Flickr dataset, nodes represent images uploaded to Flickr. If two images share common properties such as same geographic location, same gallery, comments by the same users, there is an edge between the nodes. Node features represent the 500-dimensional bag-of-words associated with the image (node). As for node labels, the authors of [29] scan over 81 tags of each image and manually merged them to 7 classes. In the Reddit dataset, nodes represent posts. If the same user left comments on two posts, then there is an edge between the two posts. Node features are the embedding of the contents of the posts. The labels of nodes indicate communities which the nodes belong to. In the ogbn-arxiv

[4] In real environments, users can measure $\alpha_{\boldsymbol{A}}, \alpha_{\boldsymbol{S}}, \alpha_{\boldsymbol{D}}, \beta_{\boldsymbol{S}}$, and $\beta_{\boldsymbol{X}}$ by monitoring the memory usage on small graphs, even if users do not know the details of their own environments.

dataset, nodes represent ARXIV papers and edges indicate that one paper cites another one. Node features represent 128-dimensional feature vectors obtained by averaging the embeddings of words in titles and abstracts. Node labels indicate subject areas of ARXIV CS papers[5]. In the ogbn-papers100M (papers100M) dataset, its graph structure and node features are constructed in the same way as ogbn-arxiv. Among its nodes, approximately 1.5 million nodes are labeled with one of ARXIV's subject areas. As in [28], Flickr and Reddit are under the inductive setting. ogbn-arxiv and ogbn-papers100M are under the transductive setting.

Baseline. We compare three types of existing methods as baselines; non-scalable GNNs, precomputation-based GNNs, and sampling-based GNNs which are scalable but inefficient (we discuss the details in Section 5). As for non-scalable GNNs, we use GCN[6] [15], JKNet[7] [27], and GPRGNN[8] [7]. As for precomputation-based GNNs, we use SGC[9] [24] and FSGNN[10] [18]. As for sampling-based GNNs, we use ShaDow-GNN[11] [28]. FSGNN and ShaDow-GNN are the state-of-the-art precomputing-based and sampling-based GNNs, respectively. We note that we use our block-wise precomputation to the precomputation-based GNNs instead of using its original CPU computation for a fair comparison.

Setup. We tune hyperparameters on each dataset by Optuna [2] and use Adam optimizer [14]. We adopt mini-batch training for precomputation-based GNNs, sampling-based GNNs, and LC-versions to deal with large-scale graphs[12]. As for ShaDow-GNN, we use the best hyperparameter sets provided by the authors and adopt GAT [22] as a backbone model since ShaDow-GAT achieves the best accuracy in most cases reported in the paper. We measure training time on a NVIDIA Tesla V100S GPU (32GB) and Intel(R) Xeon(R) Gold 5220R CPUs (378GB).

4.1 Effectiveness of LC Transformation (Q1)

Table 2 shows the test accuracy of LC versions and the baselines. LC versions (GCN_LC, JKNet_LC, and GPRGNN_LC) achieve comparable test accuracy with their original GNNs (GCN, JKNet, and GPRGNN) for all datasets. Next, Table 3 shows the training time of LC versions and the baselines. The LC versions run faster than their original GNNs. Note that LC versions tend to stop earlier than non-scalable GNNs since LC versions train their models more times due

[5] https://arxiv.org/archive/cs
[6] https://github.com/tkipf/pygcn
[7] Since official codes of JKNet from the authors are not provided, we simply implement JKNet based on the implementation of GCN.
[8] https://github.com/jianhao2016/GPRGNN
[9] https://github.com/Tiiiger/SGC
[10] https://github.com/sunilkmaurya/FSGNN
[11] https://github.com/facebookresearch/shaDow_GNN
[12] We will provide hyperparameter search space and the best parameters to reproduce experiments on our codebase that will be publicly available on acceptance.

Table 2: Comparison on test accuracy. We report the average values (standard deviation) over 5 runs.

	Flickr	Reddit	arxiv
GCN	0.525(0.003)	0.945(0.000)	0.702(0.005)
JKNet	0.526(0.004)	0.941(0.006)	0.712(0.001)
GPRGNN	0.494(0.006)	0.918(0.012)	0.694(0.006)
SGC	0.494(0.037)	0.948(0.001)	0.692(0.004)
FSGNN	0.513(0.001)	0.964(0.001)	0.722(0.003)
ShaDow-GAT	0.531(0.003)	0.947(0.003)	0.716(0.004)
GCN_LC	0.515(0.003)	0.947(0.001)	0.710(0.001)
JKNet_LC	0.517(0.004)	0.951(0.000)	0.710(0.003)
GPRGNN_LC	0.513(0.001)	0.961(0.000)	0.720(0.004)

Table 3: Comparison on training time (per epoch/total). Note that total training time includes precomputation time for SGC, FSGNN, ShaDow-GAT, GCN_LC, JKNet_LC, and GPRGNN_LC. We report the average values over 5 runs.

	Flickr	Reddit	arxiv
GCN	64.62[ms] / 129.24[s]	654.70[ms] / 1309.40[s]	210.81[ms] / 421.63[s]
JKNet	170.43[ms] / 253.25[s]	1428.51[ms] / 2552.45[s]	529.05[ms] / 1058.10[s]
GPRGNN	272.86[ms] / 539.48[s]	1456.01[ms] / 2806.62[s]	523.08[ms] / 961.76[s]
SGC	51.18[ms] / 30.31[s]	141.68[ms] / 285.43[s]	50.27[ms] / 42.23[s]
FSGNN	346.97[ms] / 133.63[s]	1066.66[ms] / 1793.91[s]	284.73[ms] / 382.67[s]
ShaDow-GAT	120.85e3[ms] / 3634.65[s]	376.42e3[ms] / 11321.09[s]	163.67e3[ms] / 4913.29[s]
GCN_LC	56.75[ms] / 49.85[s]	165.73[ms] / 212.16[s]	62.59[ms] / 120.60[s]
JKNet_LC	144.78[ms] / 78.24[s]	430.41[ms] / 865.71[s]	138.52[ms] / 277.63[s]
GPRGNN_LC	287.54[ms] / 164.88[s]	818.13[ms] / 1645.49[s]	219.66 [ms] / 204.56[s]

to mini-batch training. For example, in Flickr data LC versions more efficiently train than non-scalable GNNs even if they have similar training time per epoch. These results indicate that our framework transforms non-scalable GNNs to efficient precomputation-based GNNs with the comparable classification accuracy to the original GNNs.

Comparison on large-scale graph. Table 4 shows the performance comparison on papers100M having more than 100 million nodes and one billion edges. Non-scalable GNNs (GCN, JKNet, and GPRGNN) cannot work on papers100M since the whole graph cannot be put on GPU memory. GPRGNN_LC achieves comparable accuracy (approximate one percent difference) with FSGNN, which is the state-of-the-art precomputation-based GNN while GPRGNN_LC runs faster than FSGNN. Though ShaDow-GAT achieves the highest accuracy, it requires more than 10× total training time than other models. This is because it needs to operate graph convolutions on many enclosing subgraphs extracted from the whole graph. SGC obtains lower accuracy than GCN_LC. This result

Table 4: Results on papers100M. We show test/validation accuracy (standard deviation) and training time (per epoch / total). Total training time includes precomputation time. OOM indicates that the execution is out of memory.

	Test accuracy	Val accuracy	Time (epoch / total)
GCN	OOM	OOM	OOM
JKNet	OOM	OOM	OOM
GPRGNN	OOM	OOM	OOM
SGC	0.623(0.007)	0.667(0.002)	425.15[ms] / 2211.23[s]
FSGNN	0.665(0.003)	0.706(0.001)	3550.82[ms] / 8612.48[s]
ShaDow-GAT	0.666(0.003)	0.703(0.001)	2948.50e3[ms] / 92264.76[s]
GCN_LC	0.647(0.006)	0.688(0.002)	611.90[ms] / 2477.55[s]
JKNet_LC	0.641(0.003)	0.689(0.004)	1488.80[ms] / 3396.69[s]
GPRGNN_LC	0.658(0.002)	0.696(0.001)	2749.27[ms] / 7410.47[s]

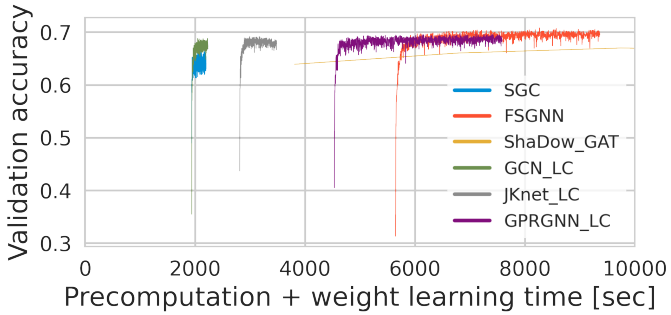

Fig. 2: Validation accuracy over training time (precomputation and weight learning time) on papers100M. Plots indicate epochs. LC versions (GCN_LC, JKNet_LC, and GPRGNN_LC) are faster than FSGNN and ShaDow-GAT while achieving competitive accuracy.

validates that non-linearity contributes to weight learning for better classification.

In order to analyze the results on papers100M in details, we show the validation accuracy at each epoch over total training time in Figure 2. Note that total training time consists of precomputation and weight learning time. We observe that GCN_LC, JKNet_LC, and GPRGNN_LC are plotted in the upper left corner of the figure. This observation indicates that they require less total training time than FSGNN and ShaDow-GAT. The LC versions achieve competitive performance with them. Through these experiments, we demonstrate that LC versions are efficient and scalable for large-scale graphs.

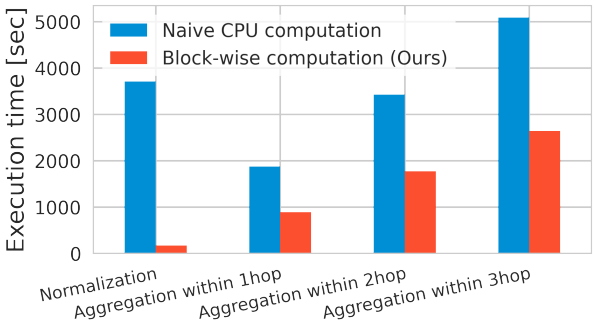

Fig. 3: Precomputation time comparison between a naive CPU computation and our block-wise computation.

4.2 Precomputation Efficiency (Q2)

To validate the efficiency of our block-wise precomputation, we compare it with naive CPU computation adopted by existing works [12, 18]. We use a large-scale graph, papers100M, which requires a 67GB normalized adjacency matrix and a 57GB feature matrix. For adjacency matrix normalization, we set the number of disjoint blocks of an adjacency matrix to $a = 3$, which satisfies Eq. (11). Also, for feature aggregation we set numbers of disjoint blocks of a normalized adjacency matrix and feature matrix to $b = 10, c = 16$, respectively, which satisfy Eq. (12).

Figure 3 shows the precomputation time for normalization and feature aggregation on CPU and GPU. The result demonstrates that our block-wise precomputation is 20× faster than CPU computation for normalization. Also, the result indicates that our precomputation is up to twice faster than CPU computation for feature aggregation. Hence, we conclude that our precomputation is more efficient than CPU computation on a single machine.

5 Related Work

Relationship between non-scalable GNNs and LC versions. We discuss the background of non-scalable GNNs and their LC versions. Graph convolution is motivated by the 1-dim Weisfeiler-Lehman (WL-1) algorithm [23] which is used to test graph isomorphism; two graphs are called isomorphic if they are topologically identical. WL-1 iteratively aggregates the labels of nodes and their neighbors, and hashes the aggregated labels into unique labels. The algorithm decides whether two graphs are isomorphic or not by using the labels of nodes at some iteration. Non-scalable GNNs such as GCN [15] replace the hash function of WL-1 with a graph convolutional layer which consists of feature aggregation, weight multiplication, and non-linear function application. As for LC versions, they replace the hash function of WL-1 with feature aggregation. These observations indicate that WL-1 is analogous to feature aggregation of LC versions, similarly to graph convolution of non-scalable GNNs.

Sampling-based GNNs. Sampling-based GNNs [10, 5, 6, 29, 28] avoid keeping a whole graph on GPU by computing node representations from enclosing subgraphs of the input graph. The major drawback of the sampling-based GNNs is that they need costly training time since they need to operate graph convolutions on many enclosing subgraphs extracted from the input graph.

GNNs dynamically modifying the importance of edges. As we discussed in Section 3.1, our transformation cannot support GNNs which dynamically control the propagation of features during weight learning. An example of such GNNs is GAT [22], which learns attention parameters controlling the importance of edges for each iteration. Another example is GIN [26] learns a parameter controlling a weight between self features and features from neighbors. One possible direction is that we first determine the parameters by training on a subset of an input graph, then fix them in order to precompute feature aggregation.

Distributed matrix operations. Matrix operations can be parallelized for distributed computing [4] [3]. For example, the authors of [11] proposed Mars which is an approach for hiding the programming complexity of MapReduce on GPU. Also, MR-Graph [20] is a customizable and unified framework for GPU-based MapReduce. It allows its users to implement their applications more flexibly. As for distributed graph neural network training, DistDGL [31] has proposed mini-batch training on graphs, which scales beyond a single machine. It suffers from an imbalance problem since it uses a typical graph clustering algorithm METIS [13] to partition large-scale graphs into subgraphs, while our scheme can partition an edge set into balanced subsets. For further scale up of graphs, it would be important to combine distributed computing and our block-wise precomputation for graphs.

6 Conclusion

We presented a framework that automatically transforms non-scalable GNNs to efficient and scalable precomputation-based GNNs. There are two major characteristics of our framework: 1) it supports a novel transformation procedure that transforms non-scalable GNNs to efficient and scalable precomputation-based GNNs having a similar functionality to the original GNNs, 2) the precomputation of the transformed GNNs can be efficiently executed by our block-wise precomputation scheme that decomposes large-scale graphs into disjoint and balanced blocks each of which can be handled on GPU memory. Through our experiments, we demonstrated that the transformed GNNs run more efficiently than their original GNNs and can be scaled to graphs with millions of nodes and billions of edges. Due to the strong performance of LC versions, we argue that LC versions will be beneficial as baseline comparisons for future research on scalable GNNs.

Acknowledgement. This work was supported by JSPS KAKENHI Grant Numbers JP20H00583 and JST PRESTO Grant Number JPMJPR21C5.

References

1. Abu-El-Haija, S., Perozzi, B., Kapoor, A., Alipourfard, N., Lerman, K., Harutyunyan, H., Ver Steeg, G., Galstyan, A.: Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In: ICML (2019)
2. Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: A next-generation hyperparameter optimization framework. In: KDD (2019)
3. Awaysheh, F.M., Alazab, M., Garg, S., Niyato, D., Verikoukis, C.: Big data resource management & networks: Taxonomy, survey, and future directions. IEEE Communications Surveys & Tutorials (2021)
4. Boehm, M., Dusenberry, M.W., Eriksson, D., Evfimievski, A.V., Manshadi, F.M., Pansare, N., Reinwald, B., Reiss, F.R., Sen, P., Surve, A.C., et al.: Systemml: Declarative machine learning on spark. PVLDB **9**(13) (2016)
5. Chen, J., Ma, T., Xiao, C.: Fastgcn: fast learning with graph convolutional networks via importance sampling. arXiv preprint (2018)
6. Chiang, W.L., Liu, X., Si, S., Li, Y., Bengio, S., Hsieh, C.J.: Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In: KDD (2019)
7. Chien, E., Peng, J., Li, P., Milenkovic, O.: Adaptive universal generalized pagerank graph neural network. In: ICLR (2021), https://openreview.net/forum?id=n6jl7fLxrP
8. Frasca, F., Rossi, E., Eynard, D., Chamberlain, B., Bronstein, M., Monti, F.: Sign: Scalable inception graph neural networks. In: ICML 2020 Workshop on Graph Representation Learning and Beyond (2020)
9. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: 10th USENIX symposium on operating systems design and implementation (OSDI 12). pp. 17–30 (2012)
10. Hamilton, W.L., Ying, R., Leskovec, J.: Inductive representation learning on large graphs. In: NeurIPS (2017)
11. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a mapreduce framework on graphics processors. In: PACT (2008)
12. Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., Leskovec, J.: Open graph benchmark: Datasets for machine learning on graphs. arXiv preprint (2020)
13. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on scientific Computing **20**(1), 359–392 (1998)
14. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint (2014)
15. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: ICLR (2017)
16. Klicpera, J., Bojchevski, A., Günnemann, S.: Predict then propagate: Graph neural networks meet personalized pagerank. In: ICLR (2019)
17. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning in the cloud. PVLDB (2012)
18. Maurya, S.K., Liu, X., Murata, T.: Improving graph neural networks with simple architecture design. arXiv preprint (2021)
19. Newman, M.E.: Networks: An Introduction. Oxford University Press (2010)

20. Qiao, Z., Liang, S., Jiang, H., Fu, S.: A customizable mapreduce framework for complex data-intensive workflows on gpus. In: IPCCC (2015)
21. Rong, Y., Huang, W., Xu, T., Huang, J.: Dropedge: Towards deep graph convolutional networks on node classification. arXiv preprint (2019)
22. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint (2017)
23. Weisfeiler, B., Lehmann, A., A.: A reduction of a graph to a canonical form and an algebra arising during this reduction. Nauchno-Technicheskaya Informatsia, 2(9):12–16 (1968)
24. Wu, F., Souza, A., Zhang, T., Fifty, C., Yu, T., Weinberger, K.: Simplifying graph convolutional networks. In: ICML (2019)
25. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y.: A comprehensive survey on graph neural networks. IEEE Transactions on Neural Networks and Learning Systems **32**(1) (2020)
26. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? arXiv preprint arXiv:1810.00826 (2018)
27. Xu, K., Li, C., Tian, Y., Sonobe, T., Kawarabayashi, K.i., Jegelka, S.: Representation learning on graphs with jumping knowledge networks. In: ICML (2018)
28. Zeng, H., Zhang, M., Xia, Y., Srivastava, A., Malevich, A., Kannan, R., Prasanna, V.K., Jin, L., Chen, R.: Deep graph neural networks with shallow subgraph samplers. CoRR **abs/2012.01380** (2020), https://arxiv.org/abs/2012.01380
29. Zeng, H., Zhou, H., Srivastava, A., Kannan, R., Prasanna, V.: GraphSAINT: Graph sampling based inductive learning method. In: International Conference on Learning Representations (2020), https://openreview.net/forum?id=BJe8pkHFwS
30. Zhang, Z., Cui, P., Zhu, W.: Deep learning on graphs: A survey. IEEE TKDE (2020)
31. Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., Gan, Q., Zhang, Z., Karypis, G.: Distdgl: distributed graph neural network training for billion-scale graphs. In: 2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3). pp. 36–44. IEEE (2020)
32. Zhu, J., Yan, Y., Zhao, L., Heimann, M., Akoglu, L., Koutra, D.: Beyond homophily in graph neural networks: Current limitations and effective designs. NeurIPS **33** (2020)

A LC Version of GPRGNN

We show an example of LC transformation for the state-of-the-art GNN model, GPRGNN [7]. We give the formulation of GPRGNN as follows:

$$\boldsymbol{H} = \sum_{k=0}^{K} \gamma_k \boldsymbol{S}^k(\sigma(\dots\sigma(\boldsymbol{X}\boldsymbol{W}_1)\dots)\boldsymbol{W}_T), \tag{13}$$

where γ_k is an attention parameter learning the importance of k-th layer and T is the number of layers for Multi-layer perceptrons. Note that S^k cannot be efficiently precomputed since the number of non-zero elements significantly increases when $k \geq 2$ for large-scale graphs. By iteratively applying f_{LC} to Eq. (13), we obtain the formulation of its LC version as follows:

$$\boldsymbol{H}^{LC} = \sum_{k=0}^{K} \gamma_k(\sigma(\dots\sigma(\boldsymbol{S}^k\boldsymbol{X}\boldsymbol{W}_1)\dots)\boldsymbol{W}_T). \tag{14}$$

$\boldsymbol{S}^k\boldsymbol{X}$ can be precomputed since it does not need to be updated when learnable weights $\boldsymbol{W}_1 \dots \boldsymbol{W}_T$ and a parameter γ are updated.