An Edge Attribute-wise Partitioning and Distributed Processing of R-GCN using GPUs

Tokio Kibata✉, Mineto Tsukada, and Hiroki Matsutani

Keio University, 3-14-1, Hiyoshi, Yokohama, Japan 223-8522
{tokio,tsukada,matutani}@arc.ics.keio.ac.jp

Abstract. R-GCN (Relational Graph Convolutional Network) is one of GNNs (Graph Neural Networks). The model tries predicting latent information by considering directions and types of edges in graph-structured data, such as knowledge bases. The model builds weight matrices to each edge attribute. Thus, the size of the neural network increases linearly with the number of edge types. Although GPUs can be used for accelerating the R-GCN processing, there is a possibility that the size of weight matrices exceeds GPU device memory. To address this issue, in this paper, an edge attribute-wise partitioning is proposed for R-GCN. The proposed partitioning divides the model and graph data so that R-GCN can be accelerated by using multiple GPUs. Also, the proposed approach can be applied to sequential execution on a single GPU. Both the cases can accelerate the R-GCN processing with large graph data, where the original model cannot be fit into a device memory of a single GPU without partitioning. Experimental results demonstrate that our partitioning method accelerates R-GCN by up to 3.28 times using four GPUs compared to CPU execution for a dataset with more than 1.6 million nodes and 5 million edges. Also, the proposed approach can accelerate the execution even with a single GPU by 1.55 times compared to the CPU execution for a dataset with 0.8 million nodes and 2 million edges.

Keywords: GPU · R-GCN · GNN · Graph data · Knowledge base

1 Introduction

In recent years, it is expected that the next step of deep learning would be responding to the various structured data. Indeed, conventional deep learning models typically use data represented in Euclidean space. Meanwhile, one of new streams of deep learning is to use graph-structured data, which is represented in non-Euclidean space, such as GNNs (Graph Neural Networks) [7]. An algorithm applying CNN (Convolutional Neural Networks) for graph-structured data, called ConvGNNs (Convolutional GNNs), demonstrates practical results [4, 3].
R-GCN (Relational-Graph Convolutional Network) [6] is a derivative model of ConvGNNs and aims at filling in a lack of knowledge base. Missing data in a

knowledge base can be classified into two types. One is a lack of attributes of nodes, and the other is a lack of links between nodes, called edges, on the graph. The edges have relational types of two nodes in some cases. Considering the edge types, R-GCN builds weight matrices for each type and direction (i.e., in and out of node) of edges. When predicting latent node attributes or edges on GNNs, the scalability problems always lie on. It is challenging to parallelize the model or graph processing of R-GCN using multiple GPUs for accelerating the execution. Particularly, R-GCN has a specific issue of scalability, because the size of weight matrices increases linearly with the number of edge types in addition to the number of nodes when these features are defined as one-hot vectors. There is a possibility that the size of weight matrices exceeds GPU device memory. To address this issue, in this paper, we propose a method to partition the graph-and-model simultaneously on R-GCN in order to accelerate the model training using one or more GPUs for large graph-structured datasets. More specifically, a node-wise partitioning was already used for [3, 10], in this paper we propose an edge-wise partitioning method.
This paper is organized as follows. As related work, GNNs are overviewed, and especially R-GCN is detailed in Section 2. Section 3 describes the proposed method, and Section 4 shows its evaluation results. Conclusions and future work are discussed in Section 5.

2 Related Work

In this section, the overview of ConvGNNs is presented. R-GCN model is then described as a target of the proposed partitioning method.

2.1 ConvGNNs

GNNs are formulated by aggregation layer and combination layer [8]. The aggregation layer defines how to aggregate adjacent nodes' features. The combination layer defines a method to concatenate the result of the aggregation layer and a target node's feature. The l-th aggregation layer's output $a_v^{(l)}$, and the l-th combination layer's output $h_v^{(l)}$ for the target node v are defined as follows:

$$a_v^{(l)} = AGGREGATE^{(l)}\ (\{h_u^{(l-1)} : u \in \mathcal{N}_{(v)}\}), \tag{1}$$

$$h_v^{(l)} = \quad COMBINE^{(l)} \quad (h_v^{(l-1)}, a_v^{(l)}), \tag{2}$$

where $\mathcal{N}(v)$ is a set of adjacent nodes of node v, $a_v^{(l)}$ is an aggregated feature vector of adjacent nodes, and $h_v^{(l)}$ is a feature vector of the node v at l-th layer. In ConvGNNs, their weight matrices are updated with those multiplied by the adjacent node's feature vectors. For example, the l-th aggregation layer and the l-th combination layer of GraphSage [5], one of ConvGNNs, are formulated as follows:

$$a_v^{(l)} = MAX(\{W_a^{(l)} \cdot h_u^{(l-1)}, \forall u \in \mathcal{N}(v)\}), \tag{3}$$

$$h_v^{(l)} = W_h^{(l)} \cdot [h_v^{(l-1)}, a_v^{(l)}], \tag{4}$$

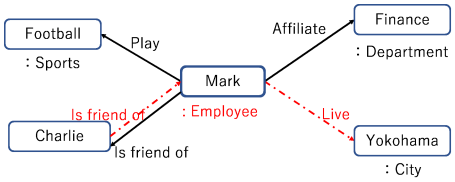


Fig. 1. Example of relational graph.

where W_a and W_h are weight matrices for aggregation and combination layers, respectively. MAX is an element-wise max-pooling and the combination layer represents a linear mapping. Such approaches have a problem with the size of graph-structured data. Especially when GPUs are used for the acceleration of the model training, the graph-structured data are required to be partitioned into smaller batches. In GraphSage, a node-wise partitioning is applied to solve the problem. The technique to make batches is based on node sampling located around the target nodes, for example, by random walk. As a result, the graph is divided into batches so that each batch can be fit into a GPU device memory. Pinsage [10], an extension of GraphSage, is an item recommendation system for a web-scale graph-structured data, which is composed of three billion nodes and 18 billion edges with data-parallel processing using multiple GPUs, where these GPUs share the same parameters and operate different batches. The size of batches is determined based on the sampling range. GIN [8] is another GNN that has shown a stable and high prediction accuracy. Several ConvGNNs have been extended to make predictions in relational graphs [9].

2.2 R-GCN

The R-GCN model aims to complete a lack of information on a knowledge base. Figure 1 illustrates an example of a knowledge base, composed by a triplet (subject, predicate, and object). In Figure 1, the graph data contains information that "(Mark) (Play)s (Football)." Knowledge base requires two prediction tasks: entity clustering and link prediction. The entity clustering task corresponds to the prediction of Mark's occupation, "Employee." Completing the link "Live" from "Mark" to "Yokohama" is one of the link predictions. Note that it needs to consider edge directions and types. R-GCN introduces these edge attributes to conventional GCNs.

R-GCN models have weight matrices W_r, which are corresponding to each edge attribute r. A set of weight matrices takes edge types and directions into account. Also, W_0 is defined as self-loops' weight matrix that is a feed-forward from the previous layer. More specifically, a hidden vector h_v of a node v on an $(l+1)$-th layer can be calculated as follows [6]:

$$h_v^{(l+1)} = \sigma\Big(\sum_{r\in\mathcal{R}}\sum_{u\in\mathcal{N}_v^r}\frac{1}{c_{v,r}}W_r^{(l)}h_u^{(l)} + W_0^{(l)}h_v^{(l)}\Big), \tag{5}$$

where $\mathcal{N}_v^r$ is a set of adjacent nodes connected to node v with edge attribute r. $c_{v,r}$ is a normalization factor for normalizing the difference of node degree, and generally it is defined as $c_{v,r} = |\mathcal{N}_v^r|$. We here define two sublayers: matrix-operation layer and adding layer. The matrix-operation layer is in charge of the computation of $\frac{1}{c_{v,r}} W_r^{(l)} h_u^{(l)}$ and $W_0^{(l)} h_v^{(l)}$. The adding layer executes the other operations. The loss function for a model training is defined as follow:

$$\mathcal{L} = -\sum_{i \in \mathcal{Y}} \sum_{k=1}^{K} t_{v,k} \ln h_{v,k}^{(L)}, \tag{6}$$

where L is the number of hidden layers, $t_{v,k}$ is the k-th cluster's label on node v, and $h_{v,k}$ is the k-th entry of the network output for the node v. $\mathcal{Y}$ is a set of nodes that have labels. For the model training, there are two regularization methods to reduce the number of learnable parameters. With the regularization of *basis*-decomposition [6], weight matrices W_r are defined as follows:

$$W_r = \sum_{b=1}^{B} a_{r,b}^{(l)} V_b^{(l)}. \tag{7}$$

This regularization means that weight matrices are defined as a linear combination of basis transformations $V_b^{(l)} \in \mathbb{R}^{d^{(l+1)} \times d^{(l)}}$ with coefficients $a_{r,b}$ dependent on each edge attribute r. Also, weight matrices consume the memory only when the operation is executed on a layer that is related to the weight matrices. However, the size of weight matrices, used at the same time, increases proportionally to the number of edge attributes. Thus, the model has a scalability issue, especially under the condition where initial node features are set as one-hot vectors. As a result, it needs to partition both a graph and a model for acceleration with GPUs, as well as typical deep learning models.

3 Proposed Method

In this section, we introduce our edge attribute-wise graph partitioning method and a graph-and-model simultaneous parallel execution on R-GCN.

3.1 Edge Attribute-wise Partitioning

In Section 2.2, we mentioned that the partitioning of both the graph and model is required to use GPUs for accelerating R-GCN execution with a large graph-structured data, because GPU device memory size is strictly limited. Generally, for executing deep learning on a GPU, the total size of a model and training data necessarily fits into the GPU device memory size. However, in the case of R-GCN model, the weight matrices W_r are required for each edge attribute r, which means that the model size increases proportionally to the number of edge attributes. This is an inherent scalability issue of R-GCN, which is different from

other ConvGNNs. There are two ways of partitioning for fitting data and model sizes into GPU device memory: node-wise partitioning and edge attribute-wise partitioning. The node-wise graph partitioning is one of the existing solutions [3, 10] to resolve the scalability problem on GNN models, making some batches by sampling adjacent nodes around target nodes. In this paper, on the other hand, we propose an edge attribute-wise partitioning to give a solution for the size of graph data as well as the size of the model. The benefit of the edge attribute-wise partitioning over the node-wise partitioning is as follows. Although the node-wise partitioning mainly aims at data-parallel computing, the partitioning results in an overlapping of weight matrices between submodels on R-GCN. In the worst case, the submodels' size is not reduced, and thus the scalability problem on R-GCN model would not always be solved. Meanwhile, the proposed edge attribute-wise partitioning aims at dividing a graph into some subgraphs in such a way that each edge attribute is exclusively divided. Here, each submodel should only have a weight matrix corresponding to the edge type that each subgraph has. Thus the size of the submodels is always scaled down. The memory space complexities of their weight matrices for input, hidden, and output layers are $O(|R_i||V_i||H|)$, $O(|R_i||H||H|)$, and $O(|R_i||H||O|)$, respectively, where $|R_i|$ is the number of edge attributes on the i-th subgraph, $|V_i|$ is the number of nodes on the i-th subgraph, $|H|$ is the number of hidden units, and $|O|$ is the dimension of output. We notice that there is no difference in the learning outcome between the division and the non-division implementations.

Figure 2 illustrates the concept of the graph partitioning, which is executed for a graph with four edge attributes. At first, a parent graph, i.e., the original graph data, is partitioned into portions, each having exactly one edge attribute. Subgraphs are finally constructed by assembling any of the portions. We propose two methods for grouping the portions into subgraphs. The first method that considers the number of edges in each subgraph and the second method that considers the number of edge attributes in each subgraph. In the first method, portions are distributed into subgraphs, minimizing the difference in the number of edges in subgraphs. In the second method, we sort portions in descending order by the number of edges. The sorted portions are assigned to one of the subgraphs in ascending order (subgraphs 1 to N) and then those in descending order (subgraphs N to 1) repeatedly. Here, we regard the portion including self-loops as a subgraph in distinction from others to reduce the size of submodels. The size of weight matrices depends on the number of nodes when initial node features are defined as one-hot vectors of local node IDs in each subgraph. Since the self-loop exists in all the nodes, the number of nodes in a self-loop subgraph is equal to that of the parent graph. A subgraph, including self-loops, increases the number of nodes in the subgraph, resulting in a larger submodel. To avoid this, we distinguish the self-loops from the others.

3.2 Graph-and-model Simultaneous Partitioning

In this section, we propose the implementation of R-GCN using multiple GPUs. We note here that R-GCN has huge weight matrices when training a large graph

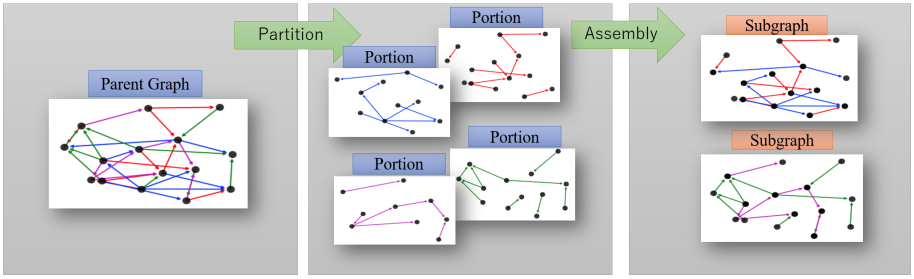


Fig. 2. Example of edge attribute-wise partitioning for graph with four edge attributes, making two subgraphs.

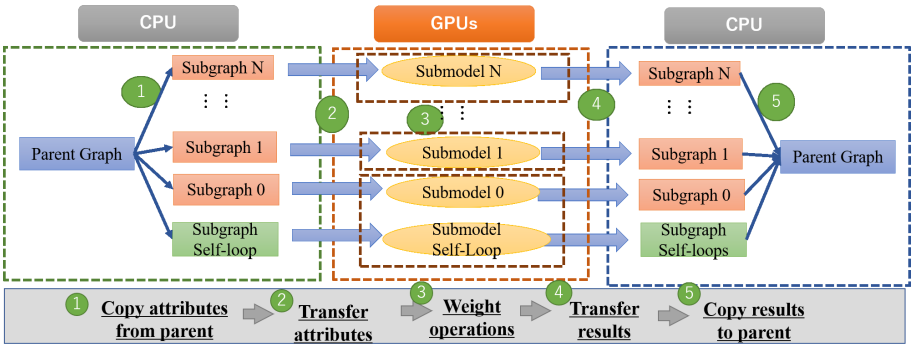


Fig. 3. Execution flow on a matrix-operation layer before an adding layer.

data, especially on the input layer, because the scale grows proportionally with the numbers of nodes and the edge attributes. The edge attribute-wise partitioning can reduce not only batch sizes in the graph but also the size of weight matrices in each submodel. We introduce the following parallel and sequential implementations:

1. *CPU+MultiGPUs* setting: parallel execution using multiple GPUs, and
2. *CPU+1GPU* setting: time-multiplexed sequential execution using a single GPU.

Figure 3 shows an execution flow on a matrix-operation layer of R-GCN under the CPU+MultiGPUs setting. In the execution, at first, subgraphs are generated by the edge attribute-wise partitioning. Then, the features of subgraphs are transferred to GPUs. In each GPU, weight matrices corresponding to the subgraph's edge attributes are set by computation of *basis*-decomposition regularization, and then the results are returned to a host CPU. In addition, the subgraph with only self-loops is operated on one of the GPUs by sharing this GPU with another subgraph. After that, the results are copied to the parent graph before executing. Then, the adding layer is executed for the parent graph.

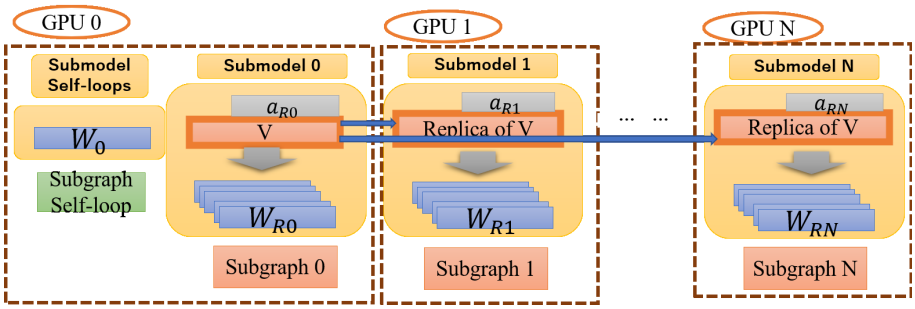


Fig. 4. Creating submodels on a matrix-operation layer under the *basis*-decomposition regularization condition, and distributing subgraphs on each GPU.

Table 1. Execution environment.

OS	Ubuntu 16.04.6 LTS
CPU	Intel Core i7-6800K (6C) 3.40GHz
GPU	NVIDIA GeForce GTX 1080Ti (11GB GDDR5X)
DRAM	32GB

Figure 4 illustrates the execution to create weight matrices in each submodel. By the *basis*-decomposition regularization, the base matrices V should be shared with all the GPUs to make each set of weight matrices. That is a reason why basis transformations V is replicated to the other GPUs. Then, each GPU prepares the set of weight matrices corresponding to a transferred subgraph, by computing a replica of V and edge attribute coefficients a_r. CPU+1GPU is also based on the edge attribute-wise partitioning. With this setting, each subgraph is transferred into a single GPU, and the submodel is executed on the GPU sequentially. We evaluate the execution time for the two implementations: CPU+MultiGPUs setting and CPU+1GPU sequential execution setting.

4 Evaluations

In this section, we evaluate the effectiveness of our proposal, the graph-and-model simultaneous partitioning on R-GCN. The evaluation environment is shown in Table 1.

4.1 Baseline

Implementation We evaluate the execution time for model training by comparing three implementations with one or more GPUs to CPU only setting. Three implementations are as follows:

1. GPUonly where all the parameters and graph data are allocated on a single GPU without CPU,

Table 2. Datasets: BGS, AM, and random graph.

	BGS	AM	Random Graph
# of nodes	333,845	1,666,764	800,000
# of edge attributes	103	120	100
# of edges	916,199	5,196,085	2,399,998
# of labeled nodes	146	1,000	400
# of classes	2	11	8

2. CPU+MultiGPUs setting using 2-4 GPUs based on the model and graph simultaneous partitioning, and
3. CPU+1GPU setting using a single GPU sequentially based on the model-and-graph simultaneous partitioning.

The second and third settings were introduced in Section 3.2. We use Pytorch as a deep learning framework. Also, DGL (Deep Graph library) [1] is used for operations on the graph-structured data, such as an aggregation of node information. As a baseline, we use a DGL's tutorial code for the implementations of CPU setting and GPUonly setting. In this paper, R-GCN model has two layers with 16 hidden units for BGS and 10 hidden units for AM and a random graph. Also, we set the number of basis transformations as 40, and we use SGD as the optimizer.

Datasets We use three datasets: BGS, AM, and a random graph. BGS and AM are provided in Resource Description Framework format [5], and the random graph is generated with the Barabasi-Albert model [2]. Table 2 lists their parameters: the numbers of nodes, edge attributes, edges, labeled nodes, and classes. The datasets are preprocessed to fit DGL graph format and R-GCN model. Firstly, self-loops are added to graph data. In addition, the edges of the graph data are duplicated by considering edge directions. The number of edge attributes on the graph data becomes twice the original dataset by this preprocessing. In the graph, distant nodes which are more than three-hop away from the target node are pruned, because we assume a 2-layer model in which the three-hop away nodes do not affect outputs of the target nodes. We delete edges whose edge attributes are applied for less than 150 edges in the case of AM.

4.2 Result of Graph Partitioning

The proposed graph partitioning method is used to generate the subgraphs. Here, the number of nodes is related to the scale of the submodels in the input layer under the condition where the initial node features are defined as one-hot vectors. The number of edges determines the computation cost, and the number of edge attributes is proportional to the scale of the submodel. In this paper, we partition graphs and models into subgraphs in two ways as proposed

Table 3. Parameters of subgraphs when applying edge attribute-wise partitioning, considering the number of edges in each subgraph.

(a) BGS

	Sub-0	Sub-1	Sub-2	Sub-3	Sub-self
# of nodes	205,111	279,072	171,265	84,573	333,017
# of edges	457,470	457,467	457,407	457,534	333,017
# of edge attributes	34	35	85	40	1

(b) AM

	Sub-0	Sub-1	Sub-2	Sub-3	Sub-self
# of nodes	1,013,531	1,073,744	1,118,582	915,739	1,203,676
# of edges	2,598,224	2,598,221	2,598,263	2,598,414	1,203,676
# of edge attributes	26	40	102	72	1

(c) Random Graph

	Sub-0	Sub-1	Sub-2	Sub-3	Sub-self
# of nodes	656,908	651,308	656,959	663,528	800,000
# of edges	798,697	785,283	798,684	817,332	800,000
# of edge attributes	50	49	50	51	1

in Section 3.1. In Section 4.3, the execution time of R-GCN is evaluated with GPUs while changing the number of subgraphs. Tables 3 and 4 show the results of partitioning each graph-structured data into four subgraphs and a subgraph that has only self-loops. We found that the way considering the number of edge attributes in each subgraph can minimize the size of weight matrices on each GPU. Thus, this approach is used in the following experiments.

4.3 Execution Time

Table 5 shows a summary of the execution time. Although the datasets were also executed with GPUonly setting, the out of GPU memory occurred in the cases of AM and random graph. Especially for AM, the size of weight matrices on the input layer was over 17GB, which explicitly demonstrates the necessity of the proposed model partitioning on R-GCN model with a large graph data. This motivates us the graph-and-model partitioning. Also, we remark that CPU+4GPUs setting can accelerate the model training for all the datasets compared to CPU setting: 3.88 times for BGS, 3.28 times for AM, and 2.60 times for the random graph. We notice that in the backward phase for updating the parameters, CPU+4GPUs setting is advantageous. On the other hand, in the forward phase, its advantage is not as much as in the backward phase, because the data transfer overhead becomes significant. Please note that, if a target graph data is small enough to execute GPUonly setting, this setting is the best choice.

Figure 5 shows the results of CPU+MultiGPUs setting and CPU+1GPU setting while changing the number of GPUs and the number of graph divisions,

Table 4. Parameters of subgraphs when applying the edge-attributes partitioning, considering the number of edge attributes in each subgraph.

(a) BGS

	Sub-0	Sub-1	Sub-2	Sub-3	Sub-self
# of nodes	177,298	145,554	258,285	286,893	333,017
# of edges	460,080	460,080	454,859	454,859	333,017
# of edge attributes	49	49	48	48	1

(b) AM

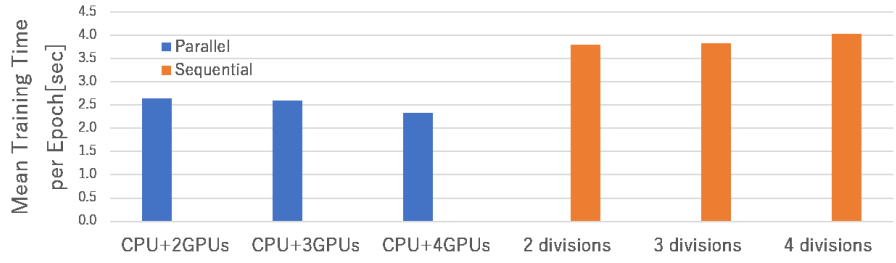
	Sub-0	Sub-1	Sub-2	Sub-3	Sub-self
# of nodes	986,425	986,250	1,035,928	1,035,928	1,203,676
# of edges	2,901,669	2901,669	2,294,892	2,294,892	1,203,676
# of edge attributes	60	60	60	60	1

(c) Random Graph

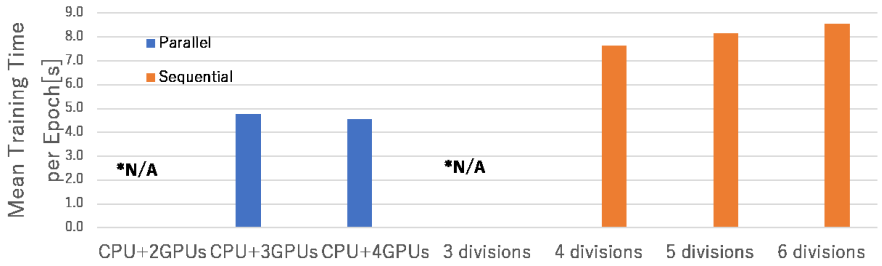
	Sub-0	Sub-1	Sub-2	Sub-3	Sub-self
# of nodes	657,309	644,850	650,788	657,179	800,000
# of edges	799,962	799,962	800,036	800,036	800,000
# of edge attributes	50	50	50	50	1

Table 5. Mean training times (Forward, Backward, and Full) per epoch [sec] for executions on CPU, GPUonly, and CPU+4GPU settings.

		BGS	AM	Random Graph
Forward	CPU	1.26	4.86	1.94
	GPUonly	0.049	N/A	N/A
	CPU+4GPU	0.80	3.68	1.34
Backward	CPU	7.78	37.07	9.92
	GPUonly	0.003	N/A	N/A
	CPU+4GPU	1.53	9.11	3.21
Full	CPU	9.06	41.92	11.86
	GPUonly	0.052	N/A	N/A
	CPU+4GPU	2.33	12.78	4.55



(a) BGS



(b) Random Graph

Fig. 5. Execution time per epoch [sec] of BGS and random graph for CPU+MultiGPUs settings with 2-4 GPUs and CPU+1GPU settings with 3-6 divisions. The number of divisions is defined as the number of subgraphs except for their self-loop subgraph, and *N/A indicates the out of memory occurred during execution.

respectively. Note that the out of memory occurred in the case of AM. We found firstly that the growth in the number of GPUs improves the performance. For BGS, the acceleration rate increases from 3.42 (CPU+2GPUs) to 3.88 times (CPU+4GPUs) compared to CPU setting. For the random graph, the acceleration rate increases from 2.48 (CPU+3GPUs) to 2.60 times (CPU+4GPUs) compared to CPU setting. The reason for the small increase in speed by increasing the number of GPUs is due to the processing of the aggregation layer on CPU and feature exchange between the parent graph and subgraphs. Although the performance of the CPU+1GPU is inferior to CPU+MultiGPUs setting, this setting accelerates the execution time by up to 2.38 and 1.55 times for BGS and the random graph, respectively, compared to CPU setting. We remark that our proposal can accelerate R-GCN even with a single GPU for training R-GCN model for a large graph. We also found here that the number of divisions is related to the performance, and minimizing the number of divisions can improve the performance. In the forward phase, the computation results are accumulated on a GPU and consume the memory capacity. As a result, CPU+1GPU setting with three divisions for the random graph introduces the out of memory even though the execution on CPU+3GPUs setting has been successfully done.

5 Conclusions

In this paper, we presented an edge attribute-wise graph partitioning and the graph-and-model simultaneous partitioning method on R-GCN to accelerate using one or more GPUs with large graph-structured data. Experimental results with CPU+MultiGPUs setting show that it can accelerate the model training of R-GCN with AM dataset with over 1.6 million nodes, 5 million edges, and 120 edge attributes. Besides, the CPU+1GPU setting outperforms CPU setting by 1.55 times for a dataset with 0.8 million nodes, 2 million edges, and 100 edge attributes even with a single GPU. The result opens up possibilities to accelerate training R-GCN by using one or multiple GPUs, each having limited device memory capacity. As future work, we need to consider smaller batches with fine-grained mini-batch execution scheduling to release the memory allocation more frequently to utilize the GPU device memory more efficiently.

Acknowledgements This work was partially supported by JSPS KAKENHI Grant Number JP19H04117.

References

1. Deep Graph Library. https://www.dgl.ai/pages/about.html
2. Albert, R., Barabasi, A.L.: Statistical Mechanics of Complex Networks. Reviews of Modern Physics (Jan 2002)
3. Hamilton, W., Ying, Z., Leskovec, J.: Inductive Representation Learning on LargeGraphs. In: Proceedings of the Neural Information Processing Systems (NeurIPS'17). pp. 1024–1034 (2017)
4. Kipf, T.N., Welling, M.: Semi-Supervised Classification with Graph Convolutional Networks. In: Proceedings of the International Conference on Learning Representations (ICLR'17) (2017)
5. Ristoski, P., de Vries, G.K.D., Paulheim, H.: A Collection of Benchmark Datasets for Systematic Evaluations of Machine Learning on the Semantic Web. In: Proceedings of the International Semantic Web Conference (ISWC'16). pp. 186–194 (2016)
6. Schlichtkrull, M., Kipf, T.N., Bloem, P., Berg, R.v.d., Titov, I., Welling, M.: Modeling Relational Data with Graph Convolutinal Networks . arXiv preprint arXiv:1703.06103v4 (Oct 2017)
7. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S.: A Comprehensive Survey on Graph Neural Networks. arXive:1901.00596v2 (Mar 2019)
8. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How Powerful Are Graph Neural Networks. In: Proceedings of the International Conference on Learning Representations (ICLR'19) (2019)
9. Ye, R., Yujie Fang, H.Z., Wang, M.: A Vectorized Relational Graph Convolutional Network for Multi-Relational Network Alignment. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'19). pp. 4135–4141 (2019)
10. Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W.L., Leskovec, J.: Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In: Proceedings of the International Conference on Knowledge Discovery & Data Mining (KDD'18). pp. 974–983 (Aug 2018)