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

Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao B. Schardl, Charles E. Leiserson, Jie Chen

Presentation by
Helen Yang

Through performance engineering, they achieve:

3x

Speedup over standard PyTorch-Geometric implementation with a single GPU

8x

Speedup over standard PyTorch-Geometric implementation on multiple GPUs

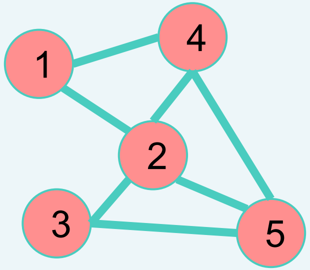
Accelerating Training and Inference of Graph Neural Networks with Fast Sampling and Pipelining

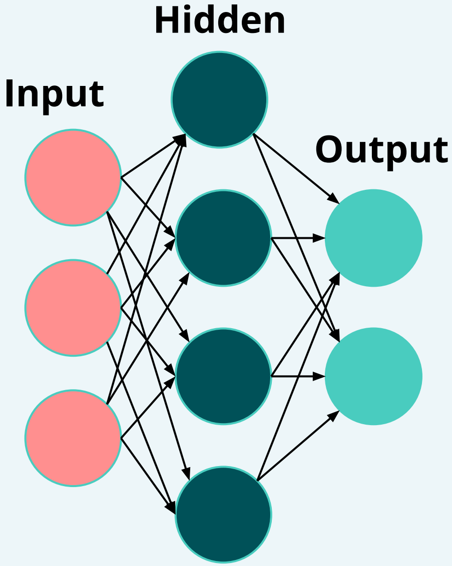
Graphs and Neural Networks

A **graph G=(V,E)** contain nodes *V* connected by edges in the set *E*.

Neural networks are a network of artificial neurons that map some given input to an output prediction.

- Usually composed of many layers with functions and nonlinearities between.
- Embedding (hidden) layers map input from high to low dimensionality.
- Ex: CNNs, RNNs, **GNNs**!!

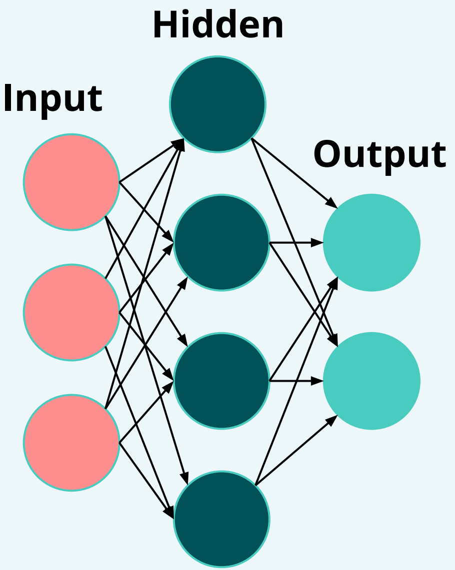




Accelerating Training and Inference of Graph Neural Networks with Fast Sampling and Pipelining

Neural Network Training and Inference

- **Training** is the process of teaching a neural network how to perform a task.
    - Ex: Classify an animal based on an image.
- Neural networks need to be trained with at least thousands of examples.
    - Ex: ChatGPT is trained on 300 billion words!
- **Inference** is the process of inputting data into a model to get a prediction



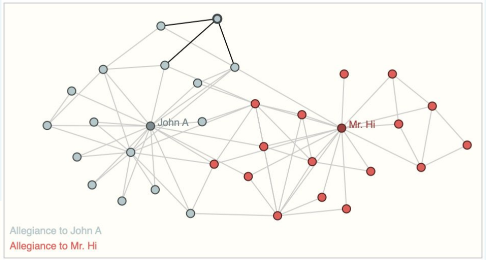
Accelerating Training and Inference of Graph Neural Networks with Fast Sampling and Pipelining

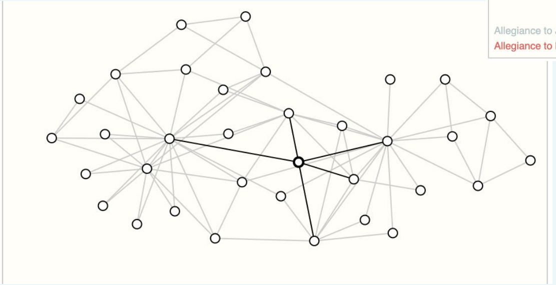
What are graph neural networks (GNNs)?

Neural networks, but with graph inputs!

Why Care About GNNs??

- Recent developments have increased GNN capabilities.
- Three main categories of GNN tasks:
  - Graph-level tasks (e.g. predicting smell of molecule)
  - Node-level tasks (e.g. predicting allegiance)
  - Edge-level tasks (e.g. predicting relationships)
- Applications include:
  - antibacterial discovery
  - physics simulations
  - fake news detection
  - traffic prediction
  - recommendation systems

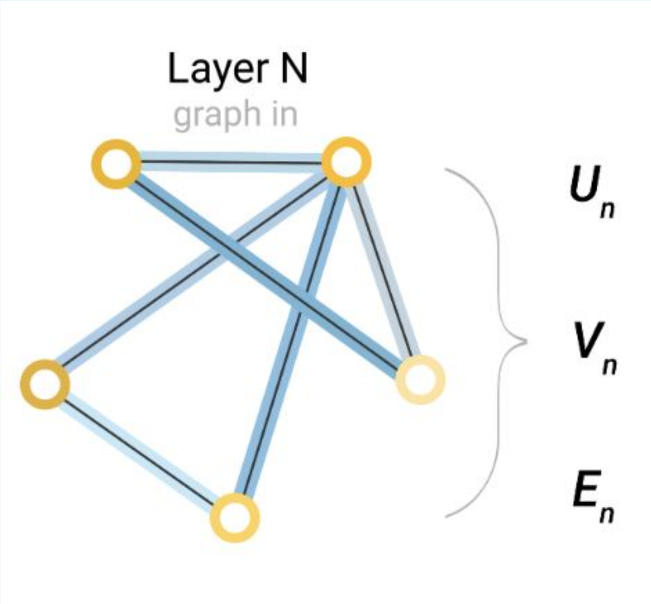




Lots of challenges though…

- How to represent the relationships in a graph in a space-efficient manner and map them to embedding space?
- Graphs can be REALLY big nowadays
  - 111M nodes, 1.6B edges

GNN Architecture – Input

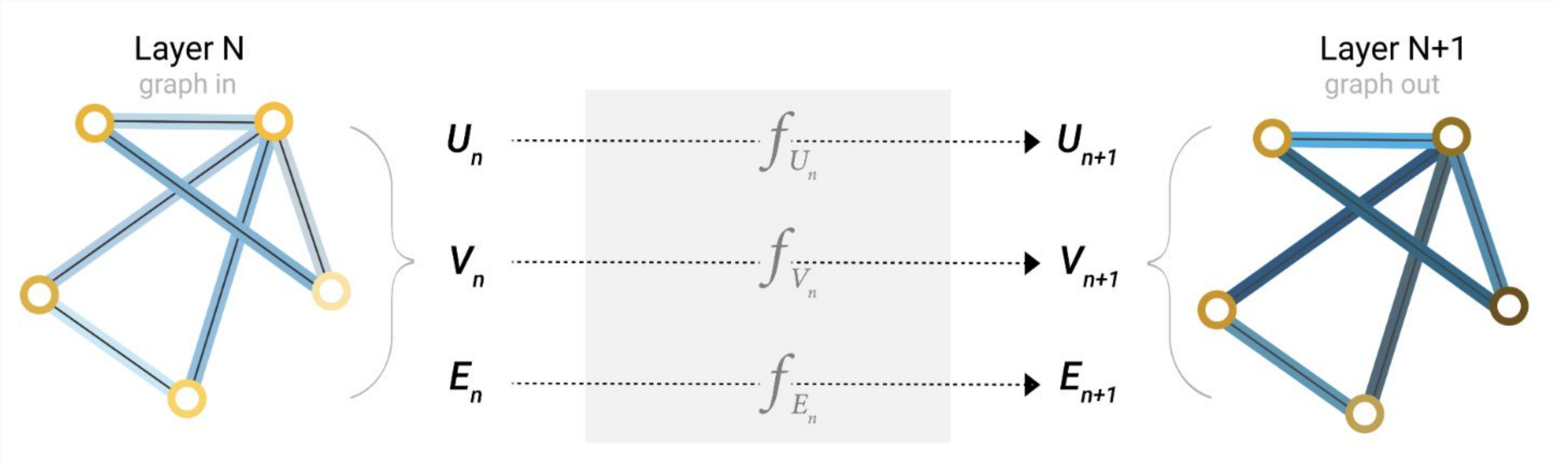


Global attribute vectors
e.g., number of nodes, longest path

Vertex (or node) attribute vectors
e.g., node identity, number of neighbors

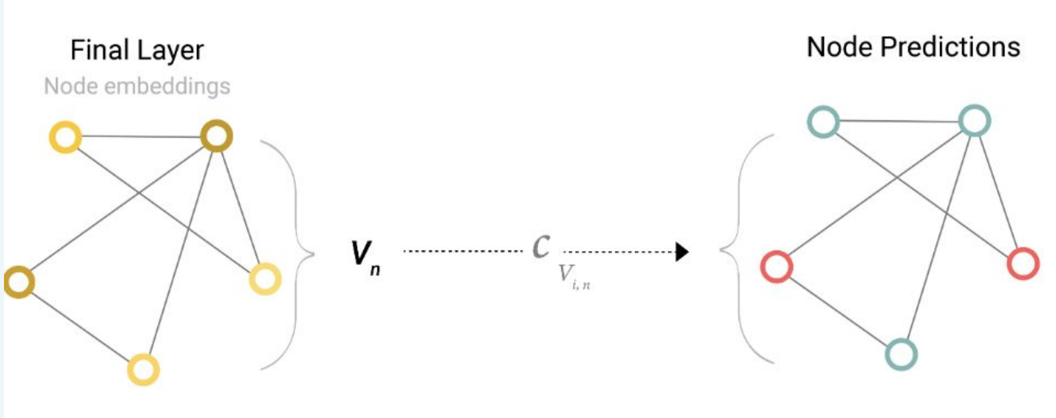
Edge attribute vectors
e.g., edge identity, edge weight

GNN Architecture – One Layer



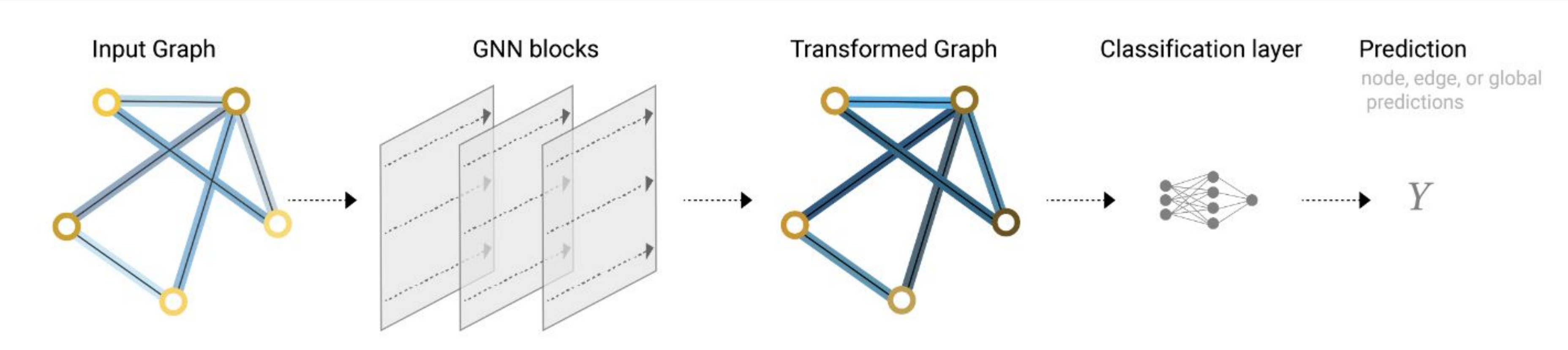
- A single layer of a simple GNN.
- Given an input graph, each component (V,E,U) gets updated by some update function to produce a new graph.
- Each function subscript indicates a separate function for a different graph attribute at the n-th layer of a GNN model.

GNN Architecture – Classification



- Let's say we want to make some binary classification on the nodes of the graph!
- For each node embedding, apply a linear classifier, ***c***.
- Can do the same if we want to make a prediction on edges or the whole graph.

GNN Architecture – Overview



- Given an input graph, we can apply several layers of update functions to get our transformed graph
- Once we have the transformed graph, we can apply some classification function to nodes, edges, or global attributes to get our final prediction.
- Note, this is a simplified overview! There are more complex techniques we could use such as pooling and aggregation.

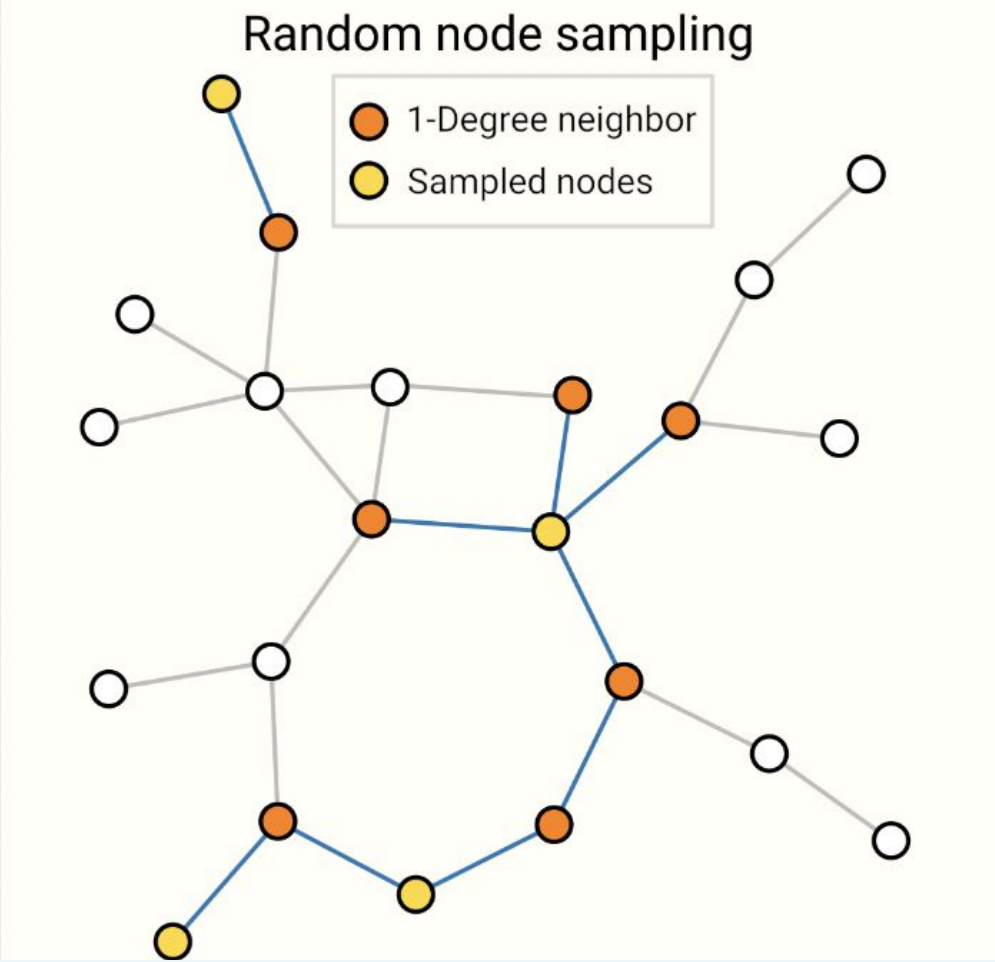
Existing GNN Frameworks

- **PyTorch Geometric (PyG):** a library built upon PyTorch to easily write and train GNNs
  - Comes with mini-batch loaders
  - multi-GPU support
- Deep Graph Library (DGL): memory-efficient message passing primitives for training GNNs

Accelerating Training and Inference of Graph Neural Networks with Fast <u>Sampling</u> and Pipelining

Sampling Graphs and Batching

- Common practice in standard NNs: update network with gradients calculated from a *subset* of the training data – we call this subset a **mini-batch**.
  - More efficient due to memory constraints
- Selecting a subset (sampling) a graph is more complicated and is an open research question.
  - Important because many graphs are too large to fit in memory.
- Idea: **neighborhoods**!
  - Randomly sample some nodes → **node-set**
  - add neighboring nodes of distance k adjacent to the node-set, including their edges



Start at a sampled node and expand outwards until all neighbors are reached

Batching continued…

- In this paper, batch preparation entails:
  - expanding the sampled neighborhood for a mini-batch of nodes
  - slicing out the feature vectors of all involved nodes
  - transfer subgraph and feature vectors to GPU

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

Standard impl. of GNN training with node features x and labels y

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

What is pipelining?

- Technique for implementing instruction-level parallelism within a single processor.
  - Key technique to building fast processors!
- Successive steps of an instruction sequence are executed in turn by modules that are able operate concurrently.
- Allows another instruction to begin before the previous one is finished.

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

Table of contents

01 Performance Benchmarking

02 SALIENT

03 Evaluation

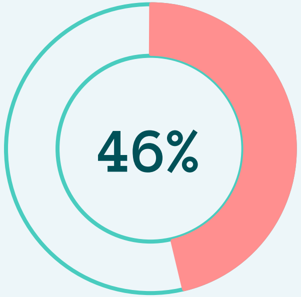
01

Performance Bottlenecks

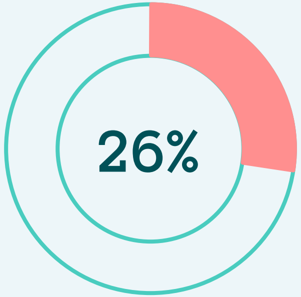
What makes GNN training and inference slow

Benchmarking

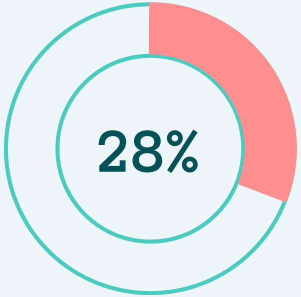
Batch Preparation



Data Transfer



GPU Training



On products dataset, with 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

Performance Analysis

- **Batch preparation and data transfer take substantially longer than core training operations (loss, gradient, etc)**
- Batch preparation time is dominated by the neighborhood sampling time, about 6x the time taken by slicing.

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

02

SALIENT

a system for fast data-parallel GNN training

Key Features of SALIENT

1 Optimized neighborhood sampling

2 Efficient parallel batch preparation

3 CPU-to-GPU data transfer optimizations

4 Seamless compatibility with PyTorch

Fast Neighborhood Sampling

Base Implementation

- Given an input graph G, a set of nodes V = {v1, . . . , vk} which define a mini-batch, and a fanout *d*.
- For each node vi ∈ V, sample *d* of its neighbors → sampled neighborhood.
- Sampled neighborhoods organized into a bipartite graph.
- Multi-hop neighborhoods form a message-flow graph (MFG).

Optimizations

- Most impactful optimizations involved **changing around data structures**.
- Data structures optimized:
  - Global-to-local node ID mapping between the input graph and sampled MFG
  - Set DS to support neighbor sampling
- C++ STL hash map and hash set → flat swiss-table
  - 2x speedup
- Array instead of a hash table for the set: 17% improvement
  - Cache locality!

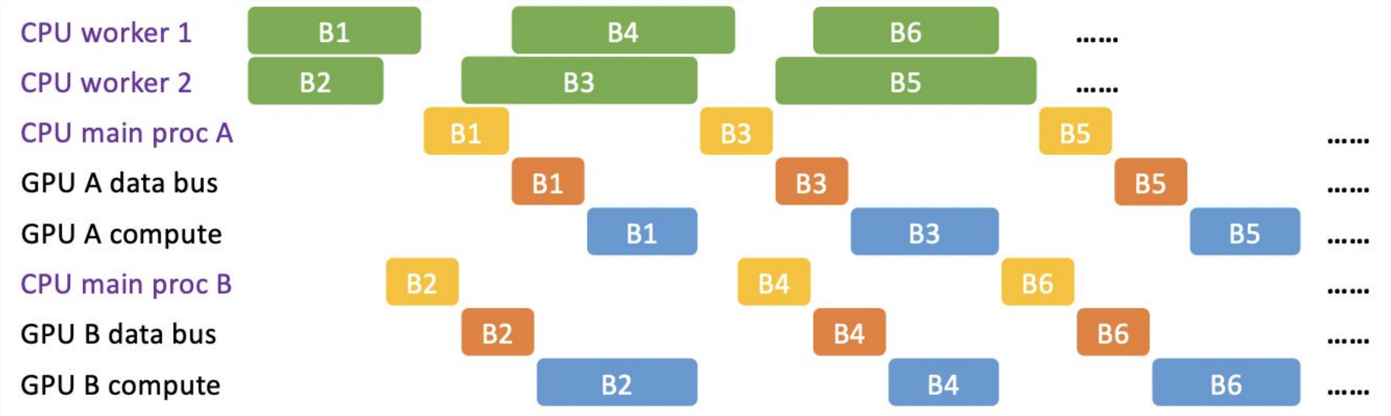
Shared-memory Parallel Batch Prep

- SALIENT uses **shared-memory multi-threading** to parallelize batch prep
- Key advantages over PyTorch multiprocessing:
    1. Lower synchronization overhead
    2. Zero-copy communication with main training process
- The 2nd key advantage allows us to perform slicing **at the same time** the main process is blocked on training.
    - worker thread writes sliced tensors directly into pinned memory accessible by the main process

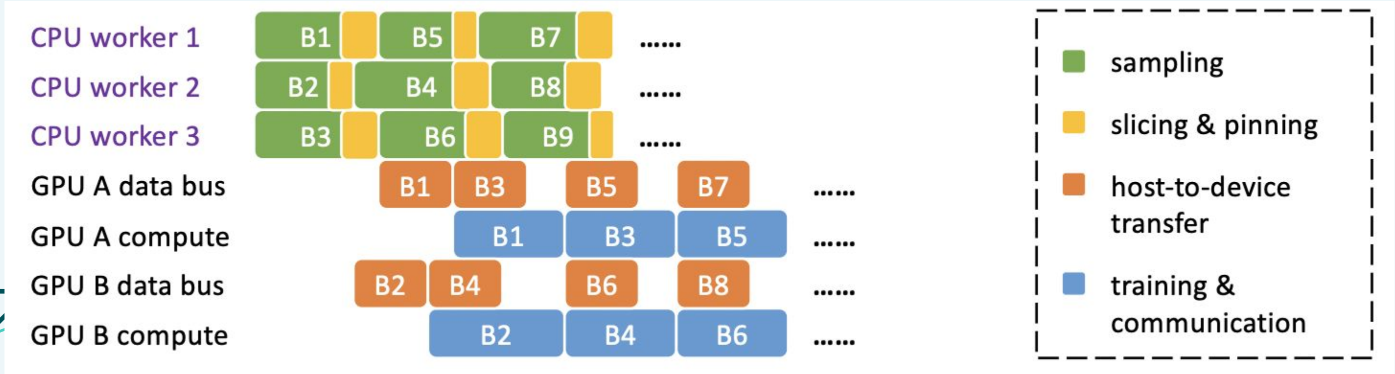
Data Transfer Optimizations and Pipelining

- Redundant assertions during data transfer in the PyG library.
- Adding an option to **skip assertions** → achieve 99% of peak data transfer throughput.
  - Significant improvement over the previous 75% throughput.
- Increase GPU utilization by **overlapping data transfers with GPU training**
  - Separate GPU streams for computation and data transfer
  - Synchronize streams to ensure a training iteration begins after the necessary data is transferred

Helpful Visualization From Paper



Standard PyTorch workflow



SALIENT implementation

The “Bi” blocks refer to operations with the i-th minibatch

Results of Optimizations

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

03

Evaluation

Experiments on SALIENT!

Specs & Models & Datasets

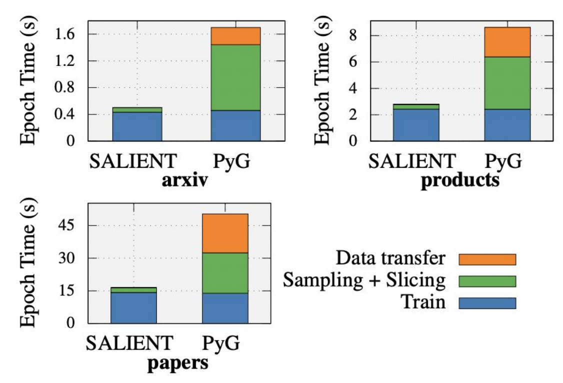
Experiments conducted on a cluster of compute nodes.

- Each compute node equipped with two 20-core Intel Xeon Gold 6248 CPUs, 384GB DRAM, and two NVIDIA V100 GPUs (32GB RAM).
- Benchmarking is based on PyTorch 1.8.1 and PyG 1.7.0
- Models: GraphSAGE, GAT, GIN, and GraphSAGE-RI

Three standard datasets used in evaluation:

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

Single GPU Improvement over PyG

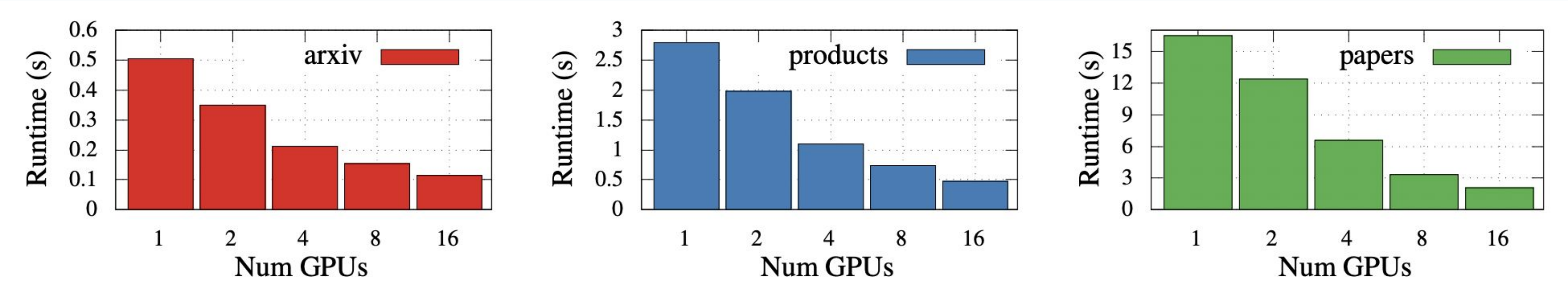


3–3.4x speedup!!

- Owes to less time spent blocked on sampling and data transfer.
- Pipelined design results in per-epoch runtime being nearly equal to GPU compute time for training

Good Multi-GPU Scaling

4.45-8.05× speedup!!



- Larger datasets see better parallel speedup since they amortize the latency of starting an epoch (time to prepare the first sets of mini-batches) over a greater amount of work per GPU

Performance Comparison Summary

On the largest data set, ogbnpapers100M, SALIENT's 2.0s per-epoch training time is orders of magnitude faster than that of other systems such as DeepGalois and DistDGL.

Final thoughts

Summary and Future Work

- Identified major bottlenecks in GNN training and inference
    - Batch preparation and data transfer
- Proposed three complementary improvements
    - Optimized neighborhood sampling, shared-memory parallel sampling and slicing, and pipelined data transfers
- SALIENT achieves near-perfect overlap of batch preparation, transfer, and training computations.
- Can easily be integrated into GNN without affecting training.
- One avenue of future work is to apply these optimizations in a distributed environment to even larger graphs.

Thank you!