GIN: High-Performance, Scalable Inference for Graph Neural Networks

Qiang Fu and H. Howie Huang

Graph Computing Lab
The George Washington University
{charlesfoo, howie}@gwu.edu

Abstract. Deep learning models have enjoyed tremendous success when applying to low-dimensional regular grid data such as image, video, and speech. Recently, graph neural networks (GNNs) have been proposed to learn from high-dimensional graph-structured data (e.g., social networks, molecular structures, and protein networks). Unfortunately, existing systems that are developed for the construction, training, and deployment of GNN models suffer from poor performance, especially when running on big graphs that exceed the size of the on-board DRAMs of computation accelerators such as GPUs. In this paper, we present Gin, a new computational framework that is able to generate highly efficient compute kernels for GNN inference. Specifically, Gin enables a user to continue to use a familiar deep learning framework (e.g., TensorFlow) as the front end, while utilizing a translator to translate the high-level representation of a GNN model into low-level codes. The back end in Gin will compile the translated code and create the optimized kernels on CPU. Our evaluation shows that Gin outperforms the state-of-art systems by up to three orders of magnitude, significantly accelerating the inference on billion-edge graphs.

Keywords: Deep Learning · Graph Neural Networks · Inference.

1 Introduction

Graph has been extensively employed to model various networks such as social networks, molecular graph structures, and biological-protein networks [6], thanks to its ability of capturing the relationships (*edges*) between different entities (*nodes*). As a result, researchers are increasingly interested in extracting useful information from graph structures, as evidenced by an emerging trend of applying deep learning (DL) techniques to graphs [2]. Similar to the tremendous success of deep learning in speech, computer vision, and natural language processing, these graph-based neural networks (GNNs) have also demonstrated promising potentials in many applications (e.g. classification, embedding, and recommendation system) [5].

In this work, we are interested in the inference performance of these graph neural networks for two reasons. First, the inference of a trained model is the

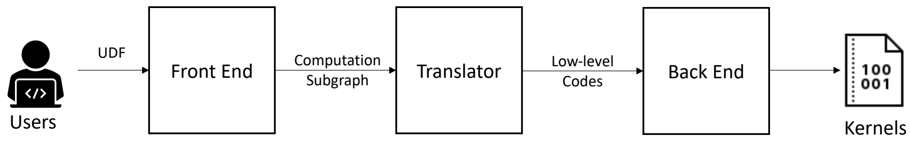

Fig. 1. Gin Architecture

process when the model is deployed to classify, recognize, and process new inputs. In fact, inference takes up most of the life cycle of deep learning models and the number of applications relying on inference is abundant. For example, Facebook services tens of trillions of inference queries per day [7, 22]. Second, it is very important to point out that the inference phase, due to the absence of backward operations and repeatedly updating on weights, is not as computational intensive as the model training phase. However, it has a much more strict requirement on throughput and latency [24], given its customer-facing nature [1]. It is also worthy noting that unlike the training phase, the inference is often done on the edge devices with less computational resources.

Unfortunately, exist systems all suffer from poor performance especially when running on big graphs that exceed DRAM size of computation accelerators such as GPUs. As we will show later in Section 2.2, it can take several minutes for current systems to process a billion-edge graph. In this paper, we believe that highly optimized kernels that are written in low-level codes are needed for faster GNN inference. To this end, we have developed Gin, a new graph inference framework that allows users to create highly-optimized inference kernels for a variety of GNN models.

As shown in Figure 1, Gin consists of three major components: a front end based on a common deep learning framework (Tensorflow used in this work), a back end that utilizes a set of graph processing techniques, and a translator that connect these two. Specifically, a user can use the dataflow framework provided by the front end to define the computation in a GNN model. The front end of Gin provides a set of message-passing-like APIs, which can be used to construct user-defined functions (UDFs) to express the NN (neural network) computations on the tensor data associated with each edge and vertex. As Gin's front end is built upon Tensorflow, users can use the operations on tensor provided by Tensorflow. From the UDFs, the computation subgraphs will be derived and fed into the translator to generate low-level codes. On the other hand, the back end is a high-performance graph processing system written in low-level codes (C++ in this work). The back-end takes the codes generated by the translator as the input; then compiles with third party libraries to generate an optimized binary. This binary reads the input tensor data for vertices or edges and yields the resulting tensor data of the graph convolution.

The remainder of the paper is organized as follows. Section 2 gives a description of the background of GNN inference and motivation of this work. Section

3 discusses the Gin framework in detail. Section 4 presents our experimental results. We conclude in Section 5.

2 Background and Motivation

2.1 Inference of Graph Neural Networks

There have been a number of GNN models that are proposed recently (interested readers may refer to [23]). In this work, we have evaluated three models, *CommetNet* [19], *GCN* [9], and *G-GCN* [17]. In CommetNet, cooperating agents learn to communicate among themselves before taking actions. This model can be used to solve multiple learning tasks such as traffic control. In contrast, GCN applies a weighted-sum convolutional operation on an arbitrary graph, and has been used in many semi-supervised or unsupervised tasks. Furthermore, incorporating the gating mechanism into graph convolution, the G-GCN model is used to extract features for community detection.

A GNN model's life cycle consists of two stages that have drastically different computational properties. During the *training* stage, a training data set is fed into the GNN model and the weights of the model are iteratively updated using the back-propagation algorithm. To deal with large input graphs, many existing models utilize vertex sampling techniques to train their models on a small portion of the input graph in each epoch. Once the model is trained, the second *inference* stage is to perform the forward pass of the model on the whole graph. Clearly, while the training is often done in an offline fashion, the inference needs to be performed quickly in real time. Therefore, improving inference performance is of vital importance to the wider application of GNNs.

In a GNN model, the input data can be divided into three classes: feature vector for vertices, feature vector for edges, and weight parameters. Same as traditional deep learning models, GNN models are also composed of a series of operations, which can be classified into two classes: normal neural network (NN) operation and graph convolution. The first class of NN operations treats the feature vectors of all vertices or edges as a feature matrix, and performs traditional NN operations, such as convolution, full-connected network, or activation function (e.g. relu, softmax).

At the heart of a GNN model is the process of *graph convolution*, which differs from NN operations in that it allows information in the feature of vertices and edges propagating along the edges to generate new information. Specifically, in graph convolution, each vertex applies the NN operations to the features of all its neighbors and the associated edges, and combines the results with its own features to produce the new feature vector for the next layer. Depending on the scale and irregularity of the graphs, such convolution could take a significant amount of time to complete.

Unlike traditional deep learning operations, such as *Convolution* and *Relu*, the graph convolution operations present a huge diversity of computation patterns among different GNN models. As a result, it is impossible to build a single

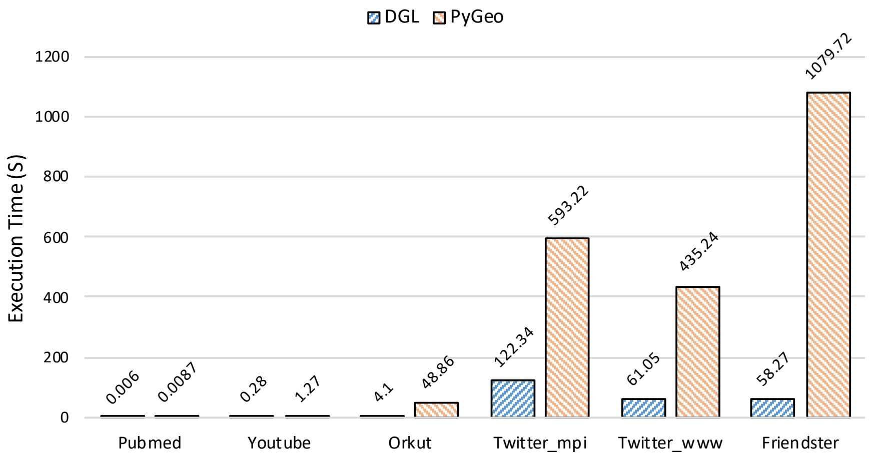

Fig. 2. Single Layer Inference Performance of DGL and Pytorch-geometric

hand-crafted kernel for all different GNN models. In other words, the diversity of GNN models demand a high-level abstraction to represent various GNN models, while the generality usually leads to bad performance, which is a classical problem in system design. In the following, we will demonstrate this problem and present our solution.

2.2 Performance Challenge of Related Works

To understand the inference performance, we have implemented the Graph Convolution Network [9] using two open-source GNN systems, DGL and Pytorch-geometric (or Pytorch in short in this paper). Here we run a forward pass of a single layer with six different input graphs (which will be presented in detail later) on a 24-core shared-memory machine, and the sizes of input and output feature vector for vertices are set to 16. It is worthy to note that only the first two small graphs can be run on the Nvidia V100 GPU, which is equipped with 32 GB memory. From Figure 2, one can observe that it takes several minutes for both systems to process large graphs, e.g., twitter graph with over a billion edges. The root cause of the poor performance lies on the fact that these GNN systems do not provide the most optimized implementation of *graph convolution*. In other words, the systems reuse the APIs offered by the existing deep learning (DL) frameworks such as Tensorflow, because of the dependence on the auto-gradient function. However, no current DL frameworks provide optimized kernels for graph convolution.

3 GIN

3.1 Bridging the Gap

GNN training systems such as Deep Graph Library (DGL) [20], NeuGraph [15], and Pytorch-geometric [3], offer a set of graph-operation based APIs to facilitate the creation and usage of different GNN models. Unfortunately, as we have discussed earlier, because these systems are built on top of popular deep learning frameworks such as Tensorflow, the graph convolution relies on the operations provided by these frameworks, and in turn are often not optimized for GNN inference.

The main observation of this work is that **vertex-centric or edge-centric APIs that are used in graph analytics systems such as [10, 8, 4, 12, 25], can be utilized to provide a highly optimized implementation of the GNN inference**. In this case, one can leverage a Gather-Apply-Scatter (GAS) or Bulk-Synchronous-Parallel (BSP) model for high efficiency. The obvious drawback here is that these systems do not provide a user-friendly interface that allows users to define the neural network (NN) operations on the vertices and edges of graph. As a result, the users need to write customized low-level codes (e.g., user-defined functions) that define the computation on each edge and vertex, which could be tedious and sub-optimized for some common NN operations (e.g., matrix multiplication).

In this work, we aim to design a new framework to bridge the gap between easy-to-use APIs and high-performance implementation of the GNN inference. Gin consists of three major components: a front end that delivers a user-friendly progrmaming interface, a back end that utilizes a set of graph processing techniques, and a translator that connect these two.

3.2 The Front End

The front end of Gin is built on top of the dataflow DL framework (e.g., TensorFlow), allowing users to define the computation in GNN models as a high-level representation. Specifically, it provides a set of message-passing like APIs, in which user-defined functions (UDFs) are needed to express the NN computations on tensor data associated with each edge and vertex. In such UDFs, a user can use the operations on tensor defined in the TensoFlow. Next, the computation subgraphs will be derived from the UDFs and be fed into the translator to generate low-level codes.

We will use Graph Convolution Network (GCN) [9] as a concrete running example to show how Gin works. GCN is one of the most representative GNN models, which performs a weighted-sum aggregator on all neighbor's feature in graph convolution. GCN can be used in application such as node embedding and classification. Let $h_u^{(l)}$ denote the feature vector of vertex u at layer l, and $W^{(l)}$ be the weights matrix that needed to be learned. For each vertex, the feature

Algorithm 1: One layer of Graph Convolution Network

```
Input  : Graph $G(V,E)$; input feature matrix $H^{(l)}$, $H_v^{(l)}$ means the feature
         vector of vertex $v$; degree array $d$ for all vertices; weight matrix
         $W^{(l)}$; non-linearity $\sigma$; neighborhood function $N : v \to 2^V$.
Output: New Feature matrix $H^{(l+1)}$ for next layer.
1  Allocate memory for result $H^{(l+1)}$;
2  $H \leftarrow H^{(l)} \cdot W^{(l)}$ ;   // Matrix multiplication of feature matrix $H$ and
     weight parameters $W$.
3  for $v \in V$ do
4      $L \leftarrow []$ ;                                   // Empty list.
5      for $u \in N(v)$ do
6          $c_{uv} \leftarrow \frac{1}{\sqrt{d_u \cdot d_v}}$ ;                  // Normalization.
7          $L$.add($c_{uv} \cdot H_u$) ;              // Message from src vertex.
8      $H_v^{(l+1)} \leftarrow$ Sum(L) ;          // Merging the incoming messages.
   // Graph Convolution ends.
9  $H^{(l+1)} \leftarrow \sigma(H^{(l+1)})$ ;                     // Activation function
10 return $H^{(l+1)}$
```

vectors for next layer can be calculated as follow:

$$h_v^{(l+1)} = \sigma\left(\sum_{u\in N(v)} \frac{1}{\sqrt{d_v \cdot d_u}} h_u^{(l)} W^{(l)}\right), \tag{1}$$

where d_v is the degree value of vertex v, $N(v)$ is the neighbor list of vertex v, $\frac{1}{\sqrt{d_v \cdot d_u}}$ serves as a normalization constant for edge (v, u), and $\sigma(\cdot)$ denotes a differentiable, non-linear activation function (e.g. Relu). According to the taxonomy we made at Section 2.1, the activation function and vector-matrix multiplication can be implemented as normal operations on the feature matrix of all vertices outside the graph convolution. Hence, the computation left in the graph convolution is similar with PageRank algorithm, in which the weighted-sum of all neighbors' feature vector is calculated as the result. Algorithm 1 provides an example implementation for one layer of GCN. Line 3-8 are the codes for graph convolution, while line 2 and 9 complete the operations of matrix multiplication and activation function.

Figure 3 shows the definition of GCN using the front end of Gin. Specifically, four UDFs are provided: ***Init***, ***Apply***, ***Compute***, and ***Gather***. Firstly, the input data is defined in the ***Init*** function, including the feature for vertices or edges, weight parameters. In the ***Apply*** function, users can define the normal NN operations on the tensor data associated with vertices and edges before the graph convolution. For GCN, a matrix multiplication of feature matrix and weight is defined, which corresponds to the line 2 in the Algorithm 1.

The ***Compute*** function is defined on each edge of the input graph structure, giving the information propagation rule along the edge. This function uses

```
// Data registration
Init() {
    vdata.H = Tensor();
    vdata.degree = Tensor();
    weights.W = Tensor();
}
// Defining the operation on features before graph convolution, corresponding to
     the line 2 of Algorithm 1.
Apply() {
    vdata.H = MatMul(vdata.H, weights.W);
}
// Propagation rule of graph convolution, corresponding to the line 6−7 of
     Algorithm 1.
Compute(edge) {
    ret = edge.src.degree ∗ edge.dst.degree;
    ret = Rsqrt(ret);
    ret = ret ∗ edge.src.H;
    return ret;
}
// Aggregation method of graph convolution and additional operations,
     corresponding to the line 8−9 of Algorithm 1.
Gather(messages) {
    ret = Sum(messages);
    return Relu(ret);
}
```

Fig. 3. Definition of GCN using front end of GIN.

an edge data structure to allow users to access the feature of the source and destination vertices and edges, and return the message passing to the destination vertex from the source. In GCN, a normalized value of the feature vector of source vertex is passed to the destination, corresponding to line 6-7 in Algorithm 1.

Lastly, the ***Gather*** function offers users a way of defining the aggregating method on the messages coming from neighbors. Specifically, this function receives a parameter *messages* representing the the array of messages coming from all the neighbors of given vertex, and users can apply a reduce operation on it to define the way of combining in-coming messages. Furthermore, some additional operations can also be used on the result of the reduce operation, such as activation functions. In the example of GCN, a *Sum* operation followed by a *Relu* is defined on the in-coming messages, referring to the line 8-9 in Algorithm 1.

3.3 The Translator and Back End

The translator is designed to bridge the gap between the high-level abstraction and high-performance implementation. For inputs, the translator takes the com-

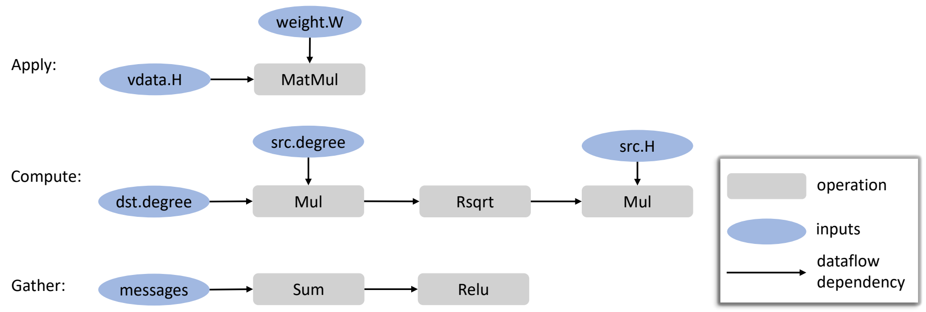

Fig. 4. Computation subgraphs extracted from front end for GCN.

putation subgraphs used by Tensorflow, representing the NN computation on each edge and vertex. The job of the translator is to produce the corresponding functions in the form of low-level codes that will be used by the back end. As a result, GIN can provide different highly optimized kernels for different GNN models, such as GCN or G-GCN.

To continue our example, after the GNN model is defined in the front end, GIN will extract three computation subgraphs from the three UDFs, excluding the ***Init***. Figure 4 shows the computation subgraphs of GCN. As one can see, each operation in the subgraphs can be mapped to a statement in the definition from the front end. Then, the translator takes as input the three computation subgraphs, analyzes them, and generates the corresponding low-level codes that should be placed at the appropriate spot of the back end. Specifically, the translator converts the three subgraphs into three corresponding low-level code blocks, and for each subgraph, iterate all the operations it contains with the topological order, allocate the memory for the output tensor and generate the codes according to some pre-defined rules. At last, GIN combines the output from translator with the back end and compile it into a binary kernel that could be used to execute single layer inference of GCN.

The back end is a high-performance graph processing code template written in low-level programming language (e.g., C++) and specified for graph convolution. Instead of letting users define the computation on edges and vertices, the back end takes the codes generated by the translator as the input; then compiles with third party libraries to generate an optimized binary. This binary reads the input tensor data for vertices or edges and yields the resulting tensor data of the graph convolution. Furthermore, to maximize the parallelism, the back end uses the OpenMP to dynamically assigns different vertices with various degrees to different threads to balance the workload, subsequently achieving better hardware utilization.

To summarize, the output of GIN is a binary kernel that takes as input the graph and the feature tensors for vertices and edges, performs one-layer GNN inference, and outputs the feature matrix for the next layer. It is important to

Table 1. Graph Datasets (K: thousand, M: million, B: billion)

Dataset	**Abbrev.**	**vertex#**	**edge#**	**avg.degree**
pubmed	PD	19.7K	108.4K	5
youtube	YT	1.1M	5.9M	6
orkut	OK	3M	117.1M	39
twitter-www	TW	41.6M	1.4B	34
twitter-mpi	TM	52.5M	1.9B	37
friendster	FD	65.6M	3.6B	56

note that this kernel can be warped as a operation by DL frameworks such as TensorFlow.

3.4 Implementation

We implement the Gin framework using Python for the front end and translator, C++ for the back end. Gin leverages TensorFlow's intermediate representation (IR) to express NN computations in the GNN models. On top of Gin, we have implemented three different GNN models, i.e. CommetNet [19], GCN [9], G-GCN [17], and evaluated with six different real-world graph datasets as shown in Table 1. It is worthy to note that although we implemented Gin on shared-memory multi-core CPU-based system due to its huge memory-capacity, the whole frameworks can be extended to GPU-based system easily just by providing another back end implementation that specified to GPU. Current DL frameworks all offer a way of customizing kernel for Cuda(GPU).

The three GNN models share some common computation properties. For each layer, the feature matrix for all vertices is multiplied with trained weight parameters, then the graph convolution takes the features as input to produce new features for vertices. Finally an activation function is applied on the output from graph convolution.

The difference of the models lies on the propagation rules of the features in graph convolution. In CommNet, there is no computation on edge for graph convolution, thus the feature from the source vertex is simply passed to the destination vertex. In contrast, GCN computes a weighted sum of all neighbors' features based on the degree of each vertex, while G-GCN utilizes the gating mechanism to control the feature propagation in graph convolution.

4 Evaluation

We evaluate GIN on a shared-memory server, which is equipped with dual 2.6GHz Intel Xeon(R) Gold 6126 processors (24 cores in total), 1.5TB memory. The installed operating system is CentOS 7. In this work, we focus on the performance of a single layer of GNN, as it is indicative of the overall performance. Beside the graph data, all the input data including feature and weight are

Table 2. Execution time (s) of DGL, Pytorch-geometric and GIN. TMO means time out of longer than one hour.

Model	System	PD	YT	OK	TW	TM	FS	Avg.speedup
CommNet	DGL	0.0062	0.34	4.5	58.23	145.22	73.56	**10.3**×
	Pytorch	0.0065	1.24	46.34	451.25	601.47	1064.45	**61.55**×
	GIN	0.0019	0.035	0.61	5.16	5.88	16.4	-
GCN	DGL	0.006	0.28	4.1	61.05	122.34	58.27	**8.8**×
	Pytorch	0.0087	1.27	48.86	435.24	593.22	1079.72	**64.81**×
	GIN	0.0014	0.052	0.51	5	5.6	15.53	-
G-GCN	DGL	0.019	0.35	7.12	80.11	156.4	153.53	**4.4**×
	Pytorch	0.18	12.52	517.93	TMO	TMO	TMO	**126.27**×
	GIN	0.0029	0.095	2.8	18.93	24.22	51.8	-

initialized to random 32-bit floating point values. We have verified that different implementations for the same model produce the same outputs.

Table 2 summarizes the execution time of GIN against DGL and Pytorch-geometric (Pytorch in short). One can see that GIN outperforms the other two baselines on every graph dataset for every GNN model and is averagely x7.8 and x83 faster than DGL and Pytorch-geometric, respectively. The performance improvements demonstrate that GIN has generated the implementation which is more optimal than other two frameworks for various GNN models.

5 Related Works

The growing scale and importance of graph data has driven the development of numerous graph processing system both on CPU and GPU platforms, including Pregel [16], GraphLab [14, 13], PowerGraph [4], Gunrock [21], GraphReduce [18] and SIMD-X [12]. Generally speaking, most of them are written as code template libraries using low-level programming language (C++), offering vertex-centric or edge-centric APIs which can be utilized to generate highly optimized kernels for various graph algorithms [11]. The obvious drawback here is that these systems do not provide a user-friendly interface that allows users to define the neural network (NN) operations on the vertices and edges of graph. As a result, the users need to write customized low-level codes (e.g., user-defined functions) that define the computation on each edge and vertex, which could be tedious and sub-optimized for some common NN operations (e.g., matrix multiplication).

On the other hand, DGL [20], Pytorch-geometric [3] and Neugraph [15] warp DL frameworks (Pytorch and MXNet) with a message-passing programming inference for users to define, train and execute GNN models. DGL employs sparse vector matrix multiplication (spmv) to accelerate the graph convolution, while Pytorch-geometric uses a *scatter-gather* operations pair. Neugraph proposes a technique called graph-aware dataflow translation and graph partition to train

GNN models on huge graphs. However, all these graph learning systems have to invoke the operations of underlying DL frameworks, which is usually sub-optimal and may leads to poor performance.

6 Conclusion

In this work, we propose Gin, a novel end-to-end framework for scalable and high-performance GNN inference. Gin is capable of generating high-efficient inference kernels of GNNs without requiring users writing low-level codes. Specifically, the front end offers a set of easy-to-use APIs for users to define a GNN model with just tens of lines of codes. Then translator convert the computation subgraph extracted from the front end to optimized low-level codes and then compile it with back end to generate the binary, which can be used as a kernel in current DL frameworks. The experiment results show that GIN achieves significant speedups over existing solutions for GNN inference.

7 Acknowledgments

The authors thank the AccML'20 reviewers for their constructive feedback. This work was supported in part by National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774.

References

1. Bernardi, L., Mavridis, T., Estevez, P.: 150 successful machine learning models: 6 lessons learned at booking. com. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. pp. 1743–1751. ACM (2019)
2. Bui, T.D., Ravi, S., Ramavajjala, V.: Neural graph learning: Training neural networks using graphs. In: Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining. pp. 64–71. ACM (2018)
3. Fey, M., Lenssen, J.E.: Fast graph representation learning with pytorch geometric. arXiv preprint arXiv:1903.02428 (2019)
4. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12). pp. 17–30 (2012)
5. Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: Advances in Neural Information Processing Systems. pp. 1024–1034 (2017)
6. Hamilton, W.L., Ying, R., Leskovec, J.: Representation learning on graphs: Methods and applications. arXiv preprint arXiv:1709.05584 (2017)
7. Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., et al.: Applied machine learning at facebook: A datacenter infrastructure perspective. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 620–629. IEEE (2018)

8. Ji, Y., Liu, H., Huang, H.H.: ispan: Parallel identification of strongly connected components with spanning trees. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 731–742. IEEE (2018)
9. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)
10. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: Large-scale graph computation on just a {PC}. In: Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12). pp. 31–46 (2012)
11. Liu, H., Huang, H.H.: Enterprise: breadth-first graph traversal on gpus. In: SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. IEEE (2015)
12. Liu, H., Huang, H.H.: Simd-x: Programming and processing of graph algorithms on gpus. In: 2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19). pp. 411–428 (2019)
13. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment **5**(8), 716–727 (2012)
14. Low, Y., Gonzalez, J.E., Kyrola, A., Bickson, D., Guestrin, C.E., Hellerstein, J.: Graphlab: A new framework for parallel machine learning. arXiv preprint arXiv:1408.2041 (2014)
15. Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., Dai, Y.: Neugraph: Parallel deep neural network computation on large graphs. In: 2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19). pp. 443–458 (2019)
16. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 135–146. ACM (2010)
17. Marcheggiani, D., Titov, I.: Encoding sentences with graph convolutional networks for semantic role labeling. arXiv preprint arXiv:1703.04826 (2017)
18. Sengupta, D., Song, S.L., Agarwal, K., Schwan, K.: Graphreduce: processing large-scale graphs on accelerator-based systems. In: SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. IEEE (2015)
19. Sukhbaatar, S., Fergus, R., et al.: Learning multiagent communication with backpropagation. In: Advances in Neural Information Processing Systems. pp. 2244–2252 (2016)
20. Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., Huang, Z., Guo, Q., Zhang, H., Lin, H., Zhao, J., Li, J., Smola, A.J., Zhang, Z.: Deep graph library: Towards efficient and scalable deep learning on graphs. ICLR Workshop on Representation Learning on Graphs and Manifolds (2019), https://arxiv.org/abs/1909.01315
21. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: A high-performance graph processing library on the gpu. In: ACM SIGPLAN Notices. vol. 51, p. 11. ACM (2016)
22. Wu, C.J., Brooks, D., Chen, K., Chen, D., Choudhury, S., Dukhan, M., Hazelwood, K., Isaac, E., Jia, Y., Jia, B., et al.: Machine learning at facebook: Understanding inference at the edge. In: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 331–344. IEEE (2019)
23. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S.: A comprehensive survey on graph neural networks. arXiv preprint arXiv:1901.00596 (2019)

24. Yadwadkar, N.J., Romero, F., Li, Q., Kozyrakis, C.: A case for managed and model-less inference serving. In: Proceedings of the Workshop on Hot Topics in Operating Systems. pp. 184–191. ACM (2019)
25. Zhu, X., Han, W., Chen, W.: Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: 2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15). pp. 375–386 (2015)