Accelerating GNN Training with Locality-Aware Partial Execution

Taehyun Kim[*]
KAIST

Changho Hwang
KAIST

KyoungSoo Park
KAIST

Zhiqi Lin
USTC, Microsoft Research

Peng Cheng
Microsoft Research

Youshan Miao
Microsoft Research

Lingxiao Ma
Microsoft Research

Yongqiang Xiong
Microsoft Research

ABSTRACT

Graph Neural Networks (GNNs) are increasingly popular for various prediction and recommendation tasks. Unfortunately, the graph datasets for practical GNN applications are often too large to fit into the memory of a single GPU, leading to frequent data loading from host memory to GPU. This data transfer overhead is highly detrimental to the performance, severely limiting the training throughput.

In this paper, we propose locality-aware, partial code execution that significantly cuts down the data copy overhead for GNN training. The key idea is to exploit the "near-data" processors for the first few operations in each iteration, which reduces the data size for DMA operations. In addition, we employ task scheduling tailored to GNN training and apply load balancing between CPU and GPU. We find that our approach substantially improves the performance, achieving up to 6.6x speedup in training throughput over the state-of-the-art system design.

CCS CONCEPTS

• **Computer systems organization → Architectures**; • **Computing methodologies → Machine learning**.

KEYWORDS

Graph Neural Networks, AI framework

[*]This work was done during the internship program in Microsoft Research.

ACM Reference Format:
Taehyun Kim, Changho Hwang, KyoungSoo Park, Zhiqi Lin, Peng Cheng, Youshan Miao, Lingxiao Ma, and Yongqiang Xiong. 2021. Accelerating GNN Training with Locality-Aware Partial Execution. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21), August 24–25, 2021, Hong Kong, China.* ACM. https://doi.org/10.1145/3476886.3477515

1 INTRODUCTION

GNNs are widely used for many applications such as e-commerce product recommendation [18, 27, 32], friend suggestion in social networks [13, 23], and traffic forecasting [30]. The input data to a GNN is represented by a graph, which consists of nodes and edges along with the feature data that describes them. GNNs can effectively excavate the hidden meaning in the relationship across the graph nodes, as it leverages deep learning techniques to improve their information extraction from a large amount of samples.

Training a GNN model operates by training a deep neural network (DNN) model with the feature data extracted from the input graph, and it can benefit from GPU for DNN acceleration. Unfortunately, the recent trend shows that practical graph datasets are too large to fit into single GPU memory. Table 1 reveals that typical GNN datasets consist of 10s to 100s of millions of nodes while the number of edges reaches a few billion. Besides, each node and edge typically requires 100s to 1000s of floating point numbers as its feature data. Therefore, the aggregate volume of the data easily exceeds 100s of GB up to a few TB.

To handle such large graphs, a widely adopted approach is partially loading the input data to GPU memory while keeping the rest in host memory [21]. Then, at each iteration, the feature data for nodes and edges sampled for computation will be transferred to GPU for DNN training. Unfortunately, the amount of data for the DMA operation is often substantial as the number of sampled nodes along with their neighbors could be fairly large, e.g., it needs to transfer 10s of MB per training iteration for computation for only a few 100s of microseconds. Not surprisingly, we observe that the limited

Dataset	Enwiki	OGB-Papers	Twitter
# nodes	12 M	111 M	53 M
# edges	44 M	1,615 M	1,963 M
Volume of node data	49 GB	455 GB	217 GB
Volume of edge data	23 GB	827 GB	1.0 TB

Table 1: Datasets of GNNs [1, 4, 17]. The total data volumes are calculated with feature dimensions of node and edge as 1024 and 128, respectively.

PCIe bandwidth becomes the performance bottleneck for GNN training with a large dataset.

In this paper, we propose a locality-aware, partial GNN code execution strategy that significantly mitigates the PCIe bandwidth bottleneck for a large graph. The key idea is to cut down the amount of data transfer to GPU by leveraging the "near-data" processors to run partial aggregation in parallel. Instead of sending the "raw" feature data, we reduce the data volume for transfer by partially aggregating them on CPU and GPU, respectively. That is, the data cached on GPU memory is computed by GPU while the data on the host memory is computed by CPU. Then, the GPU merges the transferred data from the host memory with its own intermediate data, and feeds the fully aggregated feature data into the DNN model for training. Essentially, our approach more effectively utilizes CPU, a processor resource that used to remain idle for GPU-based GNN training. While we observe that CPU-based computation often greatly improves the overall training throughput, the throughput degrades rapidly if CPU becomes overloaded. To prevent the overload, we introduce a simple scheme that balances the load across CPU cores and between CPU and GPU.

The preliminary evaluation shows that our prototype system outperforms the existing state-of-the-art strategy [21] by up to 6.6× for training of a representative GNN model [14] with the OGB-Papers dataset in Table 1. Even when the number of CPU cores is limited to 8, the performance is improved by more than 2.5×.

2 BACKGROUND

In this section, we provide a brief background on GNN algorithms and introduction of existing GNN frameworks.

2.1 GNN Model Training

Training a GNN model is similar to training a typical DNN model except that the input data to the GNN model is the feature data extracted from a target graph. Figure 1 illustrates two key operations of GNN – aggregation and combination. Aggregation refers to the process of sampling nodes in the graph and aggregating their feature data via reduction operations. Reduction operations can be simple like summation or averaging in some models while other models may employ

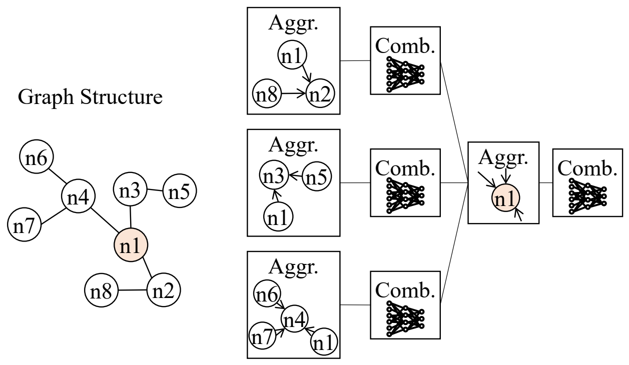

Figure 1: The forward pass of a 2-hop GNN model when the target node is n1.

complex operations such as max-pooling, attention network, or LSTM network [14, 20, 29]. Combination refers to the process of feeding the aggregated feature data into the neural network and computing the result. For an n-hop model, say the batch size is B and the neighbor sampling size is S, then it first samples B nodes (which are the target nodes) from the entire graph (e.g., n1 in the figure), and then samples up to S neighbors of each node recursively up to n-th hop nodes. [1] The n-th hop nodes first aggregate their sampled neighbors and calculate the combination, and the results are aggregated again by the $(n-1)$-th hop nodes, and so on. The final combination results by the target nodes are the forward pass output of the model. If it is a one-hop model, only the feature data of the direct neighbors of the target nodes is aggregated and combinated.

2.2 GPU-based GNN Training

GNN model training benefits from GPU for massively parallel computation. This is because a GNN model consists of multiple layers that require performing the same operation with a lot of data. According to a recent work, training a GNN model on GPU is at least 13 times and up to 37 times faster than training on CPU [28].

GPU-based GNN model training is typically implemented as follows in existing frameworks [5, 28]. First, they run sampling on the graph nodes in CPU as the first step of each training iteration. Since graph sampling requires many random memory accesses in a serial manner as explained in Section 2.1, it is more efficient to run it on CPU with multiple threads, unlike the sampling of typical DNN models that simply generates a sequence of random numbers. Next, they run aggregation and combination on GPU from the last hop based on the sampled results. For these operations, the feature data of sampled nodes and edges must be transferred

[1]Note that models may set different S for different nodes but we omit the explanation for brevity.

Models	ResNet50	WaveGlow	GraphSAGE
Data (MB)	32.0	3.4	22.7
$Time_{GPU}$ (ms)	730	1230	0.4
$PCIe\ BW_{req}$ (GB/s)	**0.043**	**0.003**	**56.8**

Table 2: Estimated PCIe bandwidth to overlap feature data copy with computation on V100 GPU. *Data* **and** $Time_{GPU}$ **refer to average volume of input data and the computation time on GPU per iteration. ResNet50 (ILSVRC2012) and WaveGlow (LJ Speech dataset) use PyTorch, while GraphSAGE (OGB-Papers) uses the prototype of our system for training (see Section 5) while caching popular nodes on GPU. We use a commonly-used training batch size for each model: 256, 12, and 1000, respectively.**

to GPU memory. The feature data transfer has to be done for every training iteration, which easily becomes the slowest point when training a large graph. Inference of a GNN model works in the same way except that it skips backpropagation and sampling. For inference, most models do not sample neighbors – rather, they aggregate all neighbors.

3 HANDLING LARGE DATASETS

The primary problem with GPU-based GNN training is that the graph dataset is often too large to fit into a single GPU memory. We explain why GNN model training is especially vulnerable to large datasets and introduce existing approaches to handle the problem.

3.1 Problem: Data Preparation Overhead

A large input dataset poses a serious problem especially for GPU-based GNN training because the data transfer time dominates the iteration time. In cases of popular CNN or RNN models [16, 24, 25], the model tends to have a larger number of layers with many heavyweight operations, thus the input data transfer of the next iteration can be effectively overlapped with the computation of current iteration. In contrast, the neural network computation in GNN takes relatively much smaller time than the data transfer. This is because GNN typically employs a lightweight neural network that has only 2 to 4 fully-connected layers [11, 12, 14, 20, 31].

Table 2 compares the average size of the input data to be transferred to GPU for each training iteration for popular CNN (ResNet50 [16]), RNN (WaveGlow [24]), and GNN (GraphSAGE [15]) models. It also shows the average iteration time on GPU as well as the amount of PCIe bandwidth required to perfectly hide the data transfer delay. The table shows that the model computation time on GPU is actually 1,825 to 3,075 times smaller than other DNN models, so it requires much higher PCIe bandwidth for overlapping with the data transmission that takes similar or few times longer time. Given that the peak bandwidth achieved with 16 lanes of PCIe v3 is 15.75 GB/s, the PCIe bandwidth is clearly the main bottleneck for GNN model training.

Another serious overhead with the data transfer lies in rearranging the feature data on contiguous memory space for efficient DMA operation. This requires copying many small feature data repeatedly, which significantly wastes the CPU cycles as well as the host memory bandwidth. For example, training a GraphSAGE model with the OGB-Papers dataset rearranges and sends 4 KB of feature data for 6,000 times on average per iteration. As the PCIe transfer throughput is 13.0 GB/s on average while the rearrangement throughput is 2 GB/s per CPU core, we need roughly 7 CPU cores for rearrangement to fully utilize the PCIe bandwidth (see the hardware setup in Section 6).

3.2 Approach 1: Leveraging Multiple GPUs

One approach to tacking the large dataset is to scale the GPU memory size with multiple GPUs [12, 19, 33]. It divides the whole graph into smaller subgraphs so that each subgraph fits into single GPU memory. For each iteration, each GPU gets sampling results for the nodes and edges in its own subgraph, executes a training iteration and updates in a synchronous manner with the other GPUs. The previous works mainly focus on minimizing the edge cuts of graph partitioning algorithms. When dividing a graph, if one edge is cut, two nodes for the edge would belong to different subgraphs. Then, the GPUs holding the nodes should communicate with each other to exchange the data of the nodes and the edge whenever the edge is sampled. Several works have attempted to address this issue [12, 19, 33].

However, this approach raises two problems. First, it would require too many GPUs to handle a very large dataset (e.g., 1-TB dataset in Table 1). For example, one would need 32 GPUs to fully cover 1 TB data with 32 GB memory per GPU, which would be overly expensive. Second, the data-parallel training throughput with a large number of GPUs is known to be suboptimal for GNN models as it would spend the majority of time on aggregating the gradients. Again, the network IO-to-computation time ratio would be very high for GNN models as the GNN models are relatively small.

3.3 Approach 2: Caching Popular Nodes

Another approach to mitigating the long data transfer is to selectively cache those nodes in the graph that are likely to be sampled more frequently than others [21]. One can conceive that the probability for sampling would be proportional to the number of neighbors of each node. This is because having more neighbors implies that the node has more relationship with many other nodes, which would end up increasing the sampling probability. By caching the feature data of such nodes, one may expect to reduce the data transfer size to

GPU. In addition, the number of neighbor nodes in real-world graphs often follow the power law, so the throughput improvement is much larger than the ratio of cached nodes over the total nodes. For example, when caching 3.9% of the nodes in OGB-Papers dataset in 32GB GPU memory, the average data transfer size of a model with the neighbor sampling size of 10 is reduced by 70.9%. In comparison with keeping all data in the host memory, this approach improves the training throughput by 1.7x to 4.8x for popular GNN models [14, 20] and datasets [3, 4].

However, the improvement for very large datasets is limited as the majority of feature data still have to be transferred to GPU. In the case of OGB-Papers above, even if the amount of data to be sent is reduced by 70.9%, it still needs to transfer 22.7 MB of data per iteration. When using the GraphSAGE model as in Table 2, it requires 56.8 GB/s PCIe bandwidth for perfect overlap.

4 LOCALITY-AWARE EXECUTION

In this section, we propose locality-aware execution to overcome the limitations of existing approaches.

4.1 Overview

Our approach leverages the "near-data" processors to execute the first few operations of each training iteration in parallel. This approach exploits the general observation in regular DNN and GNN model training – the size of the intermediate results gradually decreases as the data progresses through multiple layers. We observe that the first aggregation of each GNN training iteration reads a large amount of data and produces a relatively much smaller amount of the intermediate data. For example, the simple aggregation methods such as summation and averaging can cut the data size down to $1/(S+1)$ where S is the neighbor sampling size. In case of a complex aggregation method such as max-pooling [14] and GAT [26], we can reduce the amount of data further as they perform additional compression of the data via obtaining embedding. For these reasons, it is desirable to run the first aggregation on CPU to reduce the data copy size to GPU.

Figure 2 illustrates the workflow of this scheme. CPU not only carries out data batching that rearranges the raw feature data to send to GPU but it also executes aggregation. In each iteration, GPU runs aggregation for the nodes that are cached on GPU memory. The uncached nodes are divided into two groups, and CPU runs aggregation on one group and sends the result to GPU while the raw feature data of the other group is sent to GPU for aggregation. Section 4.2 explains why and how we divide the groups for the uncached nodes. For determining the nodes to cache on

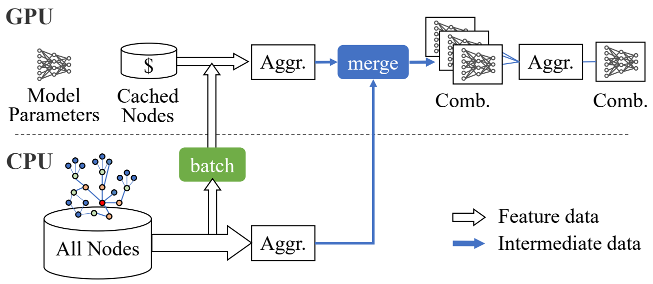

Figure 2: The workflow of our framework with data locality-aware execution.

GPU, we pick the nodes with the most number of neighbors like the approach in Section 3.3 to maximize the benefits of GPU memory caching. When the GPU receives the partially-aggregated feature data, it merges them with its own partially-aggregated data. Then, it proceeds with the combination.

This scheme can be applied to multi-GPU training in the same way as we scale out to multiple GPUs using the Approach 2 – we divide the graph into multiple subgraphs of a similar size and let each GPU cache the popular nodes in each subgraph [21]. For data in host memory, we do partial aggregation first and send the results to corresponding GPUs as before.

4.2 Challenges of GNNs on CPU

Our system runs three tasks on CPU – sampling, data batching, and near-data computation i.e. aggregation. As all these tasks are heavyweight, we need to run them as efficiently as possible. Otherwise, the potential problems below can degrade the performance.

CPU as a potential bottleneck. Blindly performing aggregation on CPU for all graph data on host memory could overload the CPU, which would degrade the overall training performance. It is necessary to carefully balance the load across CPU, PCIe, and GPU. We should adjust the load on CPU to avoid a single point bottleneck in other hardware components.

Load imbalance across CPU cores. In the existing frameworks, multiple threads perform a fraction of workload that belongs to the same iteration in parallel. They typically divide the workload to the threads by balancing the number of target nodes. However, this approach may suffer from stragglers. Since the nodes in the graph tend to have different numbers of neighbors, the total number of neighbors assigned to each thread could be drastically different. In case of near-data computation or data batching, the amount of workload for each thread is proportional to the total number of neighbors it has to process. Even for sampling, fewer neighbors mean less work. When 40 threads sample a model with the batch size of 1000 and the maximum 10 neighbors for each node in Enwiki dataset, we observe that the maximum and average neighbor node counts differ by 23%. This distortion can be even higher for multi-hop GNN models.

CPU scheduling for different tasks. We have to decide how to distribute CPU cores to the tasks. In terms of the throughput, it may be reasonable to pipeline sampling, data batching, and near-data computation by distributing cores appropriately. However, the existing frameworks do not find the best configuration automatically so users have to find the best core distribution manually.

Our execution strategy. To address the above problems, we introduce three methods. First, instead of having the threads perform a part of the task belonging to the same iteration in parallel, we make each thread (pinned on a distinct CPU core) work independently on a task for a distinct iteration. This way, the threads do not suffer from stragglers as they do not wait for synchronization, which would avoid load imbalance across the CPU cores. Second, instead of dedicating several threads only for sampling, we make all threads perform sampling independently to feed their data batching or near-data computation tasks. This approach divides the threads into two groups: a group for sampling and near-data computation, and the other group for sampling and data batching and copying to GPU. Since this approach always performs sampling with other tasks in sequence, it removes the necessity of tuning the number of cores for sampling. We also find that this approach achieves larger throughputs for data batching and near-data computation as well. Third, we fine-tune the core distribution between the data batching and near-data computation groups dynamically at runtime based on their throughput measurements. We initially allocate a half of the available CPU cores to each group and periodically compare the per-thread throughput (the number of processed iterations per second) of each group. When throughputs of both groups are imbalanced, we periodically take one core from the group with the lower throughput and give it to another group. If the PCIe transfer becomes overloaded, the data batching group will slow down, and if CPU becomes overloaded, the near-data computation group will slow down. Therefore, this approach would automatically avoid the bottleneck (either PCIe or CPU) regardless of the hardware setup and the training model.

Unfortunately, this execution strategy cannot be applied to several aggregation methods that leverage trainable parameters such as GAT [26] and max-pooling [15]. This is because the trainable parameters must be up-to-date before every iteration while the threads in our strategy work for future iterations in advance. We leave the CPU execution strategy for aggregation methods with trainable parameters as future work.

Availability. We can apply our main approach, partial aggregation, to most aggregation methods that can guarantee mathematical correctness under parallel execution. We find one corner case that uses LSTM modules for aggregation [14]. The LSTM computation cannot be parallelized because the operation is not associative, i.e. partially calculated results cannot be merged correctly. However, this method is not widely adopted in recent popular GNN models.

5 IMPLEMENTATION

We implement the prototype of our GNN framework. The main system component, data locality-aware execution, can also be integrated in other GNN frameworks like DGL [28]. However, it is challenging to implement our CPU-based optimizations in section 4.2 in current version of DGL. As DGL runs aggregation and combination for an iteration with a single execution using general DL frameworks such as PyTorch, it is hard to separate their execution to pipeline them. So we implement our own framework from scratch with CUDA toolkits and Intel AVX256 instructions. Our framework can offload some operations of GNN to CPU easily and pipeline them with sampling and the remaining operations on GPU. In addition, it supports multi-GPU training and model serving. We plan to do the further analysis and research on them. The overall implementation includes 7,454 lines of C++ code.

User Interface. Users can implement their models by describing the architecture of neural networks and an aggregator for each hop with the APIs provided by our framework. We provide both CPU-based and GPU-based implementations for some popular aggregation methods such as MEAN, SUM, and MAX in the current version and plan to provide other published methods such as GAT and max-pooling soon. However, users have to implement two implementations of CPU and GPU manually for unsupported methods for now. We plan to fix this problem by applying cross-platform deep learning APIs like OneDNN [7] to our interface so that the users can define their own aggregation methods.

6 PRELIMINARY EVALUATION

Experiment Setup. We use a server equipped with two Intel Xeon E5-2630 v4 @2.20GHz (20 logical cores each) and 512GB of DRAM. We use one NVIDIA V100 GPU with 32GB memory connected to the host via 16 PCIev3 lanes. We use the OGB-Papers and Enwiki datasets (in Table1) with node feature dimension of 1024. We take one-hop GraphSAGE with MEAN aggregator, one of the most widely adopted GNN model, for our evaluation. With 32GB of the GPU memory, we can cache 3.9% and 50.1% of node data for OGB-Papers and Enwiki, respectively, beyond carrying the model parameters.

Baselines. Our main baseline is PaGraph [21] that implements the approach described in Section 3.3. Unfortunately,

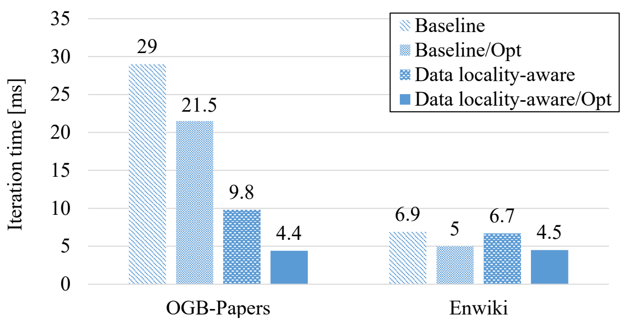

Figure 3: Performance gain of the data locality-awareness in GraphSAGE training.

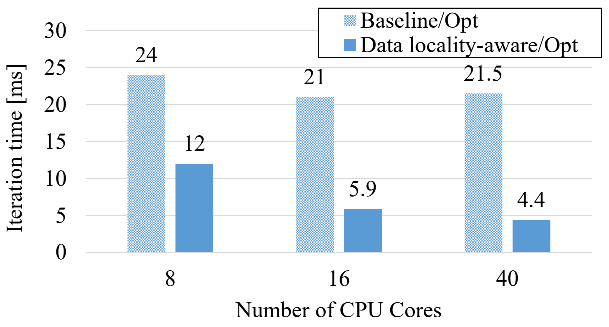

Figure 4: Performance gain of the data locality-awareness in training a GraphSAGE model for the OGB-Papers dataset while varying the number of CPU cores.

DGL that PaGraph is implemented on does not support the performance optimizations such as Just-In-Time compilation to minimize the Python interpreter overhead. This makes it hard to measure the pure gain from the changes in the system design. To avoid the problem, we implement the baseline in our own prototype GNN framework to carry the same feature of PaGraph, full GPU-based aggregation. "Baseline" denotes the full GPU-based aggregation that caches popular nodes on GPU but requires copying the raw feature data of the uncached nodes to GPU. "Opt" indicates that we add the optimizations in section 4.2. In case of the data locality-aware system without "Opt", the optimizations are disabled, so we blindly aggregate all uncached nodes on CPU.

Training throughput. Figure 3 compares the training iteration times with the OGB-Papers dataset. Our approach, the data locality-aware/Opt system, is 6.6× and 4.9× faster than the baseline and the baseline/Opt, which nicely demonstrates the performance gain from the locality-aware partial execution. We observe that the average number of aggregated nodes for a target node on the host memory is about 6. This implies that our approach reduces the amount of data transfer to GPU by a factor of 6. However, the end-to-end speed-up is lower than 6× due to the extra CPU computation overhead. The speed-ups from the optimizations in Section 4.2 are 1.35× for baseline and 2.23× for data locality-aware systems, respectively. As the data locality-aware system uses CPU more aggressively, the CPU-based optimizations bring more benefits.

The improvement with the Enwiki dataset is relatively smaller. The data locality-aware system without the optimizations is even slower than the baseline/Opt. The reason is that Enwiki caches the top 50.1% nodes of most neighbors. Because the number of neighbors in real-world graphs follows the power law [10, 21], even if 49.1% of the nodes remain in the host memory, the sampling probability for them could be small. In fact, we observe that the average data reduction ratio by computation on CPU is only 1.33. Nevertheless, the data locality-aware/Opt system outperforms the baseline and the baseline/opt by 53% and 11%, respectively, as our system can adjust to different environments and avoid any bottleneck.

Figure 4 shows the iteration times over varying numbers of CPU cores. Not surprisingly, the number of CPU cores have higher impact on the performance of our system, but our system outperforms the baseline/Opt by 2× to 4.9×. This demonstrates that our approach effectively utilizes CPU that used to be idle resources in existing GPU-based GNN model training.

While we have evaluated with only one-hop GNN models in this paper, we expect that our system works well with multi-hop GNN models as well. Since the number of neighbors typically increases much rapidly with more hops, a multi-hop GNN model could reveal a more serious bottleneck on the PCIe communication, and we believe our system is beneficial in effectively mitigating it. However, the throughput improvement can vary depending on a model's sampling policy. For node-wise sampling [15] which is the most basic and popular policy, a fixed number of neighbors are sampled at every node. So, we expect as large improvement as the result in Figure 3 regardless of the number of hops. However, this sampling policy has a potential problem that consumes too much memory due to exponentially increasing neighbor nodes. To address the problem, some previous works employs a new policy such as layer-wise sampling [11] or subgraph-wise sampling [12], which reduces the number of neighbors at the cost of some algorithmic loss. For those models, the improvement of our system could be smaller as they aggregate fewer neighbors. However, our system still provides the benefit of reducing the memory footprint on GPU, so even if a user uses such a sampling policy, we can mitigate the memory limitation and maximize the number of neighbors, which can minimize the algorithmic loss.

7 RELATED WORK

GNN frameworks. It is widely adopted for GNN model developers to use specialized deep learning libraries [5, 6, 22, 28] to introduce an efficient graph abstraction over existing general deep learning frameworks [2, 8, 9]. While GNN libraries often cannot execute training when the dataset is very large, several popular libraries such as DGL [28] or PyTorch Geometric [5] support large datasets only by keeping all feature data on the host memory. Unfortunately, this often incurs a detrimental PCIe overhead as explained in Section 3.1.

Optimization for large datasets. To handle large datasets efficiently, several recent works [12, 21, 22, 33] have suggested efficient algorithms to split the target graph into multiple subgraphs that each fits the memory of a single GPU. They split the graph to minimize the communication overhead for exchanging nodes and edges on other subgraghs on remote GPU. In particular, NeuGraph [22] presents an efficient scheduler for data transmission between CPU and GPU, especially when we need to train every node in the graph at the same time without sampling. PaGraph [21] shares the same motivation as our work to reduce data copy over PCIe, which leverages caching popular node data in the GPU memory. While our work adopts the same method as well, our approach exploits CPU to perform partial aggregation to reduce the data transfer size, which better utilizes both CPU and GPU to reduce the PCIe overhead.

8 DISCUSSION

We can apply the locality-aware execution to GNN serving applications. In cases of applications that allow the graph structure and feature data to be updated at runtime, we should infer the result of the GNN model dynamically at runtime. Since the number of neighbors of nodes in real-world graphs typically follow the power law [10], the inference often incurs a long tail latency when it performs aggregation on nodes with many neighbors, which would take a very long time to send all neighbors to GPU. We expect that we can dramatically reduce the tail latency in this case as our method largely reduces the data copy size especially when the node has many neighbors.

As explained in Section 4, our method can adopt multiple GPUs as well. The throughput gain over existing systems of our method may degrade when we use many parallel GPUs, because it increases the total PCIe bandwidth between CPU and GPU, which would degrade the benefit of reducing the PCIe bandwidth requirement. However, our approach would still outperform existing systems as it automatically avoids the PCIe or CPU bottleneck as explained in Section 4.2.

9 CONCLUSION

We have presented a new execution strategy that dramatically mitigates the PCIe bandwidth bottleneck for GNN model training with a large dataset. We have shown that locality-aware, partial aggregation has a great potential to reducing the amount of data before sending it to GPU. Also, we have presented a number of optimizations for efficient CPU cycle usage. Experimental results show that our system accelerates GNN training by up to 6.6× comparing with previous state-of-the-art.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers of APSys'21 for their valuable feedback. This work is in part supported by National Research Council of Science & Technology (NST) grant by the Korean government (MSIT) [No.CRC-15-05-ETRI] and Collaborative Research Projects with Microsoft Research Asia (IITP) grant by the Korean government (MSIT) [No.2020-0-01649].

REFERENCES

[1] 2010. Twitter follow dataset – KONECT. http://konect.cc/networks/twitter_mpi/
[2] 2015. MXNet. https://mxnet.apache.org/versions/1.8.0/
[3] 2017. LiveJournal links network dataset. http://konect.uni-koblenz.de/networks/livejournal-links
[4] 2017. Wikipedia links, English network dataset – KONECT. http://konect.cc/networks/wikipedia_link_en/
[5] 2019. PyTorch Geometric. https://pytorch-geometric.readthedocs.io/en/latest/
[6] 2020. Euler. https://github.com/alibaba/euler
[7] 2021. Intel oneDNN. https://github.com/oneapi-src/oneDNN.https://github.com/oneapi-src/oneDNN
[8] 2021. PyTorch. https://pytorch.org/
[9] 2021. TensorFlow. http://tensorflow.org/
[10] Réka Albert and Albert-László Barabási. 2001. Statistical mechanics of complex networks. *CoRR* (2001).
[11] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *Proceedings of the International Conference on Learning Representations (ICLR).*
[12] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Knowledge Discovery & Data Mining (SIGKDD).*
[13] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Yihong Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph Neural Networks for Social Recommendation. In *Proceedings of the World Wide Web Conference (WWW).*
[14] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS).*
[15] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS).*
[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE*

Conference on Computer Vision and Pattern Recognition (CVPR).

[17] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *CoRR* abs/2005.00687 (2020).

[18] Bo Huang, Ye Bi, Zhenyu Wu, Jianming Wang, and Jing Xiao. 2020. UBER-GNN: A User-Based Embeddings Recommendation based on Graph Neural Networks. *CoRR* abs/2008.02546 (2020).

[19] George Karypis and Vipin Kumar. 1998. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*.

[20] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[21] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[22] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[23] Nan Mu, Daren Zha, Yuanye He, and Zhihao Tang. 2019. Graph Attention Networks for Neural Social Recommendation. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*.

[24] Ryan Prenger, Rafael Valle, and Bryan Catanzaro. 2019. Waveglow: A Flow-based Generative Network for Speech Synthesis. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.

[25] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. *CoRR* abs/1409.4842 (2014).

[26] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2017. Graph Attention Networks. *CoRR* abs/1710.10903 (2017).

[27] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In *Proceedings of the Conference of the ACM Special Interest Group on Knowledge Discovery & Data Mining (SIGKDD)*.

[28] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *CoRR* abs/1909.01315 (2019).

[29] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019).

[30] Zhipu Xie, Weifeng Lv, Shangfo Huang, Zhilong Lu, Bowen Du, and Runhe Huang. 2020. Sequential Graph Neural Network for Urban Road Traffic Speed Prediction. *IEEE Access* 8 (2020), 63349–63358.

[31] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[32] Hongxia Yang. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. In *Proceedings of the Conference of the ACM Special Interest Group on Knowledge Discovery & Data Mining (SIGKDD)*.

[33] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *Proceedings of the IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*.