

# Accelerating GNN Dataloading on Multi-GPU Systems: dgl.graphbolt

Muhammed Fatih Balın<sup>1</sup>

---

Guest Lecture at Georgia Tech for CSE 6230  
March 5

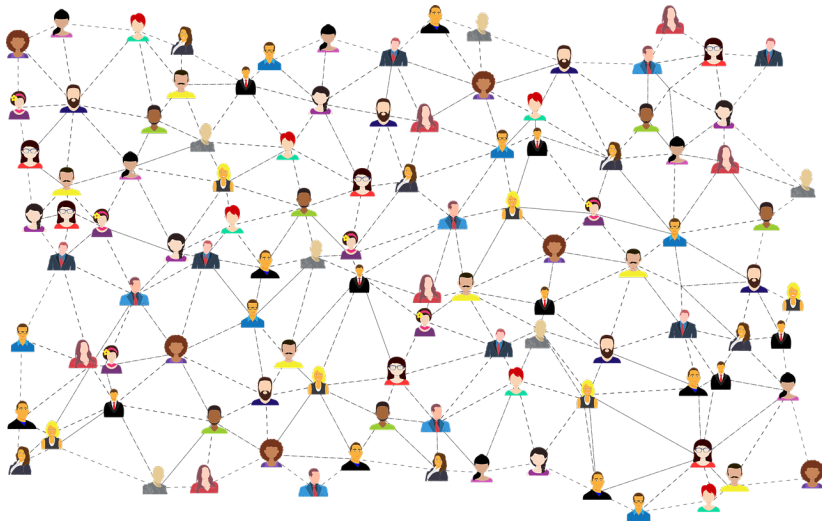
<sup>1</sup>*Computational Science and Engineering, Georgia Institute of Technology*



# Motivation

---

# Applications of GNNs



⊙ Online shopping

⊙ Content recommendation

⊙ Social media

⊙ Showing relevant ads

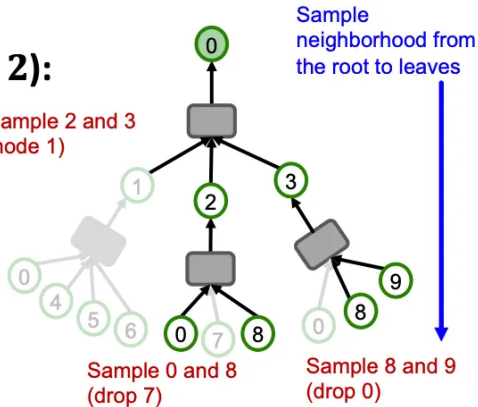
# Graph Neural Networks

- **Example ( $H = 2$ ):**

1<sup>st</sup>-hop  
neighborhood

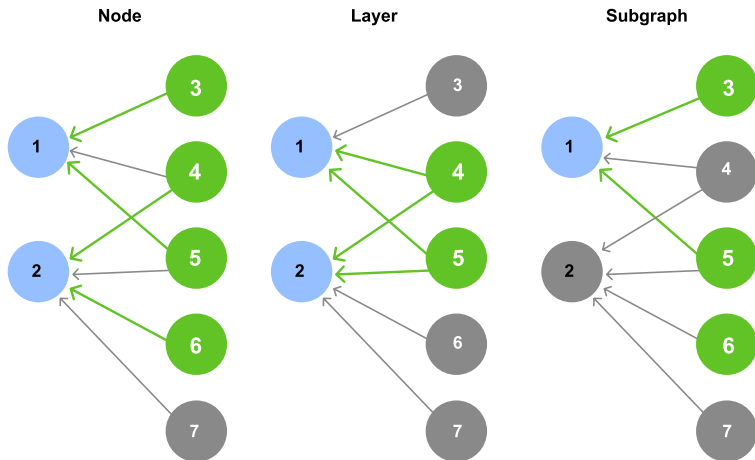
2<sup>nd</sup>-hop  
neighborhood

First, sample 2 and 3  
(drop node 1)



# Sampling strategies

Blue: seed vertices, green: sampled, gray: not sampled



# GNN Stages

1. Graph sampling
2. Feature loading
3. Forward-backward passes

Insights:

- ⊙ All are memory-bound operations.
- ⊙ Accelerators have fast memory.

1. Layer-Neighbor Sampling (NeurIPS'23)
2. Cooperative Minibatching (Under review)
3. Efficient GNN training system (dgl.graphbolt, WIP)

Background





# Graph notation

Notation	Definition
$\mathcal{G} = (V, E)$	Graph $\mathcal{G}$ consisting of vertices $V$ and edges $E \subset V \times V$
$t \rightarrow s$	Edge from vertex $t$ to seed vertex $s$
$A_{ts}$	Edge weight for edge $t \rightarrow s$
$S$	a set of vertices $S \subseteq V$
$N(s)$	1-hop neighborhood of $s$ , $\{t   (t \rightarrow s) \in E\}$
$N(S)$	1-hop neighborhood for $S$ , $\cup_{s \in S} N(s)$
$S^l$	$l$ -hop neighborhood of $S^0 = S$ , $S^{(l+1)} = S^l \cup N(S^l)$
$d_s$	the degree of $s$ , $ N(s) $

Notation	Definition
$H_t^{(l)}$	vertex embedding for $t$ at layer $l$
$f^{(l)}$	GNN layer at layer $l$
$W^{(l)}$	trainable weight matrix at layer $l$
$M_t$	message from $t$ , $M_t = H_t W$
$M_{ts}$	message over the edge $t \rightarrow s$ , $M_{ts} = A_{ts} H_t W$

# GNN sampling notation

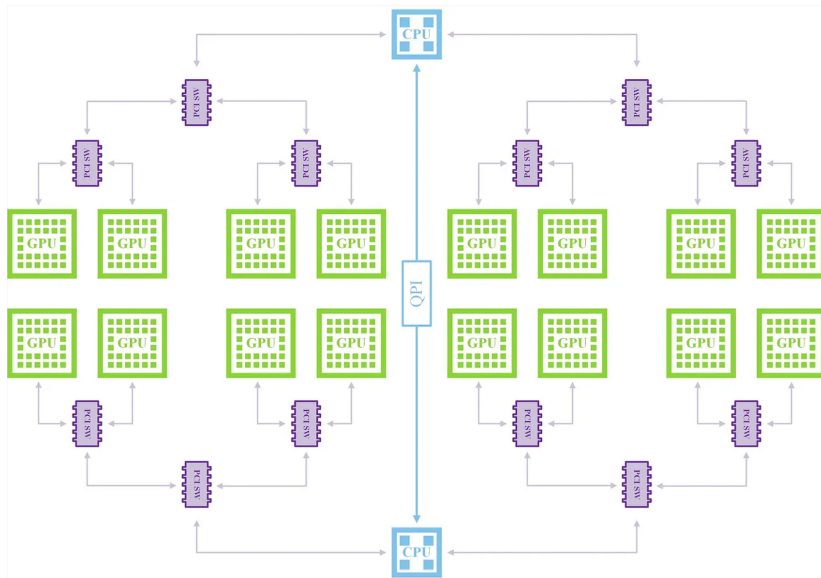
Notation	Definition
$k$	fan-out parameter or sampling budget per vertex
$\pi_t$	probability of sampling vertex $t$
$\pi_{ts}$	probability of sampling edge $t \rightarrow s$
$T$	the sampled vertices in the next layer, $T \subseteq N(S)$

# Computer Hardware

Common bandwidths and storage capacities of components in a modern PCIe-based multi-GPU computer system.

<b>Component</b>	<b>Capacity</b>
1× NVMe SSD	4 TB
System memory	2 TB
<b>GPU global memory</b>	<b>80 GB</b>
<b>Connection</b>	<b>Bandwidth</b>
1× PCI-e 4.0 lane	2 GB/s
NVMe SSD over 4× lanes	8 GB/s
GPU over 16× lanes	32 GB/s
CPU to system memory	400 GB/s
<b>GPU to GPU</b>	<b>300 GB/s</b>
<b>GPU to GPU global memory</b>	<b>2 TB/s</b>

# Example Interconnect Topology



## Layer-Neighbor Sampling (LABOR)

---

# Sampling Goals & Solution

Goals:

- ⊙ Unbiased sampling

Solution:

# Sampling Goals & Solution

Goals:

- ⊙ Unbiased sampling
- ⊙ Overlapping neighborhoods (LADIES)

Solution:



# Sampling Goals & Solution

Goals:

- ⊙ Unbiased sampling
- ⊙ Overlapping neighborhoods (LADIES)
- ⊙ **Uniformly good approximation (NS)**

Solution:

# Sampling Goals & Solution

Goals:

- ⊙ Unbiased sampling
- ⊙ Overlapping neighborhoods (LADIES)
- ⊙ Uniformly good approximation (NS)

Solution:

- ⊙ Poisson Sampling - flip biased coins:  $r \leq \pi, r \sim U(0, 1)$

# Sampling Goals & Solution

## Goals:

- ⊙ Unbiased sampling
- ⊙ Overlapping neighborhoods (LADIES)
- ⊙ Uniformly good approximation (NS)

## Solution:

- ⊙ Poisson Sampling - flip biased coins:  $r \leq \pi, r \sim U(0, 1)$
- ⊙ **Combine NS & LADIES to get best-of-both-worlds LABOR.**

# Sampling Goals & Solution

## Goals:

- ⊙ Unbiased sampling
- ⊙ Overlapping neighborhoods (LADIES)
- ⊙ Uniformly good approximation (NS)

## Solution:

- ⊙ Poisson Sampling - flip biased coins:  $r \leq \pi, r \sim U(0, 1)$
- ⊙ Combine NS & LADIES to get best-of-both-worlds LABOR.
- ⊙ **Generalizes to any unbiased sampling method.**

⊙ GCN equation:  $H_s = \frac{1}{d_s} \sum_{t \rightarrow s} H_t W = \frac{1}{d_s} \sum_{t \rightarrow s} M_t$

# GNN Sampling Background - Recap

- ⊙ GCN equation:  $H_s = \frac{1}{d_s} \sum_{t \rightarrow s} H_t W = \frac{1}{d_s} \sum_{t \rightarrow s} M_t$
- ⊙ Neighbor Sampling (NS):  
For given  $s$ , sample  $k$ -subset of  $\{t \mid t \rightarrow s\}$

# GNN Sampling Background - Recap

- ⊙ GCN equation:  $H_s = \frac{1}{d_s} \sum_{t \rightarrow s} H_t W = \frac{1}{d_s} \sum_{t \rightarrow s} M_t$
- ⊙ Neighbor Sampling (NS):  
For given  $s$ , sample  $k$ -subset of  $\{t \mid t \rightarrow s\}$
- ⊙ Layer sampling:  
Given  $S$ , sample  $T \subseteq \{t \mid t \rightarrow s \in S\}$ ,  
extract edges  $\{t \rightarrow s \mid t \in T, s \in S\}$

# Layer-Neighbor Sampling

- ⊙ LABOR-0: given  $s$  and  $r_t \sim U(0, 1)$ ,  
sample  $t \rightarrow s$  if  $r_t \leq \frac{k}{d_s}$ .



# Layer-Neighbor Sampling

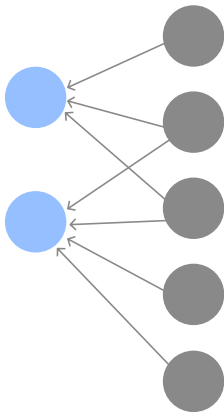
- ⊙ LABOR-0: given  $s$  and  $r_t \sim U(0, 1)$ ,  
sample  $t \rightarrow s$  if  $r_t \leq \frac{k}{d_s}$ .
- ⊙ Expected  $k$  sampled items, matching NS.

# Layer-Neighbor Sampling

- ⊙ LABOR-0: given  $s$  and  $r_t \sim U(0, 1)$ ,  
sample  $t \rightarrow s$  if  $r_t \leq \frac{k}{d_s}$ .
- ⊙ Expected  $k$  sampled items, matching NS.
- ⊙ Taking top- $k$   $r_t$  values makes # sampled items deterministic.

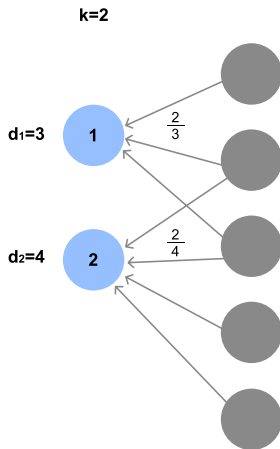
# LABOR-0 example

Blue: seed vertices, green: sampled, gray: not sampled



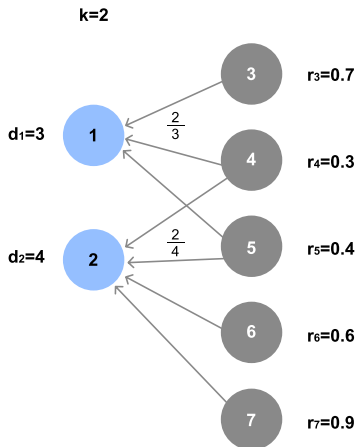
# LABOR-0 example

Blue: seed vertices, green: sampled, gray: not sampled



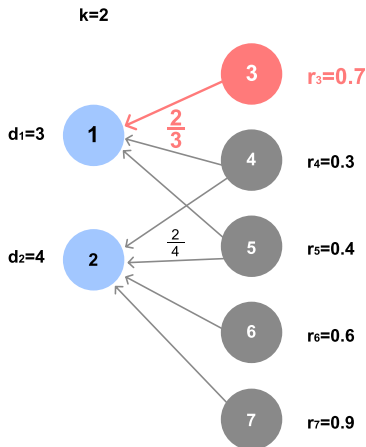
# LABOR-0 example

Blue: seed vertices, green: sampled, gray: not sampled



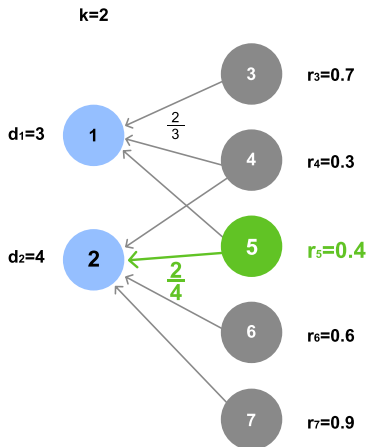
# LABOR-0 example

Blue: seed vertices, green: sampled, gray: not sampled



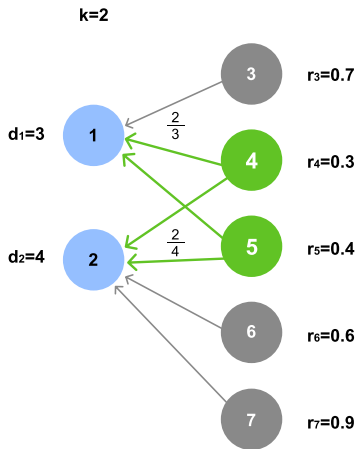
# LABOR-0 example

Blue: seed vertices, green: sampled, gray: not sampled



# LABOR-0 example

Blue: seed vertices, green: sampled, gray: not sampled





---

**Algorithm** LABOR-0 for uniform edge weights

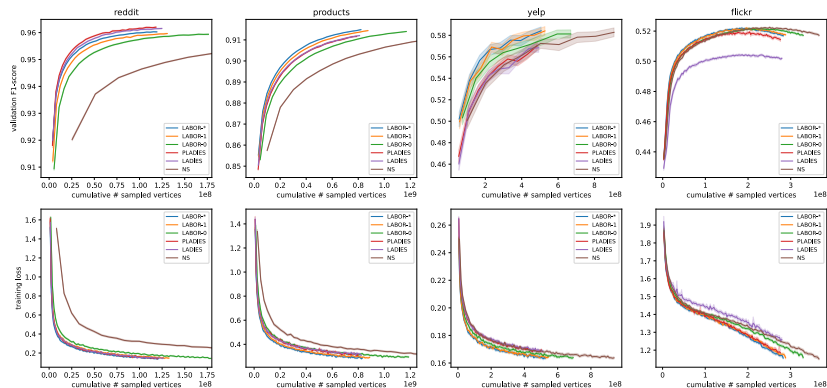
---

```
1: Input: seed vertices  $S$ , fan-out  $k$ 
2: Output: sampled edges  $E'$ 
3:  $T \leftarrow \{t \mid t \in N(S)\}$ 
4:  $r_t \sim U(0, 1), \forall t \in T$ 
5:  $E' \leftarrow []$ 
6: for all  $s \in S$  do
7:   for all  $t \in N(s)$  do
8:     if  $r_t \leq \frac{k}{d_s}$  then
9:        $E'.append(t \rightarrow s)$ 
```

---

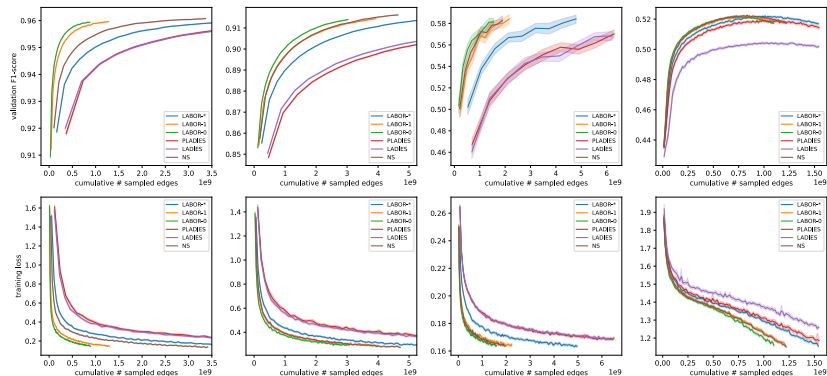
The loop corresponds to a 'copy\_if' operation.

# Experiments - Vertex efficiency



The validation F1-score and training loss curves.

# Experiments - Edge efficiency



The validation F1-score and training loss curves.

# Experiments - PLADIES and LABOR Evaluation

Dataset	Algo.	$ V^3 $	$ E^2 $	$ V^2 $	$ E^1 $	$ V^1 $	$ E^0 $	$ V^0 $	it/s	test F1-score
reddit	PLADIES	<b>24</b>	<b>2390</b>	14.1	927	6.0	33.2	1	<b>1.7</b>	96.21 ± 0.06
	LADIES	<b>25</b>	<b>2270</b>	14.5	852	6.0	32.5	1	<b>1.8</b>	96.20 ± 0.05
	LABOR-*	<b>24</b>	1070	13.7	435	6.0	26.9	1	4.1	96.23 ± 0.05
	LABOR-1	27	261	14.4	116	6.1	16.7	1	24.8	96.23 ± 0.06
	LABOR-0	36	<b>177</b>	17.8	67	6.8	9.6	1	<b>37.6</b>	96.25 ± 0.05
	NS	<b>167</b>	682	68.3	100	10.1	9.7	1	14.2	96.24 ± 0.05
products	PLADIES	<b>160</b>	<b>2380</b>	51.2	293	9.7	11.7	1	<b>4.1</b>	78.44 ± 0.24
	LADIES	<b>165</b>	<b>2230</b>	51.8	270	9.7	11.5	1	<b>4.2</b>	78.59 ± 0.22
	LABOR-*	<b>166</b>	1250	51.8	167	9.8	10.6	1	6.2	78.59 ± 0.34
	LABOR-1	178	799	53.4	136	9.8	10.5	1	21.3	78.47 ± 0.26
	LABOR-0	237	<b>615</b>	62.4	100	10.1	9.9	1	<b>32.5</b>	78.76 ± 0.26
	NS	<b>513</b>	944	95.4	106	10.6	9.9	1	24.6	78.48 ± 0.29
yelp	PLADIES	<b>100</b>	<b>1300</b>	29.5	183	6.2	6.9	1	<b>5.1</b>	61.55 ± 0.87
	LADIES	<b>102</b>	<b>1280</b>	29.7	182	6.2	6.9	1	<b>5.3</b>	61.89 ± 0.66
	LABOR-*	<b>105</b>	991	30.7	158	6.1	6.8	1	13.3	61.57 ± 0.67
	LABOR-1	109	447	31.0	96	6.2	6.8	1	<b>27.3</b>	61.71 ± 0.70
	LABOR-0	138	<b>318</b>	35.1	54	6.2	6.3	1	<b>27.2</b>	61.55 ± 0.85
	NS	<b>188</b>	392	42.5	55	6.3	6.3	1	23.0	61.50 ± 0.66
flickr	PLADIES	<b>55</b>	<b>309</b>	24.9	85	6.2	6.9	1	<b>10.2</b>	51.52 ± 0.26
	LADIES	<b>56</b>	<b>308</b>	25.1	85	6.2	6.9	1	<b>10.5</b>	<b>50.79 ± 0.29</b>
	LABOR-*	<b>57</b>	<b>308</b>	25.6	85	6.3	6.9	1	20.3	51.67 ± 0.27
	LABOR-1	58	242	25.9	73	6.3	6.9	1	<b>32.7</b>	51.66 ± 0.24
	LABOR-0	66	<b>219</b>	29.1	52	6.4	6.7	1	<b>33.3</b>	51.65 ± 0.26
	NS	<b>73</b>	244	32.8	52	6.4	6.7	1	<b>31.7</b>	51.70 ± 0.23

# Cooperative Minibatching



- ⦿ Full-batch training has no redundant computation.
- ⦿ Minibatch training requires repetitive calculations.

# Cooperative Minibatching Contributions

- ⊙ Work vs. batch size relationship
- ⊙ Data & intra-layer parallelism: *Cooperative Minibatching*.
- ⊙ Same idea in serial execution: *Dependent Minibatching*.

# Cooperative Minibatching Background

- ⊙ GNN equation:  $H_s^{(l+1)} = f^{(l)}(H_s^{(l)}, \{H_t^{(l)} \mid t \in N(s)\})$
- ⊙ Graph Sampling:  $S^0 = S, \quad S^{(l+1)} = S^l \cup N(S^l)$
- ⊙ Work of an epoch with batch size  $|S^0|$ :  
$$W(|S^0|) = \frac{|V|}{|S^0|} \sum_{l=1}^L E[|S^l|] \geq \frac{|V|}{|S^0|} \sum_{l=1}^L |S^0| = L|V|$$



# Cooperative Minibatching Background

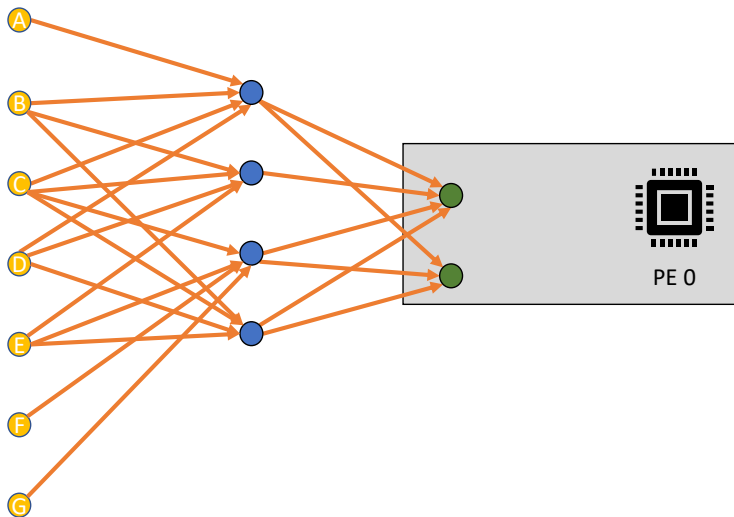
- ⊙ GNN equation:  $H_s^{(l+1)} = f^{(l)}(H_s^{(l)}, \{H_t^{(l)} \mid t \in N(s)\})$
- ⊙ Graph Sampling:  $S^0 = S, \quad S^{(l+1)} = S^l \cup N(S^l)$
- ⊙ Work of an epoch with batch size  $|S^0|$ :  
$$W(|S^0|) = \frac{|V|}{|S^0|} \sum_{l=1}^L E[|S^l|] \geq \frac{|V|}{|S^0|} \sum_{l=1}^L |S^0| = L|V|$$
- ⊙ Redundant work for layer  $l$ :  $W^l(|S^0|) \approx \frac{E[|S^l|]}{|S^0|}$

# Independent Minibatching

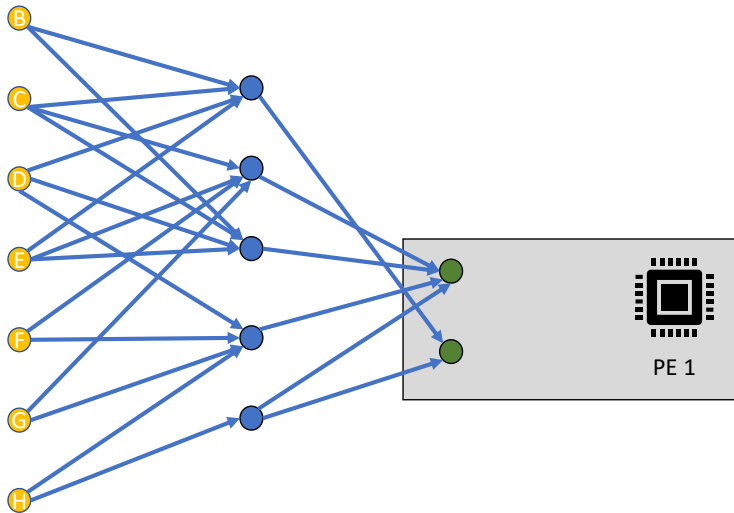
1. Each Processing Element (PE e.g., GPU) starts with its own  $S^0$ .
2. Samples  $S^1, \dots, S^L$  independently.
3. Loads input features and edge features for the sampled subgraphs independently.
4. Forward-backward independently with no communication.

⊙ **Problem: Redundant computations across PEs.**

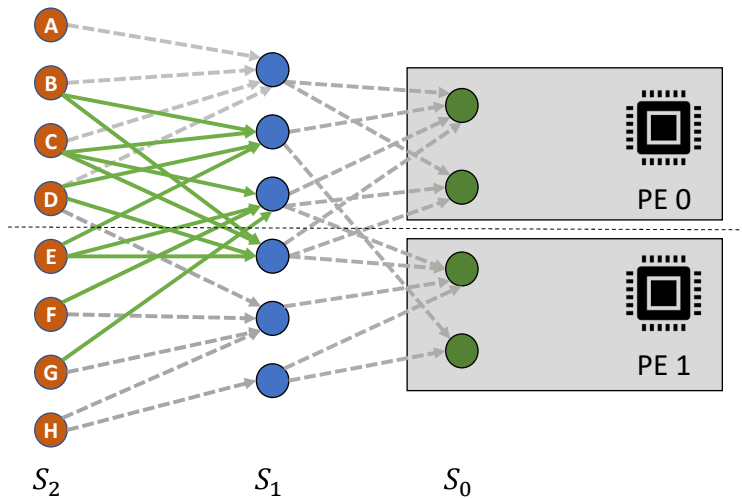
# Independent Minibatching Example with 2 PEs



# Independent Minibatching Example with 2 PEs



# Cooperative Minibatching



Green edges represent work savings.

## Theorem

*The work per epoch  $\frac{E[|S^l|]}{|S^0|}$  required to train a GNN model using minibatch training is monotonically nonincreasing as the batch size  $|S^0|$  increases.*

## Theorem

*The work per epoch  $\frac{E[|S^l|]}{|S^0|}$  required to train a GNN model using minibatch training is monotonically nonincreasing as the batch size  $|S^0|$  increases.*

## Theorem

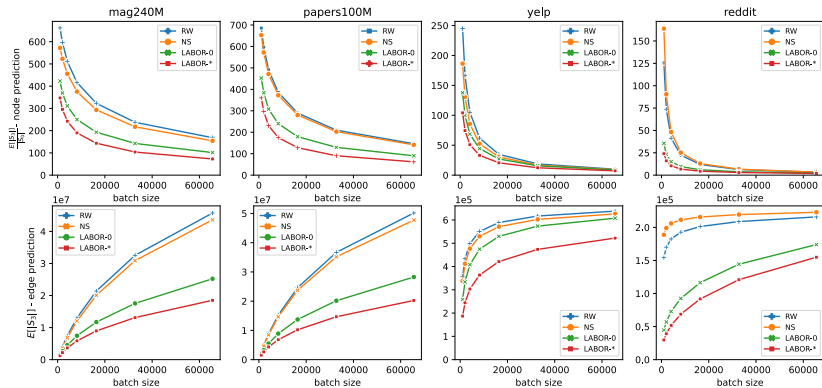
*The expected subgraph size  $E[|S^l|]$  required to train a GNN model using minibatch training is a concave function of batch size,  $|S^0|$ .*

Properties of the datasets used in experiments.

Dataset	$ V $	$ E $	$\frac{ E }{ V }$	# feats.	cache size	train - val - test (%)	# minibatches
flickr	89.2K	900K	10.09	500	70k	50.00 - 25.00 - 25.00	65
yelp	717K	14.0M	19.52	300	200k	75.00 - 10.00 - 15.00	595
products	2.45M	61.9M	25.26	100	400k	8.00 - 2.00 - 90.00	239
reddit	233K	115M	493.56	602	60k	66.00 - 10.00 - 24.00	172
papers100M	111M	3.2B	29.10	128	2M	1.09 - 0.11 - 0.19	1300
mag240M	244M	3.44B	14.16	768	2M	0.45 - 0.06 - 0.04	1215



# Empirical results about theorems



Monotonicity of the work. x-axis shows the batch size, y-axis shows  $\frac{E[|S^3|]}{|S^3|}$  (work per epoch) for node prediction (top row) and  $E[|S^3|]$  (expected subgraph size) for edge prediction (bottom row).

---

**Algorithm** Cooperative minibatching

---

- 1: **Input:** seed vertices  $S_p^0$  for each PE  $p \in P$ , # layers  $L$
  - 2: **for all**  $l \in \{0, \dots, L-1\}$  **do** {Sampling}
  - 3:   **for all**  $p \in P$  **do in parallel**
  - 4:     Sample next layer vertices  $\tilde{S}_p^{l+1}$  and edges  $E_p^l$  for  $S_p^l$
  - 5:     all-to-all to redistribute vertex ids for  $\tilde{S}_p^{l+1}$  to get  $S_p^{l+1}$
  - 6: **for all**  $p \in P$  **do in parallel** {Feature Loading}
  - 7:   Load input features  $H_p^l$  from Storage for vertices  $S_p^l$
  - 8:   all-to-all to redistribute  $H_p^l$  to get  $\tilde{H}_p^l$
  - 9: **for all**  $l \in \{L-1, \dots, 0\}$  **do** {Forward Pass}
  - 10:   **for all**  $p \in P$  **do in parallel**
  - 11:     **if**  $l+1 < L$  **then**
  - 12:       all-to-all to redistribute  $H_p^{l+1}$  to get  $\tilde{H}_p^{l+1}$
  - 13:       Forward pass on bipartite graph  $\tilde{S}_p^{l+1} \rightarrow S_p^l$  with edges  $E_p^l$  with input  $\tilde{H}_p^{l+1}$  and output  $H_p^l$
  - 14:   **for all**  $p \in P$  **do in parallel**
  - 15:     Compute the loss and initialize gradients  $G_p^0$
  - 16: **for all**  $l \in \{0, \dots, L-1\}$  **do** {Backward Pass}
  - 17:   **for all**  $p \in P$  **do in parallel**
  - 18:     Backward pass on bipartite graph  $S_p^l \rightarrow \tilde{S}_p^{(l+1)}$  with edges  $E_p^l$  with input  $G_p^l$  and output  $\tilde{G}_p^{l+1}$
  - 19:     **if**  $l+1 < L$  **then**
  - 20:       all-to-all to redistribute  $\tilde{G}_p^{l+1}$  to get  $G_p^{l+1}$
-

Any parallel algorithm can be executed sequentially.

1. Sample a mega-batch of size  $\kappa\beta$ .

# Dependent Minibatching

Any parallel algorithm can be executed sequentially.

1. Sample a mega-batch of size  $\kappa\beta$ .
2. Extract  $\kappa$  minibatches of size  $\beta$  from it.

Any parallel algorithm can be executed sequentially.

1. Sample a mega-batch of size  $\kappa\beta$ .
2. Extract  $\kappa$  minibatches of size  $\beta$  from it.
3. **Static sampled neighborhoods for  $\kappa$  consecutive minibatches.**

# Smoothed Dependent Minibatching

- ⊙ Problem: Sampled neighborhoods change instantly every  $\kappa$  iterations.

# Smoothed Dependent Minibatching

- ⊙ Problem: Sampled neighborhoods change instantly every  $\kappa$  iterations.
- ⊙ Solution: continuous change.

# Smoothed Dependent Minibatching

- ⊙ Problem: Sampled neighborhoods change instantly every  $\kappa$  iterations.
- ⊙ Solution: continuous change.
- ⊙ LABOR-0 recap:  
( $t \rightarrow s$ ) is sampled if  $r_t \leq \frac{k}{d_s}$ ,  $r_t \sim U(0, 1)$ .



# Smoothed Dependent Minibatching

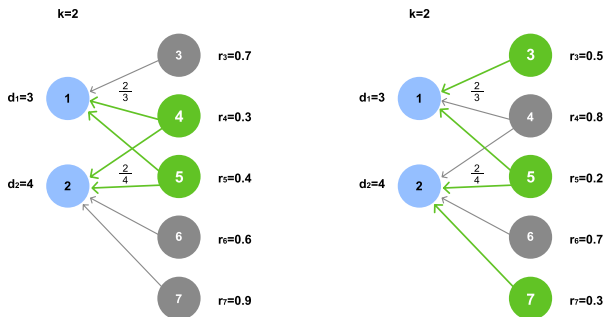
- ⊙ Problem: Sampled neighborhoods change instantly every  $\kappa$  iterations.
- ⊙ Solution: continuous change.
- ⊙ LABOR-0 recap:  
( $t \rightarrow s$ ) is sampled if  $r_t \leq \frac{k}{d_s}$ ,  $r_t \sim U(0, 1)$ .
- ⊙  $r_t$  evolves  $\kappa$  times slower.

# Smoothed Dependent Minibatching

- ⊙ Problem: Sampled neighborhoods change instantly every  $\kappa$  iterations.
- ⊙ Solution: continuous change.
- ⊙ LABOR-0 recap:  
( $t \rightarrow s$ ) is sampled if  $r_t \leq \frac{k}{d_s}$ ,  $r_t \sim U(0, 1)$ .
- ⊙  $r_t$  evolves  $\kappa$  times slower.
- ⊙ **Result: Increased temporal access locality.**

# Dependent Minibatching example for LABOR-0 with $\kappa = 2$

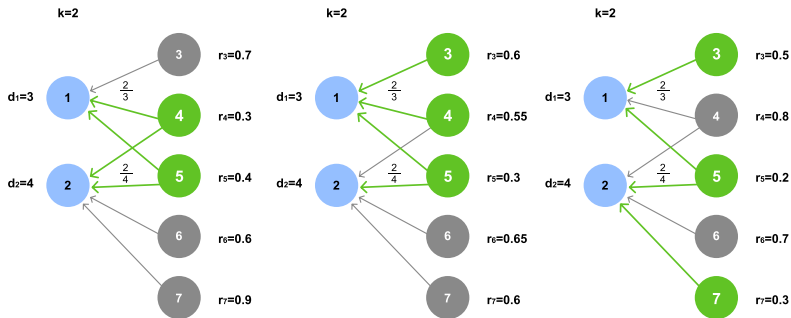
Blue: seed vertices, green: sampled, gray: not sampled



Two completely independent minibatches.

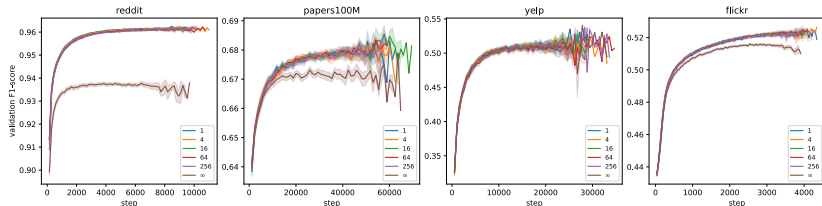
# Dependent Minibatching example for LABOR-0 with $\kappa = 2$

Blue: seed vertices, green: sampled, gray: not sampled

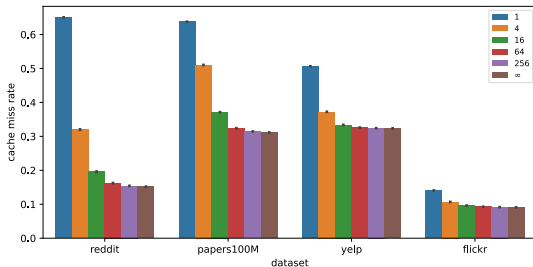


Middle minibatch is an interpolated between 2 independent minibatches.

# Experimental Results - Dependent Minibatching



The validation F1-score for LABOR-0 varying  $\kappa$ .



Cache miss rates varying  $\kappa$ .

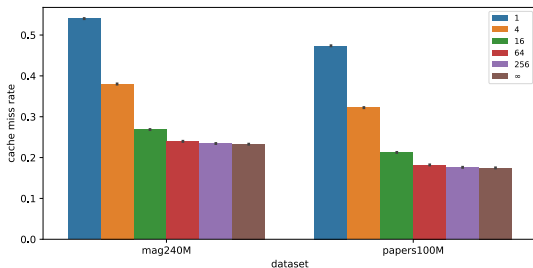
# Cooperative Minibatching Runtime Results

# PEs, $\gamma$ $\alpha, \beta,  S^0 $	Dataset & Model	Sampler	I/C	Samp.	Feature Copy			F/B	Total
					-	Cache	Cache, $\kappa$		
4 A100 $\gamma = 2\text{TB/s}$ $\alpha = 600\text{GB/s}$ $\beta = 64\text{GB/s}$ $ S^0  = 2^{12}$	papers100M GCN	LABOR-o	Indep	21.7	18.4	16.8	<b>11.2</b>	8.9	41.8
			Coop	17.7	14.0	10.1	<b>5.8</b>	13.0	36.5
	papers100M GCN	NS	Indep	16.1	26.5	<b>22.1</b>	-	10.1	48.3
			Coop	11.9	21.3	<b>12.9</b>	-	15.0	39.8
	mag240M R-GCN	LABOR-o	Indep	26.0	57.9	56.0	<b>41.0</b>	199.9	266.9
			Coop	20.0	51.1	36.9	<b>23.4</b>	183.3	226.7
mag240M R-GCN	NS	Indep	14.4	78.0	<b>71.2</b>	-	223.0	308.6	
		Coop	12.3	73.9	47.5	-	215.6	275.4	
8 A100 $\gamma = 2\text{TB/s}$ $\alpha = 600\text{GB/s}$ $\beta = 64\text{GB/s}$ $ S^0  = 2^{13}$	papers100M GCN	LABOR-o	Indep	21.3	21.1	18.7	<b>12.0</b>	9.3	42.6
			Coop	16.5	12.4	7.1	<b>4.0</b>	13.5	34.0
	papers100M GCN	NS	Indep	15.8	31.0	<b>24.5</b>	-	10.3	50.6
			Coop	12.5	19.4	9.0	-	15.6	37.1
	mag240M R-GCN	LABOR-o	Indep	30.6	70.1	66.2	<b>46.8</b>	202.1	279.5
			Coop	21.6	50.6	29.0	<b>19.3</b>	172.2	213.1
mag240M R-GCN	NS	Indep	15.0	94.9	<b>80.9</b>	-	224.8	320.7	
		Coop	14.9	71.6	39.6	-	209.0	263.5	
16 V100 $\gamma = 0.9\text{TB/s}$ $\alpha = 300\text{GB/s}$ $\beta = 32\text{GB/s}$ $ S^0  = 2^{13}$	papers100M GCN	LABOR-o	Indep	39.1	44.5	40.2	<b>29.4</b>	15.1	83.6
			Coop	26.9	22.7	10.4	<b>4.9</b>	19.1	50.9
	papers100M GCN	NS	Indep	18.0	61.3	<b>52.0</b>	-	16.2	86.2
			Coop	19.2	34.9	13.0	-	21.3	53.5
	mag240M R-GCN	LABOR-o	Indep	50.8	128.8	121.3	<b>96.2</b>	156.1	303.1
			Coop	29.2	78.1	42.8	<b>23.5</b>	133.3	186.0
mag240M R-GCN	NS	Indep	19.3	167.3	<b>152.6</b>	-	170.9	342.8	
		Coop	19.3	116.1	53.1	-	160.4	232.8	

# Speedup with changing # PEs

Improvements of cooperative batching over independent batching for 4, 8 and 16 PEs compiled from the **Total** column.

Dataset & Model	Sampler	Speedup - 4	Speedup - 8	Speedup - 16
papers100M GCN	LABOR-o	14.5%	25.3%	64.2%
	NS	21.4%	36.4%	61.1%
mag240M R-GCN	LABOR-o	17.7%	31.2%	63.0%
	NS	12.1%	21.7%	47.3%



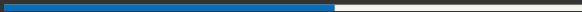
4 cooperating PEs were used,  $\kappa$  is varied (1, 4, ...).

# Cooperative Minibatching Summary

- ⊙ Work reduction with Cooperative Minibatching
- ⊙ Communication negligible in expensive GNN models
- ⊙ PEs need fast all-to-all communication
- ⊙ Less work = faster runtimes as expected.



dgl.graphbolt



- ⊙ Graph sampling (Graph caching on the accelerator)
- ⊙ Feature loading (Feature caching on accelerator and system memory)
- ⊙ Forward-backward on the accelerator

# Proposed Asynchronous Algorithm to Hide Latencies

---

## Algorithm Optimized Cooperative Minibatching

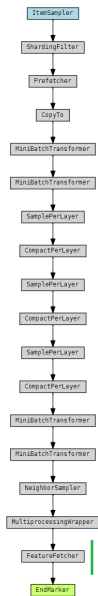
---

- 1: **Input:** seed vertices  $S_p^0$  for each PE  $p \in P$ , # layers  $L$
  - 2: **for all**  $l \in \{0, \dots, L - 1\}$  **do** {Sampling}
  - 3:   **for all**  $p \in P$  **do in parallel**
  - 4:     Query GPU graph cache and fetch vertex neighborhoods
  - 5:     co\_await Load missing vertex neighborhoods from Storage
  - 6:     Sample next layer vertices  $\tilde{S}_p^{l+1}$  and edges  $E_p^l$  for  $S_p^l$
  - 7:     co\_await all-to-all to redistribute vertex ids for  $\tilde{S}_p^{l+1}$  to get  $S_p^{l+1}$
  - 8: **for all**  $p \in P$  **do in parallel** {Feature Loading}
  - 9:   Query GPU feature cache and fetch existing input features
  - 10: co\_await Load missing features  $H_p^L$  from Storage for vertices  $S_p^L$
  - 11: co\_await all-to-all to redistribute  $H_p^L$  to get  $\tilde{H}_p^L$
-

# dgl.graphbolt - Constructing a dataloader

```
1 # We use torch datapipe below, it describes what the
  # operations are, does not run the operations.
2
3 def create_dataloader(graph, features, itemset,
4                       batch_size, fanout, device):
5     datapipe = gb.ItemSampler(
6         itemset, batch_size=batch_size, shuffle=True
7     )
8     datapipe = datapipe.copy_to(device=device,
9                               extra_attrs=["seed_nodes"])
10    datapipe = datapipe.sample_layer_neighbor(graph,
11                                             fanout)
12    datapipe = datapipe.fetch_feature(features,
13                                     node_feature_keys=["feat"])
14    return gb.DataLoader(datapipe)
```

# Pipeline



# Feature Fetching

- ⦿ 'features' is a huge tensor stored on CPU.

# Feature Fetching

- ⦿ 'features' is a huge tensor stored on CPU.
- ⦿ Compute: 'features[index].to(device)'

# Feature Fetching

- ⦿ 'features' is a huge tensor stored on CPU.
- ⦿ Compute: 'features[index].to(device)'
- ⦿ First, 'features[index]' is executed on CPU, then '.to(device)' is performed.



# Feature Fetching

- ⊙ 'features' is a huge tensor stored on CPU.
- ⊙ Compute: 'features[index].to(device)'
- ⊙ First, 'features[index]' is executed on CPU, then '.to(device)' is performed.
- ⊙ This requires multiple touches to the same data, can we do better?

- ⦿ GPUs are made of SMs (Streaming Multiprocessors), similar to cores in CPUs.

- ⦿ GPUs are made of SMs (Streaming Multiprocessors), similar to cores in CPUs.
- ⦿ **CUDAStreams are queues to schedule work to the GPU.**

- ⊙ GPUs are made of SMs (Streaming Multiprocessors), similar to cores in CPUs.
- ⊙ CUDAStreams are queues to schedule work to the GPU.
- ⊙ If the work launched in a stream does not occupy all the SMs, the work from another stream can run concurrently.

- ⦿ Compute: `'features[index].to(device)'`

## Pinned memory (Page locked memory)

- ⦿ Compute: `features[index].to(device)`
- ⦿ Pinned (page-locked) memory pages can directly be accessed by the GPU.

## Pinned memory (Page locked memory)

- ⦿ Compute: `features[index].to(device)`
- ⦿ Pinned (page-locked) memory pages can directly be accessed by the GPU.
- ⦿ **Pin the memory of features.**

## Pinned memory (Page locked memory)

- ⊙ Compute: `features[index].to(device)`
- ⊙ Pinned (page-locked) memory pages can directly be accessed by the GPU.
- ⊙ Pin the memory of features.
- ⊙ GPU can directly perform `features[index].to(device)` in a single fused kernel.



## Pinned memory (Page locked memory)

- ⊙ Compute: `features[index].to(device)`
- ⊙ Pinned (page-locked) memory pages can directly be accessed by the GPU.
- ⊙ Pin the memory of features.
- ⊙ GPU can directly perform `features[index].to(device)` in a single fused kernel.
- ⊙ **6144 threads are enough to saturate the PCI-e bandwidth.**

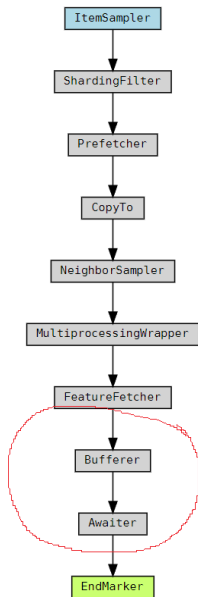
## Pinned memory (Page locked memory)

- ⊙ Compute: 'features[index].to(device)'
- ⊙ Pinned (page-locked) memory pages can directly be accessed by the GPU.
- ⊙ Pin the memory of features.
- ⊙ GPU can directly perform features[index].to(device) in a single fused kernel.
- ⊙ 6144 threads are enough to saturate the PCI-e bandwidth.
- ⊙ Separate CUDASTream → overlap PCI-e copy and computation.

# Pipelining optimizations in gb.DataLoader - Feature Fetch

```
1 torch.ops.graphbolt.set_max_uva_threads(max_uva_threads)
2 feature_fetchers = dp_utils.find_dps(
3     datapipe_graph,
4     FeatureFetcher,
5 )
6 for feature_fetcher in feature_fetchers:
7     feature_fetcher.stream = _get_uva_stream()
8     datapipe_graph = dp_utils.replace_dp(
9         datapipe_graph,
10        feature_fetcher,
11        feature_fetcher.buffer(1).wait(),
12    )
```

# Transformed Pipeline



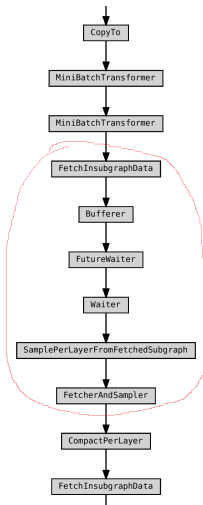
# Pipelining optimizations in gb.DataLoader - Graph Fetch

```
1 torch.ops.graphbolt.set_max_uva_threads(max_uva_threads)
2 samplers = dp_utils.find_dps(
3     datapipe_graph,
4     SamplePerLayer,
5 )
6 executor = ThreadPoolExecutor(max_workers=1)
7 for sampler in samplers:
8     datapipe_graph = dp_utils.replace_dp(
9         datapipe_graph,
10        sampler,
11        sampler.fetch_and_sample(_get_uva_stream(),
12        executor, 1),
13    )
```

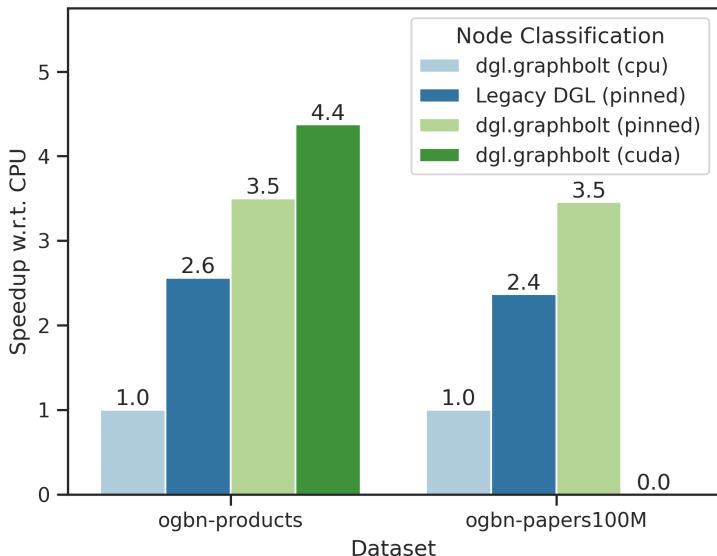
# Pipelining optimizations in gb.DataLoader - Graph Fetch

```
1 @functional_datapipe("fetch_and_sample")
2 class FetcherAndSampler(MiniBatchTransformer):
3     """Overlapped graph sampling operation replacement.
4     """
5     def __init__(self, sampler, stream, executor,
6                 buffer_size):
7         datapipe = sampler.datapipe.
8         fetch_insubgraph_data(
9             sampler, stream, executor
10        )
11        datapipe = datapipe.buffer(buffer_size).
12        wait_future().wait()
13        datapipe = datapipe.
14        sample_per_layer_from_fetched_subgraph(sampler)
15        super().__init__(datapipe)
```

# Transformed Pipeline

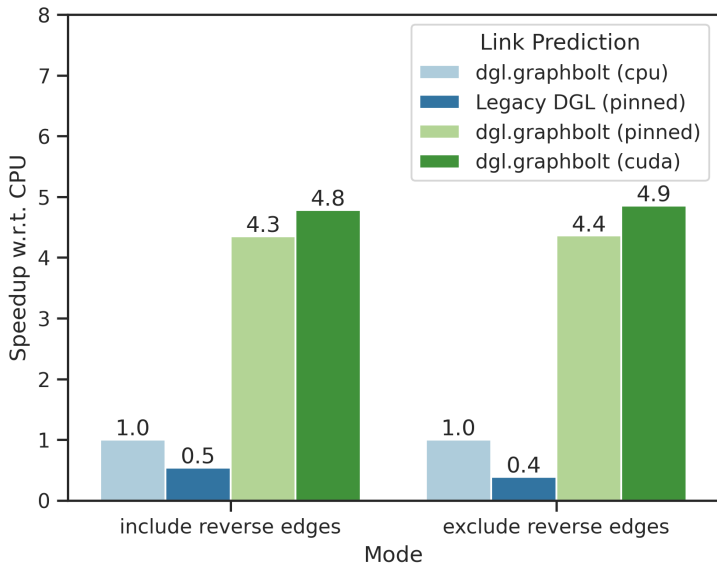


# DGL 2.1 Node Classification Speedups on ogbn-products

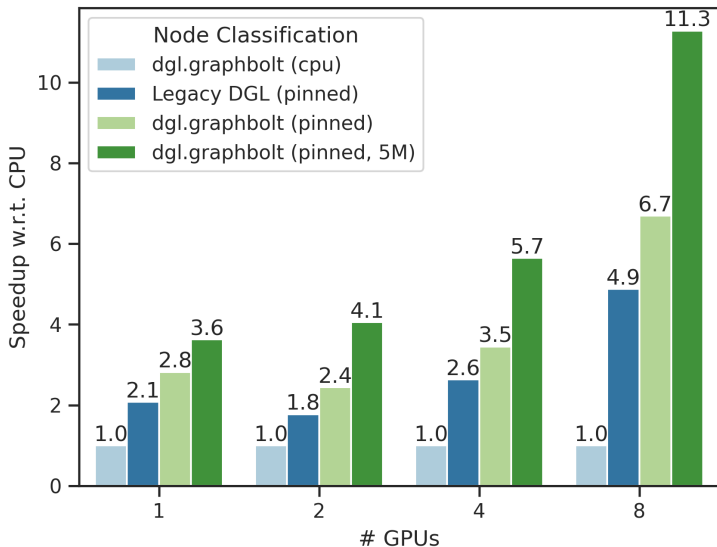




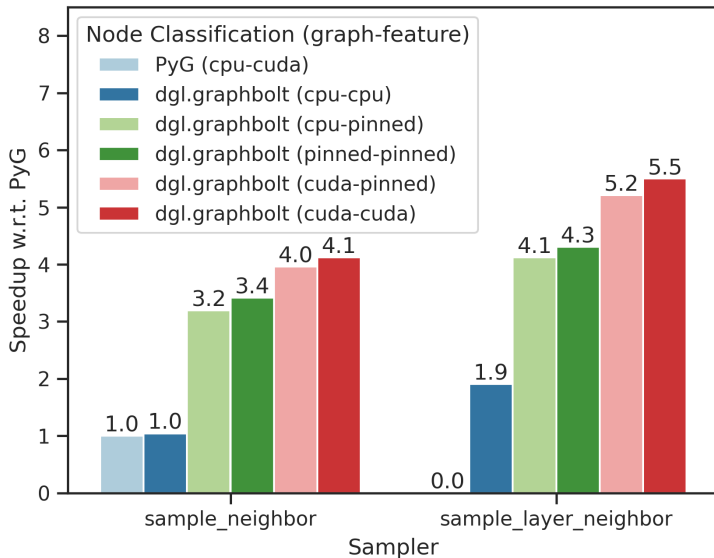
# DGL 2.1 Link Prediction Speedups on ogbl-citation2



# Multi-GPU Speedups on ogbn-papers100M



# Single-GPU Speedups with PyG on ogbn-products



- ⦿ Given a chain of datapipes, we want to enable Cooperative Minibatching seamlessly.

# Cooperative Minibatching

- ⦿ Given a chain of datapipes, we want to enable Cooperative Minibatching seamlessly.
- ⦿ Insert all-to-all operations in-between, similar to the graph and feature fetch optimizations.

# Cooperative Minibatching

- ⦿ Given a chain of datapipes, we want to enable Cooperative Minibatching seamlessly.
- ⦿ Insert all-to-all operations in-between, similar to the graph and feature fetch optimizations.
- ⦿ **Overlap communication**

# Cooperative Minibatching

- ⊙ Given a chain of datapipes, we want to enable Cooperative Minibatching seamlessly.
- ⊙ Insert all-to-all operations in-between, similar to the graph and feature fetch optimizations.
- ⊙ Overlap communication
- ⊙ Simple online graph partitioning can reduce the amount of needed communication between GPUs.

# Thanks!

- ◎ For more information
  - Email: [balin@gatech.edu](mailto:balin@gatech.edu)
  - Visit: [mfbal.in](http://mfbal.in)
  - Visit: [tda.gatech.edu](http://tda.gatech.edu)
- ◎ Acknowledgement of Support:







THANK YOU

# Variance formula derivation




$$\begin{aligned}H'_s &= \frac{1}{d_s} \sum_{t \rightarrow s} \frac{M_t}{\pi_t} \mathbb{1}[r_t \leq \pi_t] \\ \text{Var}(H'_s) &= \text{Var}\left(\frac{1}{d_s} \sum_{t \rightarrow s} \frac{M_t}{\pi_t} \mathbb{1}[r_t \leq \pi_t]\right) \\ &= \frac{1}{d_s^2} \sum_{t \rightarrow s} \frac{\text{Var}(M_t)}{\pi_t^2} \text{Var}(\mathbb{1}[r_t \leq \pi_t]) \\ &= \frac{1}{d_s^2} \sum_{t \rightarrow s} \frac{\text{Var}(M_t)}{\pi_t^2} \pi_t(1 - \pi_t) \\ &= \frac{1}{d_s^2} \sum_{t \rightarrow s} \frac{\text{Var}(M_t)}{\pi_t} (1 - \pi_t) = \frac{1}{d_s^2} \sum_{t \rightarrow s} \frac{1}{\pi_t} (1 - \pi_t) \\ &= \frac{1}{d_s^2} \sum_{t \rightarrow s} \left(\frac{1}{\pi_t} - 1\right) = \frac{1}{d_s^2} \sum_{t \rightarrow s} \frac{1}{\pi_t} - \frac{1}{d_s}\end{aligned} \tag{1}$$

## Smoothed Dependent Minibatching (cont.)



- ⊙  $r_t = PRNG(z, t)$ , where  $z$  is the random seed,  $t$  is the vertex id.
- ⊙  $r_t(c) = PRNG(z_1, z_2, c, t), \forall c \in [0, 1]$ ,  
 $PRNG(z_1, z_2, 0, t) = PRNG(z_1, t)$ ,  
 $PRNG(z_1, z_2, 1, t) = PRNG(z_2, t)$ .
- ⊙  $n_t(c) = \cos(\frac{c\pi}{2})n_t^1(z_1) + \sin(\frac{c\pi}{2})n_t^2(z_2)$
- ⊙  $r_t(c) = \Phi(n_t(c)) \sim U(0, 1)$
- ⊙ For  $i$ th minibatch,  $c = \frac{i}{\kappa}$ . When  $i = \kappa$ , we set  $z_1 \leftarrow z_2, z_2$  becomes a new random seed.

-  Muhammed Fatih Balın and Ümit V. Çatalyürek, *Layer-neighbor sampling — defusing neighborhood explosion in GNNs*, Thirty-seventh Conference on Neural Information Processing Systems, 2023.
-  Wei Lin Chiang, Yang Li, Xuanqing Liu, Samy Bengio, Si Si, and Cho Jui Hsieh, *Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks*, Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Association for Computing Machinery, jul 2019, pp. 257–266.
-  Jie Chen, Tengfei Ma, and Cao Xiao, *FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling*.


## References II


-  Yifan Chen, Tianning Xu, Dilek Hakkani-Tur, Di Jin, Yun Yang, and Ruoqing Zhu, *Calibrate and Debias Layer-wise Sampling for Graph Convolutional Networks*.
-  Jianfei Chen, Jun Zhu, and Le Song, *Stochastic training of graph convolutional networks with variance reduction*, 35th International Conference on Machine Learning, ICML 2018 3 (2018), 1503–1532.
-  Zhenkun Cai, Qihui Zhou, Xiao Yan, Da Zheng, Xiang Song, Chenguang Zheng, James Cheng, and George Karypis, *Dsp: Efficient gnn training with multiple gpus*, Proceedings of the 28th ACM SIGPLAN Annual



Symposium on Principles and Practice of Parallel Programming, PPOPP '23, 2023, p. 392–404.

-  Jialin Dong, Da Zheng, Lin F. Yang, and George Karypis, *Global Neighbor Sampling for Mixed CPU-GPU Training on Giant Graphs*, Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2021), 289–299.
-  Matthias Fey, Jan E. Lenssen, Frank Weichert, and Jure Leskovec, *Gnnautoscale: Scalable and expressive graph neural networks via historical embeddings*, Proceedings of the 38th International Conference on Machine Learning (Marina




Meila and Tong Zhang, eds.), Proceedings of Machine Learning Research, vol. 139, PMLR, Jul 2021, pp. 3294–3304.

 Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun, *Heterogeneous Graph Transformer*, The Web Conference 2020 - Proceedings of the World Wide Web Conference, WWW 2020 (2020), 2704–2710.

 Will Hamilton, Zhitao Ying, and Jure Leskovec, *Inductive representation learning on large graphs*, Advances in Neural Information Processing Systems (I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.



-  Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang, *Adaptive sampling towards fast graph representation learning*, Advances in Neural Information Processing Systems **2018-Decem** (2018), no. Nips, 4558–4567.
-  Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu, *Pagraph: Scaling gnn training on large graphs via computation-aware caching*, Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, 2020, p. 401–415.



-  Ziqi Liu, Zhengwei Wu, Zhiqiang Zhang, Jun Zhou, Shuang Yang, Le Song, and Yuan Qi, *Bandit samplers for training graph neural networks*, Advances in Neural Information Processing Systems **2020-Decem** (2020).
-  Yeonhong Park, Sunhong Min, and Jae W. Lee, *Ginex: Ssd-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching*, Proc. VLDB Endow. **15** (2022), no. 11, 2626–2639.
-  Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen mei Hwu, *Accelerating sampling and aggregation operations in gnn frameworks with gpu initiated direct storage accesses*, 2023.

## References VII

-  Zhihao Shi, Xize Liang, and Jie Wang, *LMC: Fast training of GNNs via subgraph sampling with provable convergence*, The Eleventh International Conference on Learning Representations, 2023.
-  Jie Sun, Mo Sun, Zheng Zhang, Jun Xie, Zuocheng Shi, Zihan Yang, Jie Zhang, Fei Wu, and Zeke Wang, *Helios: An efficient out-of-core gnn training system on terabyte-scale graphs with in-memory performance*, 2023.
-  Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman, *Mariusgnn: Resource-efficient out-of-core training of graph neural networks*, 2022.

-  Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou, *Gnnlab: A factored system for sample-based gnn training over gpus*, Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22, 2022, p. 417–434.
-  Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu, *Layer-dependent importance sampling for training deep and large graph convolutional networks*, Advances in Neural Information Processing Systems **32** (2019), no. NeurIPS.

-  Qingru Zhang, David Wipf, Quan Gan, and Le Song, *A Biased Graph Neural Network Sampler with Near-Optimal Regret*, no. NeurIPS, 1–25.
-  Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna, *GraphSAINT: Graph sampling based inductive learning method*, International Conference on Learning Representations, 2020.
-  Hanqing Zeng, Muhan Zhang, Yinglong Xia, Ajitesh Srivastava, Andrey Malevich, Rajgopal Kannan, Viktor Prasanna, Long Jin, and Ren Chen, *Decoupling the depth and scope of graph neural networks*, Advances in Neural

Information Processing Systems (A. Beygelzimer,  
Y. Dauphin, P. Liang, and J. Wortman Vaughan, eds.), 2021.