

**ABSTRACT**

This paper demonstrates $G^3$, a programming framework for Graph Neural Network (GNN) training, tailored for 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 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(3x): xxxx-yyyy, 2019. DOI: https://doi.org/10.14778/xxxxx.xxxxxx

**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.

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, the intersection of these two research threads (GNN and PGPS) has not yet been well studied.

**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.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. *Proceedings of the VLDB Endowment*, Vol. 12, No. xxx

ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxx

*These authors contributed equally to this work.

Husong Liu*, Shengliang Lu; Xinyu Chen, Bingsheng He
National University of Singapore
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, G³, 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. G³ 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, G³ 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 G³ with two widely applied GNNs, i.e., GCN [4] and SGC [15]. In particular, G³ 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 share the same fundamental graph operations, as well 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. TuX³ [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 [2] 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 G³ sticks to the layered model, similar to PyTorch, at users’ convenience.

3. G³ SYSTEM

The overview of G³ 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, G³ boosts the graph processing in GNN training in order to improve the overall training. As shown in Figure 3, comparing with PyTorch and Tensorflow, G³ 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, G³ embraces the modular design principle and provides flexible APIs. Descrip-
Perform a customized function on each element
Rectified linear units as activation function

# Nodes # Edges # Features
19,717 44,338 500

Filters all the nodes/edges in the frontier
Gradient updating scheme for the NN layer
Sparse dense matrix multiplication layer
Loss function, output layer

Listing 1: Building the GraphSum layer using Graph APIs

class GraphSum : G3::Layer {
  // vertex forward operation
  auto _f =[&] _G3_.(_VtxT &src, _VtxT &des) {
    float coef = 1.0 / (numNgb(src) * numNgb(des));
    for (int i = 0; i < dim; i++)
      atomicAdd(out + des * dim + i,
        *(in + src * dim + i) * coef);
  }

  void forward() {
    PGPS: :Advance(graph_.csr(), &local_, f);
  }

  // vertex backward operation
  auto _b =[&] _G3_.(_VtxT &src, _VtxT &des) {
    float coef = 1.0 / (numNgb(src) * numNgb(des));
    for (int i = 0; i < dim; i++)
      atomicAdd(out + src * dim + i,
        *(in + des * dim + i) * coef);
  }

  void backward() {
    PGPS: :Advance(graph_.csr(), &local_, b);
  }
}

Table 1: Data set statistics

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Nodes</th>
<th># Edges</th>
<th># Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pubmed</td>
<td>19,717</td>
<td>44,338</td>
<td>500</td>
</tr>
<tr>
<td>Reddit</td>
<td>233K</td>
<td>11.6M</td>
<td>602</td>
</tr>
</tbody>
</table>

Table 1: APIs in G³

<table>
<thead>
<tr>
<th>Graph APIs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>Filters all the nodes/edges in the frontier</td>
</tr>
<tr>
<td>Atomic</td>
<td>AtomicAdd, AtomicMin, AtomicMax operations</td>
</tr>
<tr>
<td>Advance</td>
<td>Performs a customized function on the nodes/edges, powered by Gunrock</td>
</tr>
</tbody>
</table>

### NN APIs

- 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

- GraphSum: Aggregates information from neighbor vertices
- RReLU: Rectified linear units as activation function
- MatMul: Dense dense matrix multiplication layer
- SoftMax: Normalizes input to a probability distribution
- DropOut: Eliminates a portion of elements randomly
- CrossEntropy: Loss function, output layer

The code snippet above demonstrates the implementation of the GraphSum layer using the Graph APIs provided by G³. The Graph APIs allow users to implement customized GNN layers to support the fast-emerging of new GNN models. Similar to the given GraphSum sample, G³ integrates them into the Advance kernel at compilation time.

### 3.1 Implementation Details

**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. G³ 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.** G³ fuses user-defined operations into GPU processing kernel and statically assembles them with pre-built layers during compilation. G³ connects the layers in order by directing the dataflow from the output of preceding layers to the input of subsequent layers.

**G³ Runtime.** G³ 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 G³ handles high-level dataflow among GNN operations provided by different libraries. G³'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, G³ enhances the existing load balancer in Gunrock by considering not only graph-structured data but also the coefficients and feature vectors. Since G³ 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 G³

Our demonstration focuses on the following two aspects.

1. How to develop a GNN application using G³?
2. How well does G³ 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.
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 G³ 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 shown in Figure 4 to assemble a GNN application. We will guide the participants to generate one layer and the speedups of G³ demo. The speedup of a framework is defined as the ratio of the execution time of PyTorch (P, running on the CPU) to that of G³. Figure 4 shows the time breakdown and the speedup for PyTorch (P), PyTorch-GPU (PG), TensorFlow (T), TensorFlow-GPU (TG), and G³ on Pubmed data set. “OOM” means the training execution has out-of-memory errors.

Secondly, we illustrate how to program a new network layer using the provided Graph/NN APIs. G³’s functor code is C/C++ sequential code with little requirement of parallel programming knowledge. Participants will be provided with code editor boxes to complete forward and backward functions, 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 G³ 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 GPU) and the execution time of the framework. G³ 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, G³ 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 G³ shows only up to 7× speedup over PyTorch. G³ 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.

6. REFERENCES


5G³ is available at github.com/husong998/g3