Characterizing the Efficiency of Graph Neural Network Frameworks with a Magnifying Glass

Xin Huang[1] Jongryool Kim[2] Bradley Rees[3] Chul-Ho Lee[1]

[1]*Texas State University*
[2]*SK hynix America*
[3]*NVIDIA*

Abstract

Graph neural networks (GNNs) have received great attention due to their success in various graph-related learning tasks. Several GNN frameworks have then been developed for fast and easy implementation of GNN models. Despite their popularity, they are not well documented, and their implementations and system performance have not been well understood. In particular, unlike the traditional GNNs that are trained based on the entire graph in a full-batch *manner, recent GNNs have been developed with different graph sampling techniques for* mini-batch *training of GNNs on large graphs. While they improve the scalability, their training times still depend on the implementations in the frameworks as sampling and its associated operations can introduce non-negligible overhead and computational cost. In addition, it is unknown how much the frameworks are 'eco-friendly' from a green computing perspective. In this paper, we provide an in-depth study of two mainstream GNN frameworks along with three state-of-the-art GNNs to analyze their performance in terms of runtime and power/energy consumption. We conduct extensive benchmark experiments at several different levels and present detailed analysis results and observations, which could be helpful for further improvement and optimization.*

1. Introduction

Graphs are everywhere, from social networks to transportation networks to biological networks. It is of vital importance to mining graph-structured data [25, 26] and learning on graphs [20, 28] since they contain rich underlying information and can be used for a wide range of applications. In particular, graph neural networks (GNNs) have attracted a lot of attention in recent years. Unlike the conventional machine learning (ML) algorithms, which assume data samples are independent and identically distributed, GNNs take graph-structured data as input for downstream tasks and capture the correlation between data samples (nodes in the graph) according to their connections (edges in the graph). GNNs have been shown to be effective for many tasks, such as representation learning, node classification, and link prediction.

Early GNN studies mainly reply on general deep learning (DL) frameworks, such as TensorFlow [1] and PyTorch [31]. It is, however, non-trivial to implement a GNN model using the DL frameworks. While they are designed and optimized for regular yet often dense data, real-world graphs often exhibit irregularity and sparsity, thereby making them inefficient for GNNs. Thus motivated, several GNN frameworks have been developed to speed up the computation and to simplify GNN implementation. The examples include Graph Nets [4], Deep Graph Library (DGL) [35], PyTorch Geometric (PyG) [16], StellarGraph [10], Spektral [17], TF-Geometric [21], and CogDL [6].

DGL and PyG are two most popular ones among them, thanks to their user-friendly designs, rich functionalities, and easy-to-follow tutorials. Inspired by NetworkX [18], DGL uses a graph-centric programming abstraction, making it easy for NetworkX users to use. It defines a 'DGLGraph' object as its key data structure for computations with graph-structured data and GNN operations. DGL also realizes the message passing operations of GNNs with generalized sparse-dense matrix multiplication (g-SpMM) and generalized sampled dense-dense matrix multiplication (g-SDDMM). Furthermore, it develops highly tuned CPU and GPU kernels for GNN operations and supports a wide range of applications for general-purpose graph learning. In addition, PyG is an extension library of PyTorch for deep learning on graph-structured data. It provides a simple 'MessagePassing' interface for the message passing operations based on a gather-and-scatter paradigm, which is built on top of its own PyTorch Scatter[1] and PyTorch Sparse[2] that provide dedicated kernels for relevant computations. It also provides a large number of off-the-shelf examples along with a lot of commonly used benchmark datasets for users to easily use and test. Both DGL and PyG have been updated and optimized significantly compared with their initial versions. However, their current implementations and system performance are not well understood.

On the other hand, '*sustainability*' becomes an important factor in both industry and academia due to climate change. Energy and power consumption ought to be critical metrics in ML/DL since training advanced models are often energy and resource hungry. Early studies in ML/DL have, however, mainly focused on improving their model accuracy to achieve state-of-the-art performance. Schwartz *et al.* [32] recently urge researchers to provide not only the accuracy, but also the efficiency in terms of carbon emission, energy

1. https://github.com/rusty1s/pytorch_scatter.
2. https://github.com/rusty1s/pytorch_sparse.

consumption, runtime, to name a few. Strubell *et al.* [33] bring power and energy concerns in ML/DL research by estimating the financial and environmental costs of building well-trained state-of-the-art natural language processing models. There is then a movement, albeit slowly, in recent studies that take power and energy consumption into consideration [32]. Nonetheless, there is no prior work to quantify the power and energy consumption of GNN models and frameworks.

In this paper, we study the two mainstream GNN frameworks – DGL and PyG, by evaluating their efficiency in terms of runtime (not only at the level of each key function but also at the level of the entire model), and power and energy consumption.[3] We benchmark their performance via functional testing on each main component of GNNs and three state-of-the-art GNNs, namely GraphSAGE, ClusterGCN, and GraphSAINT, which adopt graph sampling for mini-batch training. We further provide case studies on different implementation strategies, GPU-based sampling, and full-batch training. We provide detailed and comprehensive analysis to fully understand their performance and find opportunities for further improvement and optimization. We summarize our contributions as follows:

- First, we present the results of functional testing on each key component of building a GNN model in DGL and PyG, including data loader, sampler, and graph convolutional layers. We find that DGL is more efficient for sampling and GNN operations, especially when it comes to large graphs.
- Second, we evaluate the efficiency of sampling and GNN operations on different hardware devices (CPU vs. GPU).
- Third, we provide runtime breakdown of three state-of-the-art GNNs in both frameworks. Our results indicate that there is still a room for further improvement, especially for sampling and data movement.
- Finally, we quantify the power and energy consumption of GNN models and frameworks. To the best of our knowledge, we are the first to analyze the GNN performance from such a green computing perspective.

2. Background and Related Work

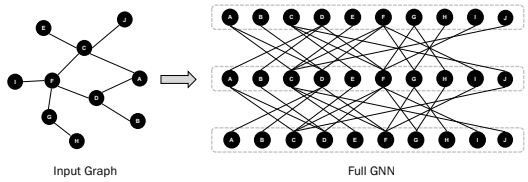
2.1. Graph Neural Networks

GNNs have emerged as an effective means for learning on graph-structured data. They commonly rely on a 'feature aggregation' mechanism, which can be written as

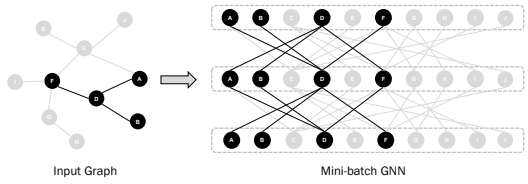
$$\mathbf{H}^{(l+1)} = f(\mathbf{G}\mathbf{H}^{(l)}\mathbf{W}^{(l)}),$$

where $f(\cdot)$ is a non-linear activation function that is applied element-wise, $\mathbf{G}$ is a graph matrix representing the graph structure, e.g., the adjacency or (normalized) Laplacian marix of the input graph, and $\mathbf{H}^{(l)}$ and $\mathbf{W}^{(l)}$ are the node feature/embedding matrix and the weight matrix of neural networks at l-th layer, respectively. In other words, it is the neighborhood aggregation or message passing as each node in the graph updates its current feature vector by aggregating the feature vectors (messages) from its neighbors.

3. Here we do not report the accuracy results of GNN models as they mainly depend on their underlying GNN methods, not the software frameworks. It has also been shown that there is no clear difference between two frameworks when it comes to the accuracy of each GNN model [31, 35, 37].



(a) Full-batch training



(b) Mini-batch training

Figure 1: Two training methods for GNNs.

Due to the feature aggregation mechanism or the interdependence of the nodes (samples), traditional GNNs such as GCN [24] and GAT [34] were trained using the *full-batch* gradient descent, as shown in Figure 1(a). In other words, they require the entire graph and node features to be maintained in memory, leading to a *scalability* issue with large graphs. To cope with the scalability issue, recent GNNs have then adopted 'sampling' techniques to construct *mini-batches* based on the graph structure to train GNNs on large graphs, as mini-batch gradient descent is used for deep neural networks. See Figure 1(b) for illustration.

Hamilton *et al.* [19] proposed GraphSAGE, which is the first work that introduces the use of sampling in training GNNs to improve the scalability. It combines neighborhood sampling, which samples k-hop neighbors with a fixed sampling size for feature aggregation, with mini-batch training. However, its resulting computation graph can be still explosive and thus cause an out-of-memory issue for large graphs. To alleviate this issue, Chen *et al.* [7] developed FastGCN, which samples a fixed number of nodes in each GNN layer independently based on a pre-computed probability distribution. Nonetheless, it can generate isolated nodes, thereby leading to an accuracy drop. Zou *et al.* [40] proposed a layer-dependent importance sampling algorithm called LADIES to resolve the sparsity issue in FastGCN, while it introduces additional computational cost and non-negligible overhead in the sampling process.

In addition, Chiang *et al.* [9] proposed ClusterGCN, which partitions the input graph into many small clusters, some of which are then randomly selected to form a subgraph – or, more precisely, mini-batch, during training. It highly improves the scalability of GNNs, although it

TABLE 1: Dataset statistics

Dataset	Description	# Nodes	# Edges	# Features	# Classes	Train / Val / Test
PPI	Protein-Protein Interactions	14,755	225,270	50	121	0.66 / 0.12 / 0.22
Flickr	Images Sharing Common Properties	89,250	899,756	500	7	0.50 / 0.25 / 0.25
ogbn-Arxiv	Citation Network of arXiv CS papers	169,343	1,166,243	128	40	0.54 / 0.29 / 0.17
Reddit	Online Communities	232,965	114,615,892	602	41	0.66 / 0.10 / 0.24
Yelp	Businesses and Reviews	716,847	13,954,819	300	100	0.75 / 0.10 / 0.15
ogbn-Products	Amazon Product Co-purchasing Network	2,449,029	61,859,140	100	47	0.08 / 0.02 / 0.90

can lead to data imbalance and information loss issues. Zeng *et al.* [38] proposed GraphSAINT, which constructs training batches by sampling subgraphs of the input graph. They leveraged graph sampling techniques, such as node sampling, edge sampling, and random walk-based sampling, to obtain subgraphs.

2.2. Related Work

There are a few GNN benchmark studies in the literature. Dwivedi *et al.* [15] introduced a benchmark framework along with a set of medium-scale graph datasets for a large collection of GNN models. They developed the benchmark framework on top of PyG and DGL and presented the results in accuracy and training time, *yet* without any detailed component analysis. Duan *et al.* [14] used a greedy *hyperparameter* search method to tune up the performance of several GNN models and reported the resulting accuracy of each model and its corresponding time and space complexity. Zhang *et al.* [39] provided a detailed workload analysis on the *inference* of GNNs. Lin *et al.* [27] focused on the *distributed training* benchmark of three GNN models implemented in PyG. The studies in [3, 22, 30] presented new datasets for GNN benchmarking.

The work by Wu *et al.* [37] is most relevant to our work as it is also concerned about the performance analysis of DGL and PyG. It was, however, based only on five datasets of *small-size* graphs with six GNN models, which are mostly traditional ones. Three datasets are for 'graph' classification as a downstream task. The smallest one has 600 graphs, each with about 30 nodes and 60 edges on average, while the largest one has 80K graphs, each with about 70 nodes and 500 edges on average. The other datasets are two small graphs (the larger one has about 20K nodes and 40K edges) for 'node' classification. For this downstream task, they focused on the *full-batch* training, not to mention lack of any detailed component analysis.

We can summarize the *differences* between our work and the GNN benchmark literature as follows. First, we provide a detailed and comprehensive analysis of DGL and PyG not only at the level of the efficiency (total training time) but also at the level of the runtime of each key component of GNN models. Second, our benchmark of the frameworks is done based on a wide range of graphs, having the largest one with about 2.4M nodes and 61M edges, and three representative GNNs that support mini-batch training for scalability. Finally, we present the energy and power efficiency of GNN models and frameworks.

3. Methodology

3.1. GNN Models

To evaluate the performance of two popular GNN frameworks – DGL [35] and PyG [16], we first consider several convolutional layers, which are key components of GNNs. We then consider three representative sampling-based GNNs, namely GraphSAGE [19], ClusterGCN [9], and GraphSAINT [38], implemented in DGL and PyG.

3.2. Datasets

We focus on supervised node classification tasks in this work. To this end, we consider six popular real-world graph datsets, each of whose description and statistics are provided in Table 1. See [38] and [22] for more details on the datasets. As for how to split each dataset for training, validation, and testing, we follow the common way in the GNN benchmark literature, which is to use 'fixed partitions' given by the original authors. The details are reported in the 'Train/Val/Test' column of Table 1.

3.3. Hardware and Software Configuration

For hardware, all experiments are conducted on a Linux server equipped with Dual Intel Xeon Silver 4114 CPUs @ 2.2GHz with 64GB RAM, and an NVIDIA Quadro RTX 8000 GPU with 48GB memory.

For software, we use Python 3.8, PyTorch v1.11.0, DGL v0.8.2, and PyG v2.0.4. All GNN models were implemented based on the official examples provided by DGL with PyTorch backend and PyG. To match the implementations in both frameworks for a fair comparison, we set the same values of the hyperparameters of samplers, convolutional layers, and other components of GNN models as long as both frameworks provide the same functional APIs. We use the default settings as provided by the frameworks otherwise. Our code is available on GitHub.[4]

Our main focus in this work is to evaluate the efficiency of the GNN frameworks in runtime and energy/power consumption. Note that we here do not consider the accuracy of each GNN model as there is no clear difference between the frameworks [16, 35, 37]. We use `pyinstrument`[5] to measure the runtime of each key function of GNNs and that of each GNN model along with its breakdown results. In addition, we use `CodeCarbon`[6] to measure power and energy

4. https://github.com/xhuang2016/GNN-Benchmark.
5. https://github.com/joerick/pyinstrument.
6. https://github.com/mlco2/codecarbon.

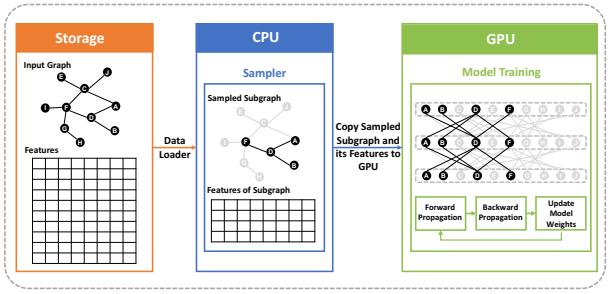


Figure 2: Workflow of sampling-based GNN training.

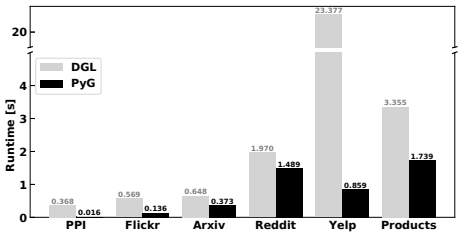
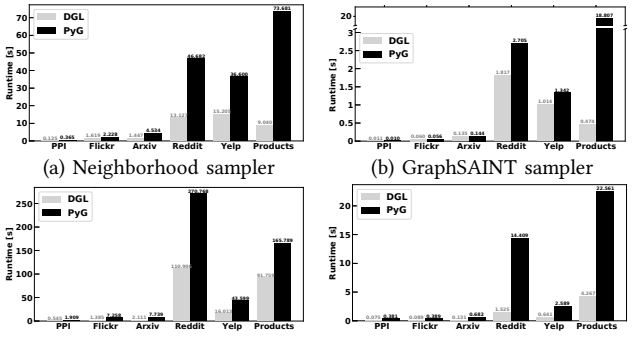


Figure 3: Runtime of data loader.



(a) Neighborhood sampler (b) GraphSAINT sampler

(c) ClusterGCN sampler: METIS (d) ClusterGCN sampler: Combining

Figure 4: Runtime comparison of graph samplers. Note that the range of y-axis is different across different figures.

consumption, which is a Python package for tracking carbon emissions produced by algorithms and programs.[7] We use the sampling interval of 0.1 seconds in this tool instead of its default setting, which is 15 seconds. Considering the fact that it is a software tool, we admit possible discrepancies between measured values and actual ones. Nonetheless, we emphasize that *relative* comparisons remain meaningful and informative regardless, not to mention that we are the first work to evaluate the power and energy consumption of GNNs and GNN frameworks.

4. Results and Discussion

In this section, we provide and discuss the detailed benchmark results on the efficiency of DGL and PyG.

4.1. Functional Testing

Figure 2 illustrates the end-to-end workflow of training a sampling-based GNN with mini-batch training. It can be divided into the following three main processes: data loading, graph sampling, and model training. We thus conduct 'functional testing' on each main process to evaluate the performance of DGL and PyG. Note that for the entire training process, data loading is a one-time operation while the other two process, i.e., graph sampling and model training, are performed repeatedly and periodically for each training batch. Note also that we do not consider the inference of each model in this paper. We repeat the experiments for each functional test for ten times and report the average values. In addition, for the functional tests, we do not include the power/energy consumption results since the runtime of some functions are too small, e.g., a few milliseconds, which can lead to incorrect power/energy measurement.

Data loader. We first compare the data loader of DGL and PyG, which is used to load the input graph and its associated node features from storage and to create a library-specific graph object for the next process of graph sampling and model training. We present the runtime results in Figure 3.

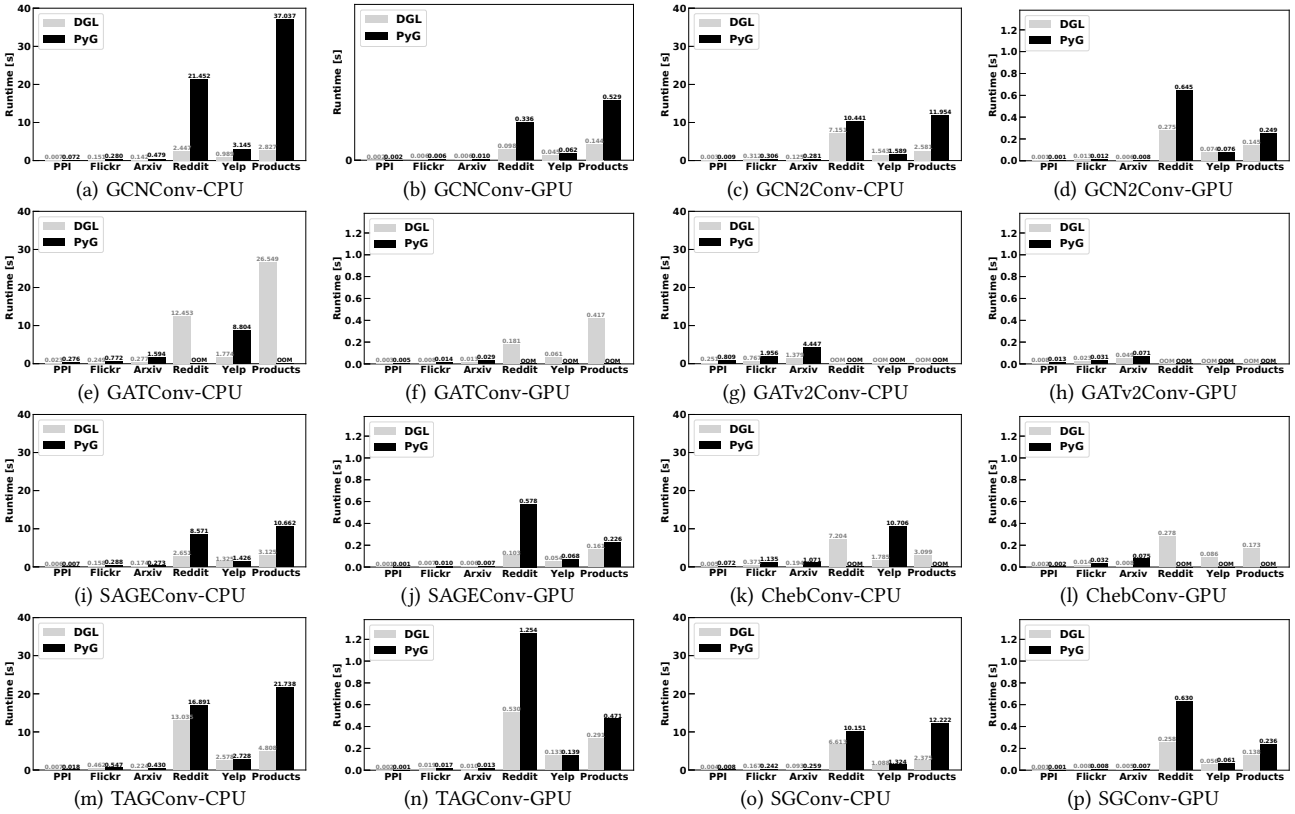
Observation 1: *PyG's data loader is more efficient and user-friendly than DGL's data loader.*

There are two main reasons. First, while both frameworks provide an easy-to-use interface to create and process the datasets, PyG integrates more datasets (around 80) into its library as compared with DGL (around 40). Specifically, five out of six datasets used in this work can be directly accessed from PyG's 'dataset' module while three datasets are already included in DGL. Note that, for the datasets that are not included in the libraries, we follow the official instructions to process the raw datasets and to create their corresponding graph objects. Second, DGL uses a graph-centric programming abstraction, which makes rich information of the input graph accessible and enables full control of manipulating the input graph. As a consequence, the workload of creating a 'DGLGraph' object is relatively higher than its counterpart in PyG.

Sampler. We then compare the performance of three different graph samplers provided by DGL and PyG, namely neighborhood sampler in GraphSAGE [19], graph clustering-based sampler in ClusterGCN [9], and random walk-based sampler in GraphSAINT [38].

For GraphSAGE sampler, we follow the settings in [19], which sample 25 and 10 neighbors per node in its first-hop and second-hop neighborhoods, respectively, with a batch size of 512. Note that each mini-batch is composed of 512 subgraphs. For ClusterCGN sampler, there are two steps, which are (1) graph partitioning with METIS algorithm

7. This profiling tool is a Python wrapper of Intel running average power limit (RAPL) interface and NVIDIA 'pynvml' library. For CPUs, it measures energy consumption by reading the Intel RAPL files and computes the power by dividing energy by time duration. For GPUs, it reads the instant power recorded by pynvml and computes the energy by multiplying power by time duration between two consecutive measurements.



(a) GCNConv-CPU (b) GCNConv-GPU (c) GCN2Conv-CPU (d) GCN2Conv-GPU

(e) GATConv-CPU (f) GATConv-GPU (g) GATv2Conv-CPU (h) GATv2Conv-GPU

(i) SAGEConv-CPU (j) SAGEConv-GPU (k) ChebConv-CPU (l) ChebConv-GPU

(m) TAGConv-CPU (n) TAGConv-GPU (o) SGConv-CPU (p) SGConv-GPU

Figure 5: Runtime of eight Conv layers. Note that the range of y-axis is different for CPU and GPU cases.

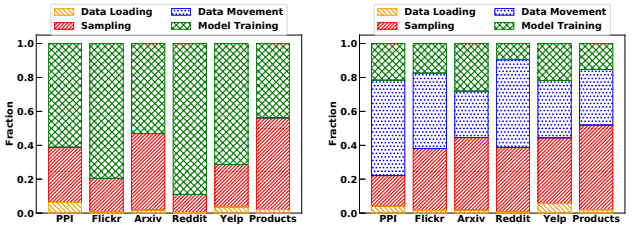
and (2) cluster aggregation. The former partitions the input graph into a given number of small clusters with METIS algorithm, while the latter is to randomly select a few of them to form a subgraph for a training batch. Note that the former is done only once, but the latter is repeated to obtain different mini-batches. In this experiment, we partition the input graph into 2000 clusters and combine 50 of them for each mini-batch. For GraphSAINT sampler, we use the random walk sampling method with 3000 roots and a walk length of two steps to construct subgraphs from the input graph for mini-batch training. While there are two other sampling methods, namely node sampling and edge sampling, in GraphSAINT, we here do not consider them as they are shown to be inferior to the random walk sampling [38]. We measure the runtime of each sampler for one training epoch, i.e., one pass over the entire graph, and report the results in Figure 4.

Observation 2: *All three samplers provided by DGL are more efficient than the ones in PyG. The performance gap is relatively small for GraphSAINT sampler since it is computationally cheaper than the other two samplers.*

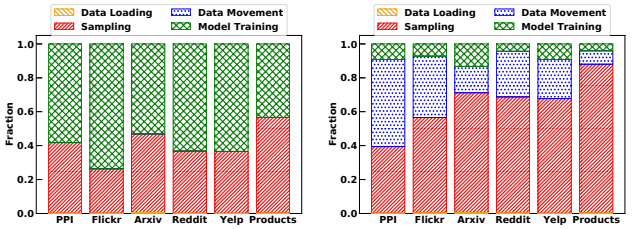
We observe that DGL implements its samplers in C++ with OpenMP, thus leading to superior performance to the ones of PyG, which are developed in Python. In addition, although the choices of hyperparameters can affect the sampling performance, GraphSAINT sampler is generally faster than GraphSAGE's neighborhood sampler and ClusterGCN sampler. It is also worth noting that the neighborhood sampler can lead to a very large computational graph for each node, while the ClusterGCN sampler can lead to information loss and data imbalance. Thus, we expect that the GraphSAINT sampler is a preferable choice in practice. Furthermore, we observe that PyG requires data format conversion to the compressed sparse column (CSC) format, e.g., if it was in the compressed sparse row (CSR) format, which turns out to be quite slow on large datasets. Finally, while all three samplers in both DGL and PyG run on CPU, DGL also provides GPU support and CUDA-Unified Virtual Addressing (UVA) support for GraphSAGE, but not for other GNN models. We shall discuss them in Section 4.3.

Graph convolutional layer. A convolutional (Conv) layer is a key and dominant component of GNNs, and its runtime performance can often reflect the overall performance. We thus conduct functional testing on a collection of Conv layers available in DGL and PyG. Both frameworks provide an 'nn' module that contains the implementations of popular Conv layers. We notice that PyG covers more than 50 Conv layers and DGL has about 30 of them. We here select eight commonly used Conv layers for functional testing. They are GCNConv [24], GCN2Conv [8], ChebConv [11], SAGEConv [19], GATConv [34], GATv2Conv [5], TAGConv [13], and SGConv [36].

We measure the runtime of executing each Conv layer on CPU and GPU. In other words, the reported runtime is



(a) DGL-CPU (b) DGL-CPUGPU



(c) PyG-CPU (d) PyG-CPUGPU

Figure 6: Runtime breakdown of GraphSAGE.

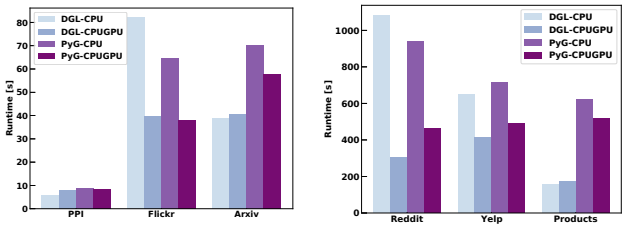


Figure 7: Total runtime time of GraphSAGE.

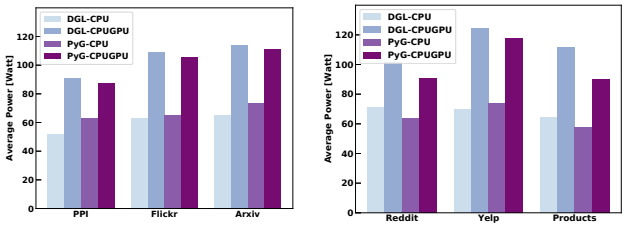


Figure 8: Average power consumption of GraphSAGE.

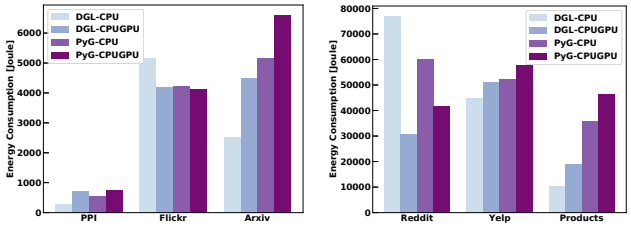


Figure 9: Energy consumption of GraphSAGE.

equivalent to the time of running *one forward propagation* over a single Conv layer with the entire input graph. We manually set the hyperparameters to be the same across the frameworks for each Conv layer. The output dimension is fixed to be 256 for all test cases. The results are presented in Figure 5.

Observation 3: *All eight Conv layers in DGL run faster than the ones of PyG on CPU. The ones in DGL also run faster than their PyG counterparts on GPU in most cases, while PyG only outperforms DGL for few cases with small graphs. Furthermore, graph convolutional operations on GPU show up to 70x speedup over them on CPU.*

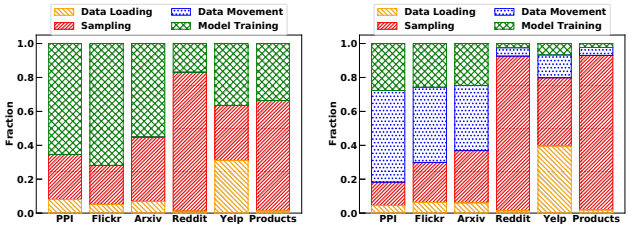
The main reason for the performance on GPU is that DGL adopts an improved CPU message passing kernel developed by [29] to boost the performance, while PyG relies on the CPU kernels included in its own PyTorch Sparse and PyTorch Scatter, where some 'scatter' operations are not well optimized on CPU. As for the performance on GPU, it is worth noting that our observation does not conflict but match with the observation in [37], which shows that PyG is more efficient than DGL, yet for small graphs. Our observation also confirms the claim in [35]. Although DGL is a bit slower on small graphs due to its framework overhead, it is generally more efficient than PyG, especially on large graphs, thanks to its highly tuned kernels. We also find that SAGEConv is relatively computationally cheaper than the other Conv layers, due to its simple aggregation operation.

In addition, we observe that both frameworks provide *fused* kernels to improve their efficiency and scalability, where two separate message-passing and aggregation operations are merged as a single message aggregation operation. DGL uses 'g.update_all()' function to invoke its g-SpMM and g-SDDMM kernels, while PyG simply calls 'matmul()' function in PyTorch Sparse. It is worth noting that PyG does not provide such fused kernel support for ChebConv, GATConv, and GATv2Conv layers. As a result, all three layers of PyG suffer from an out-of-memory issue on large graphs.

4.2. Performance Evaluation of GNNs

We evaluate three representative sampling-based GNNs, namely GraphSAGE, ClusterGCN, and GraphSAINT on CPU and GPU separately. We use 'DGL-CPU' and 'PyG-CPU' to indicate when both sampling and training are done on CPU and use 'DGL-CPUGPU' and 'PyG-CPUGPU' to indicate when sampling is done on CPU while training is done on GPU. We present their runtime breakdown, total runtime, average power consumption, and energy consumption in Figures 6–17. Note that, for all three GNNs, we use the same hyperparameters of their samplers as used in the above functional testing. We use two convolutional layers for all three models and the hyperparameters of each GNN model are set to be the same across DGL and PyG for a fair comparison. The reported results are based on the models trained by 10 epochs. We repeated the same experiments multiple times and observed more or less the same results.

As shown in Figure 6, Figure 10, and Figure 14, we break the runtime of each GNN into four parts, which are data loading, sampling, data movement, and model training. Data loading is done by 'data loader' to load the input graph and its associated node features from storage to CPU memory. Sampling is done by 'sampler' to extract subgraphs and fetch the node features of the sampled subgraphs from the entire feature matrix for mini-batch training. Data



(a) DGL-CPU (b) DGL-CPUGPU

(c) PyG-CPU (d) PyG-CPUGPU

Figure 10: Runtime breakdown of ClusterGCN.

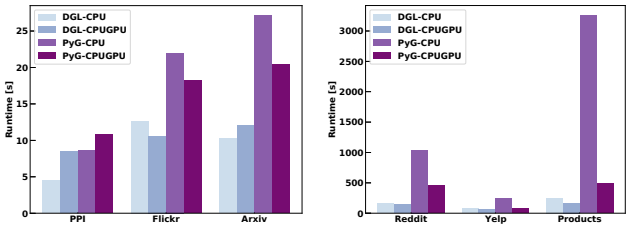


Figure 11: Total runtime of ClusterGCN.

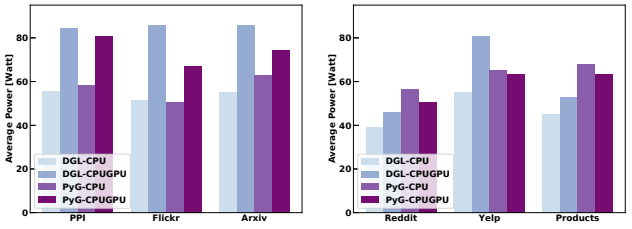


Figure 12: Average power consumption of ClusterGCN.

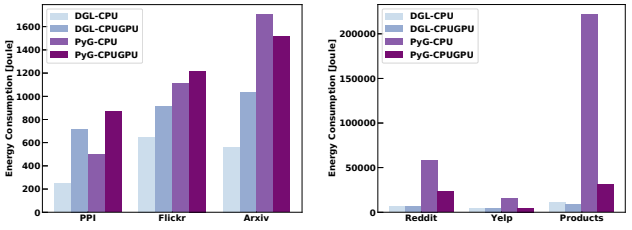
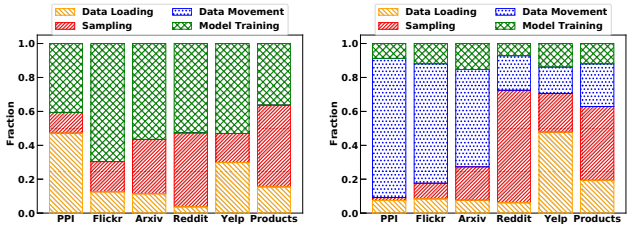


Figure 13: Energy consumption of ClusterGCN.

movement is to copy the initial weight matrices of a GNN model, each subgraph matrix, and its corresponding node features from CPU to GPU. Note that there is no data movement (from CPU to GPU) for DGL-CPU and PyG-CPU. Model training includes forward propagation, backward propagation, and update of model weights. Note that as the number of training epochs increases, the fraction of



(a) DGL-CPU (b) DGL-CPUGPU

(c) PyG-CPU (d) PyG-CPUGPU

Figure 14: Runtime breakdown of GraphSAINT.

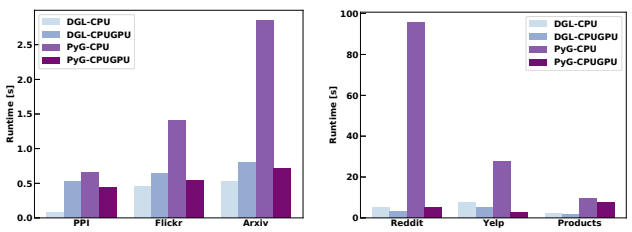


Figure 15: Total runtime of GraphSAINT.

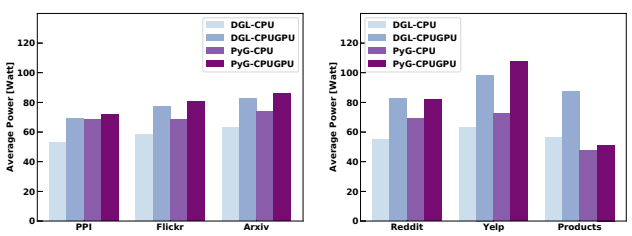


Figure 16: Average power consumption of GraphSAINT.

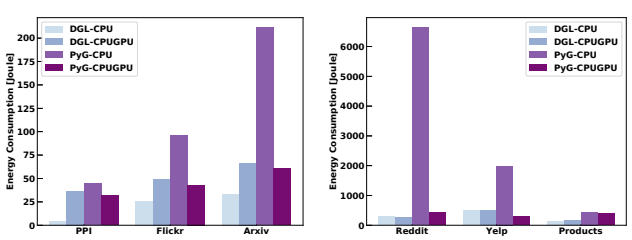


Figure 17: Energy consumption of GraphSAINT.

data loading in total runtime will decrease since it is a one-time operation. However, sampling, data movement, and model training are performed repeatedly for different mini-batches.

Observation 4: *Sampling is slow for all three GNNs and can take up to 90% of total runtime.*

This observation indicates that there is a need to optimize sampling and its associated operations. In particular, for PyG, its CPU kernel could be improved for not only sampling but also model training on CPU. In addition, we observe that data movement can also take a large portion of total runtime in both frameworks. As shall be shown in Section 4.3, data pre-loading in the frameworks can be used to mitigate this issue.

Observation 5: *DGL is generally more efficient than PyG on both CPU and GPU in terms of runtime and energy consumption, especially for large graphs.*

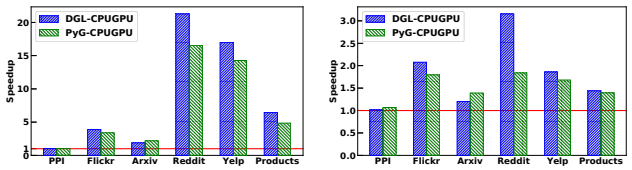
We observe that PyG is more efficient than DGL for small graphs when CPU is used for sampling and GPU is used for training, while DGL is generally more efficient for the other cases. In particular, PyG-CPUGPU is generally more efficient than DGL-CPUGPU for GraphSAINT. This behavior can be explained as follows. With mini-batch training, a GNN model is trained based on sampled subgraphs, which are much smaller than the input graph. We observe that each sampled subgraph (corresponding to a mini-batch) by GraphSAINT sampler is relatively smaller than the ones by GraphSAGE's neighborhood sampler and ClusterGCN sampler. Here the one with GraphSAGE's neighborhood sampler has multiple subgraphs for a mini-batch. Also, recall that the performance gap of the GraphSAINT sampler between DGL and PyG is insignificant, as shown in Figure 4. Since PyG is more efficient in model training with small graphs, PyG becomes more efficient even for medium-size graphs with GraphSAINT, as shown in Figure 15.

In addition, we find that there is no clear winner between DGL and PyG regarding average power consumption, which indicates that energy consumption mainly depends on overall runtime. We observe that GraphSAINT is more efficient in both runtime and energy consumption compared with the GraphSAGE and ClusterGCN, thanks to its light-weight sampling and GNN operations. Note that they are trained for the same number of epochs in our experiments. Nonetheless, we emphasize that different choices of the hyperparameters for each GNN in optimizing its accuracy would affect the efficiency in runtime and energy consumption differently.

4.3. Case Studies

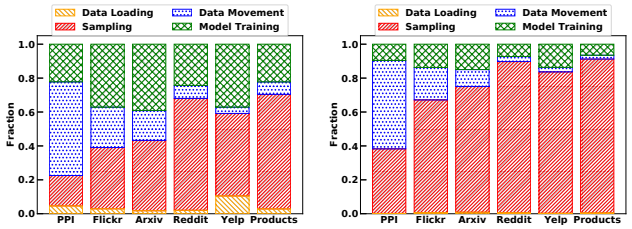
We next turn out attention to three case studies to further evaluate the performance of two GNN frameworks, regarding data pre-loading, GPU-based sampler, and full-batch model training. We focus on GraphSAGE for the case studies.

Pre-loading entire graph and node features into GPU. As shown in Section 4.2, data movement can be a problem when we use CPU for sampling and GPU for model training. We here change the implementation strategy so as to pre-load the entire graph and its associated node features into the GPU *upfront*, which can avoid the overhead of repeated data movement, i.e., the movement of the features of nodes chosen in each mini-batch. Both frameworks provide such a pre-loading option. With this option, the adjacency matrices



(a) Speedup of data movement (b) Speedup of total runtime

Figure 18: Speedup of GraphSAGE when pre-loading the input graph and node features into GPU.



(a) DGL-CPUGPU (b) PyG-CPUGPU

Figure 19: Runtime breakdown of GraphSAGE with data pre-loading.

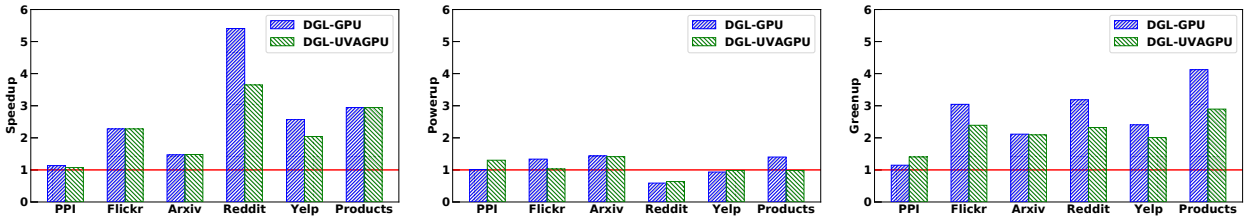
of sampled subgraphs only need to be copied from CPU to GPU for each mini-batch periodically. Note that a mini-batch is composed of a number of sampled subgraphs in GraphSAGE, where the number of sampled subgraphs is the mini-batch size. We present the resulting performance of DGL and PyG with GraphSAGE in Figure 19 for runtime breakdown and in Figure 18 for speedup results.

Observation 6: *The data pre-loading can significantly reduce overall data movement time in both frameworks.*

As expected, the pre-loading strategy saves up to 20x data movement time, thereby leading to about 2x overall speedup. Nonetheless, *it is only feasible when the GPU memory is large enough to hold the entire graph and its associated node features as well as the weight matrix of a GNN model.* It is often not the case in practice, especially for large graphs. An alternative yet effective strategy would be to cache the features of nodes that are most frequently used for model training, i.e., partial information of the graph, into GPU memory upfront, to reduce overall data movement time [12].

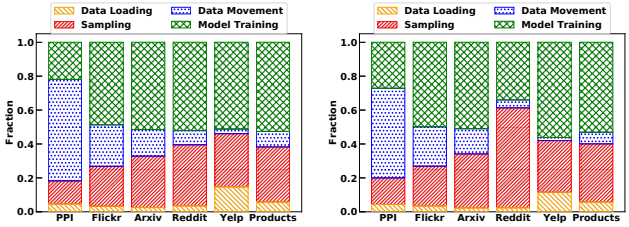
It is worth noting that DGL further provides an advanced feature called 'pre-fetching' for asynchronous data movement and model computation. We observed that the performance of DGL can be further improved, albeit a little bit, with this feature. We omit the results here for brevity.

GPU-based sampler. As mentioned before, the GPU-based neighborhood sampler, i.e., the sampler in GraphSAGE, is available in DGL to accelerate its sampling operation and eliminate the need of moving sampled subgraphs from CPU to GPU for each mini-batch. If the GPU-based sampler is used together with the pre-loading option, it can also eliminate the repeated data transfer of node features, corresponding to sampled subgraphs for each mini-batch. This combination is, however, infeasible for the cases with



(a) Speedup over DGL-CPUGPU (b) Powerup over DGL-CPUGPU (c) Greenup over DGL-CPUGPU

Figure 20: GPS-UP metrics of GraphSAGE with DGL's GPU-based sampler and UVA-based sampler.



(a) DGL-GPU (b) DGL-UVAGPU

Figure 21: Runtime breakdown of GraphSAGE with DGL's GPU-based sampler and UVA-based sampler.

large graphs and/or high-dimensional feature data that do not fit into GPU memory.

In addition, DGL supports another sampler for GrapeSAGE, which is the CUDA-Unified Virtual Addressing (UVA)-based sampler. It uses GPU to perform the sampling operation on the input graph and node features pinned on CPU memory via zero-copy access. This UVA support allows DGL to deal with much larger graphs with the benefits of using GPU for sampling and model training. Note that both UVA-based sampler and GPU-based sampler are currently only available for GraphSAGE in DGL.

We evaluate the performance of the GPU-based sampler ('DGL-GPU') and UVA-based sampler ('DGL-UVAGPU') to see how much improvement they can achieve. For the former, we also use the data pre-loading option. Their runtime-breakdown results are reported in Figure 21. Here the data movement for DGL-GPU contains two parts, which are (1) copying the input graph and node features to GPU for sampling and (2) moving the initial GNN model from CPU to GPU for training. For DGL-UVAGPU, the data movement is only for the initial model.

Observation 7: *The portion of the sampling operation in total runtime becomes smaller compared with the one with DGL-CPUGPU. However, even with GPU for sampling, it can still take up to 40% of total runtime for DGL-GPU and 60% for DGL-UVAGPU. This indicates the non-trivial overhead of the sampling operation and the potential benefit of further accelerating the sampler.*

We next use GPS-UP (Speedup, Greenup, and Powerup) metrics introduced in [2] for further efficiency analysis. The metrics are designed for comparing two different implementations. One of them is an non-optimized version (i.e. baseline) and the other is an optimized version for better performance. Specifically, they are defined as

$$\text{Speedup} = \frac{T_\phi}{T_o}, \quad \text{Greenup} = \frac{E_\phi}{E_o},$$

$$\text{Powerup} = \frac{P_o}{P_\phi} = \frac{E_o/T_o}{E_\phi/T_\phi} = \frac{\text{Speedup}}{\text{Greenup}},$$

where T_ϕ, E_ϕ, and P_ϕ are the runtime, energy consumption, and average power of the non-optimized version, respectively, and T_o, E_o, and P_o are the corresponding values of the optimized one, respectively. We here use DGL-CPUGPU as baseline and report Speedup, Greenup, and Powerup results achieved by DGL-GPU and DGL-UVAGPU over DGL-CPUGPU in Figure 20.

Observation 8: *The use of GPU for sampling saves both time and power in most cases, leading to significant energy saving.*

As can be seen from Figure 20(a), DGL-GPU achieves up to 5.5x speedup over DGL-CPUGPU. DGL-UVAGPU is sightly slower than DGL-GPU, because the former uses zero-copy access to CPU memory, which is generally slower than having access to GPU onboard memory. From Figure 20(b), we also observe that Powerup is not always above one, which implies that the power consumption of using GPU for the sampler can be higher than the CPU counterpart. It happens, especially when there are a large number of edges for each node, e.g., the case of Reddit, making the sampling computation on GPU heavier. Nonetheless, as shown in Figure 20(c), we observe that Greenup is always above one. In other words, it is *more energy-efficient* using GPU for the sampler. While GPU can consume more power than CPU for the sampling operation, it significantly reduces the total runtime, which translates into smaller overall energy consumption.

Our observations indicate the benefits of using GPU for the sampler of GNNs. Nonetheless, this GPU support is currently only limited to GraphSAGE in DGL, and there is no such support in PyG. Note that there is a recent study [23] that leverages GPUs to accelerate graph sampling for GNNs.

Full-batch training. We have focused on three sampling-based GNNs with mini-batch training to evaluate the performance of the frameworks. For a comprehensive evaluation, we here consider *full-batch* training to train a GraphSAGE model, which is done based on the entire graph *without* neighborhood sampling. Specifically, we use a GraphSAGE

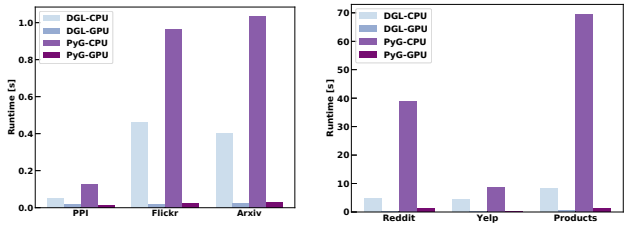


Figure 22: One epoch training time of full-batch GraphSAGE.

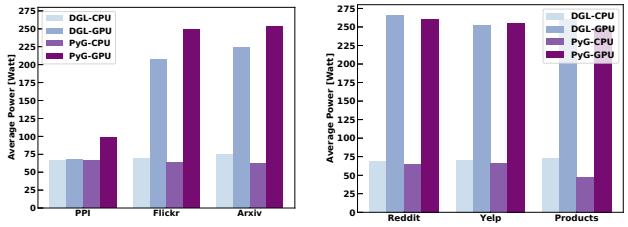


Figure 23: Average power consumption of full-batch GraphSAGE while training.

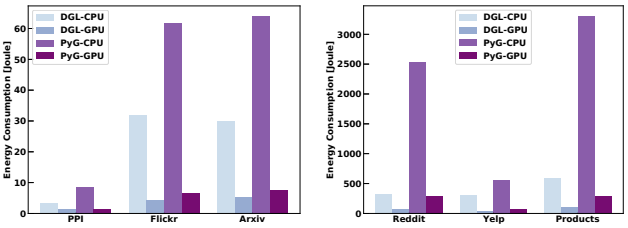


Figure 24: One epoch energy consumption of full-batch GraphSAGE.

model with two layers having mean-aggregator and train the model on CPU and GPU using DGL and PyG separately. We present the experiment results, which are the average results of 100 runs for one training epoch, in runtime, power consumption, and energy consumption in Figures 22–24, respectively.

We observe that DGL-CPU is much faster than PyG-CPU for full-bath model training. DGL-GPU training is slower than its PyG counterpart on the smallest graph PPI, while it is faster for the other five datasets. The results are consistent with our functional test results as reported above. We also observe that there is no clear difference in the average power consumption between the frameworks for model training. That is, the differences in energy consumption between the frameworks mainly come from their differences in training time.

5. Conclusion

We have characterized the efficiency of two mainstream GNN frameworks with three state-of-the-art sampling-based GNNs and six real-world graph datasets, which cover a wide range of graph sizes. We have conducted extensive experiments to evaluate the performance of each key component of GNN models and frameworks as well as each GNN's model performance from the efficiency perspective in runtime and power/energy consumption. We expect that our observations at many different levels would be useful for further improvement and optimization of GNN models and frameworks.

Acknowledgments

This work was supported in part by a grant from SK hynix America and an equipment gift from NVIDIA. This work was also supported in part by the National Science Foundation under Grant IIS-2209921.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015.

[2] S. Abdulsalam, Z. Zong, Q. Gu, and M. Qiu, "Using the Greenup, Powerup, and Speedup metrics to evaluate software energy efficiency," in *6th International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2015, pp. 1–8.

[3] T. Baruah, K. Shivdikar, S. Dong, Y. Sun, S. A. Mojumder, K. Jung, J. L. Abellán, Y. Ukidave, A. Joshi, J. Kim, and D. Kaeli, "GNNMark: A benchmark suite to characterize graph neural network training on GPUs," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 13–23.

[4] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. Allen, C. Nash, V. J. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[5] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" in *International Conference on Learning Representations*, 2022.

[6] Y. Cen, Z. Hou, Y. Wang, Q. Chen, Y. Luo, Z. Yu, H. Zhang, X. Yao, A. Zeng, S. Guo, Y. Dong, Y. Yang, P. Zhang, G. Dai, Y. Wang, C. Zhou, H. Yang, and J. Tang, "CogDL: A toolkit for deep learning on graphs," *arXiv preprint arXiv:2103.00959*, 2021.

[7] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," in *International Conference on Learning Representations*, 2018.

[8] M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li, "Simple and deep graph convolutional networks," in *International Conference on Machine Learning*. PMLR, 2020, pp. 1725–1735.

[9] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks," in *25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 257–266.

[10] CSIRO's Data61, "StellarGraph machine learning library," https://github.com/stellargraph/stellargraph, 2018.

[11] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *Advances in Neural Information Processing Systems*, vol. 29, 2016.

[12] J. Dong, D. Zheng, L. F. Yang, and G. Karypis, "Global neighbor sampling for mixed CPU-GPU training on giant graphs," in *27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 289–299.

[13] J. Du, S. Zhang, G. Wu, J. M. Moura, and S. Kar, "Topology adaptive graph convolutional networks," *arXiv preprint arXiv:1710.10370*, 2017.

[14] K. Duan, Z. Liu, W. Zheng, P. Wang, K. Zhou, T. Chen, Z. Wang, and X. Hu, "Benchmarking large-scale graph training over effectiveness and efficiency," in *Workshop on Graph Learning Benchmarks*, 2022.

[15] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson, "Benchmarking graph neural networks," *arXiv preprint arXiv:2003.00982*, 2020.

[16] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[17] D. Grattarola and C. Alippi, "Graph neural networks in TensorFlow and Keras with Spektral [application notes]," *IEEE Computational Intelligence Magazine*, vol. 16, no. 1, pp. 99–106, 2021.

[18] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using NetworkX," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[19] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[20] W. L. Hamilton, "Graph representation learning," *Synthesis Lectures on Artifical Intelligence and Machine Learning*, vol. 14, no. 3, pp. 1–159, 2020.

[21] J. Hu, S. Qian, Q. Fang, Y. Wang, Q. Zhao, H. Zhang, and C. Xu, "Efficient graph deep learning in TensorFlow with tf_geometric," in *29th ACM International Conference on Multimedia*, 2021, pp. 3775–3778.

[22] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *Advances in Neural Information Processing Systems*, vol. 33, pp. 22 118–22 133, 2020.

[23] A. Jangda, S. Polisetty, A. Guha, and M. Serafini, "Accelerating graph sampling for graph machine learning using GPUs," in *16th European Conference on Computer Systems*, 2021, pp. 311–326.

[24] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017.

[25] S. Li, X. Huang, and C.-H. Lee, "An efficient and scalable algorithm for estimating Kemeny's constant of a Markov chain on large graphs," in *27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 964–974.

[26] ——, "Estimating distributions of large graphs from incomplete sampled data," in *IFIP Networking Conference*. IEEE, 2021, pp. 1–9.

[27] H. Lin, M. Yan, X. Yang, M. Zou, W. Li, X. Ye, and D. Fan, "Characterizing and understanding distributed GNN training on GPUs," *IEEE Computer Architecture Letters*, vol. 21, no. 1, pp. 21–24, 2022.

[28] Y. Ma and J. Tang, *Deep learning on graphs*. Cambridge University Press, 2021.

[29] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha, "DistGNN: Scalable distributed training for large-scale graph neural networks," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[30] P. Mernyei and C. Cangea, "Wiki-CS: A wikipedia-based benchmark for graph neural networks," *arXiv preprint arXiv:2007.02901*, 2020.

[31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[32] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green AI," *Communications of the ACM*, vol. 63, no. 12, pp. 54–63, 2020.

[33] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," in *57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 3645–3650.

[34] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018.

[35] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep Graph Library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[36] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger, "Simplifying graph convolutional networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6861–6871.

[37] J. Wu, J. Sun, H. Sun, and G. Sun, "Performance analysis of graph neural network frameworks," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 118–127.

[38] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "GraphSAINT: Graph sampling based inductive learning method," in *International Conference on Learning Representations*, 2020.

[39] Z. Zhang, J. Leng, L. Ma, Y. Miao, C. Li, and M. Guo, "Architectural implications of graph neural networks," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 59–62, 2020.

[40] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu, "Layer-dependent importance sampling for training deep and large graph convolutional networks," *Advances in Neural Information Processing Systems*, vol. 32, 2019.