G^3: When Graph Neural Networks Meet Parallel Graph Processing Systems on GPUs

Husong Liu*, Shengliang Lu*, Xinyu Chen, Bingsheng He
National University of Singapore

ABSTRACT

This paper demonstrates G^3, a programming framework for ***G***raph Neural Network (GNN) training, tailored from ***G***raph processing systems on ***G***raphics processing units (GPUs). G^3 aims at improving the efficiency of GNN training by supporting graph-structured operations using parallel graph processing systems. G^3 enables users to leverage the massive parallelism and other architectural features of GPUs in the following two ways: building GNN layers by writing sequential C/C++ code with a set of flexible APIs (Application Programming Interfaces); creating GNN models with essential GNN operations and layers provided in G^3. The runtime system of G^3 automatically executes the user-defined GNNs on the GPU, with a series of graph-centric optimizations enabled. We demonstrate the steps of developing some common GNN structures with G^3, and the superior performance of G^3 against existing GNN training systems, i.e., PyTorch and TensorFlow.

PVLDB Reference Format:
Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. G^3: When Graph Neural Networks Meet Parallel Graph Processing Systems on GPUs. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Recent neural network (NN) models have moved beyond regular data such as image and speech, to irregular graph-structured data. Graphs, not only as of the de facto data structures in various applications such as social networks, chemistry, and weblink analysis but are also showing their essentials in problem domains across different machine learning settings. Graph Neural Network (GNN), the NN-based method on graph-structured data, attracts a surging interest due to its wide adoption and effectiveness in many applications such as node classification [4] and program verification [5]. Therefore, popular tools and libraries like PyTorch [9] and TensorFlow [1] radically enable graph-based operations for GNN training.

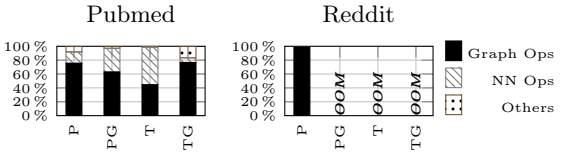

Figure 1: Time breakdown of training GCN using PyTorch (P), PyTorch-GPU (PG), TensorFlow (T), TensorFlow-GPU (TG). "OOM" means the training execution has out-of-memory errors.

However, in real-world development, the bottlenecks in GNN training begin to surface. Our experiments show that graph-structured operations take a large portion of the total workload in GNN model training. As shown in Figure 1, 44%–99% of the overall training time of Graph Convolutional Network (GCN) [4] is spent on graph-structured operations for PyTorch and Tensorflow (even with GPU accelerations). All of the tested GNN frameworks are developed based on matrix operations and message passing without specially optimized for graph structures. As shown in many previous studies on graph processing [18, 14], matrix-based graph processing has two major performance pitfalls. First, memory consumption for storage and intermediate results is prohibitively large and inefficient. Second, when dealing with graph-structured data, matrix-based operations are usually costly and contains redundant computation comparing to graph operations. As a consequence, the performance and scalability of such frameworks are lagged by inefficient graph processing.

Note that existing parallel graph processing systems (PGPS) provide high-performance and scale solutions for traditional graph tasks, e.g., breadth-first search. For example, Medusa [18] and Gunrock [14] leverage the massive parallelism of modern GPU platform, while providing flexible APIs that express a wide range of graph primitives. The success of those PGPS systems enables systems-wide opportunities in resolving the performance bottleneck of graph operations in GNN training. However, the intersection of these two research threads (GNN and PGPS) has not yet been well studied.

In this work, we advocate that by introducing PGPS to GNN, we can fundamentally improve graph-structured operations and the overall efficiency of GNN training. However, such integrations have the following technical challenges.

First, applying existing deep learning tools and frameworks trades efficiency in execution for the simplicity of programming and deploying due to the lack of native support for graph processing. Second, existing graph process-

*These authors contributed equally to this work.

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

ing frameworks hardly provide essential building blocks for GNNs. Users need to implement and optimize GPU programs from scratch for different NN operations. Even though there are NN-related libraries available as building blocks for GNN on GPUs, users have to perform memory management manually and deal with GPU specific programming details such as kernel configuration and scheduling. Third, a hand-crafted GNN on GPU with high efficiency requires explicit program optimizations for GPU architectures. Moreover, a hand-crafted GNN is limited to specific operations, which cannot fulfill the surge of new models.

To ease the pain of leverage GPUs for GNN, we propose a GNN framework, G^3, built based on PGPSs on GPUs. In this work, we use Gunrock [14], one of the state-of-the-art PGPSs on GPUs, to take over the graph-related operations in GNNs. G^3 extends the PGPS with essential NN operations (including matrix operations, SoftMax, and ReLU, to name a few) that are supported by other libraries, e.g., SuiteSparse [2] or implemented by us. Like existing frameworks, G^3 embraces the layered GNN processing model and provides flexible APIs for users.

We will demonstrate the ease-of-programming feature and the superior performance of G^3 with two widely applied GNNs, i.e., GCN [4] and SGC [15]. In particular, G^3 significantly outperforms PyTorch and TensorFlow on their CPU and GPU versions.

2. RELATED WORK AND MOTIVATION

GNN. There are three major categories of GNN models: graph convolutional networks [4], graph recursive networks [5], and graph attention networks [12]. Generally, different GNN models share the same basic operation of collectively aggregating information based on the edge connections of vertices. We refer the readers to several surveys [16, 19], which provide thorough reviews of different GNN models and applications.

Comparing with standard NN approaches, the complexity of graph-structured operations in GNNs creates a significant performance challenge. The inherent irregularity of graph data structures leads to irregularities in data access and control flow, making an efficient implementation on massively parallel architecture, such as GPUs, significantly different from standard NNs. Most of the existing tools and libraries are designed for NN models and do not efficiently express iterative graph processing models. The fundamental solution to improve the efficiency of GNN training could exist in another thread of research, i.e., PGPSs.

PGPS on GPUs. Google has pioneered the research thread of PGPS by introducing the Pregel [8] system. Since then, we have seen the development of a large number of PGPSs. The technical advance of GPU, especially the features of massive parallelism and high memory bandwidth, has attracted many research interests on accelerating graph processing using GPUs. Existing efforts have shown great success in parallelizing a plethora of graph applications [6, 11]. Many frameworks and primitives have also been presented for developing high-performance graph algorithms on GPUs [18, 3, 14].

In the past decades, researchers have paid numerous efforts in addressing the performance issues in GPU graph processing, e.g., memory accesses, workload mapping, and load balancing. Since GNN and traditional graph algorithms

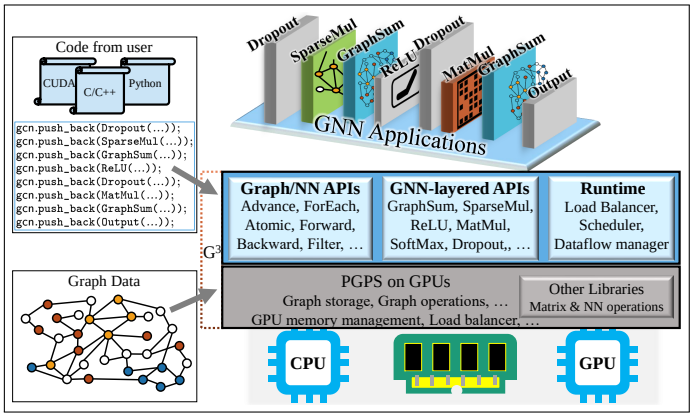

Figure 2: The architecture overview of G^3

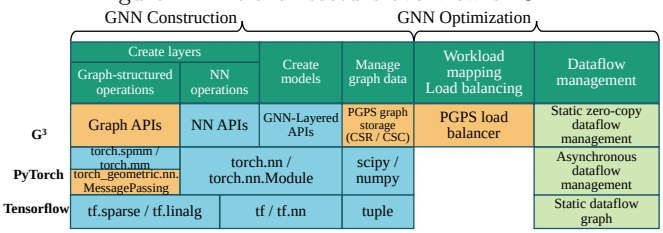

Figure 3: Components of GNN developments using G^3, PyTorch, and Tensorflow

share the same fundamental graph operations, such as transformation on vertices or edges, we can envision the systems-wide opportunities enabled by PGPSs on GPUs.

Bridging GNN and GPU graph processing. There have been some preliminary efforts in building a GNN training system based on combining existing graph systems and NN models. TuX^2 [17] makes the first effort by inheriting the benefits of graph computation model, data layout management, and balanced parallelism for distributed machine learning. DGL [13] presents a graph-oriented message-passing wrapper for deep learning systems but does not yet explore deeply the opportunities to leverage graph-aware optimizations for efficient executions. Most recently, NeuGraph [7] introduces GNN-related graph operations in TensorFlow to enable processing on large-scale graphs. Unfortunately, the system is not yet publicly available. Besides, NeuGraph replaces the layered model of NN with the Scatter-ApplyEdge-Gather-ApplyVertex graph model, which fails to ease the GNN development, while G^3 sticks to the layered model, similar to PyTorch, at users' convenience.

3. G^3 SYSTEM

The overview of G^3 is shown in Figure 2. The system is built based on a GPU-based PGPS with other libraries that support NN operations integrated. In particular, G^3 boosts the graph processing in GNN training in order to improve the overall training. As shown in Figure 3, comparing with PyTorch and Tensorflow, G^3 contains graph-aware components, including graph-structured operations, graph data management, workload mapping, and load balancing. These components are commonly available in PGPSs at high performances but were not used to previous GNN training systems.

To ease the pain of leverage GPUs, G^3 embraces the modular design principle and provides flexible APIs. Descrip-

Table 1: APIs in G^3

Graph APIs	**Description**
Filter	Filters all the nodes/edges in the frontier
Atomic	AtomicAdd/AtomicMin/AtomicMac operations
Advance	Performs a customized function on the nodes/edges, powered by Gunrock
NN APIs	**Description**
ForEach	Performs a customized function on each element
Forward	Training and inference of the NN layer
Backward	Gradient updating scheme for the NN layer
GNN Layers	**Description**
GraphSum	Aggregates information from neighbor vertices
SparseMul	Sparse dense matrix multiplication layer
ReLU	Rectified linear units as activation function
MatMul	Dense dense matrix multiplication layer
SoftMax	Normalize input to a probability distribution
DropOut	Eliminates a portion of elements randomly
CrossEntropy	Loss function, output layer

tions of the APIs are listed in Table 1. Specifically, G^3 offers three categories of APIs: 1) *Graph APIs* exposed from existing PSGS or from our extensions on PSGS based on PSGS's provided APIs; 2) *NN APIs* for manipulating the NN layers; 3) on top of *Graph APIs* and *NN APIs*, we further implement common GNN layers. Due to space limitations, we do not go into details of each layer and refer the readers to a survey paper for more details [10]. Note that these APIs require only sequential C/C++ code. Users do not have to handle GPU-related programming explicitly. G^3 automatically executes the GNN application created using these APIs on the GPU at a high performance.

Listing 1 shows the implementation of the *GraphSum* layer (graph aggregation). G^3 uses the graph intrinsics in the PGPS, e.g., *Advance*, to build graph Graph-structured operations and APIs to avoid reinventing the wheel.

The Graph/NN APIs exposed in G^3 allow users to implement customized GNN layers to support the fast-emerging of new GNN models. Similar to the given GraphSum sample, to implement customized layers, users only need to describe the behaviors of forward and backward operations applied on vertex, and G^3 integrates them into the *Advance* kernel at compilation time.

3.1 Implementation Details

We build G^3 based on Gunrock [14] as it is one of the state-of-the-art systems and satisfies our requirements of building a GPU-based GNN system. The requirements include rich graph-related intrinsics and efficient GPU managements, e.g., low-level GPU memory management, workload mapping, and load balancing. The other libraries, e.g., SparseSuite, are integrated into the Gunrock environment as header-only files, which provide essential functions within the memory space of G^3. In the rest of this section, we give implementation details of G^3.

Graph Storage. Gunrock stores graph in compressed sparse row (CSR) format and represents all per-node and per-edge data as structure-of-array (SOA) data structures that allow coalesced memory accesses with minimal memory divergence. G^3 maximizes the chances to keep the high efficiency of the Gunrock system. We reuse the graph storage provided by Gunrock and extend the support for graph storage with feature vectors and weighted matrices required for different layers of GNN.

Neural Network Generation. G^3 fuses user-defined operations into GPU processing kernel and statically assembles them with pre-built layers during compilation. G^3 connects the layers in order by directing the dataflow from the output of preceding layers to the input of subsequent layers.

Listing 1: Building the GraphSum layer using Graph APIs

```
1  class GraphSum : G3::Layer {
2  // vertex forward operation
3  auto _f=[&]__G3__(VtxT &src, VtxT &des){
4   float coef=1.0/(numNgb(src)*numNgb(des));
5   for(int i=0;i<dim;i++)
6    atomicAdd(out+des*dim+i,
7              *(in+src*dim+i)*coef);
8  };
9  void forward() {
10  PGPS::Advance(graph.csr(),&local,_f);
11 }
12 // vertex backward operation
13 auto _b=[&]__G3__(VtxT &src, VtxT &des){
14  float coef=1.0/(numNgb(src)*numNgb(des));
15  for(int i=0;i<dim;i++)
16   atomicAdd(out+src*dim+i,
17             *(in+des*dim+i)*coef);
18 };
19 void backward() {
20  PGPS::Advance(graph.csr(),&local,_b);
21 }};
```

Table 2: Data set statistics

Dataset	# Nodes	# Edges	# Features
Pubmed	19,717	44,338	500
Reddit	233K	11.6M	602

G^3 Runtime. G^3 adopts the existing GPU memory management solution provided by Gunrock. Gunrock handles the low-level GPU memory management, including memory accesses and data transfers, while G^3 handles high-level dataflow among GNN operations provided by different libraries. G^3's dataflow management avoids memory copy between different layers and minimizes data transfers between GPU global memory and host memory.

Processing graphs with a wide variance in the node degrees, such as the social network, usually causes severe load imbalance. GNNs are more complicated than traditional graph processing because that 1) vertices and edges are associated with feature vectors rather than single values, and 2) the density of graph-structured data changes among different layers. Therefore, G^3 enhances the existing load balancer in Gunrock by considering not only graph-structured data but also the coefficients and feature vectors. Since G^3 is aware of the density changes of graph-structured data among different operators, it thus manages the workload mapping and load balancing by configuring Gunrock at runtime.

4. DEMONSTRATING G^3

Our demonstration focuses on the following two aspects.

1. How to develop a GNN application using G^3?

2. How well does G^3 perform on GNN training?

Demonstration setup. We plan to conduct the evaluations with remote access to a Linux server with two 10-core Xeon E5-2640v4 CPUs, 256GB memory, and an NVIDIA Tesla P100 GPU. The GPU has 12GB global memory and 56 SMs. The demonstration is mostly based on web pages. It would be easy to access using our prepared laptops or participants' smartphones.

The statistics of data sets used for evaluations are summarized in Table 2. We use two commonly used GNN models, namely GCN (Graph Convolutional Network) and SGC (Simplifying Graph Convolutional Networks). The implementations are adopted from the original authors.

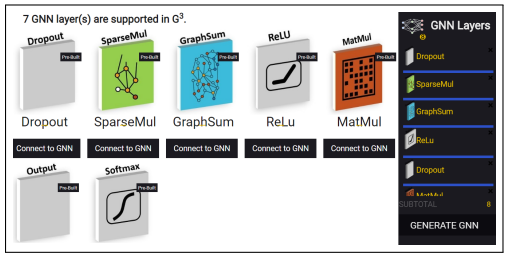

Figure 4: Web-based front-end of G^3

Ease-of-programming demonstration. Overall, we hope to demonstrate to the audience that leveraging existing PSGS enjoys their high programmability, and also the convenience of G^3 in constructing GNN models. This part of demo consists of two aspects.

Firstly, we let participants understand the layered structure of GNN. We invite participants to observe and use the web-based GUI[1] shown in Figure 4 to assemble a GNN application. We will guide the participants to generate one of the most famous GNN models by adding different layer modules to the right panel of the web page.

Secondly, we illustrate how to program a new network layer using the provided Graph/NN APIs. G^3's functor code is C/C++ sequential code with little requirement of parallel programming knowledge. Participants will be provided with code editor boxes[2] to complete forward and backward functors, which will be encapsulated as a user-defined layer. We will present the forward and backward functors of *GraphSum* as examples (Listing 1).

Performance demonstration. We shall demonstrate that the proposed system significantly improves the performance of GNN training against existing solutions. Mainly, we show the breakdown of the elapsed time on each type of layer and the speedups of G^3 over other frameworks.

Figure 5 shows the time breakdown and the speedup for two common GNN models GCN and SGC. We do not include the results for SGC on Tensorflow, because there is no publicly available implementation for SGC in Tensorflow. We hope to include our homegrown implementation in the demo. The speedup of a framework is defined as the ratio of the execution time of PyTorch (P, running on the CPU) and the execution time of the framework. G^3 significantly reduces the overall execution time cost by graph operations in GNN training (from 80% down to 20% of the total execution time), and also improve the overall performance. Specifically, G^3 can be 1.6×–101× faster than PyTorch and Tensorflow on their CPU and GPU counterparts. GPU is not fully utilized on Pubmed data set where G^3 shows only up to 7× speedup over PyTorch. G^3 shows significant speedup on the large Reddit data set, while the other counterparts run out of memory due to inefficient implementations of graph-structured operations.

5. CONCLUSIONS

In this work, we introduce G^3 for efficient GNN training on GPUs by leveraging the graph native operations in parallel graph processing systems on the GPU. This is an initial but important step for bridging the gap in GNN training towards native graph optimizations. We are actively maintaining G^3 and featuring Python interfaces for broader adoption[3].

[1] `http://137.132.92.103:8000/`

[2] `http://137.132.92.103:8000/editor.html`

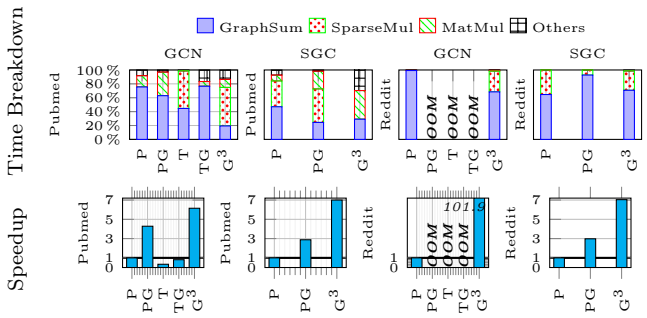

Figure 5: Performance evaluation of PyTorch (P), PyTorch-GPU (PG), TensorFlow (T), TensorFlow-GPU (TG), and G^3 on Pubmed data set. "OOM" means the training execution has out-of-memory errors.

6. REFERENCES

[1] M. Abadi et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[2] T. Davis et al. Suitesparse. 2014.

[3] F. Khorasani et al. Cusha: vertex-centric graph processing on gpus. In *HPDC*, 2014.

[4] T. N. Kipf et al. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[5] Y. Li et al. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[6] H. Liu and H. H. Huang. Enterprise: Breadth-first graph traversal on gpus. In *SC*. ACM, 2015.

[7] L. Ma et al. Neugraph: parallel deep neural network computation on large graphs. In *ATC*, 2019.

[8] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[9] A. Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*. 2019.

[10] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[11] M. Sha, Y. Li, and K.-L. Tan. Gpu-based graph traversal on compressed graphs. In *SIGMOD*, 2019.

[12] P. Veličković et al. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[13] M. Wang et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*, 2019.

[14] Y. Wang et al. Gunrock: A high-performance graph processing library on the gpu. In *PPoPP*, 2016.

[15] F. Wu et al. Simplifying graph convolutional networks. In *PMLR*, 2019.

[16] Z. Wu et al. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[17] W. Xiao et al. Tux^2: Distributed graph computation for machine learning. In *NSDI*, 2017.

[18] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE TPDS*, 25(6), 2014.

[19] J. Zhou et al. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[3] G^3 is available at `github.com/husong998/g3`