G³: 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 G³, a framework for Graph Neural Network (GNN) training, tailored from Graph processing systems on Graphics processing units (GPUs). G³ aims at improving the efficiency of GNN training by supporting graph-structured operations using parallel graph processing systems. G³ 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 G³. The runtime system of G³ 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 G³, and the superior performance of G³ against existing GNN training systems, i.e., PyTorch and TensorFlow.

PVLDB Reference Format:
Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. G³: When Graph Neural Networks Meet Parallel Graph Processing Systems on GPUs. PVLDB, 13(12): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxx

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 [2]. Therefore, popular deep learning frameworks like PyTorch [7] and TensorFlow [1] start to support GNN training.

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-
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 $12$.

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

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 PSGSs or from our extensions on PSGSs 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

Figure 2: The architecture overview of $G^3$

![Architecture Overview](image_url)

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

![Components Overview](image_url)
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, AtomicMac operations</td>
</tr>
<tr>
<td>Advance</td>
<td>Performs a customized function on the nodes/edges, powered by Gunrock</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NN APIs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForEach</td>
<td>Performs a customized function on each element</td>
</tr>
<tr>
<td>Forward</td>
<td>Training and inference of the NN layer</td>
</tr>
<tr>
<td>Backward</td>
<td>Gradient updating scheme for the NN layer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GNN Layers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphSum</td>
<td>Aggregates information from neighbor vertices</td>
</tr>
<tr>
<td>RReLU</td>
<td>Rectified linear units as activation function</td>
</tr>
<tr>
<td>MatMul</td>
<td>Dense dense matrix multiplication layer</td>
</tr>
<tr>
<td>SoftMax</td>
<td>Normalize input to a probability distribution</td>
</tr>
<tr>
<td>DropOut</td>
<td>Eliminates a portion of elements randomly</td>
</tr>
<tr>
<td>CrossEntropy</td>
<td>Loss function, output layer</td>
</tr>
</tbody>
</table>

Listing 1: Building the GraphSum layer using Graph APIs

```c
class GraphSum : G³::Layer {
  // vertex forward operation
  auto _f=[&] __G3__ (VtxT &src, VtxT &des) {
    float coef = 1.0 / sqrt(numNgb(src) * numNgb(des));
    for (int i = 0; i < dim; i++)
      atomicAdd(out + src * dim + i, *(in + des * 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);
  }
};
```

Figure 4: Web-based front-end of G³

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 E5-2640 v4 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 G³ 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. G³’s functor code is C/C++ sequential code with little requirement of parallel programming knowledge.

Performance demonstration. We shall demonstrate that G³ 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

<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>

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 overall execution time cost by graph operations in GNN 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.

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


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