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
liuhusong@u.nus.edu, {lusl,hebs,xinyuc}@comp.nus.edu.sg

ABSTRACT

This paper demonstrates G^3, a framework for Graph Neural Network (GNN) training, tailored from Graph processing systems on Graphics 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 popular GNN models 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*, 13(12): xxxx-yyyy, 2020.
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 are not only the de facto data structures in various applications such as social networks, biological networks and weblink analysis, but also show their essentials in problem domains across different machine learning settings. Graph Neural Network (GNN), the NN-based method on graph-structured data, attracts surging interests due to its wide adoption and effectiveness in many applications such as node classification [3] and program verification [4]. Therefore, popular deep learning frameworks like PyTorch [7] and TensorFlow [1] start to support GNN training.

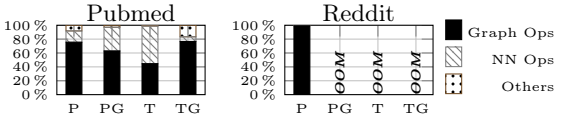

Figure 1: Time breakdown of training GCN using PyTorch (P), PyTorch-GPU (PG), TensorFlow (T), TensorFlow-GPU (TG). "OOM" means the training execution has out-of-memory errors.

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 training. As shown in Figure 1, 44%–99% of the overall training time of Graph Convolutional Network (GCN) [3] is spent on graph-structured operations for PyTorch and Tensorflow (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 [16, 12], 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.

Existing parallel graph processing systems (PGPS) provide high-performance and scale solutions for graph tasks, e.g., breadth-first search. For example, Gunrock [12] and Medusa [16] leverage the massive parallelism of modern GPU architectures, while providing flexible APIs that express a wide range of graph primitives. Those PGPS systems' success enables system-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. We have identified the following technical challenges for such integrations.

First, applying existing deep learning tools and frameworks trades efficiency in execution for the simplicity of programming and deployment 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. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

ing frameworks hardly provide essential building blocks for GNNs. Even though there are some GNN libraries as building blocks for GNN on GPUs, users have to manually perform memory management 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, such a hand-crafted GNN is inflexible and 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. G^3 extends the PGPS with essential NN operations (including matrix operations, SoftMax, and ReLU, to name a few) 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. In our implementation, we adopt Gunrock [12], one of the state-of-the-art PGPSs on GPUs, to take over the graph-related operations in GNNs.

We will demonstrate the ease-of-programming feature and the superior performance of G^3 with popularly applied GNNs, e.g., GCN [3]. Notably, 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 [3], graph recursive networks [4], and graph attention networks [10]. Generally, different GNN models share the same essential operation of collectively aggregating information based on the edge connections of vertices. We refer the readers to several surveys [13, 17], which provide thorough reviews of different GNN models and applications.

Comparing with standard NN approaches, the complexity of graph-structured operations and the inherent irregularity of graph data in GNNs lead significant performance challenges for efficient implementations on massively parallel architectures like GPUs. Most of the existing tools and libraries in NN models are designed for regular matrix operations and do not efficiently express iterative graph processing models.

PGPS on GPUs. Google has pioneered the research thread of PGPS by introducing Pregel [6]. 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 [15, 9]. Many frameworks and primitives have also been presented for developing high-performance graph algorithms on GPUs [16, 12].

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 share common graph operations, such as transformation on vertices or edges, we can take advantage of 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

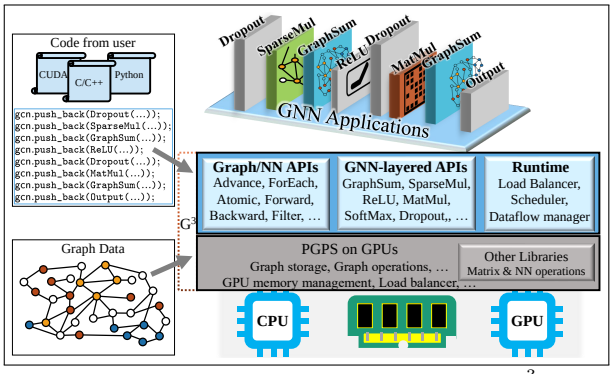

Figure 2: The architecture overview of G^3

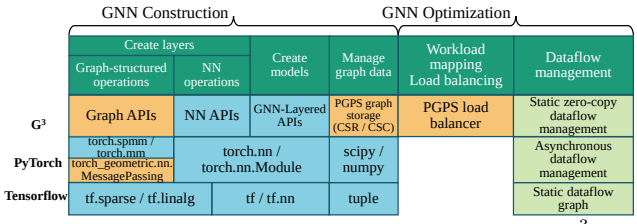

Figure 3: Components of GNN developments using G^3, PyTorch, and Tensorflow

NN models. *TuX*^{2} [14] makes the first effort by inheriting the benefits of graph computation model, data layout management, and balanced parallelism for distributed machine learning. DGL [11] presents a graph-oriented message-passing wrapper for deep learning systems but does not explore in depth the opportunities to leverage graph-aware optimizations for efficient executions. Most recently, NeuGraph [5] 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, for users' convenience.

3. G^3 SYSTEM

The architectural 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. Compared 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 in previous GNN systems.

To ease the pain of leverage GPUs, G^3 embraces the modular design principle and provides flexible APIs. Descriptions 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) *GNN layers*: common GNN layers on top of Graph APIs and NN APIs.

Due to space limitations, we do not detail each layer and refer the readers to a survey paper for more details [8]. The

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

Graph/NN APIs exposed in G^3 allow users to implement customized GNN layers to support the fast-emerging of new GNN models. 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 constructed 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.

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 [12] 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 Gunrock. In the rest of this section, we briefly 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. 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.

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

4. DEMONSTRATING G^3

Our demonstration focuses on the following two aspects.

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

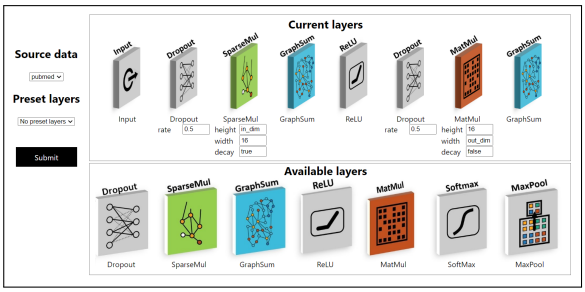

Figure 4: Web-based front-end of G^3

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 statistics of data sets used for evaluations are summarized in Table 2.

Ease-of-programming demonstration. Overall, we hope to demonstrate to the audience on the improved programmability by leveraging existing PSGS, and also the convenience of G^3 in constructing GNN models. This part of the demo consists of two parts.

Firstly, we let participants understand the layered structure of GNN by using the web-based GUI (shown in Figure 4) to reproduce a GNN training using preset layers. These preset layers, which include the popular GNN models, can be selected on the left of the GUI. Next, we will guide the participants to alter the GNN models by adding/removing layers using the GUI and re-train the model.

Secondly, we illustrate how to program a new network layer using the provided APIs. Participants will be provided with code editor boxes to complete forward and backward functors. G^3's functor code is C/C++ sequential code with little requirement of parallel programming knowledge.

Performance demonstration. We shall demonstrate that G^3 significantly improves the performance of GNN training against existing solutions.

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. The speedup of a framework is defined as the ratio of the execution time of PyTorch (P, running on the CPU) and the

Table 2: Data set statistics

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

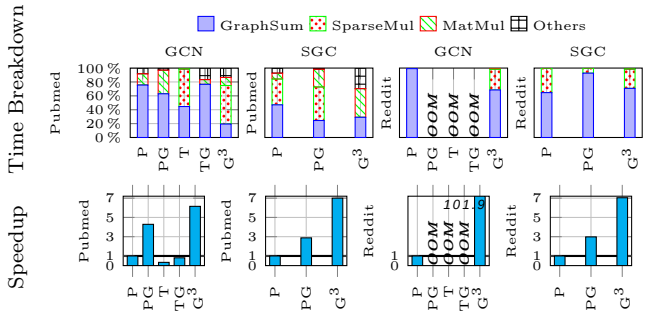

Figure 5: Performance evaluation of PyTorch (P), PyTorch-GPU (PG), TensorFlow (T), TensorFlow-GPU (TG), and G^3 on Pubmed and Reddit data set. "OOM" means the training execution has out-of-memory errors.

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

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 web-based GUI and Python interfaces for broader adoption[1].

6. ACKNOWLEDGMENTS

We acknowledge Ee Ter Low, Long Sha, and Karan Sapra for their contributions on developing the demo. This work is supported by an MoE Tier 1 grant (T1 251RES1824) in Singapore and a collaborative grant from Microsoft Research Asia.

7. REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

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

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

[4] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow. Gated graph sequence neural networks. In *Proceedings of ICLR'16*, April 2016.

[5] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai. Neugraph: parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 443–458, 2019.

[6] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[7] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037. 2019.

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

[9] M. Sha, Y. Li, B. He, and K.-L. Tan. Accelerating dynamic graph analytics on gpus. *Proc. VLDB Endow.*, 11(1):107–120, Sept. 2017.

[10] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

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

[12] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 2016.

[13] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[14] W. Xiao, J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou. Tux^2: Distributed graph computation for machine learning. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 669–682, 2017.

[15] Y. Zhang, X. Liao, H. Jin, B. He, H. Liu, and L. Gu. Digraph: An efficient path-based iterative directed graph processing system on multiple gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 601–614, New York, NY, USA, 2019. Association for Computing Machinery.

[16] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2013.

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

[1] G^3 is available at `http://github.com/Xtra-Computing/G3`