Accelerating Training and Inference of Graph Neural Networks with Fast Sampling and Pipelining

Tim Kaler [* 1 2] **Nickolas Stathas** [* 1 2] **Anne Ouyang** [* 1 2 3] **Alexandros-Stavros Iliopoulos** [1 2] **Tao B. Schardl** [1 2] **Charles E. Leiserson** [1 2] **Jie Chen** [2 3]

Abstract

Improving the training and inference performance of graph neural networks (GNNs) is faced with a challenge uncommon in general neural networks: creating mini-batches requires a lot of computation and data movement due to the exponential growth of multi-hop graph neighborhoods along network layers. Such a unique challenge gives rise to a diverse set of system design choices. We argue in favor of performing mini-batch training with neighborhood sampling in a distributed multi-GPU environment, under which we identify major performance bottlenecks hitherto under-explored by developers: mini-batch preparation and transfer. We present a sequence of improvements to mitigate these bottlenecks, including a performance-engineered neighborhood sampler, a shared-memory parallelization strategy, and the pipelining of batch transfer with GPU computation. We also conduct an empirical analysis that supports the use of sampling for inference, showing that test accuracies are not materially compromised. Such an observation unifies training and inference, simplifying model implementation. We report comprehensive experimental results with several benchmark data sets and GNN architectures, including a demonstration that, for the ogbn-papers100M data set, our system SALIENT achieves a speedup of $3\times$ over a standard PyTorch-Geometric implementation with a single GPU and a further $8\times$ parallel speedup with 16 GPUs. Therein, training a 3-layer GraphSAGE model with sampling fanout (15, 10, 5) takes 2.0 seconds per epoch and inference with fanout (20, 20, 20) takes 2.4 seconds, attaining test accuracy 64.58%.

1 Introduction

Graph neural networks (GNNs) have emerged as an important class of methods for leveraging graph structures in machine learning (Li et al., 2016; Kipf & Welling, 2017; Hamilton et al., 2017; Veličković et al., 2018; Xu et al., 2019; Morris et al., 2019). The graph structure encodes dependencies in data representations across layers of the GNN, injecting an effective relational inductive bias into the neural network design. GNNs have been shown to be successful in (semi-)supervised, unsupervised, self-supervised, and reinforcement learning (Kipf & Welling, 2017; Hu et al., 2020b; Ma & Chen, 2021; Mirhoseini et al., 2021) and have been applied in a number of domains including commerce, finance, traffic, energy, and pharmacology (Gilmer et al., 2017; Li et al., 2018; Ying et al., 2018; Weber et al., 2019; Shang et al., 2021). As graph sizes continue to grow rapidly, there is a pressing need for efficient training and inference to facilitate further study and deployment of GNNs.

One unique challenge to GNNs is the exponential increase of neighborhood size with respect to the number of network layers (i.e., hops along the input graph) (Chen et al., 2018). In a typical neural-network training scenario with stochastic gradient descent methods, e.g., Adam (Kingma & Ba, 2015), computations are organized around mini-batches: a mini-batch of training data is fed to the network to calculate the corresponding loss and gradient, which is then used to update the model parameters. Similarly, for inference, input data are processed in successive mini-batches. In GNNs, where the representation of a data point (i.e., graph node) depends on those of its neighbors, processing a mini-batch may lead to a prohibitively large expanded neighborhood. Apart from the computational cost this incurs, the features and intermediate representations of nodes in the expanded neighborhood also consume substantial memory. When using accelerators such as GPUs, the neighborhood data size can in fact exceed the accelerator memory capacity. To mitigate this issue, neighborhood sampling is a popular remedy, sometimes even a necessary rescue (Hamilton et al., 2017; Chen et al., 2018; Ying et al., 2018; Zou et al., 2019; Zeng et al., 2020; Ramezani et al., 2020; Dong et al., 2021).

In this work, we focus on GNNs with neighborhood sampling on GPUs and identify batch preparation and transfer

*Equal contribution [1]MIT CSAIL [2]MIT-IBM Watson AI Lab [3]IBM Research. Correspondence to: Jie Chen <chenjie@us.ibm.com>.

as major bottlenecks in commonly used GNN frameworks, e.g., PyTorch-Geometric (PyG) (Fey & Lenssen, 2019) and the Deep Graph Library (DGL) (Wang et al., 2019). Batch preparation entails expanding the sampled neighborhood for a mini-batch of nodes and slicing out the feature vectors of all involved nodes. The corresponding subgraph and feature vectors must then be transferred to the GPUs, since the entire graph and feature data are often too large to fit in GPU memory. Somewhat surprisingly, batch preparation and transfer take substantially longer than the core GNN training operations (loss, gradient, and model parameter computation). The latter are computed in the GPU and benefit from highly optimized libraries (e.g., BLAS (Lawson et al., 1979) and autograd (Paszke et al., 2017)). To fully reap their benefits and maintain high GPU utilization, the throughput of batch preparation and transfer needs to be increased substantially. Scaling up to use multiple GPUs makes the need for improvement even greater.

To resolve this challenge, this work presents three performance optimizations which are broadly applicable to current GNN architectures and frameworks. The first is a *fast neighborhood sampler*. We show a principled approach to exploring the space of applicable optimizations and identify settings which perform well across CPU architectures. The second is *shared-memory parallelization* for batch preparation to circumvent CPU utilization and memory bandwidth bottlenecks present in PyG and DGL. The third is *pipelined batch transfer and computation* to increase GPU utilization.

The effect of these optimizations is shown in Figure 1, which illustrates the timeline of mini-batch computations across CPU and GPU resources for a standard PyTorch workflow with and without the optimizations. The three optimizations respectively improve the CPU throughput of neighborhood sampling and expansion (green boxes in Figure 1); reduce slicing overhead (yellow boxes); and enable overlapped GPU transfers and computations (red and blue boxes). With a reasonably high CPU-to-GPU ratio, as is often the case in modern computing clusters, these optimizations almost eliminate GPU idle time, enabling fast training at a speed commensurate with that of the core training operations.

Additionally, this work studies inference. Although tradeoffs among accuracy, speed, and memory requirements have been studied extensively for training, they are relatively under-studied for inference. We conduct an empirical analysis that indicates that neighborhood sampling in inference sacrifices prediction accuracy only marginally. This suggests that mini-batch inference with neighborhood sampling is a viable alternative to layer-wise inference with full neighborhoods, yielding accuracy comparable to the latter but with a much lower memory footprint. As an added advantage, model architecture code can be reused between training and inference, simplifying development.

Our system, SALIENT, addresses and alleviates bottlenecks in SAmpling, sLIcing, and data movemENT. SALIENT's optimizations are all done over standard GNN code written in PyG, retaining the neural network module, the optimizer, and the distributed data-parallel (DDP) framework for training on multiple machines. This implementation minimizes adoption barriers for developers, who can persist with their familiar deep learning frameworks and focus on modeling and applications (e.g., experimenting with neural architectures), without being distracted by new modules and APIs.

We highlight the following contributions:

1. A careful analysis of GNN training codes operating on large graphs, identifying performance bottlenecks unique to GNNs in batch preparation and data transfer.

2. Design of an efficient batch preparation system called SALIENT that alleviates GNN training bottlenecks with broadly applicable optimizations to neighborhood sampling and GPU training workflows. We show that these improvements lead to near-optimal GPU utilization.

3. An implementation of SALIENT, whose compatibility with standard PyG code facilitates use by GNN researchers, developers, and practitioners.

4. An empirical study that suggests neighborhood sampling in inference need not sacrifice accuracy, while reducing memory usage and simplifying code development.

5. An evaluation of the end-to-end training performance of SALIENT on three benchmark data sets and four GNN architectures in both single- and multi-GPU settings. For the largest data set, ogbn-papers100M, with a 3-layer GraphSAGE model and sampling fanout (15, 10, 5), we show a training speedup of $3\times$ over a standard PyG implementation run on one GPU and a further $8\times$ speedup on 16 GPUs. Therein, training takes 2.0 seconds per epoch and inference with sampling fanout (20, 20, 20) takes 2.4 seconds, attaining test accuracy 64.58%.

2 BACKGROUND

2.1 Graph neural networks

The class of *message passing neural networks (MPNNs)* (Gilmer et al., 2017) encompasses a large number of GNNs. Let $G = (V, E)$ be a graph with node set V and edge set E. Let $X \in \mathbb{R}^{n \times f}$ be the feature matrix whose rows are node feature vectors (denoted by $x_v \in \mathbb{R}^f$ for node v). Let $\ell = 1, \ldots, L$ denote the layer index and $\mathcal{N}(v) = \{u \mid (u, v) \in E\}$ denote the one-hop neighborhood of v. MPNNs are based on the following update rule:

$$h_v^\ell = \text{UPD}^\ell \left(h_v^{\ell-1}, \text{AGG}^\ell \left(\{ h_u^{\ell-1} \mid u \in \mathcal{N}(v) \} \right) \right), \quad (1)$$

where h_v^ℓ is the layer-ℓ representation of v, AGG^ℓ is a set aggregation function, and UPD^ℓ is the update function. Ini-

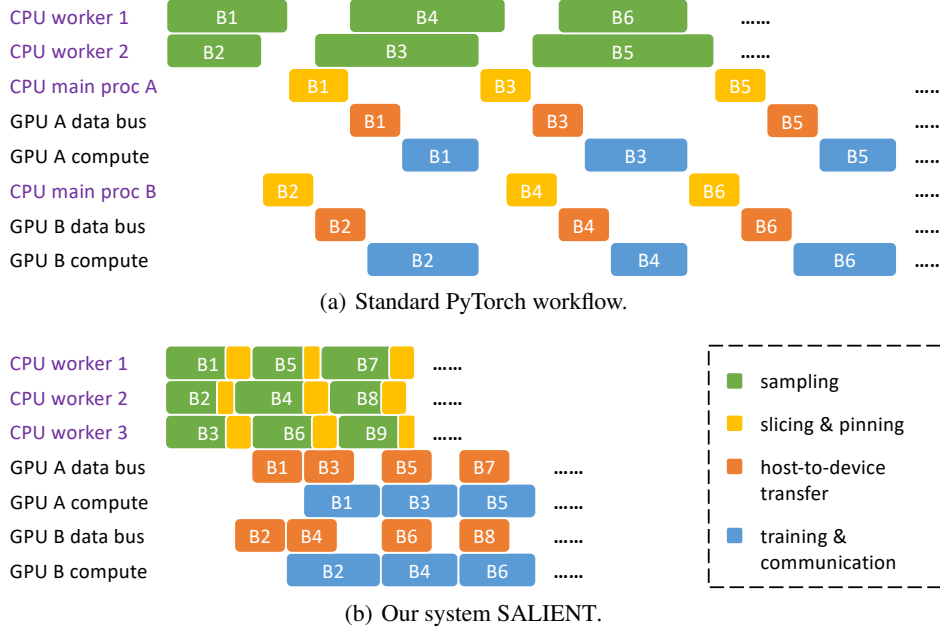(a) Standard PyTorch workflow.

(b) Our system SALIENT.

Figure 1. Illustration of mini-batch progress per training epoch: comparison between a standard PyTorch workflow and SALIENT, the optimized system detailed in this paper. The x-axis represents elapsed time. The "Bi" blocks refer to operations with the i-th mini-batch; different operations with the same batch are distinguished by color. In modern computing clusters, the number of available CPU cores is often much greater than the number of GPUs, hence CPU workers may prepare batches in parallel to try and saturate the GPUs. With respect to Listing 1, green boxes correspond to lines 1–2, yellow boxes to lines 3–4, orange boxes to lines 5, and blue boxes to lines 6–8.

tially, $h_v^0 = x_v$. After L layers of updates, h_v^L becomes the final representation. Overall, v's layer-ℓ representation h_v^ℓ depends on the previous-layer representations of v and its neighbors.

GNNs differ in their design of the two functions in (1). For example, in GraphSAGE (Hamilton et al., 2017), AGG$^\ell$ is a mean, LSTM, or pooling operator, and UPD$^\ell$ concatenates the two arguments and applies a linear layer. In GIN (Xu et al., 2019), AGG$^\ell$ is the sum of $\{h_u^{\ell-1}\}$ and UPD$^\ell$ is the sum of its arguments followed by an MLP. In GAT (Veličković et al., 2018), AGG$^\ell$ is the identity and UPD$^\ell$ computes h_v^ℓ as a weighted combination of $W^{\ell-1}h_u^{\ell-1}$ for all $u \in \{v\} \cup \mathcal{N}_v$, where the weights are attention coefficients and $W^{\ell-1}$ is the parameter matrix of the layer.

2.2 Neighborhood sampling

From (1), one sees that computing v's representation requires recursively inquiring neighbors, which may incur a prohibitively large L-hop neighborhood; similarly for a mini-batch of nodes. Restricting the neighborhood size via sampling proves to be an effective training strategy for improving memory and time efficiency. Current sampling approaches generally fall under three categories: node-wise sampling, layer-wise sampling, and subgraph sampling.

Node-wise sampling approaches, including Graph-SAGE (Hamilton et al., 2017) and PinSage (Ying et al., 2018), modify the neighborhood $\mathcal{N}(v)$ in (1) by taking a random subset containing at most d neighbors, sampled without replacement. It is typical to specify a different sample size (called *fanout*), d^ℓ, for each layer ℓ. The fanout

parameters serve as an upper bound on the effective degree during neighborhood expansion.

Layer-wise sampling approaches collect the neighbors of all nodes in a mini-batch and then sample the entire neighborhood for the batch. Sampling proceeds recursively layer by layer. Representative methods are FastGCN (Chen et al., 2018) and LADIES (Zou et al., 2019). These approaches impose a nontrivial sampling distribution over the neighborhood and rescale the neighbor representations through dividing them by their respective sampling probability, to preserve unbiasedness before activation. Nonlinear activation functions will destroy unbiasedness anyway, but training convergence results can still be established based on asymptotic consistency (Chen & Luss, 2018).

Subgraph sampling approaches, such as Cluster-GCN (Chiang et al., 2019) and GraphSAINT (Zeng et al., 2020), sample a connected subgraph and compute mini-batch loss restricted to this subgraph.

There exist other types of sampling-related approaches as well. Authors of LazyGCN (Ramezani et al., 2020) study the promise of lowering sampling frequency and propose a "lazy" sampling schedule that is applicable to all of the above categories. Inspired by LazyGCN, authors of GNS (Dong et al., 2021) further propose caching a random but sufficiently large subgraph, from which node-wise sampling is performed for each training epoch.

2.3 GNN training systems

At the system level, due to the unique characteristics of GNNs, many efforts have been made to develop training

systems scalable to large graphs, based on either mainstream deep learning frameworks or more specialized systems.

Roc (Jia et al., 2020) and DeepGalois (Hoang et al., 2021) are examples of the latter, both of which perform full-batch training as opposed to mini-batch. Also perform full-batch training are NeuGraph (Ma et al., 2019), which is based on TensorFlow (Abadi et al., 2015); and FlexGraph (Wang et al., 2021a), Seastar (Wu et al., 2021), and GNNAdvisor (Wang et al., 2021b), which are based on PyTorch (Paszke et al., 2019). On the other hand, DistDGL (Zheng et al., 2020), Zero-Copy (Min et al., 2021), GNS (Dong et al., 2021), and P^3 (Gandhi & Iyer, 2021) are based on PyTorch and the DGL module and all perform mini-batch training. In the referenced publications, the authors report results on multiple machines (CPUs only), single machines with multiple GPUs, or single machines with a single GPU. Our system, SALIENT, is based on PyTorch and the PyG module and performs mini-batch training. We demonstrate results on a single machine with a single GPU as well as multiple machines with multiple GPUs each.

3 PERFORMANCE CHARACTERISTICS OF NEIGHBORHOOD SAMPLING IN GNNS

This section summarizes our investigation into the performance bottlenecks in standard implementations of GNNs in PyTorch. Our findings underscore the gap between hardware capabilities and actualized performance and motivate the optimizations in SALIENT, which are detailed in Section 4.

For this performance study, we use as reference a standard 3-layer GraphSAGE architecture implemented in PyG, running on a 20-core Intel Xeon Gold 6248 CPU and a single NVIDIA Volta V100 GPU. At a high level, the baseline implementation for our study includes the following operations, with corresponding pseudocode in Listing 1:

1. **Batch preparation:** Sample a multi-hop neighborhood for a given mini-batch, and slice features and label tensors to obtain subtensors for nodes in the sampled neighborhood. (Lines 1–4)
2. **Data transfer:** Transfer the prepared batch (a sampled neighborhood and sliced tensors) to the GPU. (Line 5)
3. **GPU training:** Perform model evaluation, back propagation, and model update on the GPU. (Lines 6–8)

The baseline PyG code was written to be a good representation of a performance-tuned code using standard libraries. It includes the following conventional optimizations: (i) row–major representation of the feature matrix to improve CPU cache efficiency in slicing operations; (ii) CPU-to-GPU transfers via pinned memory to enable asynchronous transfer with direct memory access; and (iii) half-precision floating point for feature vectors in host memory to reduce bandwidth pressure in slicing and CPU-to-GPU data transfers

```
1  ns = NeighborSampler(G, fanouts, batch_sz)
2  for Gs, ids in ns: # A sampled subgraph Gs
3    xs, ys = x[ids],y[ids[:batch_sz]] # Slice
4    batch = (xs, ys, Gs)
5    batch = batch.to(GPU) # Transfer to GPU
6    optimizer.zero_grad() # Train on GPU
7    loss_fn(model(batch), ys).backward()
8    optimizer.step()
```

Listing 1. Reference pseudocode for a standard PyTorch implementation of GNN training with neighbor sampling on graph G with node features x and labels y.

Table 1. Per-operation performance breakdown of the baseline PyG training code. Reported runtimes correspond to blocking or non-overlapped computations among the steps outlined in Listing 1. GNN: 3-layer GraphSAGE with fanouts (15,10,5), hidden-layer feature dimensionality 256, and mini-batch size 1024. Data sets are introduced in Section 6.

Data Set	Epoch	Batch Prep.		Transfer		Train (GPU)	
	time	time	%	time	%	time	%
arxiv	1.7s	1.0s	58%	0.3s	15%	0.5s	27%
products	8.6s	4.0s	46%	2.2s	26%	2.4s	28%
papers	50.4s	18.6s	37%	17.9s	35%	13.9s	28%

Table 2. Breakdown of an ogbn-products epoch batch preparation time for PyG and SALIENT with P threads on 20 cores. Note that for PyG *Both* column, sampling and slicing occur asynchronously, each using P threads (thus $2P$ in total). SALIENT uses only P threads.

P	PyG			SALIENT		
	Sampling	Slicing	Both	Sampling	Slicing	Both
1	71.1s	7.6s	72.7s	28.3s	7.3s	35.6s
10	11.4s	1.6s	11.5s	3.3s	0.8s	4.1s
20	7.2s	1.2s	7.3s	1.9s	0.6s	2.5s

(GPU training computations are still done in single precision). In our experiments, these optimizations yield a roughly 2× speedup per epoch over a naive PyG implementation of Listing 1 or about 1.5× over a reference DGL benchmark.[1] The resulting code, hereafter referred to simply as "PyG," serves as the baseline for our performance evaluations and the starting point for SALIENT.

3.1 Observed per-operation performance

We benchmarked the per-epoch runtime of the baseline PyG code by recording the time required to execute each operation summarized in Listing 1. Our benchmarks show that batch preparation and CPU-to-GPU data transfers severely bottleneck training performance. Table 1 provides a per-

---

[1]GitHub repo: dglai/dgl-0.5-benchmark.

formance breakdown on three publicly available data sets: ogbn-arxiv, ogbn-products, and ogbn-papers100M (see Section 6 for details). The reported runtime for each operation is the amount of time spent on it from the perspective of the main thread executing the Python code. In other words, we report the *blocking* time for each operation, which is lower than its individual runtime due to computation overlap (see Figure 1(a)). Across all three data sets, only about 28% of the time is spent on GPU training. Most of the time is spent preparing batches and transferring data to the GPU.

3.2 Performance analysis of batch preparation

Batch preparation comprises two steps: (a) neighborhood *sampling* to obtain the mini-batch induced subgraph, and (b) *slicing* the feature and label tensors to extract the parts that correspond to the sampled subgraph. Both steps are parallelized: sampling uses a PyTorch DataLoader and multiprocessing, and slicing uses multiple OpenMP threads in a single process. The relative performance of sampling and slicing is not easily obtained from per-line measurements, as sampling is performed asynchronously with the main execution thread. As such, we investigate the performance of sampling and slicing using separate targeted benchmarks.

Table 2 breaks down the performance of sampling and slicing on ogbn-products for PyG. Batch preparation time is dominated by the neighborhood sampling time, requiring 7.2 seconds with 20 worker processes. Slicing, by comparison, takes just 1.2 seconds when parallelized with 20 OpenMP threads using PyTorch's parallel slicing code.

Even a conservative analysis of the performance breakdown in Tables 1 and 2 implies that neighborhood sampling is a substantial bottleneck in GNNs. For PyG to perform sampling at a pace that can keep a single GPU busy and hide sampling latency on ogbn-products, sampling throughput must be improved by at least $3\times$. When using multiple GPUs per machine, the required speedup is higher. Sections 4.1 and 4.2 discuss how SALIENT improves the performance of sampling and slicing to alleviate this bottleneck and achieve substantially higher batch preparation throughput, as previewed in the *SALIENT* columns of Table 2.

3.3 Data transfer performance

Data transfer from CPU to GPU is another bottleneck, accounting for 15–35% of the epoch time in the benchmarks of Table 1. Data transfer generally takes longer as expanded neighborhoods get larger, as seen with ogbn-papers100M, or as feature dimensionality increases.

There is potential to improve transfer time without also reducing the amount of transferred data any further. During a typical epoch with ogbn-papers100M, a total of 164GB are transferred from CPU to GPU. The peak DMA CPU-to-

GPU transfer rate on our machine is 12.3GB/s. Per Table 1, the baseline implementation attains an effective data transfer rate of 9.2GB/s or about 75% of peak. One can achieve near-optimal data transfer rates with pipelining and the elimination of redundant round-trip communications. These optimizations are discussed in Section 4.3.

4 SALIENT

We propose SALIENT, a system for fast distributed data-parallel GNN training (and inference; see Section 5) using neighborhood sampling. SALIENT combines the following features to achieve high performance:

a) an optimized implementation of neighborhood sampling and expansion;

b) an efficient parallel batch preparation scheme;

c) CPU-to-GPU data transfer optimizations that hide latency and saturate data bus bandwidth; and

d) seamless compatibility with PyTorch's DDP module to scale across multiple GPUs and machines.

Notably, SALIENT achieves the above without requiring disruptive changes to user-facing APIs. SALIENT provides a drop-in replacement for the NeighborSampler and slicing code presently used in PyG.

4.1 Fast neighborhood sampling

The base algorithm for node-wise sampling, implemented in the NeighborSampler module of PyG, is as follows. We are given an input graph G, a set of nodes $V_b = \{v_1, \ldots, v_k\}$ which define a mini-batch, and a fanout d. For each node $v_i \in V_b$, we sample d of its neighbors without replacement to get the sampled neighborhood $\mathcal{N}_d(v_i)$. The sampled neighborhoods are typically organized into a bipartite graph with source nodes $\bigcup_i \mathcal{N}_d(v_i)$ and destination nodes V_b. For multi-hop neighborhoods, the process is repeated for each source node, yielding a sequence of bipartite graphs. Together, these comprise a *message-flow graph* (MFG) for the mini-batch of nodes in V_b.

This simple algorithm for neighborhood sampling admits a variety of design and implementation choices, which may have a dramatic impact on performance. Among the most impactful ones are: a data structure for global-to-local node ID mapping between the input graph and sampled MFG; a set data structure to support neighbor sampling without replacement; and fusing the operations of sampling and MFG construction. Overall, the space of possible design choices and optimizations is too large to explore manually. We designed a parameterized implementation of sampled MFG generation to systematically explore this optimization space and identify the ones that yield high performance across compute architectures.
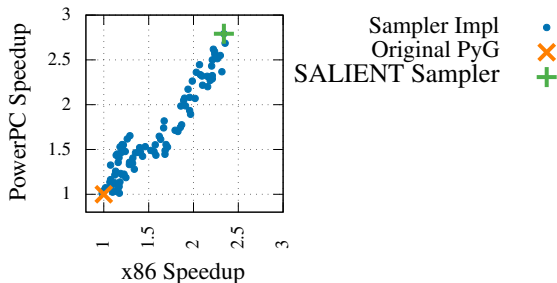
Figure 2. Exhaustive exploration of optimization parameters.

This exploration was done using a microbenchmark which executed the parameterized code on a reference hop-by-hop trace of the nodes which made up a sampled MFG for a mini-batch of nodes in ogbn-products. To mitigate sampling variability, we benchmark each individual hop of the reference trace instead of an end-to-end execution. Figure 2 shows the performance, relative to the PyG NeighborSampler implementation, of 96 instantiations of the parameterized code on two CPU architectures (x86 and PowerPC).

Analyzing the results shows that the most impactful changes, relative to the baseline PyG code, are related to data structures. Changing the C++ STL hash map and hash set to a flat swiss-table implementation (Benzaquen et al., 2018) yields a 2× speedup. Using an array instead of a hash table for the set provides a further 17% improvement. Despite its linear search complexity, the array set benefits from cache locality. As Table 2 shows, the SALIENT implementation of neighborhood sampling is 2.5× faster than that of PyG.

4.2 Shared-memory parallel batch preparation

SALIENT parallelizes batch preparation through the use of shared-memory multithreading. Shared-memory parallelization has several key advantages over PyTorch's multiprocessing, including lower synchronization overheads and, critically, the ability to perform zero-copy communication with the main training process.

To parallelize batch preparation across mini-batches, SALIENT uses C++ threads which prepare batches end-to-end, each performing sampling and slicing sequentially. Since these threads run C++ code, they are not affected by Python's global interpreter lock. By using a serial tensor-slicing code, which is otherwise parallelized in PyTorch by default, SALIENT improves cache locality and avoids contention between threads. Threads balance load dynamically via a lock-free input queue that contains the destination nodes for each mini-batch. We find that dynamic load balancing generally performs better than static partitioning schemes such as those in the PyTorch DataLoader due to the variation in final neighborhood size across mini-batches.

A particularly impactful optimization enabled by shared-

Table 3. Impact of SALIENT optimizations on per-epoch runtime.

Optimization	Per-Epoch Runtime		
	arxiv	products	papers
None (PyG)	1.7s	8.6s	50.4s
+ Fast sampling	0.7s	5.3s	34.6s
+ Shared-memory batch prep.	0.6s	4.2s	27.8s
+ Pipelined data transfers	0.5s	2.8s	16.5s

memory parallelization is the ability to perform slicing while the main process is blocked on GPU training. A batch preparation thread writes sliced tensors directly into pinned memory accessible by the main process. By comparison, slicing in PyTorch multiprocessing workers would require copying the sliced data from the worker process to the main process via POSIX shared memory, effectively halving the observed memory bandwidth and inhibiting parallel scaling.

4.3 Data transfer pipelining

Data transfers account for 15–35% of per-epoch time as shown in Table 1. To mitigate this bottleneck, SALIENT employs two optimizations to minimize data transfer latency and overlap data transfer with GPU computation.

As discussed in Section 3.3, data transfers for PyG on ogbn-papers100M are only 75% efficient. Detailed profiling reveals redundant CPU-GPU round trips which create idle time between data transfers of the MFG edges. These round trips are attributed to assertions in PyG's sparse tensor library that check the validity of the sparse adjacency matrix after it is transferred. These blocking assertions are unnecessary for data transfers, since they have already been performed when the sparse tensor was constructed on the CPU. Adding an option to skip assertions in such circumstances allows us to achieve 99% of peak data transfer throughput.

SALIENT further increases GPU utilization by overlapping data transfers with GPU training computations. Specifically, SALIENT uses separate GPU streams for computation and data transfer, synchronizing those streams to ensure a training iteration begins after the necessary data is transferred. With SALIENT's optimizations to improve the throughput of batch preparation and transfer, these operations generally take less time than the GPU training computations. Consequently, overlapping transfers with GPU computations nearly eliminates latency outside the GPU computations.

4.4 Summary

Our design decisions in SALIENT are informed by a careful analysis of existing bottlenecks in standard workflows. We find that it is possible to get a highly efficient system with targeted optimizations in neighborhood sampling, shared-memory parallelization for slicing directly into pinned mem-
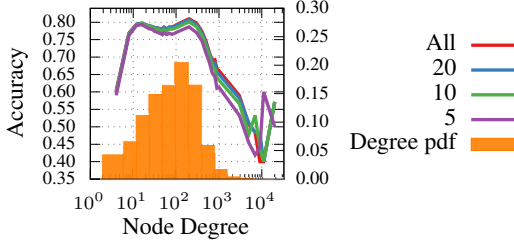
Figure 3. Test accuracy and node count versus node degree. Data set: ogbn-products; GNN: GraphSAGE. Legend: "all" indicates full neighborhood (non-sampling); number indicates sampling fanout for each layer.

Table 4. Summary of data sets.

Data Set	#Nodes	#Edges	#Feat.	Train. / Val. / Test
arxiv	169K	1.2M	128	91K / 30K / 48K
products	2.4M	62M	100	197K / 39K / 2.2M
papers	111M	1.6B	128	1.2M / 125K / 214K

ory, and pipelining data transfer and GPU computations. Figure 1(b) illustrates the timeline of GNN training with SALIENT and contrasts it to that of a standard PyTorch workflow (Figure 1(a)). Table 3 quantifies this comparison, listing the incremental impact of each optimization category. These optimizations do not require fundamental changes to the basic workflow structure and are orthogonal to other improvements in the training process itself.

5 INFERENCE WITH SAMPLING

While neighborhood sampling is extensively used for training, it is unclear if this approach compromises prediction accuracy in inference. Note that these two phases are rather different in nature. The goal of training is to optimize a loss function and identify model parameters, whereas the goal of inference is to predict labels for the test-set nodes. In deep learning, de facto choices of optimizers are stochastic gradient methods, where the loss function and the prediction need not be accurately evaluated in every gradient step to achieve convergence; e.g., the mini-batch gradient is only an estimator of, but is not exactly, the loss gradient. As long as sampling is done sufficiently many times, the average will converge to the probability expectation. Sampling in inference. however, is one-shot and the sample average may be rather different from the mean. Will sampling produce predictions as accurate as the case of non-sampling?

Theoretical analysis is beyond our scope, but we investigate empirical data. As a typical example, Figure 3 shows the degree distribution of the test set of ogbn-products overlaid with the prediction accuracy distribution obtained by using a 3-layer GraphSAGE architecture. One observes that when the full neighborhood is used, high-degree nodes tend to be predicted less accurately, but such nodes are few in the test set. In other words, it suffices to maintain the prediction quality of the low-degree nodes to achieve a comparable overall accuracy. Moreover, the figure clearly shows that a small sampling fanout already approximates well the left half of the accuracy distribution. As the fanout increases, the right half is approximated increasingly well, too.

For this reason, we apply neighborhood sampling to inference as well. It enjoys several advantages. First, it allows reusing the model architecture code and a majority of the mini-batch training code. Second, it reduces memory consumption. Because of the explosive size of multi-hop neighborhoods, a mini-batch is unlikely to fit in GPU memory without sampling. Then, inference must be conducted alternatively by evaluating the network layer by layer and storing layer-wise results in host memory. For some model architectures (e.g., dense connections), all layer results must be stored, demanding multiple times more storage. Finally, as opposed to the layer-by-layer approach, mini-batch inference can trivially be run on a select subset of nodes and can be executed in a distributed data parallel context.

6 EVALUATION

We conduct a comprehensive set of experiments to evaluate the performance of SALIENT and demonstrate substantial improvement over a baseline performance-engineered PyG implementation. All experiments are conducted on a cluster of compute nodes in a 10GigE network, each equipped with two 20-core Intel Xeon Gold 6248 CPUs, 384GB DRAM, and two NVIDIA V100 GPUs (32GB RAM). The benchmarking is based on PyTorch 1.8.1 and PyG 1.7.0. The C++ code for batch preparation is compiled with GCC 7.5.0 and optimization flags `-O3 -march=native`.

Data sets. We evaluate on three standard benchmark data sets: ogbn-arxiv, ogbn-products, and ogbn-papers100M (Hu et al., 2020a). The graph and training set in these data sets vary in size, with ogbn-papers100M being one of the largest open benchmarks at the time of this work. See Table 4 for detailed information. All graphs were made undirected (if originally not) as is common practice.

GNN architectures. We experiment with a variety of architectures to demonstrate the wide applicability of SALIENT: GraphSAGE (Hamilton et al., 2017), GAT (Veličković et al., 2018), GIN (Xu et al., 2019), and GraphSAGE-RI. The latter adds residual connections to GraphSAGE and employs an Inception-like structure for final prediction.[2] Details for each GNN atchitecture are given in the appendix. Table 5 lists key hyperparameters that impact training time and accuracy. All experimental results for ogbn-papers100M, except

---

[2]This architecture is similar to that in the GitHub repo mengyangniu/ogbn-papers100m-sage.

Table 5. GNN hyperparameters for our experiments. Fanout is for training. For inference fanout, see Table 6. Batch size is per GPU.

Data Set	GNN	#Layers	Hidden	Fanout	Batch
arxiv	SAGE	3	256	(15, 10, 5)	1024
products	SAGE	3	256	(15, 10, 5)	1024
papers	SAGE	3	256	(15, 10, 5)	1024
papers	GAT	3	256	(15, 10, 5)	1024
papers	GIN	3	256	(20, 20, 20)	1024
papers	SAGE-RI	3	1024	(12, 12, 12)	1024

Table 6. Test accuracy under various neighborhood fanouts for inference. GNN: GraphSAGE with training fanout (15, 10, 5). For ogbn-papers100M, the "fanout: all" case runs out of memory and we report the result with fanout (100, 100, 100) instead.

Data Set	Accuracy			
	fanout: all	*(20, 20, 20)*	*(10, 10, 10)*	*(5, 5, 5)*
arxiv	.7074±.005	.7054±.005	.6980±.005	.6849±.004
products	.7749±.004	.7755±.003	.7708±.003	.7558±.003
papers	.6491±.005*	.6458±.004	.6379±.004	.6163±.005

Figure 4. Performance improvement of SALIENT over standard PyG workflow. Timing measurements on one machine with one GPU. GNN: GraphSAGE with fanout $(15, 10, 5)$.

for Figure 6, are obtained with GraphSAGE.

Single-GPU improvement over PyG. We first compare the performance of SALIENT and PyG on a single GPU. Figure 4 suggests a $3\times$ to $3.4\times$ speedup across data sets, owing to the diminishing percentage of time blocked on sampling and data transfer. SALIENT's optimizations improve the overall efficiency of these stages, and its pipelined design results in the overall per-epoch runtime being nearly equal to the GPU compute time for training.

Multi-GPU scaling. We now scale the training to multiple GPUs. A maximum of 16 GPUs are used, spanning eight machines. The effective batch size is proportional to the number of GPUs. SALIENT straightforwardly applies the PyTorch DDP module and performs distributed communications with the NCCL backend. Figure 5 shows generally good scaling in the distributed setting. Larger data sets, such as ogbn-papers100M, tend to see greater parallel speedup due to having higher computational density and larger training sets. As such, bigger graphs amortize the latency of starting an epoch (e.g., the time to prepare the first sets of mini-batches) over a greater amount of work per GPU. The sampled neighborhoods of batches also tend to be larger for bigger and well-connected graphs, which increases the amount of GPU computations per mini-batch and better

shadows communication and synchronization overheads. With 16 GPUs, the speedup ranges from $4.45\times$ to $8.05\times$.

Neighborhood sampling for inference. We investigate the effectiveness of applying neighborhood sampling for inference. Table 6 lists the test accuracies for all data sets either using either full or sampled neighborhoods. Each accuracy result is obtained through five repetitions of training and inference. Full-neighborhood inference uses layer-wise computation and stores intermediate layer results in host memory. One observes that for the ogbn-arxiv and ogbn-products data sets, a fanout of 20 for each layer is sufficient to match full-neighborhood accuracy. For ogbn-papers100M, layer-wise inference with full neighborhood runs out of memory. Hence, we report the accuracy with fanout 100 instead. We see that the accuracy has been saturated and conclude that fanout 20 is sufficient for this data set as well.

Performance of varying GNNs. A feature of SALIENT is that the GNN architecture implementation is independent of performance engineering in batch preparation and transfer. Hence, a PyG developer can keep using exactly the same API to design and tune GNNs. This feature brings in the benefit of fast prototyping for an application. We experiment with a number of architectures and report the training time (with 16 GPUs) and test accuracy for the largest data set ogbn-papers100M in Figure 6.

Several observations follow. First, the training time for different architectures varies significantly, affected by multiple factors such as the complexity of the architecture and the choice of hyperparameters. Second, speedup over PyG also varies significantly. Computation density (relative to data transfer and communication) is highest for GraphSAGE-RI, medium for GAT and GIN, and lowest for GraphSAGE. GraphSAGE enjoys the greatest improvement (approximately $2.3\times$) due to our performance engineering on sampling and transfer, while GraphSAGE-RI and GAT have the least improvement, which however is still over $1.4\times$. Third, architectures achieve different accuracies. With only moderate tuning, GraphSAGE-RI performs noticeably better than the other three. These accuracy numbers are on par with those appearing in the literature or public GitHub repos.
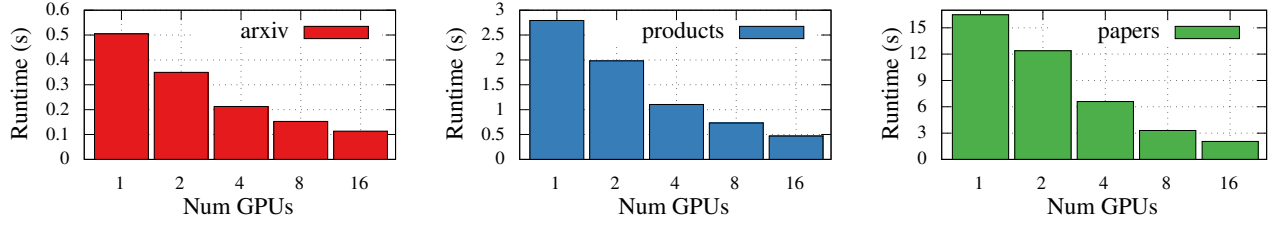
Figure 5. Epoch time when scaling to multiple GPUs with proportionately scaled batch size using the SAGE architecture from Table 5.
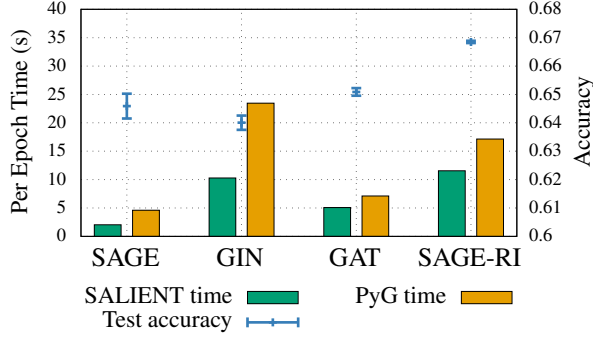

Figure 6. Per epoch training time and test accuracy after 25 epochs for ogbn-papers100M on several GNN models, trained by using 16 GPUs. Test inference fanouts were $(20, 20, 20)$ for SAGE, GIN, and GAT, and $(100, 100, 100)$ for SAGE-RI.

7 COMPARISON WITH EXISTING SYSTEMS

It is important to put the results in Section 6 in perspective. Table 7 summarizes the reported performance of several representative systems. On the largest data set, ogbn-papers100M, our 2.0s per-epoch training time is orders of magnitude faster than that of the listed systems. This record, however, is achieved in an incomparable environment (differing in hardware, software framework, model architecture, or batching scheme) from those of existing systems. Since most referenced systems are not publicly available or readily usable, we note a few differentiating points.

We adopt mini-batch training, as opposed to full-batch training appearing in several prior systems. One reason is that the former converges faster and generalizes better (Bottou et al., 2018). On the system level, these two batching schemes have drastically different computation patterns and may suffer different bottlenecks.

SALIENT is built on PyTorch and PyG, a framework less used in system-oriented publications. We consider that PyG enjoys a large user base[3] and it benefits from a demonstration of improvement that encourages widespread attraction.

---

[3]At the time of this writing, pyg-team/pytorch_geometric has 12.8K stars and 2.2K forks on GitHub, while dmlc/dgl has 8.2K stars and 1.8K forks and alibaba/euler has 2.7K stars and 534 forks.

Meanwhile, it should be noted that SALIENT's optimizations are general and can be applied to other frameworks.

We demonstrate experiments in a multi-machine, multi-GPU environment, with attractive speedup, using PyTorch's DDP module for distributed training. Most of the systems summarized in Table 7 demonstrate no such experiments and/or are not readily extensible to such an environment. The only exception, adopting mini-batch training, is P^3 (Gandhi & Iyer, 2021). This system addresses a different bottleneck than we do—the communication cost and partitioning overhead. The techniques proposed therein are independent of SALIENT and can be incorporated into our sampling pipeline for a further efficient system.

8 CONCLUSIONS AND FUTURE WORK

In this work, we identify major bottlenecks in GNN training and inference—batch preparation and transfer—and propose three complementary improvements, namely optimized neighborhood sampling, shared-memory parallel sampling and slicing, and pipelined data transfers. We also find that neighborhood sampling impacts inference accuracy only minimally. We build our system SALIENT based on PyTorch and PyG and showcase that changing the GNN architecture can be easily done as usual, without interfering with the training/inference code.

We demonstrate that SALIENT achieves near-perfect overlap of batch preparation, transfer, and training computations. That is, the end-to-end training time per epoch is nearly equal to the time for the slowest of these components in isolation. The limiting factor for batch preparation is the number of CPU cores or the DRAM bandwidth; for data transfer it is the peak CPU-to-GPU memory bandwidth. As feature vector size increases, or with higher fanout, memory bandwidth may become insufficient. Then, one must avail of additional techniques such as GPU-based slicing (Min et al., 2021) or caching data on the GPU (Dong et al., 2021) to reduce the slicing or data transfer volume.

An additional avenue of future work is distributing the graph and node data in a distributed computing environment to accommodate processing even larger graphs. Graph parti-

Table 7. Representative GNN training systems and their performance on either ogbn-papers100M or the largest graph reported, whichever is larger, for each system.

System	Framework	Batching	GNN	Machines	Data Set	Speed (s/epoch)	Acc. (%)
NeuGraph	TensorFlow	full-batch	GCN, $L = 2$	1 machine with 28 Intel cores, 512GB DRAM, 8 P100 GPUs	amazon: $\|V\| = 8.6M$, $\|E\| = 231.6M$, $f = 96$ (McAuley et al., 2015)	0.655 [a]	N/A
Roc	FlexFlow, Lux	full-batch	GCN	4 machines, each has 20 x86 cores, 256GB DRAM, 4 P100 GPUs; 100Gbps InfiniBand	amazon: $\|V\| = 9.4M$, $\|E\| = 231.6M$, $f = 300$ (He & McAuley, 2016)	0.526 [b]	N/A
DistDGL	PyTorch, DGL, METIS	mini-batch, size 2000, $d^\ell = (15, 10, 5)$	GraphSAGE, $L = 3$, $f_{\text{hidden}} = 256$	16 EC2 instances, each has 96 vCPUs, 384GB DRAM; 100Gbps network	ogbn-papers100M: $\|V\| = 111M$, $\|E\| = 1.6B$, $f = 128$ (Hu et al., 2020a)	13 [c]	N/A
DeepGalois	Galois, GuSP, Gluon	full-batch	GraphSAGE, $L = 2$, $f_{\text{hidden}} = 16$	32 machines, each has 48 x86 cores, 192GB DRAM; 100Gbps Omni-Path	same as above	70 [d]	N/A
Zero-Copy	PyTorch, DGL	mini-batch	GraphSAGE	1 machine with 24 AMD cores, 256GB DRAM, 2 RTX3090 GPUs	same as above	648 [e]	N/A
GNS	PyTorch, DGL	mini-batch, size 1000, $d^\ell = (\text{cache}, 15, 10)$	GraphSAGE, $L = 3$, $f_{\text{hidden}} = 256$	1 EC2 instance with 32 CPU cores, 256GB DRAM, 1 T4 GPU	same as above	98.5 [f]	63.31 [f]
P^3	PyTorch, DGL	mini-batch, size 1000, $d^\ell = (25, 10)$	GraphSAGE, $L = 2$, $f_{\text{hidden}} = 32$	4 machines, each has 1×12-core Intel CPUs, 441GB DRAM, 4 P100 GPUs; 10Gbps Ethernet	same as above	3.107 [g]	N/A
SALIENT	PyTorch, PyG, DDP	mini-batch, size 1024, $d^\ell_{\text{train}} = (15, 10, 5)$, $d^\ell_{\text{infer}} = (20, 20, 20)$	GraphSAGE, $L = 3$, $f_{\text{hidden}} = 256$	8 machines, each has 2×20-core Intel CPUs, 384GB DRAM, 2 V100 GPUs; 10GigE network	same as above	Train: 2.0 Infer: 2.4s on test set	64.58 ±0.40

[a] Estimated as 6.55/10, where 6.55 comes from the TF-SAGA section of Table 2 and 10 is estimated from Figure 17 of Ma et al. (2019).
[b] Estimated as 1/1.9, where 1.9 is estimated from Figure 5 of Jia et al. (2020).
[c] Reported in Figure 8 of Zheng et al. (2020).
[d] Estimated from Figure 4 of Hoang et al. (2021). Note that the referenced article demonstrates that under the same full-batch setting, DeepGalois may be several times faster than DistDGL.
[e] Estimated from Figure 11 of Min et al. (2021).
[f] Reported in Table 3 of Dong et al. (2021).
[g] Reported in Table 4 of Gandhi & Iyer (2021).

tioning (Karypis & Kumar, 1999) will likely be invoked, but the objective may consider not only edge cut and load balance but also the cost of multi-hop neighborhood sampling. Sampling approaches will need to be re-investigated in a distributed environment, to minimize communication. Partitioning along the feature dimension is another promising technique for long feature vectors (Gandhi & Iyer, 2021).

ACKNOWLEDGEMENTS

REFERENCES

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

Benzaquen, S., Evlogimenos, A., Kulkunidis, M., and Pereplitsa, R. Swiss tables and absl::hash, Sep 2018. URL https://abseil.io/blog/20180927-swisstables.

Bottou, L., Curtis, F. E., and Nocedal, J. Optimization methods for large-scale machine learning. *SIAM Rev.*, 60 (2):223–311, 2018.

Chen, J. and Luss, R. Stochastic gradient descent with biased but consistent gradient estimators. Preprint arXiv:1807.11880, 2018.

Chen, J., Ma, T., and Xiao, C. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *ICLR*, 2018.

Chiang, W.-L., Liu, X., Si, S., Li, Y., Bengio, S., and Hsieh, C.-J. Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks. In *KDD*, 2019.

Dong, J., Zheng, D., Yang, L. F., and Karypis, G. Global neighbor sampling for mixed CPU-GPU training on giant graphs. In *KDD*, 2021.

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

Gandhi, S. and Iyer, A. P. P3: Distributed deep graph learning at scale. In *OSDI*, 2021.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *ICML*, 2017.

Hamilton, W. L., Ying, R., and Leskovec, J. Inductive representation learning on large graphs. In *NIPS*, 2017.

He, R. and McAuley, J. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *WWW*, 2016.

Hoang, L., Chen, X., Lee, H., Dathathri, R., Gill, G., and Pingali, K. Efficient distribution for deep learning on large graphs,. In *GNNSys*, 2021.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. Open graph benchmark: Datasets for machine learning on graphs. Preprint arXiv:2005.00687, 2020a.

Hu, W., Liu, B., Gomes, J., Zitnik, M., Liang, P., Pande, V., and Leskovec, J. Strategies for pre-training graph neural networks. In *ICLR*, 2020b.

Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with Roc. In *MLSys*, 2020.

Karypis, G. and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

Lawson, C. L., Hanson, R. J., Kincaid, D., , and Krogh, F. T. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. Gated graph sequence neural networks. In *ICLR*, 2016.

Li, Y., Yu, R., Shahabi, C., and Liu, Y. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. In *ICLR*, 2018.

Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., and Dai, Y. NeuGraph: Parallel deep neural network computation on large graphs. In *USENIX ATC*, 2019.

Ma, T. and Chen, J. Unsupervised learning of graph hierarchical abstractions with differentiable coarsening and optimal transport. In *AAAI*, 2021.

McAuley, J., Targett, C., Shi, Q., and van den Hengel, A. Image-based recommendations on styles and substitutes. In *SIGIR*, 2015.

Min, S. W., Wu, K., Huang, S., Hidayetoğlu, M., Xiong, J., Ebrahimi, E., Chen, D., and mei Hwu, W. Large graph convolutional network training with GPU-oriented data communication architecture. Preprint arXiv:2103.03330, 2021.

Mirhoseini, A., Goldie, A., Yazgan, M., Jiang, J. W., Songhori, E., Wang, S., Lee, Y.-J., Johnson, E., Pathak, O., Nazi, A., Pak, J., Tong, A., Srinivasa, K., Hang, W., Tuncer, E., Le, Q. V., Laudon, J., Ho, R., Carpenter, R., and Dean, J. A graph placement methodology for fast chip design. *Nature*, 594:207–212, 2021.

Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., and Grohe, M. Weisfeiler and Leman go neural: Higher-order graph neural networks. In *AAAI*, 2019.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop*, 2017.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *NIPS*, 2019.

Ramezani, M., Cong, W., Mahdavi, M., Sivasubramaniam, A., and Kandemir, M. GCN meets GPU: Decoupling "when to sample" from "how to sample". In *NeurIPS*, 2020.

Shang, C., Chen, J., and Bi, J. Discrete graph structure learning for forecasting multiple time series. In *ICLR*, 2021.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. In *ICLR*, 2018.

Wang, L., Yin, Q., Tian, C., Yang, J., Chen, R., Yu, W., Yao, Z., and Zhou, J. FlexGraph: a flexible and efficient distributed framework for GNN training. In *EuroSys*, 2021a.

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

Wang, Y., Feng, B., Li, G., Li, S., Deng, L., Xie, Y., and Ding, Y. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In *OSDI*, 2021b.

Weber, M., Domeniconi, G., Chen, J., Weidele, D. K. I., Bellei, C., Robinson, T., and Leiserson, C. E. Anti-money laundering in Bitcoin: Experimenting with graph convolutional networks for financial forensics. In *KDD Workshop on Anomaly Detection in Finance*, 2019.

Wu, Y., Ma, K., Cai, Z., Jin, T., Li, B., Zheng, C., Cheng, J., and Yu, F. Seastar: vertex-centric programming for graph neural networks. In *EuroSys*, 2021.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *ICLR*, 2019.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. Graph convolutional neural networks for web-scale recommender systems. In *KDD*, 2018.

Zeng, H., Zhou, H., Srivastava, A., Kannan, R., and Prasanna, V. GraphSAINT: Graph sampling based inductive learning method. In *ICLR*, 2020.

Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., Gan, Q., Zhang, Z., and Karypis, G. DistDGL: Distributed graph neural network training for billion-scale graphs. In *IA3*, 2020.

Zou, D., Hu, Z., Wang, Y., Jiang, S., Sun, Y., and Gu, Q. Layer-dependent importance sampling for training deep and large graph convolutional networks. In *NeurIPS*, 2019.

A   ARTIFACT APPENDIX

A.1   Abstract

This section describes the software artifacts associated with this paper for the purpose of replicating the presented experimental results. The code is distributed via GitHub at `https://github.com/MITIBMxGraph/SALIENT_artifact` and can be used to perform the experiments presented in the paper. To streamline the process of exercising the software to reproduce key experimental results, we have provided scripts in the `experiments` directory of the repository to run: (a) single GPU experiments that produce data for Table 1 and Figure 4; and, (b) distributed multi-GPU experiments that produce data for Figure 5 and Figure 6. Detailed instructions for running these scripts are provided in a dedicated readme file for artifact evaluation located at `https://github.com/MITIBMxGraph/SALIENT_artifact/blob/main/README.md`.

A.2   Artifact check-list (meta-information)

- **Algorithm:** PyG and SALIENT training algorithms for GNNs with node-wise sampling.

- **Program:** PyTorch, CUDA

- **Compilation:** `gcc/g++` version 7 or greater; `nvcc` version 11.

- **Data set:** Node classification benchmark data sets from OGB.

- **Run-time environment:** Ubuntu 18.04 (or modern linux distribution) with NVIDIA drivers installed.

- **Hardware:** NVIDIA GPU with sufficient memory. Distributed experiments require SLURM cluster with GPU nodes.

- **Experiments:** Single GPU performance comparisons, and distributed multi-GPU experiments

- **How much disk space required (approximately)?:** 100 GB for all experiments, 10 GB for a subset thereof.

- **How much time is needed to prepare workflow (approximately)?:** 1–2 hours with prior experience and access to hardware/clusters.

- **How much time is needed to complete experiments (approximately)?:** 1 hour for single GPU experiments and 4–12 hours for full set of distributed experiments.

- **Publicly available?:** Yes

- **Code licenses (if publicly available)?:** Apache License 2.0

- **Data licenses (if publicly available)?:** Amazon license and ODC-BY.

- **Archived (provide DOI)?:** `https://doi.org/10.5281/zenodo.6332979`

A.3   Description

A.3.1   How delivered

The code may be obtained from GitHub at `https://github.com/MITIBMxGraph/SALIENT_artifact`. Within the repository, scripts for streamlining the process of exercising the artifact are provided in the `experiments` directory. A dedicated readme file that documents the use of these scripts is provided at `https://github.com/MITIBMxGraph/SALIENT_artifact/blob/main/README.md`.

A.3.2   Hardware dependencies

The minimum requirements for exercising the software artifact are as follows. The single GPU experiments require one NVIDIA GPU with sufficient memory, and one multi-core CPU that uses either the x86 or PowerPC architecture. We recommend using x86 CPUs as we have tested the installation process more thoroughly for them.

The distributed multi-GPU experiments require a SLURM cluster with GPU nodes. Such a cluster may be obtained through cloud services and accompanying software packages. For example, on Amazon Web Services one can use the ParallelCluster software to launch a SLURM cluster.

Depending on the used hardware and available disk space, certain experiments may not be feasible. We have made an effort to reduce the disk space and memory requirements needed for running experiments on the largest data set, and we expect that GPUs with 16GB of memory and machines with 128GB of main memory will be able to run all or almost all of the experiments. For the distributed multi-GPU experiments, the PyG implementation often requires more than 128GB of memory when running on the ogbn-papers100M data set. Exercising the distributed experiments for PyG on this graph may require compute nodes with 256GB or more main memory.

A.3.3   Software dependencies

Reasonably up-to-date NVIDIA drivers must be installed on the machine. For the distributed experiments, a SLURM cluster is required. The remaining software dependencies can be resolved using the `conda` package manager or by using the provided Dockerfile. If using docker, one must have `nvidia-docker` installed for GPU support within the container.

A.3.4 Data sets

Graph data sets for node property prediction are taken from Open Graph Benchmark (OGB). To decrease the time and minimum hardware resources required for experiments, we have provided, as an option, the ability to download pre-processed versions of the graph data. If not electing to download the preprocessed graphs, the first execution of the code on a new graph will download it from OGB and perform preprocessing locally.

A.4 Installation

We recommend referring to the installation instructions provided at https://github.com/MITIBMxGraph/SALIENT_artifact/blob/main/README.md. We summarize the installation process here.

Installation using Docker: We provide a docker container that can be used for running experiments on a single machine. Although the container could also be used to run distributed experiments, we have not tested this option. To use the docker container with NVIDIA GPUs, one should install docker and the NVIDIA Container Toolkit.

```
# Pull the container
docker pull nistath/salient:cuda-11.1.1

# Clone the code repository outside of the container
git clone \
    git@github.com:MITIBMxGraph/SALIENT_artifact.git

# Run docker container with host code folder mounted
docker run --ipc=host --gpus all -it \
    -v `pwd`/SALIENT_artifact:/salient \
    nistath/salient:cuda-11.1.1

# Install fast sampler
cd /salient/fast_sampler && python setup.py develop
```

Installation in Python environment: We provide instructions to install the artifact in a Python environment. Such installation can be used for both the single GPU and distributed multi-GPU experiments (assuming access to a SLURM cluster). The instructions for installing in a conda environment are provided at https://github.com/MITIBMxGraph/SALIENT_artifact/blob/main/INSTALL.md and are summarized below.

```
# Install miniconda
wget https://repo.anaconda.com/miniconda/\
    Miniconda3-py38_4.10.3-Linux-x86_64.sh
bash Miniconda3-py38_4.10.3-Linux-x86_64.sh

# Create a conda environment for experiments
conda create -n salient python=3.8 -y
conda activate salient

# Install Pytorch, PyG, OGB, prettytable
conda install pytorch torchvision \
    torchaudio cudatoolkit=11.3 -c pytorch
conda install pyg -c pyg -c conda-forge
pip install ogb
conda install prettytable -c conda-forge
```

```
# Install patched PyTorch-Sparse
pip uninstall torch_sparse
FORCE_CUDA=1
pip install \
    git+git://github.com/rusty1s/pytorch_sparse.git@master

# Install fast_sampler
cd fast_sampler
python setup.py install
cd ..
```

A.5 Experiment workflow

We recommend referring to the documentation for performing artifact evaluation located in the repository at https://github.com/MITIBMxGraph/SALIENT_artifact/blob/main/README.md. We summarize the experimental workflow here. Unless otherwise noted, all file paths are relative to the experiments directory in the repository.

Initial setup: The script initial_setup.sh can be executed to configure the number of sampling workers based on the hardware, and determine what data sets to download based on the available disk space. It will then, by default, download the appropriate preprocessed data sets.

Single GPU experiments: We provide the script run_all_single_gpu_experiments.sh to run all single GPU experiments and display the final table of results. Additional scripts, documented in the artifact evaluation guide in the repository, are provided to run these experiments manually and regenerate the summary table of results.

Distributed multi-GPU experiments: These experiments require the use of a SLURM cluster. The file all_dist_benchmarks.sh must be modified to account for the configuration of the cluster at hand. Additional instructions and guidance are provided in the artifact evaluation guide in the repository. The final table of results can be generated with the command:

```
python helper_scripts/parse_times.py \
    distributed_job_output/
```

A.6 Evaluation and expected result

Upon completion of the single GPU experiments, a table will be produced that provides a performance breakdown of per-epoch runtime that reproduces the breakdowns provided in Table 1 and Figure 4 of the paper.

Upon completion of the distributed multi-GPU experiments, a table will be produced that provides the data needed to reproduce Figure 6. Specifically, the table includes the per-epoch runtime and test accuracy for SALIENT and PyG across four GNN architectures shown in Figure 6. The scripts may be modified to run on a different number of GPUs to reproduce the scalability experiment shown in Figure 5.

A.7 Experiment customization

The following experiment customizations are possible. The software may be directly exercised without the use of the dedicated artifact evaluation scripts. The scripts for single GPU experiments may be modified to use different GNN architectures, sampling fanouts, and hidden feature sizes by modifying the parameters in `performance_breakdown_config.cfg`. The distributed multi-GPU experiments may be modified to run on different data sets and different numbers of GPUs. The `fast_sampler` extension can be integrated to other codes.

B CODE REPOSITORY

In addition to the artifact repository that focuses on benchmarking and reproducibility, an implementation of SALIENT for general-purpose usage is available at `https://github.com/MITIBMxGraph/SALIENT`.

C ARCHITECTURES FOR EXPERIMENTS

See listings 1, 2, 3, and 4 for the model definitions written in PyG.

```python
1  def __init__(self, in_channels, hidden_channels, out_channels, num_layers):
2      kwargs = dict(bias = False)
3      conv_layer = SAGEConv
4      super().__init__()
5      self.num_layers = num_layers
6      self.convs = torch.nn.ModuleList()
7      self.hidden_channels = hidden_channels
8
9      self.convs.append(conv_layer(in_channels, hidden_channels, **kwargs))
10     for _ in range(num_layers - 2):
11         self.convs.append(conv_layer(hidden_channels, hidden_channels, **kwargs))
12     self.convs.append(conv_layer(hidden_channels, hidden_channels, **kwargs))
13     self.reset_parameters()
14
15 def forward(self, x, adjs):
16     end_size = adjs[-1][-1][1]
17     for i, (edge_index, _, size) in enumerate(adjs):
18         x_target = x[:size[1]]
19         x = self.convs[i]((x, x_target), edge_index)
20         if i != self.num_layers - 1:
21             x = F.relu(x)
22             x = F.dropout(x, p=0.5, training=self.training)
23     return torch.log_softmax(x, dim=-1)
```

Listing 1. GraphSAGE model definition.

```python
1  def __init__(self, in_channels, hidden_channels, out_channels, num_layers):
2      kwargs = dict(bias = False, heads = 1)
3      conv_layer = GATConv
4      super().__init__()
5      self.num_layers = num_layers
6      self.convs = torch.nn.ModuleList()
7      self.hidden_channels = hidden_channels
8
9      self.convs.append(conv_layer(in_channels, hidden_channels, **kwargs))
10     for _ in range(num_layers - 2):
11         self.convs.append(conv_layer(hidden_channels, hidden_channels, **kwargs))
12     self.convs.append(conv_layer(hidden_channels, out_channels, **kwargs))
13     self.reset_parameters()
14
15 def forward(self, x, adjs):
16     for i, (edge_index, _, size) in enumerate(adjs):
17         x_target = x[:size[1]]
18         x = self.convs[i]((x, x_target), edge_index)
19         if i != self.num_layers - 1:
20             x = F.relu(x)
21             x = F.dropout(x, p=0.5, training=self.training)
22     return torch.log_softmax(x, dim=-1)
```

Listing 2. GAT model definition.

```python
def __init__(self, in_channels, hidden_channels, out_channels, num_layers):
    kwargs = dict()
    conv_layer = GINConv
    super().__init__()
    self.num_layers = num_layers
    self.convs = torch.nn.ModuleList()
    self.hidden_channels = hidden_channels

    self.convs.append(GINConv(Sequential(Linear(in_channels, hidden_channels),
                      BatchNorm1d(hidden_channels), ReLU(),
                      Linear(hidden_channels, hidden_channels), ReLU())))
    for _ in range(num_layers - 2):
        self.convs.append(GINConv(Sequential(Linear(hidden_channels, hidden_channels),
                          BatchNorm1d(hidden_channels), ReLU(),
                          Linear(hidden_channels, hidden_channels), ReLU())))
    self.convs.append(GINConv(Sequential(Linear(hidden_channels, hidden_channels),
                      BatchNorm1d(hidden_channels), ReLU(),
                      Linear(hidden_channels, hidden_channels), ReLU())))
    self.lin1 = Linear(hidden_channels, hidden_channels)
    self.lin2 = Linear(hidden_channels, out_channels)
    self.reset_parameters()

def forward(self, x, adjs):
    end_size = adjs[-1][-1][1]
    for i, (edge_index, _, size) in enumerate(adjs):
        x_target = x[:size[1]]
        x = self.convs[i]((x, x_target), edge_index)
    x = self.lin1(x).relu()
    x = F.dropout(x, p=0.5, training=self.training)
    x = self.lin2(x)
    return torch.log_softmax(x, dim=-1)
```

Listing 3. GIN model definition.

```python
def __init__(self, in_channels, hidden_channels, out_channels, num_layers):
    conv_layer = SAGEConv
    kwargs = dict(bias = False)
    super().__init__()
    self.num_layers = num_layers
    self.convs = torch.nn.ModuleList()
    self.bns = torch.nn.ModuleList()
    self.res_linears = torch.nn.ModuleList()
    self.hidden_channels = hidden_channels

    self.convs.append(conv_layer(in_channels, hidden_channels, **kwargs))
    self.bns.append(torch.nn.BatchNorm1d(hidden_channels))
    self.res_linears.append(torch.nn.Linear(in_channels, hidden_channels))
    for _ in range(num_layers - 2):
        self.convs.append(conv_layer(hidden_channels, hidden_channels, **kwargs))
        self.bns.append(torch.nn.BatchNorm1d(hidden_channels))
        self.res_linears.append(torch.nn.Identity())
    self.convs.append(conv_layer(hidden_channels, hidden_channels, **kwargs))
    self.bns.append(torch.nn.BatchNorm1d(hidden_channels))
    self.res_linears.append(torch.nn.Identity())

def forward(self, _x, adjs):
    collect = []
    end_size = adjs[-1][-1][1]
    x = F.dropout(_x, p=0.1, training=self.training)
    collect.append(x[:end_size])
    for i, (edge_index, _, size) in enumerate(adjs):
        x_target = x[:size[1]]
        x = self.convs[i]((F.dropout(x,p=0.1,training=self.training),
                           F.dropout(x_target, p=0.1, training=self.training)), edge_index)
        x = self.bns[i](x)
        x = F.leaky_relu(x)
        x = F.dropout(x, p=0.1, training=self.training)
        collect.append(x[:end_size])
        x += self.res_linears[i](x_target)
    return torch.log_softmax(self.mlp(torch.cat(collect, -1)), dim=-1)
```

Listing 4. GragSAGE-RI model definition.