PCGCN: Partition-Centric Processing for Accelerating Graph Convolutional Network

Chao Tian
Peking University
Beijing, China
VivaLaTian@pku.edu.cn

Lingxiao Ma
Peking University
Beijing, China
xysmlx@pku.edu.cn

Zhi Yang*
Peking University
Beijing, China
yangzhi@pku.edu.cn

Yafei Dai
Peking University
Beijing, China
dyf@pku.edu.cn

***Abstract*—Inspired by the successes of convolutional neural networks (CNN) in computer vision, the convolutional operation has been moved beyond low-dimension grids (e.g., images) to high-dimensional graph-structured data (e.g., web graphs, social networks), leading to graph convolutional network (GCN). And GCN has been gaining popularity due to its success in real-world applications such as recommendation, natural language processing, etc. Because neural network and graph propagation have high computation complexity, GPUs have been introduced to both neural network training and graph processing. However, it is notoriously difficult to perform efficient GCN computing on data parallel hardware like GPU due to the sparsity and irregularity in graphs. In this paper, we present PCGCN, a novel and general method to accelerate GCN computing by taking advantage of the locality in graphs. We experimentally demonstrate that real-world graphs usually have the clustering property that can be used to enhance the data locality in GCN computing. Then, PCGCN proposes to partition the whole graph into chunks according to locality and process subgraphs with a dual-mode computing strategy which includes a *selective* and a *full* processing methods for sparse and dense subgraphs, respectively. Compared to existing state-of-the-art implementations of GCN on real-world and synthetic datasets, our implementation on top of TensorFlow achieves up to $8.8\times$ speedup over the fastest one of the baselines.**

***Index Terms*—graph convolutional network (GCN), GPU, graph, deep learning**

I. Introduction

Deep neural networks (DNNs) have achieved a lot of advances in many areas, e.g., image and speech recognition, natural language processing, which makes it be applied in many real-world applications, like self-driving, search engine, recommendation and so on. Specifically, the convolutional neural networks (CNNs) have achieved a huge success in computer vision. Inspired from CNNs, researchers have proposed many methods to move the notation of convolution to graph-structured data (e.g., social networks, knowledge graphs, etc.), known as graph convolutional networks (GCNs) [1]. In GCN, a single convolution operation on graph transforms and aggregates feature information from a node's one-hop graph neighborhood, and multiple such convolutions are stacked to propagate nodes' information across far reaches of a graph. In this way, GCNs leverage both feature information as well as graph structure. GCNs have achieved convincing model accuracy in many real-world applications. For example, in recommendation systems, it can learn features from the user-item graph for higher-quality recommendation [2].

Recent advances in GPU hardware technologies offer potentially new avenues to accelerate the inference and training of GCNs, for example, the newest NVIDIA Turing GPU has aggregated about 18MB register file space and over 100 tera-flops computation power. Different from the regular grid structures in both the feature map and the convolution kernel of CNNs, there is a graph structure in GCN. The sparsity and irregularity in graphs make it notoriously difficult to perform efficient GCN computing on data parallel hardware like GPU. Existing open-source GCN implementations [1], [3]–[5] are based on general sparse matrix operations and ignores graph-aware properties, resulting in unpredictable and fine-grained random memory accesses, thereby decreasing the utility of large on-chip memory resource offered by these new architectures.

In this paper, we present PCGCN, a partition-centric GCN framework that enhances cache-efficiency and memory performance to efficiently execute GCNs. PCGCN executes a GCN model as bulk synchronous steps of message exchange between vertex subsets called partitions. PCGCN propagates updates from nodes to partitions and reduces the redundancy associated with vertex-centric processing model by leveraging the graph locality and the cache We also adopt a hybrid partition-centric processing strategy that can adaptively select the optimal mode for each pair of graph partitions with different computation characteristics. In these ways, PCGCN extracts high performance from the memory hierarchy.

To summarize, our contributions are listed as follows.

- We propose PCGCN, a partition-centric processing framework for training GCNs on massive parallel devices like GPU, and implemented PCGCN on top of popular deep learning framework (TensorFlow [6]). This is particularly important for users to leverage GCNs in more data analysis applications and make compatible with existing programs.
- We design various optimizations based on graph properties to resolve the efficiency challenge in GCN training on GPUs, including the partition-centric processing framework, dual-mode subgraph computing strategy and GPU-specific optimizations.

* Corresponding Author

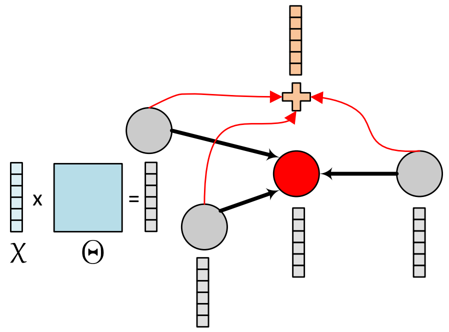


Fig. 1. Graph convolutional network

- We evaluate the performance of PCGCN by executing a typical GCN algorithms on a variety of real-world and synthetic datasets. We show that PCGCN outperforms the best one of the baselines, including DGL [3], PyG [5] and the open-source GCN implementation [1], with an up-to $8.8\times$ speedup.

II. Preliminaries

A. Graph Convolutional Networks

Just like CNNs in computer vision, graph convolutional networks [1] aggregate information for a vertex and transform the features to learn the vertex representation. A GCN model is built with multiple layers of GCN, and Equation 1 shows one layer of GCN.

$$Z = \widetilde{D}^{-\frac{1}{2}} A \widetilde{D}^{-\frac{1}{2}} X \Theta \tag{1}$$

where A is the adjacent matrix of the graph, $\widetilde{D}^{-\frac{1}{2}}$ is the matrix used in the normalized graph Laplacian, X is the vertex hidden state of the previous layer and Θ is the weight matrix.

This formula can be described from the graph view as shown in Figure 1 when treating $\widetilde{D}^{-\frac{1}{2}} A \widetilde{D}^{-\frac{1}{2}} = \widetilde{A}$ as the adjacent matrix with normalized edge value: the vertex gathers information from neighbors which has made a transformation over the vertex feature to generate its hidden state. Therefore, in GCN, information is extracted via neural networks on vertices and propagated via the graph structure.

B. Real-world Graphs

Graphs are popular in many fields, e.g., social network, Internet, biology, etc. However, real-world graphs usually have some properties. The locality, i.e., the small-world property, appears in most real-world graphs [7], [8]. It means that nodes tend to create tightly knit groups rather than randomly connect each other in most real-world graphs. The locality can be benchmarked with the clustering coefficient [7], and we show the measurement of several real-world and synthetic graphs in Table I. We also visualize the pubmed citation graph [9] in Figure 2 by visualizing its adjacent matrix where each pixel indicates an edge. It shows that most edges are distributed among diagonal, which means that these edges are connected in some clusters of vertices.

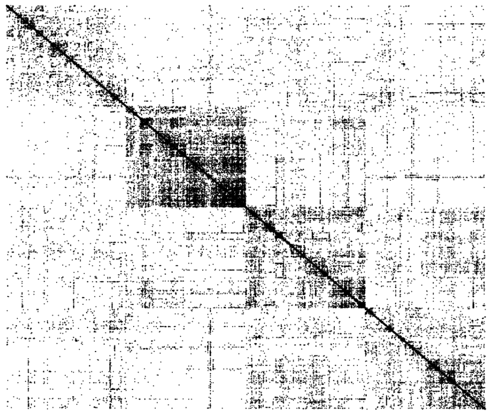
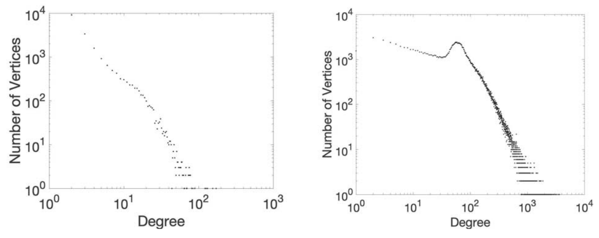


Fig. 2. Visualization of the adjacent matrix of the pubmed graph



(a) pubmed (b) reddit

Fig. 3. Degree distribution of real-world graphs in TableI.

Furthermore, a real-world graph usually has an irregular degree distribution [10] such as the skewed power-law distribution that a few vertices connect many neighbors while most have only a few neighbors. Figure 3 shows the distribution of the pubmed and reddit-s datasets described in Table I.

C. GPU Architecture

GPU is a SIMT-architecture parallel device and has been widely used in accelerating machine learning training and inference, especially for deep learning. However, in order to maximize the power of GPU, the program should be carefully designed with considering regular computation patterns including but not limited to: coalesced memory access, memory hierarchy, branch divergence, etc.

Unlike image, audio, or text that have clear grid structures and are friend to SIMT-architecture devices like GPU, graphs are irregular, making it hard to conduct GCN computation efficiently on GPU. PCGCN exploits the locality property in real-world graphs to make the computation more regular to accelerate GCN on GPU.

III. Partition-Centric Graph Convolutional Network

In PCGCN, we propose to leverage the locality of real-world graphs to accelerate GCN computing by accelerating the graph propagation. Specifically, PCGCN introduces a partition-centric processing scheme in the graph propagation stage, which makes PCGCN achieve locality-friendly in processing graphs. Moreover, a dual mode subgraph computing

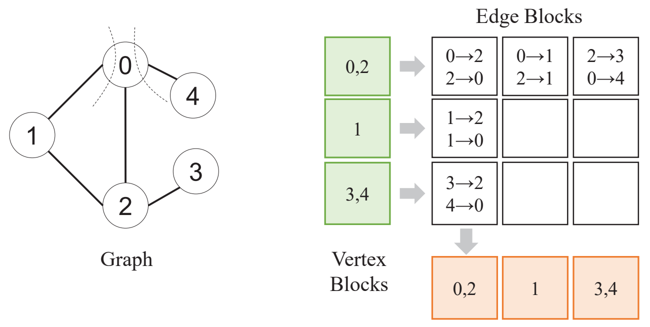


Fig. 4. Graph partition and partion-centric processing for an undirected graph.

method is introduced to further accelerate the graph propagation by design different computing mode according to the density of a subgraph.

A. Partition-Centric Graph Propagation

PCGCN modifies the graph propagation stage of GCN from a whole graph computing scheme to a subgraph-centric one. Algorithm 1 shows the feed-forward computation steps of a L layer model. The input graph is partitioned into a set of subgraphs. For each GCN layer, the hidden state from the previous is transformed by a fully connected neural network, which is the same as the original GCN. Then, for each subgraph, it gathers and accumulates states from itself and neighbor subgraphs to execute the partition-centric graph propagation. Finally, the outputs of each subgraph are combined as the output of this layer. This procedure is repeated over all the layers of the model.

a) Graph Partition: For a given graph, in order to process it in the partition-centric computing scheme, the input graph should firstly be partitioned into a set of subgraphs. Therefore, PCGCN applies a 2D graph partitioning. As shown in Figure 4, it partitions the whole graph into K subgraphs and thus creates K disjoint vertex blocks and $K \times K$ edge blocks where $E_{i,j}$ represents edges between two vertex blocks V_i and V_j.

Recently, there are plenty of graph partition algorithms, e.g., random partitioning, min-cut partitioning, etc. We notice that a random partitioning will hurt the locality of graphs because it ignores the locality and randomly assigns vertices to partitions. Thus, we choose the locality-aware algorithm (e.g., METIS [11]) as the graph partition method to enhance the locality.

b) Partition-Centric Processing: After partitioning the graph into a set of subgraphs, PCGCN can apply the partition-centric processing. For layer ℓ of the model, PCGCN calculates the neural network transformation a^ℓ and then partitions it to corresponding subgraphs. Then, PCGCN processes these subgraphs one by one. For subgraph S_k, PCGCN traverses all the subgraphs that have edges $E_{k,i}$ connected to S_k, and processes the edge block. For each edge in $E_{k,i}$, it calculates the multiplication of data in the source vertex and the edge, and accumulates it in h_k^ℓ. After all of the neighbor subgraphs being processed, it will generate the hidden state h_k^ℓ for vertices in

Algorithm 1: Forward Computation of Partition-Centric Graph Convolutional Network

```
Symbols: input graph: $G = (V, E)$, layers:
         $\ell = \{1, \cdots, L\}$, subgraphs:
         $\{S_k = (V_k, E_k) | k = 1, \cdots, K\}$, vertices in
         subgraph $k$: $V_k$, edges in subgraph $k$: $E_k$,
         edges between subgraph $i$ and $j$: $E_{i,j}$,
         features of layer $\ell$: $h^\ell$ ($h^0$ indicates the
         input features), weight of layer $\ell$: $w^\ell$
Partition $G \rightarrow \{S_k | k = 1, \cdots, K\}$;
// calculate a $L$ layer GCN model
for $\ell = 1, \cdots, L$ do
    $a^\ell = h^{\ell-1} \times w^\ell$;
    Split $a^\ell \rightarrow \{a_k^\ell | k = 1, \cdots, K\}$ according to
      subgraphs;
    // execute graph propagation for each
       subgraph
    for $k = 1, \cdots, K$ do
        // gather and accumulate states from
           neighbor subgraphs
        $h_k^\ell = \sum_{i=1}^{K} f(E_{k,i}, a_i^\ell)$;
    // combine hidden states of each subgraph
    $h^\ell = \text{concat}(h_k^\ell, \forall k \in \{1, \cdots, K\})$;
return $h^L$;
```

S_k. Finally, the concatenated $h_k^\ell (\forall k \in \{1, \cdots, K\})$ is the output hidden state h^ℓ of layer ℓ.

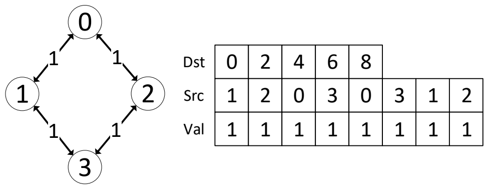
Note that the partition-centric processing scheme is friendly to memory hierarchy, because the range of source and destination vertices is limited to the subgraph. Therefore, processing edges of the same sources or destinations can load data from the cache, especially for graphs with higher locality.

c) Equivalence of PCGCN and GCN: PCGCN only changes the order of calculating neighbors of a given vertex compared to the original GCN. The operations in the graph propagation stage of GCN are element-wise multiplication and add, which are commutative and associative. Thus, the changed order does not affect the results because of the property of commutative and associative operations [12]. Therefore, the PCGCN is equal to the original GCN and can produce the same numerical results, hence the same per-epoch convergence.

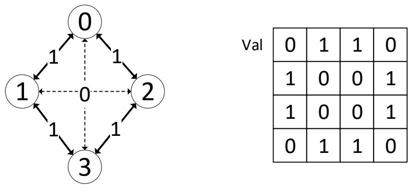
B. Dual Mode Subgraph Computing Strategy

For each layer, PCGCN processes the GCN from the subgraph perspective to leverage the graph locality. According to Section II, real-world graphs usually have irregular distributions. The irregularity of a graph often leads to different density in subgraphs. To further accelerate the computation of a subgraph, PCGCN introduces a dual-mode subgraph computing scheme according to the density.

a) Graph Propagation in Selective *Mode:* When there are a few edges in the edge block $E_{k,i}$ of the subgraph S_k and S_i in the layer ℓ of the model, PCGCN uses a *selective* mode



(a) CSR format for *selective* mode.



(b) Dense format for *full* mode. The dotted line indicates the inserted edge with value 0.

Fig. 5. Subgraph layouts in sparse and dense formats for *selective* and *full* modes, respectively.

to process this edge block. Specifically, PCGCN scans all the edges in the edge block. For edge $e_{p\to q}$ (i.e., edge connects the source vertex v_p and the destination vertex v_q), PCGCN decodes the source and destination indexes of the edge and fetches the corresponding hidden state $h_p^{\ell-1}$. Then, according to GCN, the features of the source vertex will be multiplied with the scalar value on the edge on the fly, and the result will be added to the hidden state h_q^{ℓ} of the destination vertex v_q.

b) Graph Propagation in Full *Mode:* When there are lots of edges in a subgraph, the decoding procedure in the *selective* mode (i.e., read the index of the source vertex and fetch the related vertex data) may lead to a huge impact on the processing efficiency. In dense subgraphs, lots of edges can be placed in a sequential order, e.g., when vertex v_0 connects the vertex v_1, v_2, there are edges $e_{1\to 0}$, $e_{2\to 0}$, $e_{3\to 0}$. In this case, the decoding procedure in the *selective* is redundant.

PCGCN introduces a *full* mode for this situation. It assumes the subgraph S_k and S_i is fully connected, i.e., each vertex in S_k is connected to all the vertices in S_i. If there is no edge $e_{p\to q}$ between vertex p and q, it will insert this edge with value 0 on the edge (e.g., $e_{1\to 2}$ in Figure 5(b)). This insertion does not affect the correctness of the result, because the edge value 0 makes the multiplication of the source vertex and the edge be 0. Different with the *selective* mode, the *full* mode processes all the edges in the fully connected chunk sequentially instead of decoding the indexes of edges and fetching the corresponding vertices.

c) Hybrid Graph Propagation: The above *selective* mode is suitable for sparse edge blocks, while the *full* mode is better for dense edge blocks. Due to the irregularity, a real-world graph may contain both sparse and dense edge blocks. The computing complexity of both modes mostly relates to the sparsity of an edge block. So, we take the sparsity to select the processing mode for a given edge block. Note that the

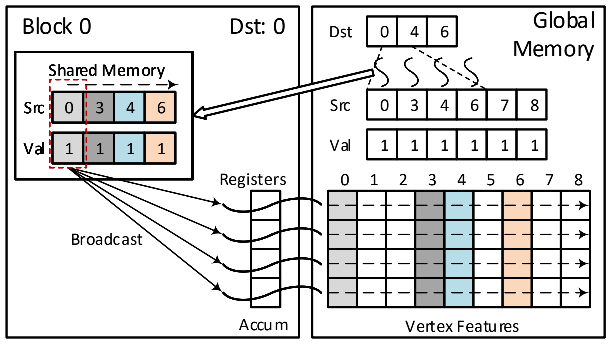


Fig. 6. Graph propagation in the *selective* mode of PCGCN. Assume that this subgraph has 9 vertices where each of them has a 4-dimension feature vector, and we schedule 4 GPU threads to process this subgraph.

same sparsity results in the same processing time on a fixed hardware platform. Specifically, we profile the runtime of full and selective modes for a block of sparsity p, denoted by T_f and T_s, respectively. Then, the runtime for a block of sparsity p' in selective mode can be estimated as $T_s * (p'/p)$. We choose full mode if $T_f < T_s * (p'/p)$, and get $p' > T_f * p/T_s$. Therefore, we take $\theta = T_f * p/T_s$ to select the processing mode for a given edge block.

IV. Accelerating PCGCN Computation on GPU

Section III has presented the design of partition-centric processing. Note that PCGCN can be implemented on various hardware platforms, e.g., CPU, GPU, etc., and benefit from the partition-centric processing. This is because most hardware has a memory hierarchy that can leverage the graph locality. As GPUs have been widely used for neural network computing, in this section, we present the implementation details of PCGCN to accelerate the GCN computation on GPU. To make compatible with current deep learning systems (e.g., TensorFlow [6], MXNet [13], PyTorch [14], etc.), we implement PCGCN and warp it as an operator in TensorFlow. Furthermore, considering the overhead of general-purpose deep learning systems, the implementation of PCGCN on top of TensorFlow can make a fair comparison with the baselines described in Section V.

A. Graph Representation Format

A good data format can not only reduce the storage usage but also improve the computation efficiency. As Figure 5 shown, PCGCN leverages a hybrid graph representation format.

For *selective* mode, as shown in Figure 5(a), the compressed sparse row (CSR) is used to store the subgraph. CSR stores a graph as three arrays of destination pointers (Dst), source indices (Src) and edge values (Val). Destination node indices are given implicitly by the Dst array, containing the starting index in the Src array for its neighbors. Therefore, the decoded tuples $(Src[idx], v, Val[idx]), idx \in [Dst[v], Dst[v+1])$ can represent in-coming edges of vertex v. In other words, the incoming neighbors for node v are at positions $Dst[v]$ up to but not including $Dst[v+1]$ in the Src arrays. For example, in Figure 5, vertex 0 has 2 edges (encoded with $Dst[1]=2$) connected with vertex 1 (encoded with $Src[0]=1$) and vertex 2 (encoded with $Src[1]=2$) with value 1 on the edge (encoded with $Val[0]=1, Val[1]=1$). In this format, threads within the same GPU thread block can share the same row index, which reduces both storage space and memory accesses.

For *full* mode, as shown in Figure 5(b), the dense format is used to store the subgraph, where the edge values are stored into a matrix. As introduced in Section III-A, each vertex in this subgraph are assumed to connect to all of the other vertices, where nonexistent edges are added with value 0. Therefore, this fully connected subgraph are stored into a dense matrix where the (p,q)th element indicates the value on edge $e_{p\to q}$. In dense format, the edge indexes are removed, and the edges are stored sequentially.

B. Graph Propagation in Selective *Mode*

As introduced in Section II, GPU is a SIMT-architecture device, requiring careful designs for massive parallel threads. To leverage the massive parallelism, chunks scheduled to *selective* mode are merged into a sparse graph and are processed by a GPU kernel.

Figure 6 shows the kernel design of the graph propagation in the *selective* mode. PCGCN schedules vertices and its corresponding computation to one GPU thread block. First of all, the edge data in the CSR format are sequentially loaded into shared memory for fast accesses. Then, because GCNs usually have a high dimension feature vector on each vertex, consecutive GPU threads are scheduled to process the multiplication of vertex feature vector and edge data, where consecutive threads will process consecutive dimensions of the feature vector and update the accumulated intermediate data in registers. In this case, the memory accesses are sequential and there is no branch divergence, leading to high performance processing on GPU. Finally, after all of the in-coming edges of a vertex are processed, the accumulated intermediate data in registers will be written back to global memory. The usage of registers can avoid redundant memory writes in global memory.

C. Graph Propagation in Full *Mode*

In *full* mode, PCGCN schedules one subgraph to one GPU thread block for processing. Similar to the *selective* kernel, the edge data in dense format will be loaded into shared memory for fast accesses. After that, consecutive GPU threads are scheduled to process the multiplication of vertex feature vector and edge data and update the accumulated intermediate data in registers, leading to coalesced memory accesses and reduced write operations in global memory. As the *full* mode takes the subgraph as a fully connected graph, each vertex should calculate edges to all of the other vertices. Therefore, it is clear that computing subgraphs in *full* mode converts irregular sparse computation to regular dense computation. So, PCGCN applies a set of optimizations for the fixed computation logic, e.g., loop unroll, double buffering, data prefetching, vectorized memory access, etc.

D. Tensor Core Acceleration

NVIDIA's latest generation Volta and Turing GPUs introduce hardware acceleration of matrix multiplication on 16-bit/8-bit/4-bit mixed precision by providing the WMMA warp matrix instructions. Recall that the *full* mode of PCGCN computes a fully connected edge block between two subgraphs and processes the edges sequentially. This procedure can be mapped to a warp matrix multiplication instruction and thus can be accelerated by the tensor core.

However, the tensor core requires 16-bit/8-bit/4-bit computation. In order to reduce the impact of lower precision on training, PCGCN leverages a mixed precision policy that the original 32-bit is used for the *selective* mode and the 16-bit is used for the tensor core based *full* mode, respectively. And there is a transformation between 32-bit and 16-bit when combining the results from both modes.

V. Evaluation

In this section, we demonstrate the efficiency of PCGCN by evaluating it on real-world and synthetic datasets. Section V-A describes the methodology of the experiments. Section V-B shows the overall runtime comparisons of PCGCN and the baselines. Section V-C investigates the exact graph propagation runtime where PCGCN targets to. Section V-E studies the impact of the tensor core acceleration on convergence. Section V-D shows the performance under different configurations.

A. Experiment Setup

a) Environment and Baselines: We evaluate PCGCN on two platforms with different GPUs: one is equipped with dual 2.2 GHz Intel Xeon E5-2650v4 processors (24 cores in total), 256 GB memory, and NVIDIA GTX 1080Ti GPU; another is equipped with single 2.0 GHz Intel Xeon E5-2683v3 processors (14 cores in total), 64 GB memory, and NVIDIA RTX 2080Ti GPU. The installed operating system is Ubuntu 16.04, using libraries CUDA 10.0, and cuDNN 7.5.

We compare PCGCN to the open-source GCN implementation [1] on TensorFlow v1.14 [6] (GCN), Deep Graph Library (DGL) [3] v0.4.0 (PyTorch v1.2 [14] backend) and PyTorch Geometric (PyG) [5] v1.3.2 (PyTorch v1.2 [14] backend). For fair comparison, all of the baseline systems are the latest stable versions [1] , and we also take minor optimizations [2] for GCN, marked as GCN-opt.

We focus on metrics for system performance, for example, time to scan one epoch of data. We have proved the equivalence of GCN and PCGCN in Section III-A. Thus, PCGCN (without Tensor Core acceleration) produces the same numerical results with GCN, DGL and PyG, hence the same

[1]The latest stable versions in the original submission of IPDPS 2020.

[2]Replace inefficient *feed_dict* with preloaded data tensors as recommended by TensorFlow official document (https://github.com/tensorflow/docs/blob/r1.11/site/en/api_guides/python/reading_data.md#preloaded-data).

TABLE I
DATASETS USED IN EVALUATION. (K: THOUSAND, M: MILLION)

Dataset	#vertex	#edge	#feature	#label	max degree	avg. degree	clustering coefficient
pubmed	19.7K	108.3K	500	3	172	5	0.06
blog	10.4K	678.3K	128	32	3993	65	0.46
youtube	15.1K	11.1M	128	32	6746	739	0.55
C1000-9	1.0K	901.1k	128	32	925	900	0.90
MANN-a81	3.3k	11.0M	128	32	3318	3310	1.00
reddit-s	10.2K	3.3M	602	41	4490	326	0.70
reddit	233.0K	23.4M	602	41	3650	100	0.17
RMAT-0.1	10.2K	216.7K	128	32	700	21	0.03
RMAT-0.5	10.2K	992.4K	128	32	2414	96	0.11
RMAT-1	10.2K	2.1M	128	32	4081	205	0.20
RMAT-5	10.2K	10.5M	128	32	8461	1024	0.51
RMAT-10	10.2K	21.0M	128	32	9628	2048	0.66

TABLE II
THE OVERALL RUNTIME (MS) OF GCN, GCN-OPT, DGL, PYG AND PCGCN ON GTX 1080TI AND RTX 2080TI. "OOM" INDICATES THE OUT-OF-MEMORY ERROR.

Time (ms)	NVIDIA GTX 1080Ti						NVIDIA RTX 2080Ti (Tensor Core Acceleration)					
Dataset	GCN	GCN-opt	DGL	PyG	PCGCN	speedup	GCN	GCN-opt	DGL	PyG	PCGCN	speedup
pubmed	46.83	4.97	10.89	7.56	**3.01**	1.7×	27.31	3.12	10.43	6.37	**2.23**	1.4×
blog	32.01	15.02	9.85	30.22	**4.46**	2.2×	22.99	5.92	9.36	15.78	**3.9**	1.5×
youtube	563.82	201.01	25.14	444.19	**22.17**	1.1×	337.86	54.15	19.88	222.57	**9.86**	2.0×
C1000-9	35.55	18.05	9.11	67.78	**1.15**	7.9×	24.03	5.99	8.72	21.57	**1.02**	5.9×
MANN-a81	637.58	206.06	20.39	1023.34	**2.32**	8.8×	336.69	51.9	16.17	309.4	**2.1**	7.7×
reddit-s	179.51	64.39	12.2	139.97	**8.58**	1.4×	135.46	18.52	10.56	69.75	**6.11**	1.7×
reddit	1744.05	470.58	147.83	oom	**113.48**	1.3×	956.23	188.78	96.47	oom	**63.06**	1.5×
RMAT-0.1	13.96	5.54	9.16	10.72	**2.78**	2.0×	11.14	3.78	9.23	6.9	**2.51**	1.5×
RMAT-0.5	41.95	19.43	9.82	41.01	**4.63**	2.1×	30.56	7.67	9.42	21.91	**3.36**	2.3×
RMAT-1	110.12	39.51	10.53	85.33	**7.07**	1.5×	82.71	12.98	10.71	43.75	**5.27**	2.0×
RMAT-5	518.18	192.26	24.44	423.76	**12.01**	2.0×	326.29	52.04	17.68	209.79	**6.62**	2.7×
RMAT-10	1051.38	382.39	42.97	oom	**13.05**	3.3×	601.19	101.97	34.76	oom	**7.14**	4.9×

per-epoch convergence. We report the average results over 100 epochs if no specified.

b) Datasets: Table I lists the real-world and synthetic datasets used for evaluation, including PubMed citation network (pubmed) [9], BlogCatalog social network (blog) [15], Reddit online discussion forum (reddit, reddit-s [3]) [16], YouTuBe interaction network (youtube) [17], DIMACS graphs C1000-9 [18] and MANN-a81 [18]. To evaluate the superiority of PCGCN in graphs of different sparsity, we use the RMAT [19], a widely used graph generator, to generate synthetic graphs characterized by the skewed distribution and fractal community structure which are similar to real-world graphs. In RMAT, we set the number of vertices as 10240 and generate different number of edges to achieve the graph density of $\{0.1\%, 0.5\%, 1\%, 5\%, 10\%\}$ (marked as RMAT-density, e.g., RMAT-0.1 indicates RMAT of 0.1% density). As some datasets do not have features or labels that are required by the GCN, we use node2vec [20] to generate the vertex features and community clustering to generate the vertex labels, respectively. The column *feature* in Table I represents the size of vertex feature vector, and the *label* column means the number of label classes. The degree column collects the average and maximum statistics of in-degree.

[3]As the reddit dataset is too large to be processed by some baselines, we sample a smaller dataset from the original reddit dataset.

B. Overall Performance

We evaluate the overall model performance by comparing it with state-of-the-art implementations of GCN on TensorFlow, DGL and PyG on two GPU platforms. We enable tensor core acceleration on the RTX 2080Ti GPU We set the num of layers $\ell = 4$ and the hidden size $hid = 64$. We will evaluate the performance under different hidden sizes and number of layers latter in Section V-D.

Table II shows the end-to-end performance. Overall, PCGCN achieves an average 2.9× speedup (up to 8.8×) over the best one of baselines on GTX 1080Ti, and an average 2.9× speedup (up to 7.7×) over the best one of baselines on RTX 2080Ti, respectively. Note that, in some datasets like pubmed and RMAT-0.1, the GCN-opt outperforms other baselines including DGL and PyG which are specifically designed for GNN training, and becomes the fastest baseline, after our optimization. Furthermore, when we compare the results among graphs of different density, PCGCN can achieve a better performance on graphs of higher clustering coefficient (e.g., blog, C1000-9 and RMAT-5). This is because graphs of higher clustering coefficient usually result in higher locality during graph propagation where PCGCN targets to.

Note that, PyG runs out-of-memory on large graphs such as reddit and some RMAT graphs, while PCGCN can process all of them. This is because PCGCN implements the edge

TABLE III
THE OVERALL RUNTIME (MS) OF PCGCN-SEL, PCGCN-FULL, AND PCGCN ON GTX 1080TI AND RTX 2080TI. "OOM" INDICATES THE OUT-OF-MEMORY ERROR.

Time (ms)	NVIDIA GTX 1080Ti			NVIDIA RTX 2080Ti (Tensor Core Acceleration)		
Dataset	PCGCN-sel	PCGCN-full	PCGCN	PCGCN-sel	PCGCN-full	PCGCN
pubmed	3.01	7.1	3.01	2.23	3.54	2.23
blog	4.46	14.02	4.46	3.9	7.2	3.9
youtube	23.92	23.91	22.17	16.67	11.34	9.86
C1000-9	3.46	1.15	1.15	3.05	1.02	1.02
MANN-a81	21.25	2.32	2.32	15.33	2.1	2.1
reddit-s	10.01	14.37	8.58	7.61	6.52	6.11
reddit	113.48	oom	113.48	63.06	oom	63.06
RMAT-0.1	2.78	10.91	2.78	2.51	4.99	2.51
RMAT-0.5	4.63	12.56	4.63	3.36	6.14	3.36
RMAT-1	7.32	13.37	7.07	5.82	7.02	5.27
RMAT-5	23.66	13.59	12.01	16.24	7.81	6.62
RMAT-10	43.85	13.7	13.05	29.52	8.3	7.14

TABLE IV
THE GRAPH PROPAGATION KERNEL TIME (MS) OF GCN, GCN-OPT, DGL, PYG AND PCGCN ON GTX 1080TI AND RTX 2080TI. "OOM" INDICATES THE OUT-OF-MEMORY ERROR.

Time (ms)	NVIDIA GTX 1080Ti						NVIDIA RTX 2080Ti (Tensor Core Acceleration)					
Dataset	GCN	GCN-opt	DGL	PyG	PCGCN	speedup	GCN	GCN-opt	DGL	PyG	PCGCN	speedup
pubmed	0.42	0.41	0.84	1.19	**0.25**	1.7×	0.31	0.31	0.62	0.74	**0.19**	1.6×
blog	2.56	2.51	0.46	4.25	**0.32**	1.4×	0.68	0.68	0.39	2.17	**0.23**	1.7×
youtube	32.94	32.88	3.43	60.08	**2.51**	1.4×	7.89	7.85	2.49	32.35	**1.03**	2.4×
C1000-9	2.86	2.84	0.39	14.12	**0.05**	7.8×	0.79	0.79	0.26	3.69	**0.03**	8.7×
MANN-a81	29.21	29.19	2.57	219.84	**0.22**	11.7×	7.45	7.41	2.31	59.23	**0.15**	15.4×
reddit-s	10.65	10.61	1.46	18.14	**0.72**	2.0×	2.64	2.59	1.07	9.6	**0.42**	2.5×
reddit	56.68	56.57	23.9	oom	**10.84**	2.2×	19.87	19.82	12.25	oom	**5.1**	2.4×
RMAT-0.1	0.79	0.79	0.32	1.51	**0.09**	3.5×	0.22	0.22	0.23	0.89	**0.06**	3.7×
RMAT-0.5	3.22	3.17	0.58	5.66	**0.32**	1.8×	0.83	0.83	0.49	3.04	**0.23**	2.1×
RMAT-1	6.47	6.42	0.94	11.59	**0.58**	1.6×	1.87	1.86	0.84	6.29	**0.41**	2.0×
RMAT-5	32.03	32.01	3.16	56.64	**1.49**	2.1×	7.73	7.7	2.46	30.36	**0.70**	3.5×
RMAT-10	64.43	64.39	6.05	oom	**1.66**	3.6×	14.80	14.79	4.72	oom	**0.73**	6.5×

updating on-the-fly in the graph propagation stage, which avoids redundant memory copies in the GPU global memory. PyG decouples the graph propagation stage into three individual procedures (i.e., scatter message from source to edge, update edge with the message and gather updated messages to destination) and thus take extra memory to store the intermediate data.

To evaluate the contribution of *selective* and *full* execution modes, we also run PCGCN under exact each mode, marked as PCGCN-sel and PCGCN-full, respectively. Table III shows the results. PCGCN tends to process most subgraphs within the *selective* mode for sparse graphs (e.g., pubmed, blog, etc..) and the *full* mode for dense graphs (e.g., C1000-9, MANN-a81 etc.), respectively. Moreover, PCGCN tends to process more subgraphs in the *full* mode on the RTX 2080Ti GPU due to the tensor core acceleration. Take reddit-s and youtube on GTX 1080Ti for case study, we find that PCGCN uses the *full* mode to process about 6% subgraphs (28% edges) of reddit-s, and 96% subgraphs (99% edges) of youtube, respectively.

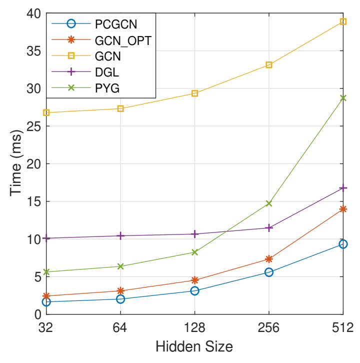
C. Efficient Graph Propagation

The superiority of PCGCN benefits from the acceleration in the graph propagation stage of GCN. To evaluate the exact efficiency from PCGCN, we evaluate the runtime of the graph propagation stage in the feed-forward computation of the first layer of the GCN models in Section V-B. We leverage the nvprof [21] tool and measure the GPU kernel time of graph propagation.

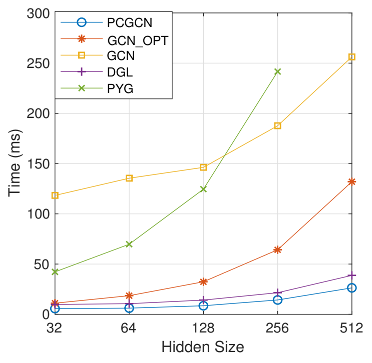
Table IV shows the results on both GTX 1080Ti GPU and RTX 2080Ti GPU. It is clear that PCGCN significantly outperforms the baselines with an up-to 11.7× and 15.4× speedup on GTX 1080Ti and RTX 2080Ti, respectively. Comparing with Table II, the propagation take a larger proportion in graphs of higher density. GCN and GCN-opt have the same graph propagation time, because we only optimize the implementation of feeding data without any modification on GCN computing.

D. Sensitivity Study

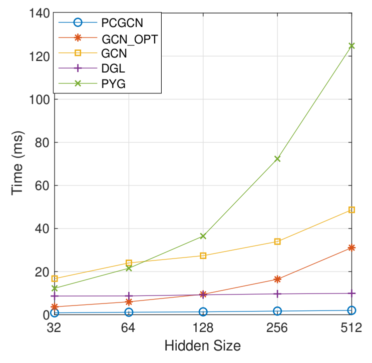
We have shown the superiority of PCGCN in Section V-B with a fixed hidden size $hid = 64$ and a fixed number of layers $\ell = 4$. Note that the performance of GCN is affected by both the hidden size and the number of layers. Thus, we evaluate the performance of hidden size $hid \in \{32, 64, 128, 256, 512\}$ and number of layers $\ell \in \{2, 4, 6, 8, 10\}$ in this section. We take the pubmed, reddit-s and C1000-9 datasets on the GTX



(a) pubmed

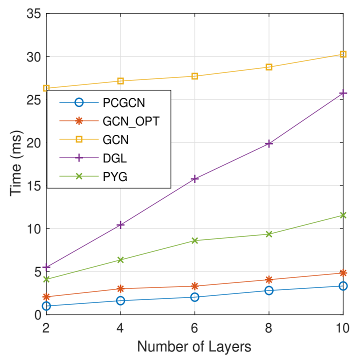


(b) reddit-s

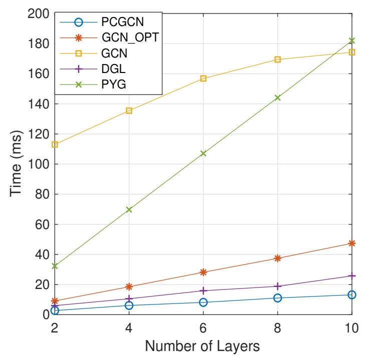


(c) C1000-9

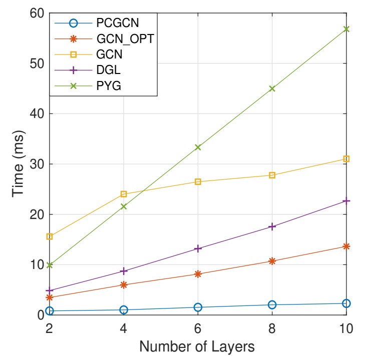
Fig. 7. Runtime (ms) of different hidden sizes on pubmed, reddit-s and C1000-9 on RTX 2080Ti. PyG runs out-of-memory on reddit-s when $hid = 512$.



(a) pubmed



(b) reddit-s



(c) C1000-9

Fig. 8. Runtime (ms) of different layers on pubmed, reddit-s and C1000-9 on RTX 2080Ti.

2080Ti GPU as the example, because these datasets have an increasing density and clustering coefficient. The results on other graphs are similar.

Figure 7 and Figure 8 shows that PCGCN outperforms all the baselines within any hidden size and number of layers configurations. On the one hand, it is clear that the runtime of PCGCN has a nearly linear correlation with both the hidden size and the number of layers. On the other hand, compared results on different dataset, PCGCN achieves a higher speedup on graphs with higher clustering coefficient.

E. Convergence Study under Tensor Core Acceleration

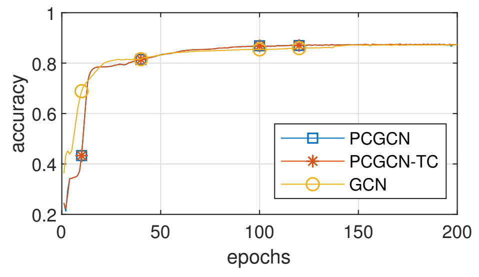
In Section IV, we introduce the tensor core acceleration technique for PCGCN on GPU. The tensor core uses the 16-bit float instead of the 32-bit, and thus may lead to accuracy loss in training. To evaluate the impact, we compare the per-epoch convergence of GCN, PCGCN and PCGCN-TC (tensor core enabled) on the pubmed and reddit-s datasets. We partition both datasets into 80% training set and 20% test set, and evaluate the per-epoch testing accuracy under the same model configurations as Section V-B.

As Figure 9 shown, GCN, PCGCN and PCGCN-TC have a similar per-epoch accuracy on the test set on both datasets and achieve a similar convergence. Note that there is a little variance at beginning on the pubmed dataset because of the random weight initialization. Finally, PCGCN-TC has an accuracy drop of 0.08% and 0.19% for pubmed and reddit-s, respectively. So, the tensor core acceleration can accelerate PCGCN-TC with little impact on the convergence.

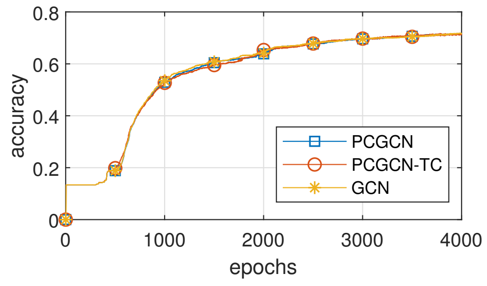
VI. Related Work

a) Graph Neural Networks: Graph neural networks (GNNs) which apply deep neural networks (DNNs) on graph-structured data, was first proposed in [22]. Most recently, driven by the widely used graph data (e.g., knowledge graphs, social networks, protein networks, etc.), applying deep learning models on graphs [1], [16], [22]–[28] is becoming an emerging trend, and moving the state-of-the-art prediction results in their targeted applications such as vertex classification [1], link prediction [29], recommendation [2], graph embedding [30], and question-answering [31], etc. Graph convolutional networks [1] is one of the most widely used models, combining merits and wisdoms from CNNs in computer vision.

b) GPU-accelerated Graph Processing: The growing need for efficient large-scale graph processing has driven the design and implementation of many specialized graph processing systems, e.g., Pregel [32], GraphLab [33], Pow-



(a) pubmed



(b) reddit-s

Fig. 9. Per-epoch convergence of GCN and PCGCN within the same model configuration. Little variance at beginning due to random weight initialization.

erGraph [12], GraphX [34] and Gemini [35]. Triggered by the advances of high-performance GPUs, there are many other works that focus on exploiting GPUs for large-scale graph processing. CuSha [36] exploits new graph representations for fast GPU-based graph processing. GTS [37] leverages asynchronous GPU streams for out-of-GPU-core graph processing. Garaph [38] exploits replication-based strategy to resolve conflicts in processing irregular graphs. LuX [39] leverages the memory hierachy for better data placement in distributed GPU-based graph processing. However, all of these works are focused on basic graph applications like PageRank and shortest path, and lack the capacity for neural network computing. PCGCN exploits optimizations from graph locality to resolve the efficiency challenge in GCN training.

c) GCN Implementations on GPU: Implementing GCN is challenging, as high GPU throughput needs to be achieved on the highly sparse and irregular data of varying sizes. Deep learning (DL) frameworks such as TensorFlow [6], PyTorch [14], and MXNet [13] are designed for DNN computing, but cannot naturally express and efficiently process graph propagation in GCN.

Most recently, driven by the requirement of processing GNN models, there are a series of system work that focus on GNN training, e.g., NeuGraph [40], Deep Graph Library (DGL) [3], [4], PyTorch Geometric (PyG) [5], Euler [41], and AliGraph [42]. However, they focus on designing a graph-oriented interface for GNN programming rather than drilling down to the performance issues that derived from graph-aware operations in GCN. Euler [41] is a distributed CPU-based GNN training system for industrial-scale graphs, but it does not support GPU-acceleration. NeuGraph [40] and DGL [3], [4] introduces the operator fusion and dataflow optimization techniques to accelerate GCNs from dataflow level on both CPU and GPU, but they lack optimizations from graph structures. PCGCN addresses the computing efficiency challenge in GCN computation by leveraging graph properties to design system-level optimizations.

Besides, to handle large-scale graphs, graph sampling approaches such as GraphSAGE [16] are developed to alleviate the scalability challenge in GNN processing. GraphSAGE [16] learns node's embedding by neighbor sampling and aggregator, which reduces the computation overhead at the expense of model capacity and convergence guarantee. FastGCN [43] proposes importance sampling to fix the number of nodes per GCN layer. VR-GCN [44] developes a variance reduction technique to reduce the number of neighborhood sampling nodes. Cluster-GCN [45] samples graphs by leveraging graph clustering to achieve higher model accuracy. These approaches are orthogonal to and compatible with PCGCN. PCGCN can further accelerate GCN computing by leveraging graph locality.

VII. Conclusion and Future Work

In this paper, we presented the PCGCN for fast GCN computation, which perceives a graph as a set of links between nodes and partitions instead of a sparse adjacent matrix. We presented and developed system-level optimizations to achieve high performance by leveraging graph properties. PCGCN is a stepping stone in building systems for GNNs. In the future, we would like to extend this partition-centric processing scheme to a general GNN framework so that more GNN models can benefit from the acceleration.

Acknowledgements

Authors would like to thank the anonymous reviewers for their insightful comments. This work was supported by the National Natural Science Foundation of China under Grant No. 61972004.

References

[1] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, ser. ICLR'17, 2017.

[2] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD'18. ACM, 2018, pp. 974–983.

[3] "Deep graph library," https://github.com/dmlc/dgl, Retrieved October, 2019.

[4] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," *arXiv preprint arXiv:1909.01315*, 2019.

[5] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI'16. USENIX Association, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[7] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *nature*, vol. 393, no. 6684, p. 440, 1998.

[8] S. H. Strogatz, "Exploring complex networks," *nature*, vol. 410, no. 6825, p. 268, 2001.

[9] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI magazine*, vol. 20, no. 1, pp. 61–80, 2008.

[10] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM computer communication review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[11] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs." in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI'12. USENIX Association, 2012, pp. 17–30.

[13] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," in *NIPS Workshop on Machine Learning Systems*, ser. LearningSys'16, 2016.

[14] "PyTorch," http://pytorch.org, Retrieved October, 2019.

[15] L. Tang and H. Liu, "Relational learning via latent social dimensions," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD'09. ACM, 2009, pp. 817–826.

[16] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, ser. NIPS'17, 2017, pp. 1024–1034.

[17] L. Tang, X. Wang, and H. Liu, "Uncoverning groups via heterogeneous interaction analysis," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ser. ICDM'09. IEEE, 2009, pp. 503–512.

[18] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[19] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.

[20] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD'16. ACM, 2016, pp. 855–864.

[21] Nvidia Corporation, "Profiler :: Cuda toolkit documentation," https://docs.nvidia.com/cuda/profiler-users-guide/index.html, Retrieved October, 2019.

[22] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, ser. IJCNN'05. IEEE, 2005, pp. 729–734.

[23] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in Neural Information Processing Systems*, ser. NIPS'16, 2016, pp. 3844–3852.

[24] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[25] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *arXiv preprint arXiv:1901.00596*, 2019.

[26] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun, "Graph neural networks: A review of methods and applications," *arXiv preprint arXiv:1812.08434*, 2018.

[27] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *arXiv preprint arXiv:1812.04202*, 2018.

[28] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[29] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European Semantic Web Conference*. Springer, 2018, pp. 593–607.

[30] T. N. Kipf and M. Welling, "Variational graph auto-encoders," *arXiv preprint arXiv:1611.07308*, 2016.

[31] M. Narasimhan, S. Lazebnik, and A. Schwing, "Out of the box: Reasoning with graph convolution nets for factual visual question answering," in *Advances in Neural Information Processing Systems*, ser. NIPS'18, 2018, pp. 2654–2665.

[32] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD'10. ACM, 2010, pp. 135–145. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[33] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[34] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI'14. USENIX Association, 2014, pp. 599–613. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[35] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 301–316.

[36] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: Vertex-centric graph processing on gpus," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14. ACM, 2014, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600227

[37] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim, "GTS: A fast and scalable graph processing method based on streaming topology to gpus," in *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '16. ACM, 2016, pp. 447–461. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915204

[38] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication," in *Proceedings of the 2017 USENIX Annual Technical Conference*, ser. USENIX ATC'17. USENIX Association, 2017, pp. 195–207.

[39] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken, "A distributed multi-gpu system for fast graph processing," *Proceedings of the VLDB Endowment*, vol. 11, no. 3, pp. 297–310, Nov. 2017. [Online]. Available: https://doi.org/10.14778/3157794.3157799

[40] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "NeuGraph: Parallel deep neural network computation on large graphs," in *2019 USENIX Annual Technical Conference*, ser. USENIX ATC'19. USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/ma

[41] "Euler," https://github.com/alibaba/euler, Retrieved October, 2019.

[42] H. Yang, "Aligraph: A comprehensive graph neural network platform," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2019, pp. 3165–3166.

[43] J. Chen, T. Ma, and C. Xiao, "FastGCN: fast learning with graph convolutional networks via importance sampling," in *International Conference on Learning Representations*, ser. ICLR'18, 2018.

[44] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," in *International Conference on Machine Learning*, 2018, pp. 941–949.

[45] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: ACM, 2019, pp. 257–266.