NeuGraph: Parallel Deep Neural Network Computation on Large Graphs

Lingxiao Ma and Zhi Yang, ***Peking University;*** **Youshan Miao, Jilong Xue, Ming Wu, and Lidong Zhou,** ***Microsoft Research;*** **Yafei Dai,** ***Peking University***

https://www.usenix.org/conference/atc19/presentation/ma

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

NeuGraph: Parallel Deep Neural Network Computation on Large Graphs

Lingxiao Ma[†*], Zhi Yang[†*]
Peking University

Youshan Miao
Microsoft Research

Jilong Xue
Microsoft Research

Ming Wu
Microsoft Research

Lidong Zhou
Microsoft Research

Yafei Dai
Peking University

Abstract

Recent deep learning models have moved beyond low dimensional regular grids such as image, video, and speech, to high-dimensional graph-structured data, such as social networks, e-commerce user-item graphs, and knowledge graphs. This evolution has led to large graph-based neural network models that go beyond what existing deep learning frameworks or graph computing systems are designed for. We present NeuGraph, a new framework that bridges the graph and dataflow models to support efficient and scalable parallel neural network computation on graphs. NeuGraph introduces graph computation optimizations into the management of data partitioning, scheduling, and parallelism in dataflow-based deep learning frameworks. Our evaluation shows that, on small graphs that can fit in a single GPU, NeuGraph outperforms state-of-the-art implementations by a significant margin, while scaling to large real-world graphs that none of the existing frameworks can handle directly with GPUs.

1 Introduction

Graphs are natural representations of many real-world data; examples include web graphs, social networks, e-commerce user-item graphs, and knowledge graphs. With a graph representation, graph-based learning tasks, such as vertex classification and link prediction, can be optimized effectively. There has been a recent surge of interest in extending neural network models to graph data [7, 8, 13, 17–19, 23, 25, 29, 37]. These methods, known as graph neural networks (GNNs), combine standard neural networks with iterative graph propagation: the property of a vertex is computed recursively (with neural networks) from the properties of its neighbor vertices.

However, neither the existing deep learning frameworks nor the existing graph systems could support GNN algorithms sufficiently. The lack of system support has seriously limited the ability to explore the full potentials of GNNs at scale. Deep learning (DL) frameworks such as TensorFlow [4], PyTorch [2], MXNet [12], and CNTK [50] are designed to express deep neural networks (DNNs) but do not naturally express and efficiently execute graph propagation models. Deep graph library (DGL) [1] supports programming GNNs by wrapping DL systems with a graph-oriented message-passing interface. While DGL addresses the expressiveness challenge, it does not yet explore deeply the opportunities to leverage graph-aware operations for efficient executions. Furthermore, none of these frameworks, including DGL, offer the needed scalability to handle large graphs: The highly connected nature of graphs means that graph propagation could easily involve a large portion of a large graph, especially for power-law or dense graphs. Processing even a single vertex requires that deep learning frameworks load a large amount of graph-related data (e.g., structure and feature data) into limited GPU memory.

With the vertex-program abstraction and graph-specific optimizations, existing graph processing systems [10, 15, 26, 28, 47] can naturally express iterative graph algorithms like PageRank and community detection, and scale them to graphs with billions of vertices and edges. But graph systems can hardly express neural networks (NNs) and lack key capabilities required by efficient DNN executions, such as the tensor abstraction, automatic differentiation and dataflow programming model.

We therefore advocate bridging deep learning systems and graph processing systems to enable a new framework for scalable GNN training. In this paper, we explore the design of a GNN processing framework on top of dataflow-based DL systems. We argue that by introducing the graph model to dataflow and recasting graph-specific optimizations as dataflow optimizations, we can enable the DL frameworks to support efficient and scalable DNN computation on graphs. To support this argument, we developed NeuGraph, an efficient GNN processing framework built on top of an existing dataflow engine.

[†] National Engineering Laboratory for Big Data Analysis and Applications, Center for Data Science, Peking University.

[*] Lingxiao Ma and Zhi Yang equally contributed to this work.

The work is done when Lingxiao Ma is an intern and Zhi Yang is a visiting researcher at Microsoft Research.

NeuGraph combines the dataflow abstraction with the vertex-program abstraction in a new programming model called SAGA-NN (Scatter-ApplyEdge-Gather-ApplyVertex with Neural Networks). SAGA can be considered as a variant of graph-parallel abstraction (e.g., GAS [26]). Unlike a traditional system where user-defined functions (UDFs) express vertex programs, UDFs in SAGA-NN express NN computation on tensors as vertex or edge data, e.g., vertex or edge data. With the new programming model, NeuGraph allows users to express a GNN algorithm without worrying about the underlying system implementation (e.g., GPU memory management or scheduling). The graph-aware dataflow engine in NeuGraph judiciously partitions the graph data (vertex and edge data) into *chunks* (subgraphs), constructs the dataflow that operates at the chunk granularity, and schedules parallel executions of the dataflow on GPUs.

Naively adapting optimizations developed in the context of graph processing systems can lead to inefficient dataflow executions on DL frameworks. NeuGraph achieves high efficiency by introducing a range of optimizations both in the scheduling of parallel chunk processing, as well as the execution of core graph propagation procedures (i.e., Scatter-ApplyEdge-Gather stages) over the often-sparse graph structure. With fine-grained graph partitioning, NeuGraph achieves efficient *selective scheduling* and *pipeline scheduling* on top of the dataflow, to hide data movement between GPU and host when scaling a model out of the GPU core. To continue performance scaling, NeuGraph further adopts a new topology-aware scheduling strategy to efficiently distribute GNN models over modern multi-GPU systems. Finally, NeuGraph introduces computation-related optimizations for graph propagation, which is often hard to accelerate using GPUs.

We implemented NeuGraph on top of TensorFlow. We show that NeuGraph can support a variety of GNN algorithms on large graphs with millions of vertices and hundreds of millions of edges, as well as hundreds of feature dimensions over vertices, which existing DL frameworks cannot directly handle with GPUs. Compared on large graphs that TensorFlow can handle only with CPUs, NeuGraph achieves $16 \sim 47\times$ speedups. Even on small graphs that can fit into a GPU's memory, NeuGraph can still achieve a up to $5\times$ speedup over the state-of-art implementation on TensorFlow and a up to $19\times$ speedup over DGL [1]. Moreover, NeuGraph achieves nearly linear scalability over multiple GPUs.

As one of our key contributions, NeuGraph bridges two largely parallel threads of research, graph processing systems and dataflow-based DL frameworks, in the new GNN setting. NeuGraph significantly expands the capabilities of existing DL frameworks to support GNNs in the following key dimensions: programming model, graph partition and dataflow translation, graph propagation operations, and execution scheduling. We have also demonstrated, through extensive evaluation on real graphs with typical GNNs, significant benefits in scalability and efficiency by connecting graph processing and DL frameworks.

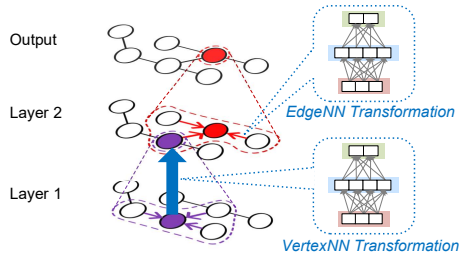

Figure 1: Feed-forward computation of a 2-layer GNN.

The rest of the paper is organized as follows. Section 2 introduces the SAGA-NN programming abstraction. Section 3 describes the optimizations in the NeuGraph system. Section 4 discusses the implementation and Section 5 presents our experimental results. We discuss related work in Section 6 and conclude in Section 7.

2 NeuGraph Programming Abstraction

In this section, we first reveal the essential structure of graph neural networks, and then propose our programming model that combines graph-parallel and dataflow abstractions.

2.1 Graph Neural Networks

Deep learning, in the form of deep neural networks, is a class of machine learning algorithms that use a cascade of multiple layers of nonlinear processing units for feature extraction and transformation. Each successive layer uses the output from the previous layer as input. Deep learning has been gaining popularity due to its success in areas such as speech, vision, and natural language processing. In these areas, the coordinates of the underlying data representation often have a regular grid structure, which is friendly to hardware accelerators (e.g., GPU) with massive SIMD-style parallelisms.

Graph neural networks are deep learning based methods that operate neural networks on graph data, and have been adopted for many applications due to convincing in terms of model accuracy. Recently, several surveys [5, 46, 52, 54] provided a thorough review of different graph neural network models as well as a systematic taxonomy of the applications. A majority of GNN models can be categorized into *graph convolutional networks* [7, 9, 13, 19, 23], *graph recursive networks* [25, 33], and *graph attention networks* [43, 51].

We discuss 3 representative categories of GNNs with 3 representative models: (1) GCN [23] is a graph convolutional network that generalizes the notion of the convolution operation, typically for image datasets, and applies it to an arbitrary graph (e.g., a knowledge graph). GCN has been widely used in real-world scenarios like recommendation [6, 49]. Initially, each vertex in the graph has a feature vector. First, each vertex collects its neighbor vertices' feature vectors along edges, and

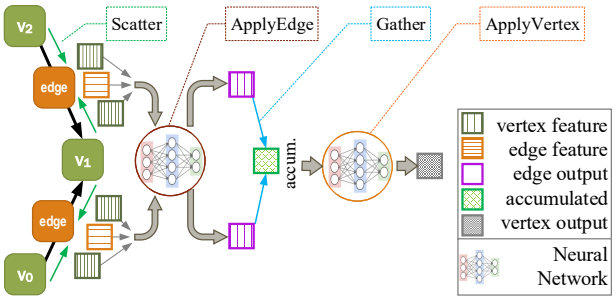

Figure 2: SAGA-NN stages for each layer of GNN.

sums the collected vectors (weighted by edge values). Then, a fully-connected NN is used to compute the vertex feature vector as the output. This is a layer of GCN. Stacking multiple GCN layers makes the vertex features representative enough for tasks. Taking the recommendation system as an example, a bipartite graph is constructed from the user-item ratings: There will be an edge with the rating as the edge value between the user vertex and the item vertex if a user rates an item. Then, the embeddings of both users and items can be learned by the GCN from the graph and the features of users and items. Finally, these embeddings are used to predict the missing user-item ratings to make a recommendation. (2) GG-NN [25] is a graph recursive network. It has an architecture similar to GCN, but uses different parameters for different edge types, as well as a Gated Recurrent Unit (GRU) in the NN to process accumulated features. (3) As a graph attention network, GAT [43] differs GCN mainly in that it computes an *attention* value for each edge during transferring vertex features.

In general, these GNN models share the same basic idea of collectively aggregating information following the graph structure. Specifically, each vertex or edge in the graph can be associated with a set of tensor data (normally a vector) as its feature or embedding. A GNN can consist of multiple layers, with an iterative propagation procedure conducted layer-by-layer over the same graph, as illustrated in Figure 1. At each layer, the vertex or edge features are transformed and propagated along edges, and then aggregated at the target vertices to produce new features for the next layer. Different from traditional graph algorithms (e.g., PageRank), the transformation on either vertices or edges can be arbitrary DNN computation. The GNN may also contain a label for each vertex, each edge, or the entire graph, for computing a loss function at the top layer. A feed-forward computation is then performed from the bottom layer to the top, with back-propagation conducted reversely.

Comparing with DNNs, the complexity due to graphs in GNNs creates a significant scalability challenge. First, real-world graphs, such as social networks or e-commerce networks, can easily have millions of nodes and edges. Second, vertices and edges in the graph are interconnected and need to be modeled as a whole neural network (i.e., a large, sparse neural network architecture defined according to a graph structure). This is particularly challenging on GPUs given the limited GPU memory capacity. Finally, unlike image, audio, or text that have clear grid structures, graph data are irregular, making it hard to conduct parallel GNN computation efficiently on GPUs.

2.2 A Running Example

We take the Gated Graph ConvNet (G-GCN) algorithm [7,29] as a concrete running example (see Example 2.1). G-GCN incorporates the gating mechanism into graph convolution. This model can be used to extract vertex features for community detection.

Example 2.1. *Let $\mathbf{h}_u^\ell$ denote the feature vector of a vertex u at layer ℓ, and W^ℓ, W_H^ℓ, and W_C^ℓ be the weight parameters to learn. G-GCN recursively defines the feature of a vertex u as follows:*

$$\mathbf{h}_u^{\ell+1} = ReLU\left(W^\ell \otimes \left(\sum_{v\to u} \eta_{vu} \odot \mathbf{h}_v^\ell\right)\right) \quad (1)$$

where $\otimes$ refers to matrix multiplication, $\odot$ refers to element-wise multiplication, and η_{vu} (for each edge $v \to u$) acts as edge gate,

$$\eta_{vu} = sigmoid\left(W_H^\ell \otimes \mathbf{h}_u^\ell + W_C^\ell \otimes \mathbf{h}_v^\ell\right) \quad (2)$$

where ReLU and sigmoid are nonlinear activation functions in neural networks.

G-GCN can be mapped to the pattern of computing a layer in Figure 1: Equation 2 represents the EdgeNN to compute the edge weight. $\sum_{v\to u} \eta_{vu} \odot \mathbf{h}_v^\ell$ in Equation 1 collects features from neighbors, and ReLU$\left(W^\ell \otimes \cdots\right)$ in Equation 1 is the VertexNN to process the accumulated features.

2.3 SAGA-NN Model

Based on the common pattern observed in GNN models, we propose SAGA-NN (Scatter-ApplyEdge-Gather-ApplyVertex with Neural Networks) as a new programming model for GNNs. It combines dataflow and vertex-program to express the recursive parallel computation at a layer of a GNN. SAGA-NN splits the feed-forward computation into four stages: *Scatter*, *ApplyEdge*, *Gather*, and *ApplyVertex*, as illustrated in Figure 2.

SAGA-NN provides two user-defined functions (UDFs) for ApplyEdge and ApplyVertex respectively, for users to declare neural network computations on edges and vertices. The **ApplyEdge** function defines the computation on each edge, which takes `edge` and `p` as input, where `edge` refers to the edge data and `p` contains the learnable parameters of the GNN model. Each `edge` is a tuple of tensors [`src`, `dest`, `data`] representing the associated data of the source and destination vertices connected by the edge, as well as the edge associated data (e.g., edge weight). This function can be used to apply a

```
G-GCN(vertex^ℓ):  // computing vertex^{ℓ+1}
  params p = [W_H^ℓ  W_C^ℓ  W^ℓ]
  // Passing data over edges
  edge^ℓ=Scatter(vertex^ℓ)
  // edge-parallel computation
  acc = ApplyEdge(edge^ℓ, p):
    η = sigmoid(p.W_H^ℓ ⊗ edge^ℓ.dest + p.W_C^ℓ ⊗ edge^ℓ.src)
    return η ⊙ edge^ℓ.src
  set Gather.accumulator = sum
  accum = Gather(acc)
  // compute new vertex data
  vertex^{ℓ+1} = ApplyVertex(vertex^ℓ, accum, p):
    return ReLU(p.W^ℓ ⊗ accum)
  return vertex^{ℓ+1}
```

Figure 3: Gated Graph ConvNet at layer ℓ in *SAGA-NN* model.

neural network model on `edge` and `p`, and outputs an intermediate tensor data associated with the edge. The **ApplyVertex** function defines the computation on a vertex, which takes as input a vertex tensor data `vertex`, the vertex aggregation `accum`, and learnable parameters `p`, and returns the new vertex data after applying a neural network model. The SAGA-NN abstraction builds on a dataflow framework, so users can symbolically define the dataflow graphs in UDFs by connecting mathematical operations (e.g., *add*, *tanh*, *sigmoid*, *matmul*) provided by the underlying framework.

The other two stages, **Scatter** and **Gather**, perform data propagation and prepare data collections to be fed to ApplyEdge and ApplyVertex as input. They are triggered and conducted by the system implicitly. We chose not to expose UDFs for Scatter and Gather, because these functions, if provided, are highly coupled with the propagation procedure, whose computations flow through the irregular graph structure and are difficult to express as dataflow that NeuGraph optimizes—users would have to implement the corresponding derivative functions of the UDFs, a serious burden. Following the same principle, NeuGraph also avoids exposing user-defined aggregation methods. It provides a set of default ones instead, including *sum*, *max* (e.g., *max-pooling* operator [18]), and concatenation, which can be chosen by setting **Gather**.`accumulator`.

NeuGraph models a GNN as a sequence of SAGA stages. The *Scatter* passes the vertex data `vertex` onto its adjacent edges to construct edge data `edge`, including both the source and destination vertex data. The subsequent *ApplyEdge* then invokes a parallel computation defined by the UDF on the edge data to produce an intermediate tensor value for each edge as its outputs. The *Gather* then propagates those outputs along the edges and aggregates them at the destination vertices through commutative and associative accumulate operations. Finally, the *ApplyVertex* executes the computation defined in UDF on all vertices to produce updated vertex data for the next layer. The procedure in Figure 1 fits in the SAGA-NN model: The *ApplyEdge* and *ApplyVertex* represent the EdgeNN and VertexNN, respectively; the *Scatter* and *Gather* perform the propagation along edges. This mapping indicates that the GNNs following the procedure in Figure 1 could be implemented with SAGA-NN model, hence presents the generality of SAGA-NN.

Figure 3 illustrates the description of G-GCN (at layer l) in the SAGA-NN model. Scatter gives each edge $v \rightarrow u$ with vertices data $[\mathbf{h}_v^\ell, \mathbf{h}_u^\ell]$, and ApplyEdge computes per-edge update $\texttt{acc}_{vu} = \eta_{vu} \odot \mathbf{h}_v^\ell = \texttt{sigmoid}\left(W_H^\ell \otimes \mathbf{h}_u^\ell + W_C^\ell \otimes \mathbf{h}_v^\ell\right) \odot \mathbf{h}_v^\ell$. Next, Gather performs $\texttt{accum}_u = \sum_{v:v\rightarrow u} \texttt{acc}_{vu}$, and ApplyVertex computes $\mathbf{h}_u^{\ell+1} = \texttt{ReLU}\left(W^\ell \otimes \texttt{accum}\right)$.

The dataflow abstraction makes it easy to express neural network architectures and leverage auto-differentiation. With the dataflow abstraction in SAGA-NN, NeuGraph enjoys the flexibility of executing operations on vertices or edges in batch for increasing efficiency. The vertex-program in SAGA-NN allows users to express computations naturally by thinking like a vertex, and models common patterns in GNNs as well-defined stages, thereby enabling optimizing in both graph computation and dataflow scheduling.

3 NeuGraph System

NeuGraph provides a combination of the dataflow and vertex-program abstractions as the user interface. Under this abstraction, NeuGraph proposes graph-aware optimizations for GNN processing to achieve efficiency and scalability.

At a high level, NeuGraph consists of: 1) a translation engine that translates GNN expressed by the SAGA-NN model into a dataflow graph at chunk-granularity to enable GNN computation over large graphs in GPUs; 2) a streaming scheduler that minimizes data movement across the host and GPU memory and maximizes its overlap with computation. The scheduler also needs to be topology-aware for use of multiple GPUs; 3) a graph propagation engine for deep learning that employs a set of fast propagation kernels and fuses operations to remove redundant memory copies; 4) a dataflow execution runtime. NeuGraph requires no modifications to existing dataflow-based DL frameworks, offering a general method to combine graph and NN computation within existing DL frameworks. In this section, we focus on the first three design points as they are main contributions of NeuGraph.

3.1 Graph-Aware Dataflow Translation

Just as with DNNs, efficient use of GPUs is critical to the performance of GNNs, especially for large graphs. However, existing DL frameworks cannot handle large graphs directly on a GPU because graph data cannot fit into GPU memory.

To achieve scalability beyond the physical limitation of GPU memory, NeuGraph introduces graph-specific partitioning on top of the dataflow abstraction. Note that both vertex feature data and graph structure data can be large. NeuGraph thus applies a 2D graph partitioning: As illustrated in Figure 4, it slices vertex data into P equally-sized disjoint vertex chunks, and tiles the adjacency matrix (representing edges)

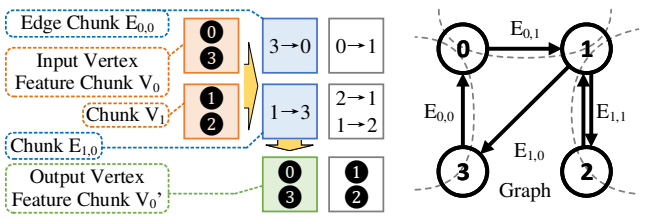

Figure 4: 2D Partitioning of a graph, here $P = 2$.

into $P \times P$ edge chunks. Edges in an edge chunk E_{ij} connect vertices in two vertex chunks V_i and V_j , respectively. By splitting graph data into chunks, NeuGraph can process edge chunks one by one, with only the source and destination vertex chunks needed for the edge chunk being processed. To achieve this, NeuGraph generates a dataflow graph with operators on data chunks, each of which fits in GPU memory, as illustrated in Figure 5.

For the forward computation at a layer, NeuGraph translates a dataflow subgraph for each destination vertex interval (e.g., a column in Figure 4): The Scatter operator inputs a specific edge chunk, i.e., the edge chunk in the i-th row and j-th column, and the associated i-th and j-th vertex chunks, and outputs an edge data chunk containing tuples in the form of [`src`, `dest`, `data`]. Each edge data chunk can be processed by operators specified in the ApplyEdge UDF to produce another edge data chunk with the result data `acc` (as in Figure 3). The operators at the Gather stage accumulate each edge's data based on its destination vertex to generate the corresponding vertex accumulation data chunk. After the processing of all the edge chunks for a destination vertex interval is done, the operators specified in the ApplyVertex UDF process the vertex accumulation chunks and output new vertex data chunks for the next layer.

For back-propagation, as the UDFs for ApplyEdge and ApplyVertex are expressed as dataflow computations over regular tensors, NeuGraph can leverage auto-differentiation provided by the DL frameworks. Additionally, NeuGraph further provides the *backward*-Gather operator to distribute the accumulation gradient returned by the *backward*-ApplyVertex stage across edges, and the *backward*-Scatter operator to accumulate all the partial gradients returned by the *backward*-ApplyEdge stage for a vertex in the previous layer.

Note that it is not necessary to enforce strict global barriers between stages in the SAGA-NN model. NeuGraph can flexibly schedule the chunk-based operators simply based on the data dependencies described in the dataflow graph. The system maintains the working set of operators within GPU memory by employing explicit device-to-host (D2H) and host-to-device (H2D) operators to conduct data swapping between the host and GPU memory. Also, during a training process, some intermediate feature data (e.g., the result of matrix multiplication in the ApplyEdge stage as in Figure 5) relevant to vertex chunks or edge chunks will be used in back-

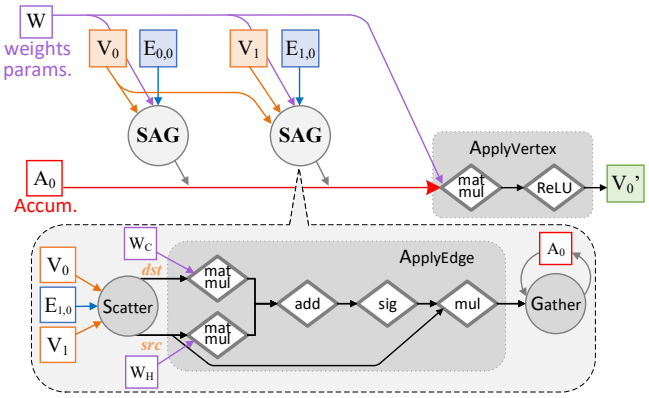

Figure 5: Chunk-based dataflow graph for a destination interval V_0 at a G-GCN layer. The backward dataflow graph and the swapping of intermediate results to host memory for backward are omitted for a clear visualization.

propagation. To save GPU memory, they are swapped out to host memory during the feed-forward computation and swapped back in during the back-propagation.

Discussion. The source vertex determines the row of the edge chunk and the destination vertex determines the column of the edge chunk. For every GNN layer, edge processing can be done in either a row-oriented or a column-oriented manner, based on the update pattern. For the forward computation, data flows from the source vertex to the destination vertex. With this pattern, row-oriented processing loses the opportunity of reusing the accumulated vertex data chunks, whose total size can be larger than the size of GPU memory. NeuGraph therefore adopts a column-oriented approach as illustrated in Figure 5, where it continuously executes operators in the Scatter-ApplyEdge-Gather (SAG) stages for V_0 and V_1 to produce A_0, which is subsequentially consumed by operators in the ApplyVertex stage. The destination vertex chunk and the corresponding accumulated vertex data chunk (e.g., A_0 in the figure) can be reused in GPU memory when NeuGraph processes edge chunks in the same column, so that data movement can be minimized.

By contrast, for the backward computation, a vertex gradient is propagated from the destination vertex to the source vertex. In this case, row-oriented processing is preferred. The vertex gradient data chunk can be reused from GPU memory when NeuGraph processes edge chunks in the same row. In the rest of this section, we focus on the discussion of the forward-pass execution of chunk-based dataflow, the backward-pass execution is done in a similar manner.

Besides the chunk processing order, determining the number of vertex chunks P is also important. Assuming edge chunks are accessed in the column-oriented manner in the forward pass, each edge chunk is accessed once, and each source vertex chunk is loaded P times. Thus, a smaller P is preferred to reduce I/O. NeuGraph selects P as the minimum integer to fit each chunk in GPU memory. Given a chunk-size choice

and the scheduling plan of the dataflow graph, NeuGraph computes the GPU memory requirement of the execution. If this requirement is beyond GPU's capacity, NeuGraph shrinks the chunk size by increasing P.

3.2 Streaming Processing out of GPU Core

For each layer, NeuGraph can scale GNN computation beyond the GPU core by processing the dataflow subgraph for a column of edge chunks (illustrated in Figure 5) in a column-by-column way. As we show later in the experiments (Table 2), the CPU-GPU data transfer has a significant impact on the overall performance, especially for sparse graphs. NeuGraph introduces a streaming scheduler with two innovations: selective scheduling that reduces data transfer on unnecessary vertices, and pipeline scheduling that maximizes the overlap between computation and data transfer.

Selective Scheduling. Unlike traditional graph algorithms (e.g., PageRank), the vertex data in GNNs can be much larger due to their high-dimensional feature vectors. To reduce the transfer cost of vertex chunks, NeuGraph exploits sparsity inherent in real-world graphs: To compute a specific edge chunk, not all vertices in the corresponding vertex chunks will be used due to the sparse graph structure (e.g., some vertices have no edges in this chunk). So, when processing an edge chunk E, NeuGraph applies a filter in CPU to select the useful vertices from E's source vertex chunk, and only transfers the selected vertex data into GPU.

We notice that a random graph partition (e.g., a permutation of the vertices) makes selective scheduling inefficient. Therefore, NeuGraph adopts a locality-aware graph partitioning algorithm (e.g., Kernighan-Lin algorithm) to condense as many edges that are connected to the same vertex as possible into one chunk (e.g., a diagonal one in the matrix of edge chunks). In this way, better access locality can be achieved for vertex data and hence more potential in selective scheduling.

Interestingly, when the majority of the vertices are useful (e.g., in a dense subgraph), directly transferring the full vertex chunk can be faster as it does not require additional memory copies for filtering. So for an edge chunk, we dynamically determine whether to apply the filtering in CPU based on the fraction θ of useful vertices. Given the host memory copy throughput T_{copy} on the CPU side, the filtering cost is $\frac{\theta}{T_{copy}}$. Let T_{trans} be the bulk transfer throughput from CPU to GPU. For a vertex chunk, if $\theta < \frac{T_{copy}}{T_{copy}+T_{trans}}$, NeuGraph chooses to apply filtering as it benefits the overall data transfer efficiency. Otherwise, NeuGraph skips the filtering and directly loads the entire vertex chunk into GPU.

Pipeline Scheduling. Besides the filtering optimization, NeuGraph further overlaps data transfer and computation through a pipeline scheduling to hide the transfer latency. Instead of streaming one edge chunk each time into GPU, NeuGraph can stream multiple chunks into the GPU device memory.

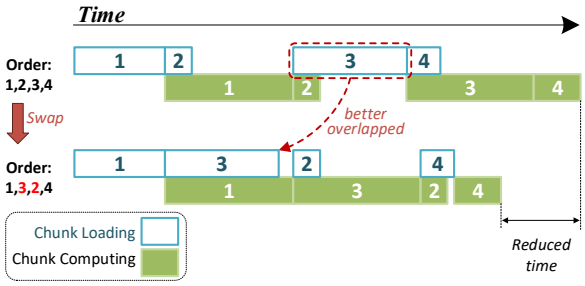

Figure 6: The swapping heuristic for a case of streaming two edge sub-chunks ($k = 2$).

In this case, a smaller chunk size can increase overlapping potential, which seems opposite to the requirement of a large chunk size to reduce vertex access I/O.

To deal with this dilemma, we apply the second-level partitioning over the edge grid to improve streaming efficiency without increasing the total I/O amount. Specifically, we horizontally partition an edge chunk and its associated source vertex chunk into k ($k \geq 2$) fine-grained sub-chunks, which enables parallel streaming processing of k sub-chunks. While performing computation on an edge sub-chunk, NeuGraph can simultaneously stream in other edge sub-chunks and their associated source vertex sub-chunks.

Recall that different edge sub-chunks could have distinct data transfer and computation cost due to different sparsity levels. NeuGraph carefully makes a scheduling plan for streaming heterogeneous sub-chunks. Given a column of edge sub-chunks, the system first generates the initial schedule plan by assigning a random order for processing. Next, it repeatedly swaps the order of a pair sub-chunks such that a better schedule plan with less time can be obtained. This process stops when it converges or reaches maximum iterations.

Then, NeuGraph exploits the cyclic pattern inherent in GNNs: Both the computation time and data transfer time of each sub-chunk can be profiled in the first several iterations and used in refining the scheduling plan for processing in the following iterations. Specifically, the system simulates the execution of the current schedule order based on the profiled execution information of individual sub-chunks. As illustrated in Figure 6, by examining the overlapping result in this simulation, the system finds a sub-chunk whose data transfer time is much shorter than the computation time, and within the same chunk, another sub-chunk is an opposite case. By swapping the order of these two heterogeneous edge sub-chunks, the system enables a better balance between the computation and data transfer.

3.3 Parallel Multi-GPU Processing

To improve scalability further, we can parallelize the training by partitioning the chunk-based dataflow (model parallelism) over multi-GPUs. Our dataflow graph is easy to parallelize due to its parallel nature, where GPUs can be assigned

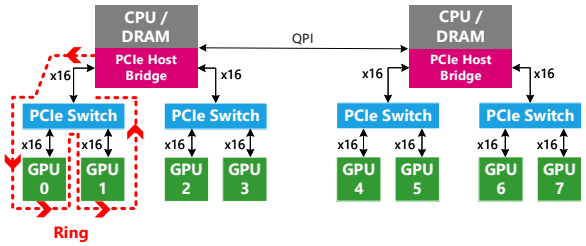

Figure 7: Multi-GPU architecture

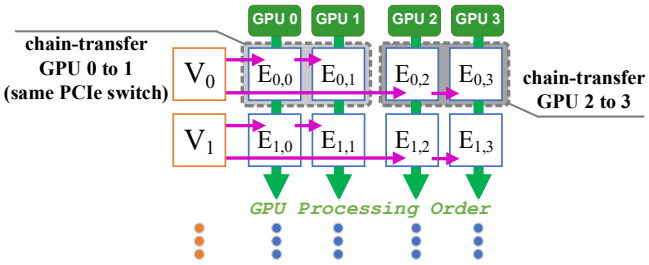

Figure 8: NeuGraph transfers vertex chunks along the chain.

dataflow subgraphs for different columns for cooperative processing.

However, with recent advances in hardware, modern multi-GPU systems introduce complex inter-connections among GPUs and across GPUs and CPUs, which presents new challenges to parallelize a dataflow graph. To illustrate this issue, Figure 7 shows the topology of a typical 8-GPU server, where GPUs are connected to CPU/DRAM (host memory) via a multi-level PCIe interface hierarchy. The upper level links that are shared by multiple communication paths can easily become a bottleneck. For example, GPUs 0 and 1 can only reach half of their peak bandwidth when reading edge/vertex data from host memory simultaneously, as limited by the link from the left-most PCIe switch to DRAM. Connecting the host to an accelerator like GPU via PCIe is the most common channel at present. We start from a common case, which may apply to other architectures.

To maximize the parallelism degree on multiple GPUs and prevent shared inter-connection links from becoming a bottleneck, NeuGraph employs a chain-based streaming scheduling scheme. Note that a vertex chunk is required by all the GPUs processing different columns of edge chunks. So, our idea is to let a GPU forward the vertex chunk (once loaded to its memory) to its neighbor GPU under the same PCIe switch, which can eliminate the bandwidth contention on the upper-level shared inter-connection link. NeuGraph therefore *logically* considers the GPUs under the same PCIe switch as a large *virtual GPU* and enables them to share data in a chain order as illustrated by the red dotted line in Figure 7.

In chain-based scheduling, each GPU streams one column of edge chunks and all vertex chunks to compute a destination vertex chunk. Note that the vertex data chunk for the destination interval can be initially loaded and cached in GPU memory. For simplicity, we assume that only the source vertex data is required for the computation. In particular, a GPU needs to take the following two operations: 1) loading an edge chunk from the host memory, and a data chunk from the host memory or from the device memory of its previous GPU in the chain, and 2) performing local computations. NeuGraph employs a coordinated scheduling to better overlap the two operations. As illustrated in Figure 8, we group GPUs into multiple virtual GPUs according to the inter-connection topology; e.g., GPUs 0 and 1 constitute one virtual GPU; GPUs 2 and 3 constitute another. Initially, GPUs 0 and 2 load vertex data chunk V_0 from the host memory. After loading, GPUs 0 and 2 start computing over chunk V_0, and also begin loading chunk V_1 from the host memory. Meanwhile, GPUs 1 and 3 start fetching chunk V_0 from GPUs 0 and 2, respectively. Next, GPUs 1 and 3 drop the data chunk V_0 after processing it locally as the chunk has already been consumed by all virtual GPUs. The whole process continues in such a pipelining fashion until all vertex data chunks have been loaded and processed.

In Section 3.2, we introduce the selective scheduling that can help reduce data movement between the host and GPU device memory. However, to apply selective scheduling in chain-based streaming, we need to select the useful vertex data required by the corresponding edge chunks in a virtual GPU; e.g., $E_{0,0}$ and $E_{0,1}$ in Figure 8. In a multi-GPU execution, we use the threshold $\theta = \frac{T_{copy}}{T_{copy}+T_{trans}}$ to determine whether or not to apply selective scheduling, where T_{copy} and T_{trans} are aggregative memory-copy and aggregative data-transfer throughput on both the CPU and GPU sides, respectively. Thus, given limited CPU resources shared by a large number of GPUs, NeuGraph applies selective scheduling on more sparse chunks with a larger θ.

3.4 Graph Propagation Engine

Besides ensuring high streaming efficiency, NeuGraph also introduces several important optimizations to reduce computation time in the execution of the Scatter-ApplyEdge-Gather (SAG) stages, which are not easily amenable to efficient GPU acceleration due to the often sparse edge structure of a graph.

First, NeuGraph incorporates a *dataflow graph optimization* to remove redundant computations in the SAG stage by considering the semantics of the SAGA-NN model. Consider the matrix multiplication operations in the ApplyEdge stage in Figure 5. These operations are conducted on vertex data that are scattered to a subset of edges and the learnable parameters W_C or W_H that are shared by all edges. Because a vertex may have multiple edges to which that the vertex data can be scattered, such a multiplication for a vertex can be conducted multiple times, leading to redundancies. NeuGraph therefore moves the computations that are related only to the source or destination vertices from the ApplyEdge stage of the current layer to the ApplyVertex stage of the previous layer.

Second, to support the Scatter and Gather stages efficient-

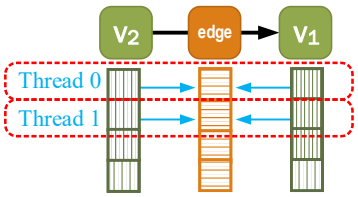

Figure 9: Parallelism along the dimension of feature vector.

ly on GPUs, NeuGraph provides *scatter/gather operation kernels* optimized for GPU executions. The design carefully considers the data structure layout to allow the kernel to better leverage the massive parallelism provided by GPU. In most GNNs, the data of each vertex is a dense vector rather than a scalar. We therefore exploit parallelism in per-vertex data access that fits better to GPU with SIMD architectures. Figure 9 illustrates the scatter kernel passing the vertex data, from both the source and the destination, onto an edge to form the edge data. We assign a thread block to process incoming edges with the same destination vertex. For vertices with a large in-degree, we divide the incoming edges into consecutive subgroups to be processed by multiple thread blocks. In a thread block, threads copy the source/destination vertex data into the edge data in parallel along the dimension of the vertex feature vector, ensuring good coalesced memory access. The gather kernel reduces the partial accumulation vectors `acc` from a set of edges that end at the same destination vertex `accum` into an accumulated vector. We employ a similar principle of exploiting parallelism for the scatter operator. A block of threads first cooperatively enumerate an edge group, accumulate the features of every edge into a temporary vector in GPU register, and finally write the result back to the corresponding destination vertex.

Finally, NeuGraph supports *Scatter-ApplyEdge-Gather (SAG) stage fusion* as another kernel optimization on execution of the propagation procedures. We find that, on most GNN applications, especially after the dataflow graph optimization, the ApplyEdge function only performs element-wise operations, such as +, -, ×, ÷, *tanh*, *sigmoid*, *ReLU*. In this case, we can optimize SAG stages by allowing the vertex/edge data to be directly updated with element-wise operations in GPU registers and then written back to their destination vertices in a single pass, without any extra cost of creating intermediate edge data in the GPU global memory. To achieve that, NeuGraph automatically detects such a case and replaces the whole SAG stages using a specially customized operation called *Fused-Gather*. This operation processes each edge chunk as follows: It first loads the inputs of Scatter; i.e., source vertices and edge data, into GPU registers, and then uses GPU threads to perform in-place updates directly on elements in registers based on user-defined element-wise operations in ApplyEdge. It finally produces the vector `acc`, which is summed onto the corresponding vertex accumulation vector `accum` with the user-defined Gather.accumulator.

4 Implementation

We implemented NeuGraph on top of TensorFlow (v1.7) with about 5,000 lines of C++ code and 3,000 lines of Python code. NeuGraph uses TensorFlow as the dataflow execution runtime, and additionally provides three specialized modules for GNN applications: (1) an engine translating a vertex-centric symbolic program into dataflow; (2) a streaming scheduler implementing the core scheduling logic; (3) a graph propagation engine with optimized kernels for the proposed Gather/Scatter operators. We discuss several important aspects of our implementation next.

Dataflow Translation. NeuGraph provides a base class `GNNlayer` in addition to the conventional operators; users can easily define each layer of a GNN algorithm by providing a symbolic vertex-program. Then NeuGraph divides vertices and edges into chunks, and generates a chunk-based dataflow graph by appropriately connecting GNN-layers with Gather and Scatter according to the user program. NeuGraph preprocesses the graph using the min-cut partition of METIS [21], and organizes each edge chunk in the compressed sparse column (CSC) format for the feed-forward computation, while using the compressed sparse row (CSR) format for back-propagation computation.

Streaming Scheduler. To improve performance, the streaming scheduler first analyzes the received dataflow graph and incorporates the optimizations described in Section 3.2. NeuGraph implements a filtering operator running on the CPU side, and determines whether to apply it before the H2D operator of each vertex chunk based on the percentage of relevant vertices (i.e., selectivity). Also, NeuGraph profiles the transfer/computation information of edge chunks and revises the dataflow graph based on the refined scheduling plan discussed in Section 3.2.

Multi-GPU Execution. Different devices in NeuGraph need to communicate with one another for coordination. In existing DL frameworks, an operator is usually dispatched to a specific device, with its input and output tensors on the same device. The multi-GPU communication in NeuGraph is executed by a series of concurrent operators from different devices. In each operator, after memory is allocated on a device for communication, it will exchange addresses with other devices for upcoming device-to-device data transfer. Parameters in different GPUs also need synchronization in each iteration. This is implemented by all-reduce.

Graph Propagation Engine. The graph engine contains graph-specific operator kernels. NeuGraph has optimized implementations for the proposed operators (*gather*, *scatter*, *fused-gather*). Specifically, *scatter* is a map operator that turns vertex data into edge data, and *gather* is a reduce operator that accumulates edge data for each vertex. Also, NeuGraph implements *fused-gather* operator described in Section 3.4 to enable one-pass edge computation when the edge computa-

```
CommNet(v^ℓ): // computing v^{ℓ+1}
  params p = [W_H^ℓ, W_C^ℓ]
  // Passing data over edges
  edge^ℓ = Scatter(v^ℓ)
  // no edge-parallel computation
  acc = ApplyEdge(edge^ℓ):
    return edge^ℓ.src
  set Gather.accumulator = sum
  accum = Gather(acc)
  // compute new vertex data
  v^{ℓ+1} = ApplyVertex(v^ℓ, accum, p):
    return ReLU(p.W_H^ℓ ⊗ v^ℓ + p.W_C^ℓ ⊗ accum)
  return v^{ℓ+1}
```

Figure 10: CommNet in SAGA-NN

```
GCN(v^ℓ): // computing v^{ℓ+1}
  params p = W^ℓ
  // Passing data over edges
  edge^ℓ = Scatter(v^ℓ)
  // edge.data is static weight
  acc = ApplyEdge(edge^ℓ):
    return edge^ℓ.src × edge^ℓ.data
  set Gather.accumulator = sum
  accum = Gather(acc)
  // compute new vertex data
  v^{ℓ+1} = ApplyVertex(v^ℓ, accum, p):
    return ReLU(p.W^ℓ ⊗ accum)
  return v^{ℓ+1}
```

Figure 11: GCN in SAGA-NN

```
GG-NN(v^ℓ): // computing v^{ℓ+1}
  // different for each edge type
  params p, A
  edge^ℓ = Scatter(v^ℓ)
  // edge.data is edge type
  acc = ApplyEdge(edge^ℓ, A):
    return A(edge^ℓ.data) ⊗ edge^ℓ.src
  set Gather.accumulator = sum
  accum = Gather(acc)
  // compute new vertex data with GRU
  v^{ℓ+1} = ApplyVertex(v^ℓ, accum, p):
    return GRU(vertex^ℓ, accum)
  return v^{ℓ+1}
```

Figure 12: GG-NN in SAGA-NN

tion is element-wise.

5 Evaluation

In this section, we demonstrate the efficiency and scalability of NeuGraph by evaluating it on multiple GNNs and datasets.

GNN Models. NeuGraph can support many different types of graph-based neural networks [7, 8, 13, 18, 19, 23, 25, 29, 41]. We use the following three representative GNN models.

Communication neural network (*CommNet*) [41] is a model with which cooperating agents learn to communicate among themselves before taking actions. This network can be used to solve multiple learning communication tasks like traffic control. In CommNet, there is no computation on the edge, so the ApplyEdge stage is simply a passthrough (see Figure 10).

Graph convolutional network (*GCN*) [19,23] applies convolutional operations to an arbitrary graph, and has been used in many semi-supervised or unsupervised graph clustering problems, such as entity classification in a knowledge graph. GCN (see Figure 11) has a computation (without neural networks) on the edge for weighted neighbor activation.

Gated graph sequence neural network (*GG-NN*) [25] applies recurrent neural networks (RNNs) to graph data and is used for NLP tasks. GG-NN performs NN-based edge computation (see Figure 12), with different parameters for different edge types. It also performs a heavy Gated Recurrent Unit (GRU) computation on vertices.

We chose these GNNs as the benchmark algorithms in the evaluation not only because of their different computation patterns, but also for the purpose of comparing with TensorFlow: the propagation stage in these cases can be treated as a sparse matrix multiplication and therefore expressible in TensorFlow. Certain algorithms such as G-GCN in our running example cannot be directly supported using the TensorFlow multiplication operators.

Datasets. Table 1 lists the real-world datasets used for evaluation, including the PubMed citation network (pubmed) [38], the BlogCatalog social network (blog) [42], the Reddit online discussion forum (reddit-small, reddit-full) [18], the Wikipedia data dump (enwiki) [3], and the Amazon data dump (amazon) [30]. The column *feature* in Table 1 reports the sizes of the vertex feature vectors, and the *label* column contains the numbers of label classes. As different GNN tasks share the same GNN architecture and differ only on the output layer, we tested the performance of our system on the task of vertex classification (e.g., classifying academic papers into different subjects in the PubMed citation dataset, which contains sparse bag-of-words feature vectors for each document and a list of citation links between documents) and set the number of layers $\ell = 2$ in experiments.

Dataset	vertex#	edge#	feature	label	avg. degree
pubmed	19.7K	108.4K	500	3	5
blog	10.3K	668.0K	128	39	65
reddit-small	58.2K	1.4M	300	41	25
reddit-full	2.4M	705.9M	300	50	292
enwiki	3.6M	276.1M	300	12	77
amazon	8.6M	231.6M	96	22	27

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

Environment and Baselines. We evaluated NeuGraph on a multi-GPU server, which is equipped with dual 2.6 GHz Intel Xeon E5-2690v4 processors (28 cores in total), 512 GB memory, and 8 NVIDIA Tesla P100 GPUs. The installed operating system is Ubuntu 16.04, using the libraries CUDA 9.0 and cuDNN 7.0.

We compared NeuGraph (NG) with TensorFlow v1.7 (TF) [4], GraphSAGE [18] (TensorFlow backend) and DGL v0.1.3 (PyTorch v1.0 [2] backend) [1]. GraphSAGE is a modeling framework for inductive representation learning on graphs and is widely used to generate low-dimensional vector representations for vertices. DGL is a Python package that serves as an interface between any existing tensor libraries and data expressed as graphs, thereby making it easy to implement GNNs.

We took the existing open-source implementations [1, 18, 23] [i]. We also implemented a basic extension, integrating TensorFlow with the chunk-based dataflow translation (TF-SAGA). The TF-SAGA can support larger GNN models, but

[i] For fair comparison, we took minor optimizations (e.g., replacing inefficient *feed_dict* with preloaded data tensors in memory to avoid redundant memory copies from python runtime to TensorFlow runtime).

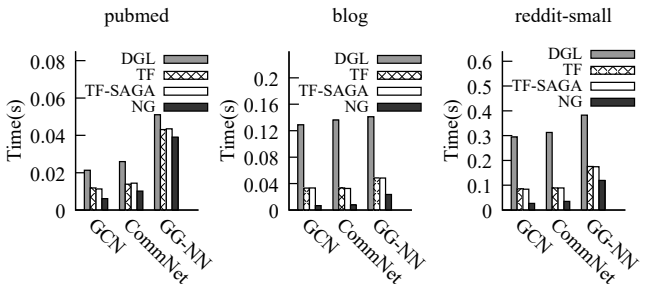

Figure 13: End-to-end performance comparison among DGL, TensorFlow (TF), TF-SAGA and NeuGraph (NG) on small datasets. GraphSAGE runs OOM.

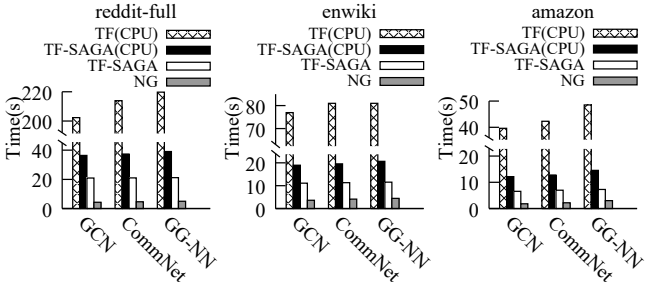

Figure 14: NeuGraph end-to-end performance comparisons on different large datasets. TensorFlow uses CPU-only mode as OOM occurs on GPU. TF-SAGA (CPU) is configured to run on CPU only, whereas TF-SAGA is GPU-enabled.

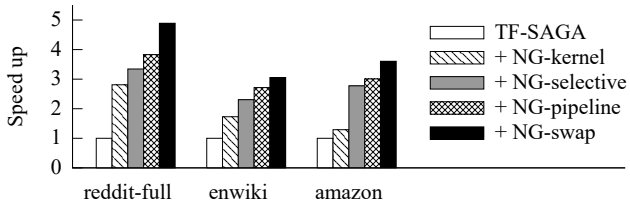

Figure 15: NeuGraph performance improvement breakdown of end-to-end on GCN model over different large datasets. The speedup is measured over the TF-SAGA (speedup = 1).

with all other optimizations described in Section 3 disabled. The comparison with TF-SAGA can reveal how much each optimization contributes to the overall performance.

We focused on metrics for system performance; e.g., time to scan one epoch of data. NeuGraph produces the same numerical results as TensorFlow and DGL, and hence has the same per-epoch convergence. All performance numbers in our experiments are calculated by computing the averages over 10 epochs.

5.1 Performance on a Single GPU

First, we evaluated NeuGraph by comparing it with the state-of-the-art frameworks TensorFlow, DGL, and GraphSAGE. As TensorFlow and DGL can only process graphs that fit in the device memory of a single GPU, we conducted these experiments on the first three small graphs in Table 1.

Figure 13 shows the end-to-end comparison results among different models and datasets. Overall, NeuGraph achieves on average a 2.5× speedup (up to 5.0×) compared with TensorFlow, and on average an 8.1× speedup (up to 19.2×) compared with DGL. We found that the properties of both graphs and models impact performance. NeuGraph achieves the largest speedup with GCN on the blog dataset. This is mainly because the high average vertex degree of the blog graph leads to greater graph propagation (i.e., SAG stages) costs, which NeuGraph can optimize more effectively.

Due to lack of graph support on TensorFlow, GraphSAGE implements GNNs through sampling neighbors and padding to convert irregular graphs to regular tensors. It leads to out of memory even on small graphs using the same evaluation setup (i.e., processing the whole graph with the sampler disabled). Moreover, it still runs about 5× slower than NeuGraph for GCN on pubmed even if the sampler is set to sample exactly one neighbor per vertex.

5.2 Scaling-up on a Single GPU

Since TensorFlow failed to process large graphs on GPU due to the out of memory (OOM) exceptions, we ran TensorFlow only on CPU. Accordingly, besides running TF-SAGA on GPU, we also ran it on CPU. DGL also experienced OOM exceptions when directly processing large graphs on GPU, therefore requiring additional graph sampling to alleviate memory pressure at the expense of model capacity and convergence guarantee. By contrast, NeuGraph can scale GNNs beyond GPU memory without loss on the model scale. Note that NeuGraph can also support the same graph sampling approaches as DGL. In this case, the results in Section 5.1 have already demonstrated that NeuGraph significantly outperforms DGL for small model scales on a single GPU. Hence, we do not compare them again here but focus instead on model scales that cannot fit in GPU memory.

End-to-end Comparison. Figure 14 shows the end-to-end comparison results among different models and datasets. Under the same CPU-only mode, TF-SAGA can achieve on average a 4.3× speedup over TensorFlow. That is because TF-SAGA on CPU contains finer grained chunk-level operators, which can be processed concurrently on the CPUs and make better use of the CPU resources. Moreover, NeuGraph achieves 16 ∼ 47× speedups compared to TensorFlow-CPU, which is the current solution for large graphs.

Compared with TF-SAGA on GPU, NeuGraph could provide even better performance with its additional optimizations. Figure 14 shows that NeuGraph achieves 2.4 ∼ 4.9× speedups over the GPU-enabled TF-SAGA on different models and datasets. Similar to those on small graphs, the speedups on large graphs depend on the graph structure. The average speedup across all models on the reddit-full graph with the highest vertex degree is 4.6× over the GPU-enabled

Time (s)	TF-SAGA			NeuGraph		
Dataset	IO	Comp.	Runtime	IO	Comp.	Runtime
reddit-full	7.67	13.27	20.94	3.84	2.46	4.28
enwiki	5.93	5.13	11.07	3.24	1.77	3.63
amazon	5.11	1.44	6.55	1.56	1.18	1.82

Table 2: GCN on large graphs: TF-SAGA vs. NeuGraph. NeuGraph overlaps I/O and computation time.

TF-SAGA, as opposed to 2.8× on the enwiki graph with moderate vertex degree and 3.1× for the amazon graph with the lowest vertex degree.

Breakdown Comparison. Both streaming and kernel optimizations can play important roles in achieving good overall performance after scaling GNN out of GPU core. To understand how much each optimization contributes to the overall performance, we disabled the graph propagation kernel optimization (NG-kernel) described in Section 3.4, as well as selective scheduling (NG-selective) and pipeline scheduling (NG-pipeline and NG-swap) described in Section 3.2. It effectively turns NeuGraph into the TF-SAGA. We then turned on these optimizations one by one and measured the resulting speedups they brought. To better understand the improvement, we also profiled the GCN execution on both TF-SAGA and NeuGraph with nvprof [32].

Figure 15 shows the improvement of each optimization over TF-SAGA for GCN. The results under other models are similar. We found that the graph kernel optimization works better on dense graphs (like reddit-full), whereas selective scheduling is more effective on sparse graphs (like amazon). For example, the graph kernel optimization can achieve a 2.8× speedup on the reddit-full graph, but only a 1.2× speedup on the amazon graph. However, selective scheduling can still bring an additional 2.6× speedup on the amazon graph. That is because a high-density graph leads to a higher computation cost on SAG stage, which is the target of the graph kernel optimization, whereas a low-density graph with selective scheduling can filter more unnecessary vertices. The figure also shows that our swap-based pipeline scheduling can bring significant improvement by effectively overlapping data transfer and computation, especially on the reddit-full graph where data chunks highly heterogeneous.

Table 2 shows the time of the host-device data transfer (I/O) and computation (Comp.) for TF-SAGA and NeuGraph. Compared to TF-SAGA, the optimizations in NeuGraph reduce both I/O and computation significantly and achieve good overlapping with pipeline scheduling.

As described in Section 3.1, the processing order of chunks may also impact performance. To examine the exact effect of processing order, we ran NeuGraph with the streaming processing optimizations described in Section 3.2 disabled. Figure 16 shows that, for the forward-backward pass, the column-row-oriented strategy is 1.4 ∼ 1.7× faster than the row-column-oriented one.

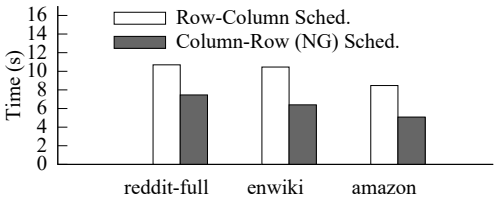

Figure 16: NeuGraph with row/column-oriented chunk scheduling: GCN on large graphs.

5.3 Scaling-out on Multiple GPUs

As described in Section 3.3, we can easily extend TF-SAGA from one GPU to multiple GPUs by allowing each GPU to process a dataflow subgraph, without considering the bandwidth contention. We compared it to NeuGraph with the chain-based scheduling disabled or enabled, in order to understand the performance of our topology-aware scheduling.

Figure 17 shows the results of the GCN model on three large graphs; the results of other GNN models are similar. NeuGraph significantly outperforms the multi-GPU TF-SAGA with the chain-based scheduling enabled or disabled. The average speedup of NeuGraph is 3.6×/2.7× over multi-GPU TF-SAGA with varying numbers of GPUs.

The benefit of the chain-based scheduling is highlighted in the comparison between enabling and disabling this topology-aware scheduling. For example, when scaling from 1 GPU to 2 GPUs, the average speedup of the disabled case even decreases, whereas the enabled one can improve from 3.8× to 5.5× over the single GPU TF-SAGA. This is mainly because, without the chain-based scheduling, two GPUs within the same PCIe switch need to load input edge/vertex data through a shared link concurrently, which can easily become the bottleneck. By contrast, the chain-based mechanism allows the second GPU to load vertex data directly from the first one, reducing the pressure on the shared PCIe link.

We observed that the chain-based scheduling achieves nearly linear speedup on the reddit-full and enwiki graphs, but exhibits less optimal results on the relatively sparse amazon graph. The reason is that NeuGraph tends to apply selective scheduling on relatively sparse graphs. However, given the limited CPU resources shared by an increasing number of GPUs, NeuGraph has to decrease usage of the CPU for per-GPU filtering. Also, the current TensorFlow implementation cannot support NUMA-aware tensor allocation well, which imposes a performance impact on the CPU filtering, especially on sparse graphs like the amazon where the filtering is often used.

6 Related Work

The growing scale and importance of graph data has driven the development of numerous specialized graph processing systems, including Pregel [28], GraphLab [26], Power-

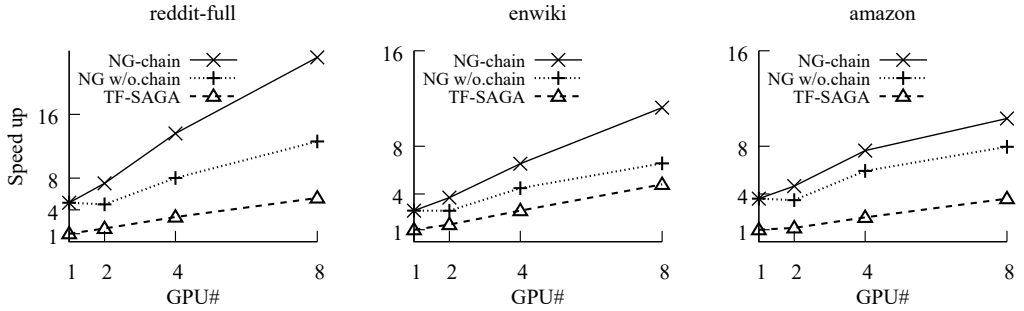

Figure 17: Scaling out GCN with NeuGraph on large graphs (w/o refers to without). The speedup is measured over the single GPU TF-SAGA (speedup = 1). Chain-base scheduling works on multi-GPU, resulting in the same 1 GPU point with it enabled/disabled.

Graph [15] and GraphX [16]. There are many other following works with optimizations on different aspects including graph layout, sequential data access, and secondary storage (e.g., GraphChi [24], Grace [34], FlashGraph [53], XStream [36] and Chaos [35]), distributed shared memory and RDMA (e.g., Grappa [31] and GraM [45]), NUMA-awareness, scheduling, and graph partitioning (e.g., PowerLyra [10] and BiGraph [11]). All these works focus on CPU based computation.

There is another series of system works that focus on exploiting GPU for large graph processing. GraphReduce [39] can process out-of-memory graphs on a single GPU and optimize memory coalescing by using two different formats. GTS [22] can also process out-of-memory graphs on multiple GPUs by fully exploiting the asynchronous GPU streams. Garaph [27] exploits edge-centric parallelism and dynamic scheduling to achieve the best performance on the CPU/GPU hybrid platform. Lux [20] investigates the placement of graph data over the CPU memory hierarchy on multiple nodes. All these graph processing systems are driven by basic graph benchmarks such as PageRank and shortest path, but lack the support for neural network computation, such as the tensor abstraction and auto-differentiation. To be compatible with existing DL libraries, NeuGraph chooses to recast the graph-specific optimizations as dataflow optimizations on top of DL frameworks (e.g., TensorFlow). This does not limit the capability of expressing a general DL computation, and allows users to benefit from both graph and DL optimizations.

TuX^2 [47] aims to bridge the gap between graph and traditional machine learning computation, while NeuGraph targets neural network computation on graphs, which connects graph processing and deep learning supported by the dataflow frameworks like TensorFlow [4], PyTorch [2], MXNet [12], and CNTK [50], etc. Most recently, Cavs [48] introduces the vertex-centric programming model into dynamic neural networks to address the problems that each sample has a unique dataflow graph and the training is iterative on batches of samples. NeuGraph addresses different problems and challenges regarding scalability and performance in supporting GNN models on large real-world graphs. DGL [1] wraps DL systems with a message-passing programming interface for GNNs, while NeuGraph addresses the system challenges (e.g., scalability and efficiency) by translating graph-aware computation on dataflow and recasting graph optimizations.

From the modeling perspective, there are several modeling works (e.g., GraphSAGE [18], MPNN [14], and GN-Block [5]) that attempt to unify existing GNNs into a single modeling framework. These generalized modeling frameworks can be implemented easily and executed efficiently at scale by NeuGraph. Recently developed graph sampling approaches (e.g., DGL [1], GraphSAGE [18], PinSAGE [49], and FastGCN [9]) alleviate scalability challenges of GNNs at the expense of model capacity and convergence guarantee. These approaches are orthogonal to and compatible with our work. NeuGraph frees users from choosing appropriate sample sizes and worrying about GPU memory limitations.

7 Conclusion and Future Work

GNN is an emerging computation model that arises naturally from the need to apply neural network models on graphs. Supporting efficient and scalable parallel computation for GNN training is demanding due to its inherent complexity. Given this new requirement, we advocate unifying graph computation and deep learning systems for GNNs. NeuGraph represents a critical step in this direction by showing not only the feasibility, but also the potential of such unification. We accomplish this by defining a new, flexible SAGA-NN model to express GNN algorithms by fusing graph-related optimizations into the management of data partitioning, scheduling and parallelism in deep learning frameworks.

One potential future direction is to scale GNN further to multiple servers, by leveraging the work in distributed graph systems [40, 44, 45].

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions. We are particularly grateful to our shepherd Harry Xu for his detailed guidance in the final revision process.

References

[1] Deep graph library. https://github.com/dmlc/dgl, Retrieved January, 2019.

[2] PyTorch. http://pytorch.org, Retrieved January, 2019.

[3] Wikimedia downloads. https://dumps.wikimedia.org/, Retrieved May, 2018.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 265–283. USENIX Association, 2016.

[5] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[6] Rianne van den Berg, Thomas N Kipf, and Max Welling. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263*, 2017.

[7] Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

[8] Thang D. Bui, Sujith Ravi, and Vivek Ramavajjala. Neural graph learning: Training neural networks using graphs. In *Proceedings of 11th ACM International Conference on Web Search and Data Mining*, WSDM'18, pages 64–71. ACM, 2018.

[9] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations*, ICLR'18, 2018.

[10] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys'15, pages 1:1–1:15. ACM, 2015.

[11] Rong Chen, Jiaxin Shi, Binyu Zang, and Haibing Guan. Bipartite-oriented distributed graph partitioning for big learning. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys'14, pages 14:1–14:7. ACM, 2014.

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *NIPS Workshop on Machine Learning Systems*, LearningSys'16, 2016.

[13] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, NIPS'16, pages 3844–3852, 2016.

[14] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, ICML'17, pages 1263–1272. JMLR. org, 2017.

[15] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'12, pages 17–30. USENIX Association, 2012.

[16] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 599–613. USENIX Association, 2014.

[17] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, IJCNN'05, pages 729–734. IEEE, 2005.

[18] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, NIPS'17, pages 1024–1034, 2017.

[19] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*, 2015.

[20] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment*, 11(3):297–310, November 2017.

[21] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs.

SIAM Journal on scientific Computing, 20(1):359–392, 1998.

[22] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook 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*, SIGMOD '16, pages 447–461. ACM, 2016.

[23] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, ICLR'17, 2017.

[24] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'12, pages 31–46. USENIX Association, 2012.

[25] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. In *International Conference on Learning Representations*, ICLR'16, 2016.

[26] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[27] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication. In *Proceedings of the 2017 USENIX Annual Technical Conference*, USENIX ATC'17, pages 195–207. USENIX Association, 2017.

[28] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD'10, pages 135–145. ACM, 2010.

[29] Diego Marcheggiani and Ivan Titov. Encoding sentences with graph convolutional networks for semantic role labeling. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, EMNLP'17, pages 1506–1515. Association for Computational Linguistics, 2017.

[30] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'15, pages 43–52. ACM, 2015.

[31] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC'15, pages 291–305. USENIX Association, 2015.

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

[33] Nanyun Peng, Hoifung Poon, Chris Quirk, Kristina Toutanova, and Wen-tau Yih. Cross-sentence n-ary relation extraction with graph lstms. *Transactions of the Association for Computational Linguistics*, 5:101–115, 2017.

[34] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haradasan. Managing large graphs on multi-cores with graph awareness. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, pages 41–52. USENIX Association, 2012.

[35] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 410–424. ACM, 2015.

[36] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 472–488. ACM, 2013.

[37] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[38] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 20(1):61–80, 2008.

[39] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. GraphReduce: Processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 28:1–28:12. ACM, 2015.

[40] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 317–332. USENIX Association, 2016.

[41] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. Learning multiagent communication with backpropagation. In *Advances in Neural Information Processing Systems*, NIPS'16, pages 2244–2252, 2016.

[42] Lei Tang and Huan Liu. Relational learning via latent social dimensions. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'09, pages 817–826. ACM, 2009.

[43] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, ICLR'18, 2018.

[44] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. Fast and concurrent rdf queries using rdma-assisted gpu graph exploration. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC'18, pages 651–664. USENIX Association, 2018.

[45] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. GraM: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC'15, pages 408–421. ACM, 2015.

[46] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[47] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. TuX2: Distributed graph computation for machine learning. In *Proceedings of the the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'17, pages 669–682. USENIX Association, 2017.

[48] Shizhen Xu, Hao Zhang, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P. Xing. Cavs: An efficient runtime system for dynamic neural networks. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC'18, pages 937–950. USENIX Association, 2018.

[49] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure 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*, KDD'18, pages 974–983. ACM, 2018.

[50] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. An introduction to computational networks and the computational network toolkit. Technical report, Microsoft Technical Report MSR-TR-2014–112, October 2014.

[51] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. GaAN: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018.

[52] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *arXiv preprint arXiv:1812.04202*, 2018.

[53] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. FlashGraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 45–58. USENIX Association, 2015.

[54] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.