IMPROVING THE ACCURACY, SCALABILITY, AND PERFORMANCE OF GRAPH NEURAL NETWORKS WITH ROC

Anonymous Authors[1]

ABSTRACT

Graph neural networks (GNNs) have been demonstrated to be an effective model for learning tasks related to graph structured data. Different from classical deep neural networks which handle relatively small individual samples, GNNs process very large graphs, which must be partitioned and processed in a distributed manner. We present ROC, a distributed multi-GPU framework for fast GNN training and inference on graphs. ROC is up to 4.6× faster than existing GNN frameworks on a single machine, and can scale to multiple GPUs on multiple machines. This performance gain is mainly enabled by ROC's graph partitioning and memory management optimizations. Besides performance acceleration, the better scalability of ROC also enables the exploration of more sophisticated GNN architectures on large, real-world graphs. We demonstrate that a class of GNN architectures significantly deeper and larger than the typical two-layer models can achieve new state-of-the-art classification accuracy on the widely used Reddit dataset.

1 INTRODUCTION

Graphs provide a natural way to represent real-world data with relational structures, such as social networks, molecular networks, and webpage graphs. Recent work has extended deep neural networks (DNNs) to extract high-level features from data sets structured as graphs, and the resulting architectures, known as graph neural networks (GNNs), have recently achieved state-of-the-art prediction performance across a number of graph-related tasks, including vertex classification, graph classification, and link prediction (Kipf & Welling, 2016; Hamilton et al., 2017; Xu et al., 2019).

GNNs combine DNN operations (e.g., convolution and matrix multiplication) with iterative graph propagation: in each GNN layer, the activations of each vertex are computed with a set of DNN operations, using the activations of its neighbors from the previous GNN layer as inputs. Figure 1 illustrates the computation of one vertex (in red) in a GNN layer, which aggregates the activations from its neighbors (in blue), and then applies DNN operations to compute new activations of the vertex.

Existing deep learning frameworks do not easily support GNN training and inference at scale. TensorFlow (Abadi et al., 2016), PyTorch (PyTorch), and Caffe2 (Caffe2) were originally designed to handle situations where the model and data collection can be *large*, but each sample of the collection is relatively *small* (e.g., a single image). These systems typically leverage data and/or model parallelism, by partitioning the batch of input samples or the DNN models across multiple devices such as GPUs. Each input sample is stored on a single GPU and not partitioned. However, GNNs use *small* DNN models (typically a couple of layers) on very *large* and irregular input samples — graphs. These large graphs must be partitioned and processed in a distributed manner without fitting in a single device. Recent GNN frameworks such as DGL (DGL, 2018) and PyG (Fey & Lenssen, 2019) are implemented on top of PyTorch (PyTorch), and have the same scalability limitation. NeuGraph (Ma et al., 2019) stores intermediate GNN data in the host CPU DRAM to support multi-GPU training, but it

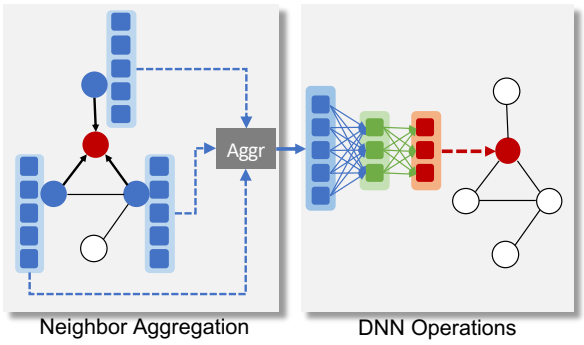


Figure 1. Computation of one vertex (in red) in a GNN layer by first aggregating its neighbors' activations (in blue), and then applying DNN operations.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the Systems and Machine Learning (SysML) Conference. Do not distribute.

is still limited to the compute resources on a single machine. AliGraph (Yang, 2019) is a distributed GNN framework on CPU platforms, which does not exploit GPUs for performance acceleration.

Such lack of system support has limited the potential application of GNN algorithms on large-scale graphs, and has also prevented the exploration of larger and more complicated GNN architectures. To alleviate the system limitation, various *sampling* techniques (Hamilton et al., 2017; Ying et al., 2018) were introduced to first down-sample the original graphs before applying the GNN models, so that the data can fit in a single device. Sampling allows the existing frameworks to train larger graphs, but at the potential cost of model accuracy loss (Hamilton et al., 2017).

In this paper, we propose ROC, a distributed multi-GPU framework for fast GNN training and inference on large-scale graphs. ROC can scale to multiple GPUs on multiple machines to run GNN models on the *full* graphs, and achieves up to 4.6× performance improvement over existing GNN frameworks. Despite its use of the full graphs, ROC also achieves better time-to-accuracy performance compared to existing sampling techniques. Moreover, the better scalability allows ROC to easily support larger and more sophisticated GNNs than those possible in existing frameworks. To demonstrate this, we build a class of deep GNN architectures formed by stacking multiple GCN layers (Kipf & Welling, 2016). By using significantly larger and deeper GNN architectures, we improve the classification accuracy over state-of-the-art sampling techniques by 1.5% on the widely used Reddit dataset (Hamilton et al., 2017).

To achieve these results, ROC tackles two significant system challenges for distributed GNN computation.

Graph partitioning. Real-world graphs could have arbitrary sizes and variable per-vertex computation loads, which are challenging to partition in a balanced way (Gonzalez et al., 2014; Zhu et al., 2016). GNNs mix compute-intensive DNN operations with data-intensive graph propagation, making it hard to statically compute a good load-balancing partitioning. Furthermore, GNN inference requires partitioning new input graphs that only run for a few iterations, such as predicting the properties of newly discovered proteins (Hamilton et al., 2017), in which case existing dynamic repartitioning approaches do not work as well (Venkataraman et al., 2013). ROC uses an *online linear regression model* to optimize graph partitioning. During the training phase of a GNN architecture, ROC jointly learns a *cost model* for predicting the run time of performing a GNN operation on an input (sub)graph. To capture the runtime performance of a GNN operation, the cost model includes both graph-related features such as the number of vertices and edges in the graph, and hardware-related features such as the number of GPU memory accesses to execute the operation. During online learning, for each training iteration, ROC computes a graph partitioning using the run time predictions from the cost model, and uses the partitioning to parallelize training. At the end of each training iteration, the actual run time of the subgraphs is sent back to the graph partitioner, which updates the cost model by minimizing the difference between the actual and predicted run times. We show that this learning-based graph partitioner outperforms existing graph partitioning strategies by 1.4×.

Memory management. In GNNs, computing on even a single vertex needs to access a potentially large number of neighbor vertices that may span across multiple GPUs and machines. These data transfers have a high impact on overall performance. The framework thus must carefully decide in which device memory (CPU or GPU) to store each intermediate tensor, in order to minimize data transfer costs. The memory management is hard to optimize manually as the optimal strategy depends on the input graph size and topology as well as the device constraints such as memory capacity and communication bandwidth. We formulate minimizing data transfers as a cost minimization problem, and introduce a dynamic programming algorithm to quickly find a globally optimal strategy. We compare the ROC memory management strategy with existing heuristic approaches, and show that ROC reduces data transfer costs between CPU and GPU by 2×.

Overall, compared to NeuGraph, ROC improves the runtime performance by up to 4.6× for multi-GPU training on a single compute node. Beyond improved partitioning and memory management, ROC sees smaller performance improvements from use of a more efficient distributed runtime (Jia et al., 2019) and the highly optimized GPU kernels adopted from Lux (Jia et al., 2017).

Besides performance acceleration, ROC also enables exact GNN computation on *full* original graphs without using sampling techniques, as well as the exploration of more sophisticated GNN architectures beyond the commonly used two-layer models. For large real-world graphs, we show that performing exact GNN computation on the original graphs and using larger and deeper GNN architectures can increase the model accuracy by 1.5% compared to existing sampling techniques.

To summarize, our contributions in this work are:

- On the systems side, we present ROC, a distributed multi-GPU framework for fast GNN training and inference on large-scale graphs. ROC uses a novel online linear regression model to achieve efficient graph partitioning, and introduces dynamic programming to minimize data transfer cost.

Table 1. The graph partitioning strategies used by different frameworks. Balanced training/inference indicates whether an approach can achieve balanced partitioning for GNN training/inference.

Frameworks	Partitioning Strategies	Balanced Training	Balanced Inference
TensorFlow, NeuGraph	Equal		
GraphX, Gemini	Static		
Presto, Lux	Dynamic	✓	
ROC (ours)	Online learning	✓	✓

- On the machine learning side, ROC removes the necessity of using sampling techniques for GNN training on large graphs, and also enables the exploration of more sophisticated GNN architectures. We demonstrate this potential by achieving new state-of-the-art classification accuracy on the Reddit dataset.

2 Background and Related Work

2.1 Graph Neural Networks

A GNN takes graph-structured data as input, and learns a representation vector for each vertex in the graph. The learned representation can be used for down-stream tasks such as vertex classification and link prediction (Kipf & Welling, 2016; Xu et al., 2019).

As shown in Figure 1, each GNN layer gathers the activations of the neighbor vertices from the previous GNN layer, and then updates the activation of the vertex, using operations such as convolution or matrix multiplication. Formally, the computation in a GNN layer is:

$$a_v^{(k)} = \text{AGGREGATE}^{(k)}\left(\{h_u^{(k-1)} | u \in \mathcal{N}(v)\}\right) \quad (1)$$

$$h_v^{(k)} = \text{UPDATE}^{(k)}(a_v^{(k)}, h_v^{(k-1)}) \quad (2)$$

where $h_v^{(k)}$ is the learned activation of vertex v at the k-th layer, $h_v^{(0)}$ is the input feature of v. $\mathcal{N}(v)$ denotes v's neighbors in the graph. For each vertex, AGGREGATE gathers the activations of its neighbors using an accumulative function such as average or summation. UPDATE computes new activation $h_v^{(k)}$ by combing its previous activation $h_v^{(k-1)}$ and the neighborhood aggregation $a_v^{(k)}$. Eventually, $h_v^{(k)}$ captures the structural information for all neighbors within k hops of v.

2.2 Related Work

Distributed ML training. In the terminology of Jia et al. (2019), ML problems can be partitioned in the *sample, operator, attribute* and *parameter* dimensions for parallel and distributed execution. The vast majority of existing deep learning frameworks (Abadi et al., 2016; PyTorch) use the sample (i.e., data parallelism) and operator dimensions (i.e., model parallelism) to parallelize training, but some recent works exploit multiple dimensions (Jia et al., 2019). One of the key differences with GNN is that partitioning in the attribute dimension (i.e., partitioning large individual samples) is necessary.

GNN frameworks. Most of the existing GNN frameworks, such as DGL (DGL, 2018) and PyG (Fey & Lenssen, 2019) that extend PyTorch (PyTorch), do not support graphs where the data cannot fit in a single device. NeuGraph (Ma et al., 2019) supports GNN computation on multiple GPUs in a single machine.AliGraph (Yang, 2019) is a distributed GNN framework but only uses CPUs rather than GPUs.

Sampling in GNNs. As discussed in Section 2.1, due to the highly connected nature of real-world graphs, computing $h_v^{(k)}$ may require accessing more data than can fit in one GPU memory. A number of sampling techniques have been proposed to support GNN training on larger input graphs, by down-sampling the neighbors of a vertex (Hamilton et al., 2017; Ying et al., 2018; Chen et al., 2018). The method is formalized as follows.

$$a_v^{(k)} = \text{AGGREGATE}^{(k)}\left(\{h_u^{(k-1)} | u \in \widehat{\mathcal{N}}(v)\}\right) \quad (3)$$

where $\widehat{\mathcal{N}}(v)$ is the sampled subset of $\mathcal{N}(v)$ with a size limit. For example, GraphSAGE (Hamilton et al., 2017) samples at most 25 neighbors for each vertex (i.e., $|\widehat{\mathcal{N}}(v)| \leq 25$), while a vertex may actually contain thousands of neighbors.

In our evaluation, we show that existing sampling techniques come with potential model accuracy loss for large real-world graphs. This observation is consistent with previous work (Hamilton et al., 2017). ROC provides an orthogonal approach to enable large-scale GNN processing. Any existing sampling technique can be additionally applied in ROC to further accelerate large-scale GNN training.

Graph frameworks and graph partitioning. A number of distributed graph processing frameworks (Malewicz et al., 2010; Gonzalez et al., 2014; Jia et al., 2017) have been proposed to accelerate data-intensive graph applications. These systems generally adopt the Gather-Apply-Scatter (GAS) (Gonzalez et al., 2012) vertex-centric programming model. GAS can naturally express the data propagation in GNNs, but cannot support many neural network operations. For example, computing the attention scores (Veličković et al., 2018) between vertices not directly connected cannot be easily expressed in the GAS model.

Table 1 summarizes the graph partitioning strategies used in existing deep learning and graph processing frameworks. Deep learning frameworks (Abadi et al., 2016; Ma et al., 2019) typically partition data (e.g., tensors) *equally* across GPUs. On the other hand, graph processing frameworks use more complicated strategies to achieve load balance. For

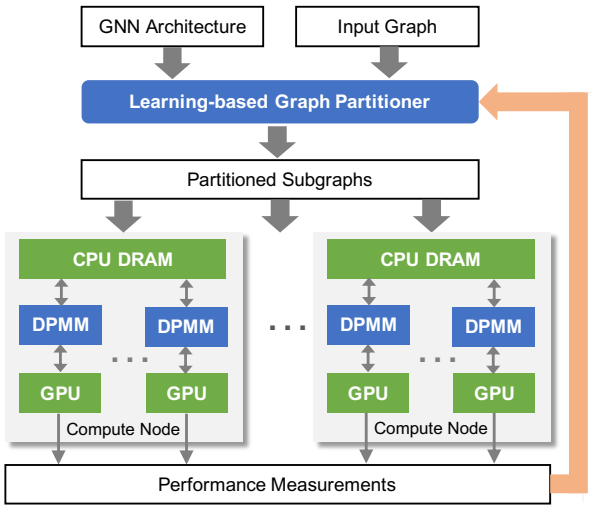


Figure 2. ROC overview. DPMM is the dynamic programming based memory manager.

example, GraphX (Gonzalez et al., 2014) and Gemini (Zhu et al., 2016) *statically* partition input graphs by minimizing a heuristic objective function, such as the number of edges spanning different partitions. These simple objective functions can achieve good performance for data-intensive graph processing, but they do not work well for compute-intensive GNNs due to highly varying per-vertex computation loads. *Dynamic* repartitioning (Venkataraman et al., 2013; Jia et al., 2017) exploits the iterative nature of many graph applications and rebalances the workload in each iteration based on the measured performance of previous iterations. This approach converges to a balanced workload distribution for GNN training, but is much less effective for inference which computes the GNN model only once for each new graph. ROC uses an online-learning-based approach to achieve balanced partitioning for both GNN training and inference, through jointly learning a cost model to predict the run time on arbitrary graphs.

3 ROC OVERVIEW

Figure 2 shows an overview of the ROC distributed runtime. ROC takes a GNN architecture and a graph as inputs, and distributes the GNN computations across multiple GPUs (potentially on different host machines) by partitioning the input graph into multiple subgraphs. Each GPU worker executes the GNN architecture on a subgraph, and communicates with CPU DRAM to obtain input tensors and write output tensors. The communication is optimized by a per-GPU memory manager to minimize communication costs.

ROC uses an online-linear-regression-based graph partitioner to address the unique load imbalance challenge of distributed GNN inference, where a trained GNN model is used to provide inference service on previously unseen graphs (Section 4). This problem exists today in real-world GNN inference services (Hamilton et al., 2017), and our partitioning technique improves the inference performance by 1.4× compared to existing strategies. The graph partitioner is trained jointly with the training phase of the GNN architecture, and also used to partition the training workload when a new graph is encountered in the training set.

The subgraphs are then sent to multiple GPUs to process in parallel. Instead of requiring all the intermediate results related to each subgraph to fit in GPU device memory, ROC uses the much larger CPU DRAM on the host machines to hold all the data, and treats the GPU memories as caches. Such a design allows us to support much larger graphs. However, transferring the tensors between GPUs and the host DRAM can have a major impact on execution performance. ROC introduces a dynamic programming algorithm to quickly find a memory management strategy to minimize these data transfers (Section 5).

4 GRAPH PARTITIONER

The goal of the ROC graph partitioner is discovering balanced partitioning for GNN training and inference on arbitrary input graphs. This is especially challenging for distributed inference on new graphs, where no existing performance measurements of the new graphs are available. We introduce an *online-linear-regression-based graph partitioner* that uses runtime performance measurements as training samples, and uses the learned performance to enable efficient partitioning on arbitrary graphs.

We formulate graph partitioning for GNNs as an *online learning* task. The performance measurements on graph partitioning are training samples to this task. Each training iteration produces new data points, and the graph partitioner should compute a balanced graph partitioning based on *all* existing data points.

4.1 Cost Model

The key component in the ROC graph partitioner is a *cost model* that predicts the execution time of performing a GNN operator on an arbitrary graph, which could be the whole or any subset of the input graph. Note that the cost model learns to predict the execution time of a GNN operator instead of an entire GNN architecture for two reasons. First, ROC exploits the composability of neural network architectures and the learned model can be applied to a variety of GNN architectures. Second, this allows ROC to gather more training data in each training iteration. For a GNN model with N operators and P partitions, ROC is able to gather $N \times P$ training data points, while modeling the entire GNN

Table 2. The vertex features used in the current cost model. The semantics of the features are described in Section 4.1.

	Definition	Description
x_1	1	the vertex itself
x_2	$\lvert\mathcal{N}(v)\rvert$	number of neighbors
x_3	$\lvert\mathcal{C}(v)\rvert$	continuity of neighbors
x_4	$\sum_i \lceil \frac{c_i(v)}{\texttt{WS}} \rceil$	# mem. accesses to load neighbors
x_5	$\sum_i \lceil \frac{c_i(v) \times d_{in}}{\texttt{WS}} \rceil$	# mem. accesses to load the activations of all neighbors

only provides P data points.

As collecting new training data points is expensive, which requires measuring GNN operations on a GPU, we employ a linear regression model to minimize the number of trainable parameters. Our model assumes that the cost to perform a DNN operation on a vertex is linear in a set of vertex features (e.g., number of neighbors), and the cost to run an arbitrary graph is the summation of its vertices' cost.

The cost model for a DNN operation is formalized as follows.

$$t(p, v) = \sum_i w_i(p) x_i(v) \quad (4)$$

$$t(p, \mathcal{G}) = \sum_{v \in \mathcal{G}} t(p, v) = \sum_{v \in \mathcal{G}} \sum_i w_i x_i(v) \quad (5)$$

$$= \sum_i w_i \sum_{v \in \mathcal{G}} x_i(v) = \sum_i w_i x_i(\mathcal{G}) \quad (6)$$

where $w_i(p)$ is a trainable parameter for operation p, $x_i(v)$ is the i-th feature of vertex v, and $x_i(\mathcal{G})$ sums up the i-th feature of all vertices in $\mathcal{G}$. $t(p, \mathcal{G})$ estimates the time to run operation p on input graph $\mathcal{G}$.

Our model minimizes the mean square error over all available data points.

$$\mathcal{L}(p) = \frac{1}{N} \sum_{i=1}^{N} \big(t(p, \mathcal{G}_i) - y(p, \mathcal{G}_i)\big)^2 \quad (7)$$

where N is the total number of available data points for operation p, and $y(p, \mathcal{G}_i)$ is their performance measurements.

Table 2 lists the vertex features used in the cost model. $x_1(v)$ and $x_2(v)$ capture the computation workload associated with vertex v and its edges, respectively. The remaining features estimate the required memory accesses to GPU device memory. When multiple threads in a GPU warp issue memory references to consecutive memory addresses, the GPU device wherever possible automatically *coalesces* these references to a single memory access that will be handled more efficiently. To describe continuity of a vertex's neighbors, we define $\mathcal{C}(v) = \{c_1(v), ..., c_{|C|}(v)\}$, where $c_i(v)$ is a range of consecutively numbered vertices, and $\mathcal{C}(v)$ includes all of v's neighbors. For example, for vertex v_1 with neighbors $\{v_3, v_4, v_6, v_8\}$, we have $c_1(v_1) = \{v_3, v_4\}$, $c_2(v) = \{v_6\}$, and $c_3(v) = \{v_8\}$. The feature $x_3(v)$ is the number of consecutive blocks in v's neighbors, which is 3 in the example. In addition, $x_4(v)$ and $x_5(v)$ estimate the number of GPU memory accesses to load all neighbors and their input activations.

The cost model can be easily extended to include new features to capture additional operation-specific and hardware-specific information.

4.2 Partitioning Algorithm

Using the learned cost model as an oracle, the ROC graph partitioner computes a graph partitioning that tries to achieve balanced workload distribution under the cost model.

ROC uses the graph partitioning strategy proposed by Jia et al. (2017) to maximize coalesced accesses to GPU device memory, which is critical to achieve optimized GPU performance. Each vertex in a graph is assigned a unique number between 0 and $V - 1$, where V is the number of vertices in the graph. In ROC, each partition holds consecutively numbered vertices, which allows us to use $N - 1$ numbers $\{p_0, p_1, ..., p_{N-1}\}$ to identify a graph partitioning.

ROC preprocesses an input graph by computing the partial sums of each vertex feature, which allows ROC to estimate the runtime of a subgraph in $O(1)$ time. In addition, ROC uses *binary search* to find a splitting point p_i in $O(\log V)$, and therefore computing balanced partitioning only takes $O(N \log V)$ time, where N and V are the number of partitions and number of input vertices, respectively.

5 MEMORY MANAGER

As discussed in Section 3, ROC performs the GNN operations on the GPU devices to maximize the runtime performance, but only requires all the GNN data to fit in the host CPU DRAM to support larger graphs. The GPU memory therefore only needs to cache a subset of intermediate tensors, whose corresponding data transfers between the CPU and GPU can be saved to reduce communication. How to select this subset of tensors to minimize the data transfers within the limited GPU memory becomes a critical memory management problem that ROC must address. This task is challenging as the optimal management strategy depends not only on the GPU device memory capacity and the sizes of the input graph and GNN tensors, but also the topology of the GNN architecture which determines the reuse distance for each tensor.

ROC formulates the GPU memory management as a cost minimization problem: given an input graph, a GNN architecture, and a GPU device, finding the subset of tensors to be cached in the GPU memory that minimizes data trans-

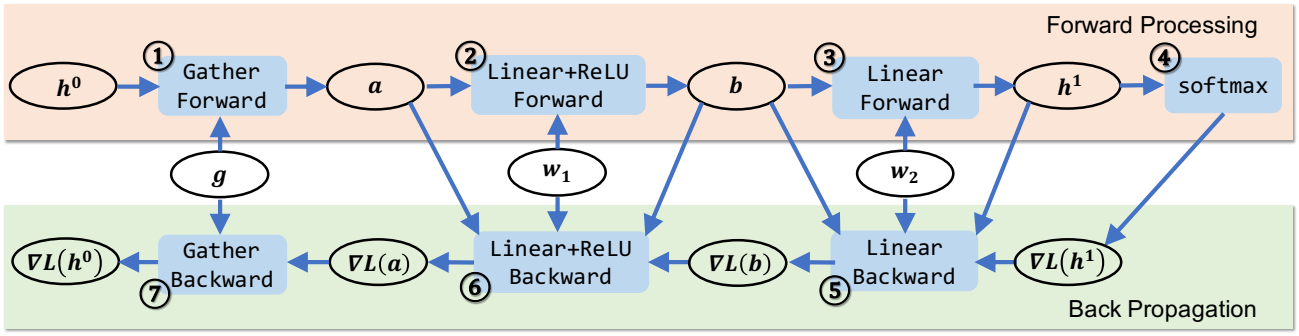


Figure 3. The computation graph of a toy 1-layer GIN architecture (Xu et al., 2019). A box represents an operation, and a circle represents a tensor. Arrows indicate dependencies between tensors and operations. The `gather` operation performs neighborhood aggregation. The `linear` and the following `ReLU` are fused into a single operation as a common optimization in existing frameworks.

Table 3. All the valid states and their activation tensors for the GNN architecture in Figure 3.

Valid State $\mathcal{S}$	**Activation Tensors** $\mathcal{A}(\mathcal{S})$
{①}	$\{a, g\}$
{①, ②}	$\{g, a, b, w_1\}$
{①, ②, ③}	$\{a, b, g, h^1, w_1, w_2\}$
{①, ②, ③, ④}	$\{a, b, g, w_1, w_2, \nabla L(h^1)\}$
{①, ②, ③, ④, ⑤}	$\{a, b, g, w_1, \nabla L(b)\}$
{①, ②, ③, ④, ⑤, ⑥}	$\{a, g, \nabla L(a)\}$
{①, ②, ③, ④, ⑤, ⑥, ⑦}	$\{\}$

fers between the CPU and GPU. ROC introduces a dynamic programming algorithm to quickly find the globally optimal solution.

The key insight of the dynamic programming algorithm is that, at each stage of the computation, we only need to consider caching the tensors that will be used by the future operations. For a GNN architecture $\mathcal{G}$, we define a *state* $\mathcal{S}$ to be the set of operations that have already been performed in $\mathcal{G}$. A state is valid only if the operations it contains preserve all the data dependencies in $\mathcal{G}$, i.e., for any operation in $\mathcal{S}$, all its predecessor operations in $\mathcal{G}$ must be also in $\mathcal{S}$. Such a definition allows the valid states to capture all the possible execution orderings for operations in $\mathcal{G}$. For each state $\mathcal{S}$, we define its *active tensors* $\mathcal{A}(\mathcal{S})$ to be the set of tensors that were produced by the operations in $\mathcal{S}$ and will be consumed as inputs by the operations outside of $\mathcal{S}$. Intuitively, $\mathcal{A}(\mathcal{S})$ captures all the tensors we can cache in the GPU to eliminate future data transfers at the stage $\mathcal{S}$.

Figure 3 shows the computation graph of a toy 1-layer GIN architecture (Xu et al., 2019), whose computation can be formalized as following.

$$h_v^{(1)} = W_2 \times \text{RELU}(W_1 \times \sum_{u \in \mathcal{N}(v)} h_u^{(0)}) \quad (8)$$

For this GNN architecture, all the valid states and their active tensors are listed in Table 3.

Since the valid states represent all the possible execution orderings of the GNN, we can use dynamic programming to compute the optimal memory management strategy associated with each execution state. Algorithm 1 shows the pseudocode. $\text{COST}(\mathcal{S}, \mathcal{T})$ computes the minimum data transfers required to compute all the operations in a state $\mathcal{S}$, with $\mathcal{T}$ being the set of tensors cached in the GPU memory. $\mathcal{T}$ should be a subset of $\mathcal{A}(\mathcal{S})$. We reduce the task of computing $\text{COST}(\mathcal{S}, \mathcal{T})$ in to smaller tasks by enumerating the last operation to perform in $\mathcal{S}$ (Line 10). The cost is the specific data transfers to perform this last operation (*xfer* in Line 14) adding the cost of the corresponding previous state $(\mathcal{S}', \mathcal{T}')$. To improve performance, we leverage memorization to only evaluate COST once for each $\mathcal{S}$ and $\mathcal{T}$ pair.

Time and space complexity. Overall, the time and space complexity of Algorithm 1 are $O(S^2T)$ and $O(ST)$, respectively, where S is the number of possible execution states for a GNN architecture, and T is the maximum number of available tensor sets for a state. We observed that S and T are at most 16 and 4096 for all GNN architectures in our experiments, making it practical to use the dynamic programming algorithm to minimize data transfer cost.

6 IMPLEMENTATION

ROC is implemented on top of FlexFlow (Jia et al., 2019), a distributed multi-GPU runtime for high-performance DNN training . We extended FlexFlow in the following aspects to support efficient GNN computation. First, we replaced the equal partitioning in FlexFlow with a fine-grained partitioning interface to support splitting tensors at arbitrary points. This extension is critical to efficient partitioning for GNNs. Second, we have added a graph propagation engine to support neighborhood aggregation operations in GNNs, such as the `gather` operation in Figure 3. We have reused

Algorithm 1 A recursive dynamic programming algorithm for computing minimum data transfers. IN and OUT return the input and output tensors of an operation, respectively, and size($\mathcal{T}$) returns the memory space required to save tensors in $\mathcal{T}$.

Input: An input graph g, GNN architecture $\mathcal{G}$, and GPU memory capacity *cap*.
Output: Minimum data transfers to compute $\mathcal{G}$ on g within capacity *cap*.
▷ $\mathcal{D}$ is a database storing all computed COST functions.
function COST($\mathcal{S}, \mathcal{T}$)
  if $(\mathcal{S}, \mathcal{T}) \in \mathcal{D}$ **then**
    return $\mathcal{D}(\mathcal{S}, \mathcal{T})$
  if $\mathcal{S}$ is $\varnothing$ **then**
    return size($\mathcal{T}$)
  $cost \leftarrow \infty$
  for $o_i \in \mathcal{S}$ **do**
    if $(\mathcal{S} \setminus o_i)$ is a valid state **then**
      $\mathcal{S}' \leftarrow \mathcal{S} \setminus o_i$
      $\mathcal{T}' \leftarrow \big(\mathcal{T} \setminus \text{OUT}(o_i)\big) \cap \mathcal{A}(\mathcal{S}')$
      $xfer \leftarrow \text{size}\big(\text{IN}(o_i) \setminus \mathcal{T}'\big)$
      if $\text{size}\big(\mathcal{T} \cup \text{IN}(o_i) \cup \text{OUT}(o_i)\big) \leq cap$ **then**
        $cost = \min\{cost, \text{COST}(\mathcal{S}', \mathcal{T}') + xfer\}$
  $\mathcal{D}(\mathcal{S}, \mathcal{T}) \leftarrow cost$
  return $\mathcal{D}(\mathcal{S}, \mathcal{T})$

the highly optimized CUDA kernels in Lux (Jia et al., 2017) to perform graph propagation, which allows ROC to directly benefit from all kernel-level optimizations in Lux.

7 EVALUATION

In this section, we aim to evaluate the following points:

- Can ROC achieve comparable training performance compared to state-of-the-art GNN frameworks on a single GPU?
- Can ROC improve the end-to-end performance of distributed GNN training and inference?
- Can we improve the model accuracy on existing datasets by using larger and more sophisticated GNNs?

7.1 Experimental Setup

GNN architectures. We use three real-world GNN architectures to evaluate ROC. GCN is a widely used graph convolutional network for semi-supervised learning on graph-structured data (Kipf & Welling, 2016). GIN is provably the most expressive GNN architecture for the Weisfeiler-Lehman graph isomorphism test (Xu et al., 2019). CommNet consists of multiple cooperating agents that learn to communicate amongst themselves before taking actions (Sukhbaatar et al., 2016).

Table 4. Datasets used in our evaluation.

Dataset	Vertex	Edge	Feature	Label
Pubmed	19,717	108,365	500	3
PPI	56,944	1,612,348	700	121
Reddit	232,965	114,848,857	602	41
Amazon	9,430,088	231,594,310	300	24

Datasets. Table 4 lists the GNN datasets used in our evaluation. Pubmed is a citation network dataset (Sen et al., 2008), containing sparse bag-of-words feature vectors for each document (i.e., vertex), and citation links between documents (i.e., edges). PPI contains a number of protein-protein interaction graphs, each of which represents a human tissue (Hamilton et al., 2017). Reddit is a dataset for online discussion forum, with each node being a post, and each edge being a comment between posts (Hamilton et al., 2017). Amazon is the product dataset from Amazon (He & McAuley, 2016). Each node is a product, and each edge represents also-viewed information between products. The task is to categorize a product using its description and also-viewed relations.

All experiments were performed on a GPU cluster with 4 compute nodes, each of which contains two Intel 10-core E5-2600 CPUs, 256GB DRAM, and four NVIDIA Tesla P100 GPUs. GPUs on the same node are connected with NVLink, and nodes are connected with 100GB/s EDR Infiniband.

For each training experiment, the ROC graph partitioner learned a new cost model by only using performance measurements obtained during the single experiment. For each inference experiment, the graph partitioner used the learned cost model from the training on the same dataset.

Unless otherwise stated, all experiments use the same training/validation/test splits as prior work (Hamilton et al., 2017; Kipf & Welling, 2016; He & McAuley, 2016). All training throughput and inference latency were measured by averaging 1,000 training iterations.

7.2 Single-GPU Results

First, we compare the end-to-end training performance of ROC with existing GNN frameworks on a single GPU. Due to the small device memory on a single GPU, we limited these experiments to graphs that can fit in a single GPU.

Figure 4 shows the results among TensorFlow, DGL, PyG, and ROC. We expected that ROC would be slightly slower than the other frameworks on a single GPU, since it writes the output tensors of each operation back to DRAM for distributed computation, while other frameworks keep all

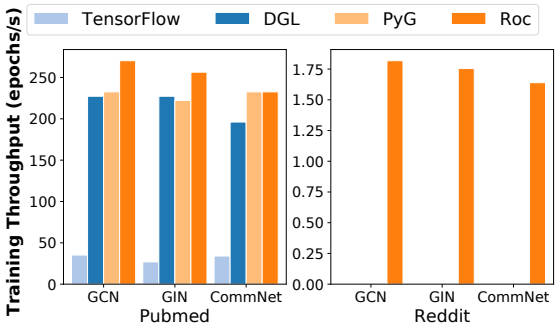


Figure 4. End-to-end training performance comparison between existing GNN frameworks and ROC on a single P100 GPU.

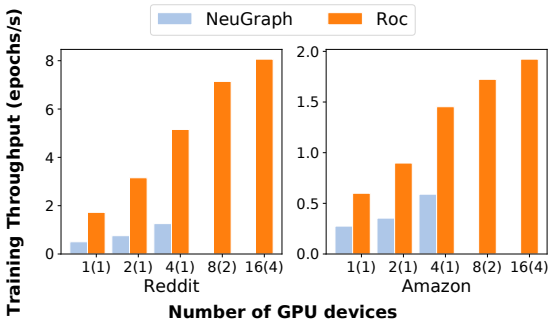


Figure 5. Training throughput comparison between NeuGraph and ROC using different numbers of GPUs. Numbers in parenthesis are the number of compute nodes used in the experiments.

tensors in a single GPU, and do not involve such data transfers. However, for these graphs, ROC reuses cached tensors on the GPU to minimize data transfers from DRAM to GPU, and overlaps the data transfers back to DRAM with subsequent GNN operations.

TensorFlow, DGL, and PyG were not able to run the Reddit dataset due to out-of-device-memory errors. ROC can still train Reddit on a single GPU, by using DRAM to save the intermediate tensors.

7.3 Multi-GPU Results

Second, we compare the end-to-end training performance of ROC with NeuGraph. NeuGraph supports GNN training across multiple GPUs on a single machine.

A NeuGraph implementation is not yet available publicly, so we ran ROC using the same GPU version and software library versions cited in Ma et al. (2019) and directly compared with the performance numbers reported in the paper.

Figure 5 shows the results. For experiments on a single compute node, ROC outperforms NeuGraph by up to 5×. The speedup is mainly because of the graph partitioning and memory management optimizations that are not available in NeuGraph. First, NeuGraph uses the *equal vertex partitioning* strategy that equally distributes the vertices across multiple GPUs. Section 7.6 shows that the ROC graph partitioning improves training throughput by up to 1.4× compared to the equal vertex partitioning strategy. Second, NeuGraph uses a streaming processing approach that partitions each GNN operation into multiple chunks, and sequentially streams each chunk along with its input data to GPUs. Therefore, it does not consider the memory management optimization as ROC, and Section 7.7 shows that the ROC memory manager improves training throughput by up to 2×.

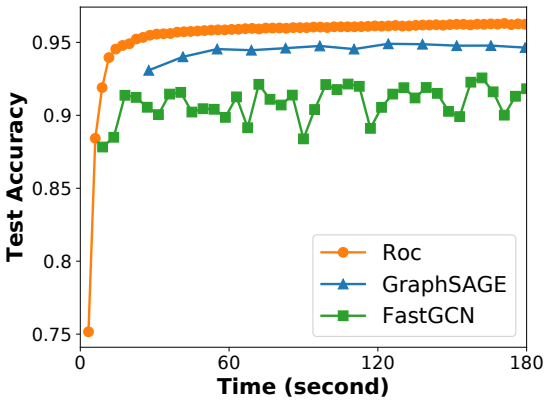


Figure 6. Time-to-accuracy comparison between state-of-the-art sampling techniques and ROC on the Reddit dataset (Hamilton et al., 2017). All experiments ran the same GCN model. ROC performed full-batch training on the entire graph, while GraphSAGE and FastGCN performed mini-batch sampling. For GraphSAGE and FastGCN, each dot indicates one training epoch, while for ROC, each dot represents five epochs.

The remaining performance improvement is likely due to other implementation optimizations in ROC, such as the use of the Lux high optimized GPU kernels for fast neighborhood aggregations. However, we were not able to further investigate the performance difference due the absence of a publicly available implementation of NeuGraph.

7.4 Comparison with Graph Sampling

We compare the training performance of ROC with state-of-the-art graph sampling approaches on the Reddit dataset. All frameworks use the same GCN model (Kipf & Welling, 2016). ROC performs full-batch training on the entire graph by following Kipf & Welling (2016), while GraphSAGE and FastGCN uses mini-batch sampling with a batch-size of 512.

Figure 6 shows the time-to-accuracy comparison on a single P100 GPU, where x-axis reports the end-to-end training time for each epoch, and y-axis shows the test accuracy of

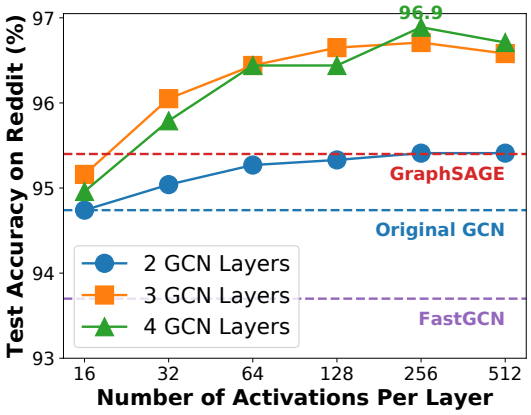


Figure 7. Test accuracy on the Reddit dataset using deeper and larger GNN architectures. The dotted lines show the best test accuracy achieved by GraphSAGE (95.4%), FastGCN (93.7%), and the original GCN architecture (94.7%), respectively.

the current model at the end of each epoch. For GraphSAGE and FastGCN, each dot indicates one training epoch, while for ROC each dot represents five training epochs for simplicity. Note that GraphSAGE and FastGCN can achieve relatively high accuracy within a few training epochs. For example, GraphSAGE achieves 93.4% test accuracy in two epochs. However, ROC requires around 20 epochs to achieve the same test accuracy. This is because ROC uses full-batch training following Kipf & Welling (2016), and only updates parameters once per epoch, while existing sampling approaches generally performs mini-batch training and have more frequent parameter updates. Even though ROC uses more epochs, it is still as fast or faster than GraphSAGE and FastGCN to a given level of accuracy.

7.5 Deeper and Larger GNN Architectures

ROC enables the exploration of larger and more sophisticated GNN architectures for large-scale graphs. To demonstrate, we consider a class of deep GNN architectures formed by stacking multiple GCN layers (Kipf & Welling, 2016). We use residual connections (He et al., 2016) between different GCN layers to facilitate training of deeper GNN architectures by allowing the model to preserve information learned from previous layers.

Formally, each layer of our GNN architecture is defined as follows.

$$H^{(k+1)} = \begin{cases} GCN(H^{(k)}) + H^{(k)} & d(H^{(k+1)}) = d(H^{(k)}) \\ GCN(H^{(k)}) + WH^{(k)} & d(H^{(k+1)}) \neq d(H^{(k)}) \end{cases}$$

where *GCN* is the original GCN layer (Kipf & Welling, 2016), and $d(\cdot)$ returns the number of activations in the input tensor. When $H^{(k)}$ and $H^{(k+1)}$ have the same number of activations, we directly insert a residual connection between the two layers. When $H^{(k)}$ and $H^{(k+1)}$ have different numbers of activations, we use a linear operation to transform $H^{(k)}$ to the expected shape. This design allows us to add residual connections for all GCN layers.

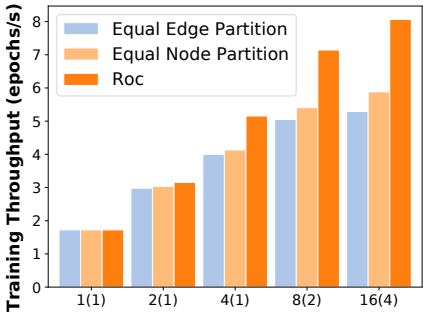


Figure 8. Training throughput comparison among different graph partitioning strategies on the Reddit dataset. Numbers in parenthesis are the number of compute nodes used in the experiments.

We increase the *depth* (i.e., number of GCN layers) and *width* (i.e., number of activations per layer) to obtain larger and deeper architectures beyond the commonly used 2-layer GNNs. Figure 7 shows the accuracy achieved by our GNN architectures on the Reddit dataset. The figure shows that improved accuracy can be obtained by increasing the depth and width of a GNN architecture. As a result, our GNN architectures achieve up to 96.9% test accuracy on the Reddit dataset, outperforming state-of-the-art sampling techniques results by 1.5%.

7.6 Graph Partitioning

To evaluate the learning-based graph partitioner in ROC, we compare the performance of the graph partitioning achieved by ROC with (1) *equal vertex partitioning* and (2) *equal edge partitioning*. (1) is used in NeuGraph to parallelize GNN training, and (2) has been widely used in existing graph processing systems.

Figure 8 shows the training performance comparison on different sets of GPUs. Neither of these baseline strategies perform as well as the ROC linear regression-based partitioner.

To evaluate the distributed inference performance on new graphs not used during training, we used the PPI dataset containing 24 protein graphs. Following prior work (Hamilton et al., 2017), we trained the GIN architecture on 20 graphs, and measured the inference latency on the remaining four graphs, by using the graph partitioner learned during training.

Figure 9 shows that the the learned cost model enables the graph partitioner to discover efficient partitioning on new

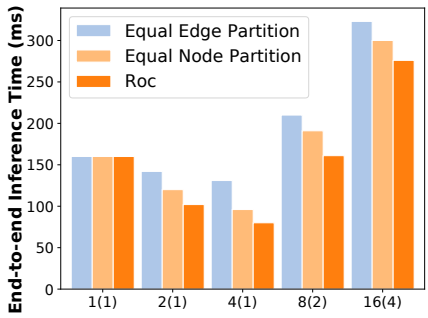
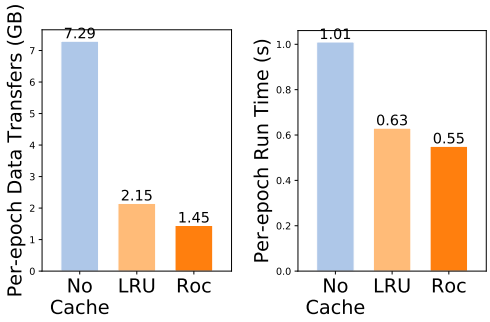


Figure 9. End-to-end inference time for the test graphs in the PPI dataset. The numbers were measured by averaging the inference time of the four test graphs.



(a) Data transfers. (b) Training time.

Figure 10. Performance comparison among different memory management strategies. All numbers are measured by training GCN on the Reddit dataset on a single GPU.

graphs for inference services, by reducing the inference latency by up to 1.2×. For the PPI graphs, the distributed inference across multiple compute nodes achieves worse performance than the inference on a single node. This is due to the small sizes of the inference graphs.

7.7 Memory Management

We now evaluate the performance of the ROC memory manager by comparing it with (1) the streaming processing approach in NeuGraph that streams input data along with computation (i.e., no caching optimization) and (2) the *least-recently-used* (LRU) cache replacement policy.

Figure 10 shows the comparison results for training GCN on the Reddit dataset on a single GPU. The dynamic programming-based memory manager reduces the data transfers between GPU and DRAM by 1.4-5× and reduces the per-epoch training time by 1.2-2×.

8 CONCLUSION

ROC is a distributed multi-GPU framework for fast GNN training and inference. ROC partitions an input graph onto multiple GPUs on multiple machines using an online-linear-regression-based strategy to achieve load balance, and coordinates optimized data transfers between GPU devices and host CPU memories with a dynamic programming algorithm. ROC increases the performance by up to 4.6× over existing GNN frameworks, and offers better scalability. The ability to process larger graphs and GNN architectures additionally enables model accuracy improvements. We achieve new state-of-the-art classification accuracy on the Reddit dataset by using significantly deeper and larger GNN architectures.

REFERENCES

Deep Graph Library: towards efficient and scalable deep learning on graphs. https://www.dgl.ai/, 2018.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016.

Caffe2. A New Lightweight, Modular, and Scalable Deep Learning Framework. https://caffe2.ai, 2016.

Chen, J., Ma, T., and Xiao, C. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations*, 2018.

Fey, M. and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, 2014.

Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems 30*. 2017.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2016.

He, R. and McAuley, J. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16. International World Wide Web Conferences Steering Committee, 2016.

Jia, Z., Kwon, Y., Shipman, G., McCormick, P., Erez, M., and Aiken, A. A distributed multi-gpu system for fast graph processing. *Proc. VLDB Endow.*, 11(3), November 2017.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., and Dai, Y. Neugraph: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 2019.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, 2010.

PyTorch. Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org, 2017.

Sen, P., Namata, G., Bilgic, M., Getoor, L., Galligher, B., and Eliassi-Rad, T. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

Sukhbaatar, S., szlam, a., and Fergus, R. Learning multiagent communication with backpropagation. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 29*. Curran Associates, Inc., 2016.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. *International Conference on Learning Representations*, 2018.

Venkataraman, S., Bodzsar, E., Roy, I., AuYoung, A., and Schreiber, R. S. Presto: Distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, 2013.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

Yang, H. Aligraph: A comprehensive graph neural network platform. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining - KDD 19*, 2019. doi: 10.1145/3292500.3340404. URL http://dx.doi.org/10.1145/3292500.3340404.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pp. 974–983, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5552-0. doi: 10.1145/3219819.3219890. URL http://doi.acm.org/10.1145/3219819.3219890.

Zhu, X., Chen, W., Zheng, W., and Ma, X. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.