GNNLab: A Factored System for *Sample-based GNN Training* over GPUs

RONG CHEN

IPADS, SJTU

ASAIL 2022

Joint work with *Jianbang, Dahai, Xiaoniu @IPADS, Qiang @BASICS, and Lei, Wenyuan* @Ailibaba

Who AM I

Rong Chen (陈榕) / SJTU IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

https://ipads.se.sjtu.edu.cn/rong_chen

- Institute of Parallel And Distributed Systems (IPADS)
- Best paper award: EuroSys 2015, APsys 2017, and ICPP 2007
- 7 OSDI/SOSP papers and 9 EuroSys/Usenix ATC papers
- Huawei OlympusMons Pioneer Award, 2020

Research Interest

- Building *efficient*, scalable and reliable distributed framework

GNN Training

Whole-graph training: hard to scale

GNN Training

Whole-graph training: hard to scale

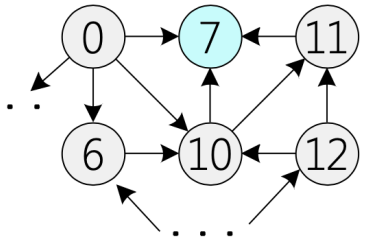
Sample-based training

- Systems: DGL, PyG, AliGraph[VLDB'19], P3[OSDI'21], . . . , GNNLab
- ***Friendly to GPU***: massive parallelisms and limited GPU memory
- SET model: Sample, Extract and Train

Sample-base GNN Training

SET model

Graph topc



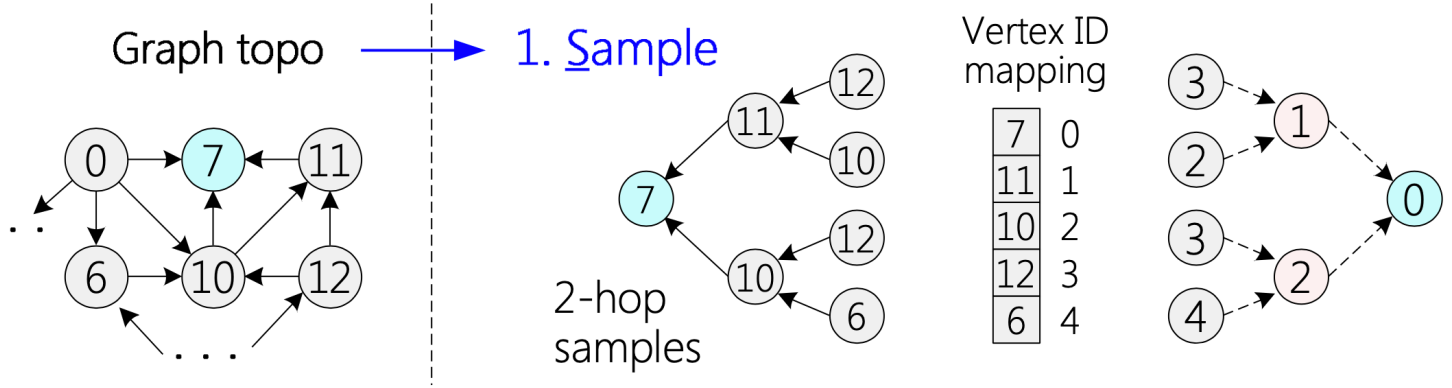
Features

0	··	6	7	··	10	11	12
A_1	··	G_1	H_1	··	K_1	L_1	M_1
A_2	··	G_2	H_2	··	K_2	L_2	M_2
A_3	··	G_3	H_3	··	K_3	L_3	M_3
A_4	··	G_4	H_4	··	K_4	L_4	M_4

Sample-base GNN Training

SET model

1. Sample



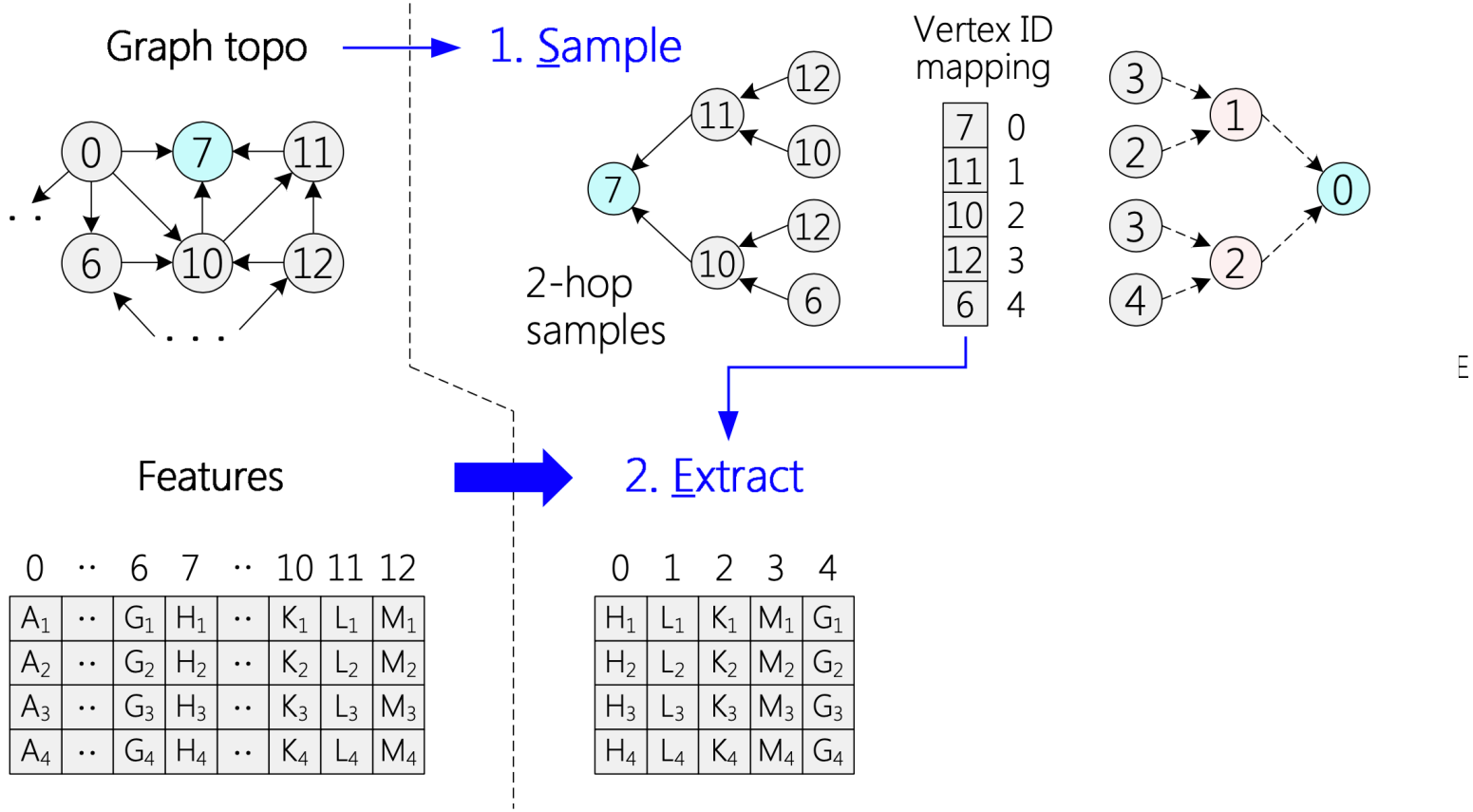
Features

0	..	6	7	..	10	11	12
A_1	..	G_1	H_1	..	K_1	L_1	M_1
A_2	..	G_2	H_2	..	K_2	L_2	M_2
A_3	..	G_3	H_3	..	K_3	L_3	M_3
A_4	..	G_4	H_4	..	K_4	L_4	M_4

Sample-base GNN Training

SET model

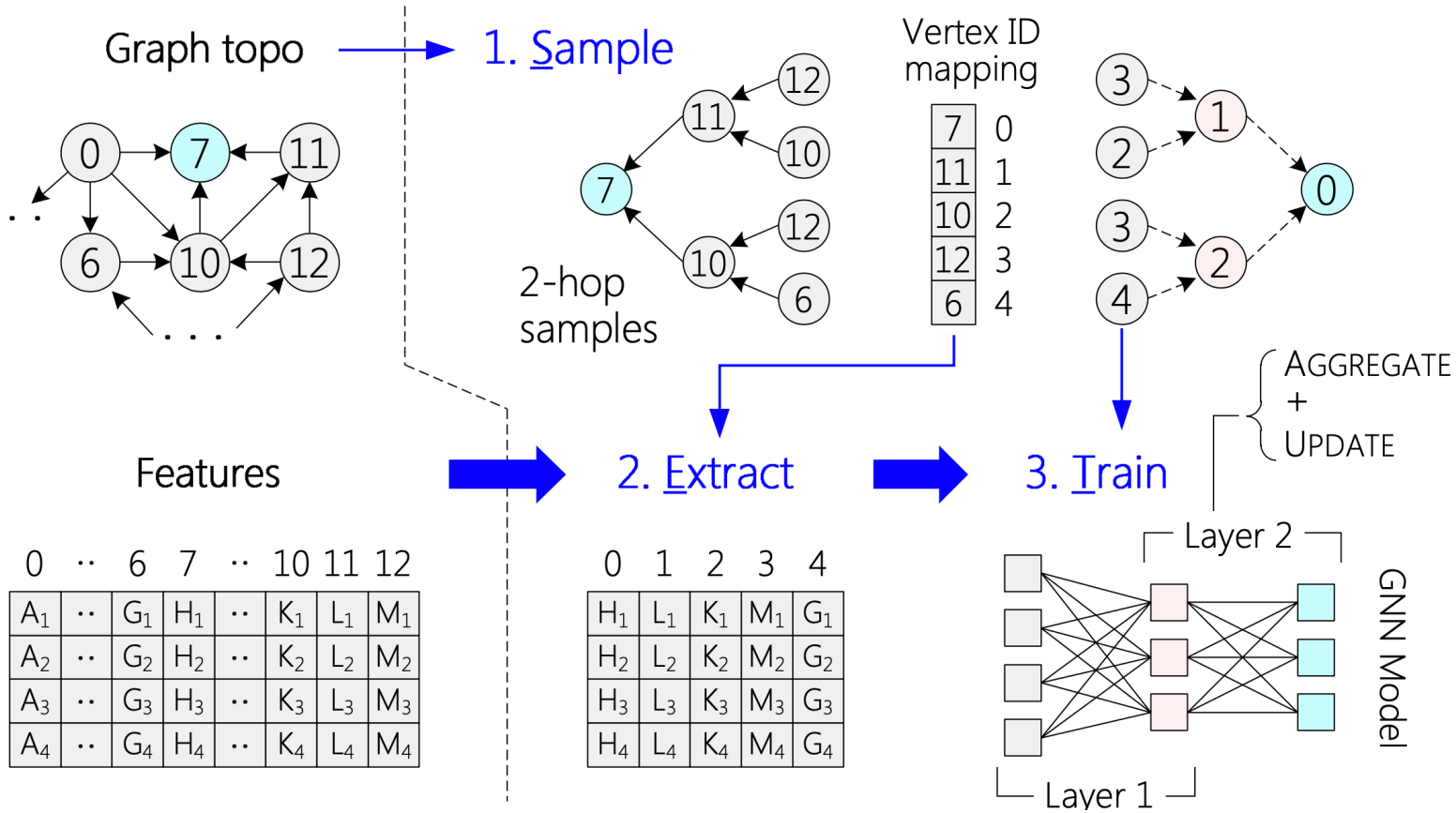
1. Sample
2. Extract



Sample-base GNN Training

SET model

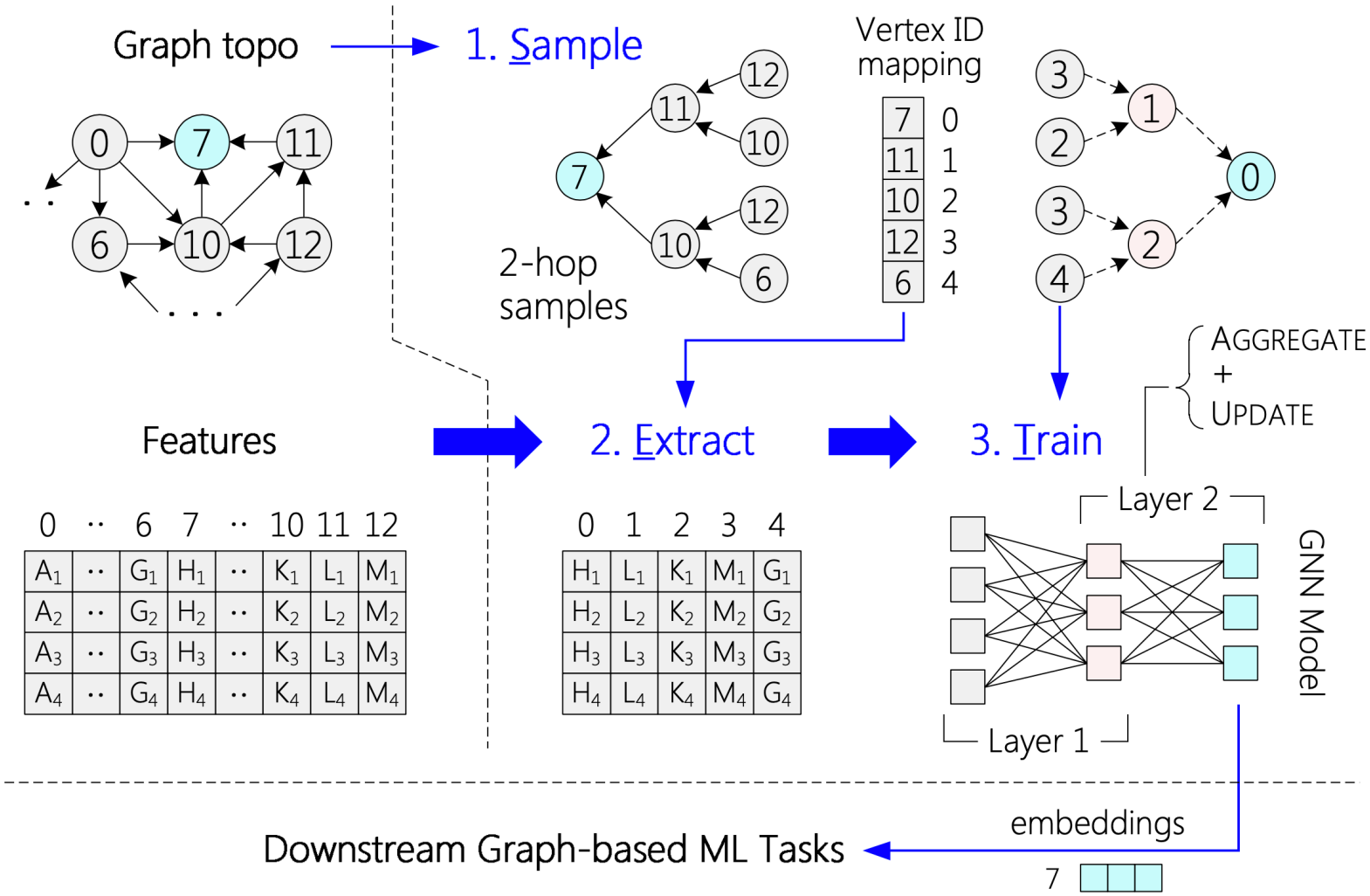
1. Sample
2. Extract
3. Train



Sample-base GNN Training

SET model

1. Sample
2. Extract
3. Train



GPU-based GNN Training

GPUs have been widely exploited to accelerate GNN training

- Train: almost

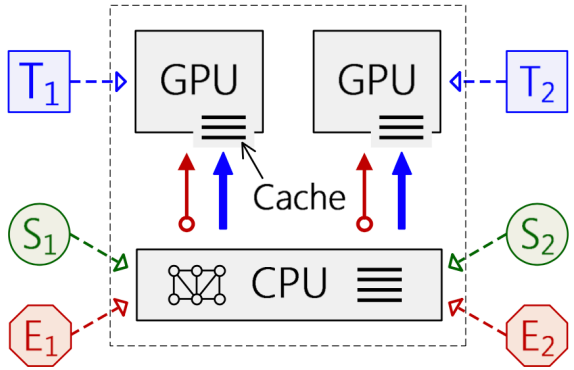
GNN Systems	Sample	Extract	Train	TOT
T_{SOTA}	2.93	5.55	**4.00**	12.50

GCN w/ 3-hop sampling on OGB-Papers100M

GPU-based GNN Training

GPUs have been widely exploited to accelerate GNN training

- Train: almost
- Extract: PaGraph[SOCC'20]



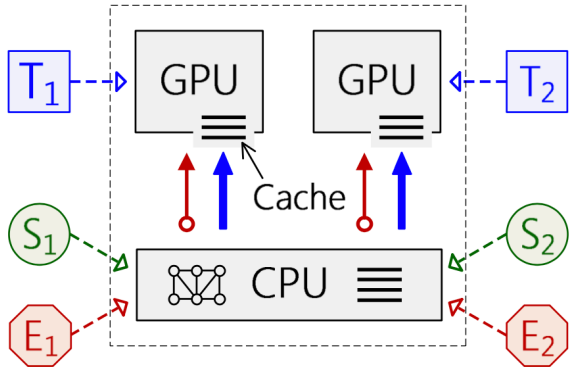
GNN Systems	Sample	Extract	Train	TOT
T_{SOTA}	2.93	5.55	4.00	12.50
w/ GPU-based Caching	2.88	**1.73**	4.00	8.62

GCN w/ 3-hop sampling on OGB-Papers100M

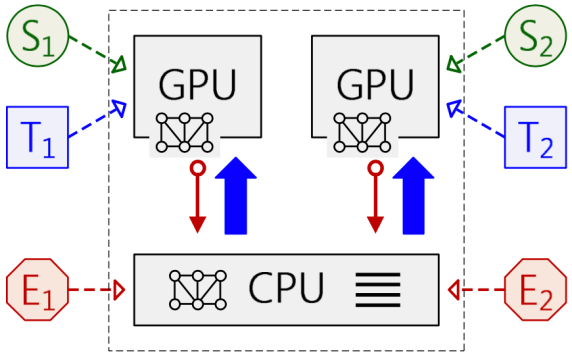
GPU-based GNN Training

GPUs have been widely exploited to accelerate GNN training

- Train: almost
- Extract: PaGraph[SOCC'20]
- Sample: DGL, NextDoor[EuroSys'21]



GNN Systems	Sample	Extract	Train	TOT
T_{SOTA}	2.93	5.55	4.00	12.50
w/ GPU-based Caching	2.88	1.73	4.00	8.62
w/ GPU-based Sampling	**0.70**	5.46	4.01	10.21

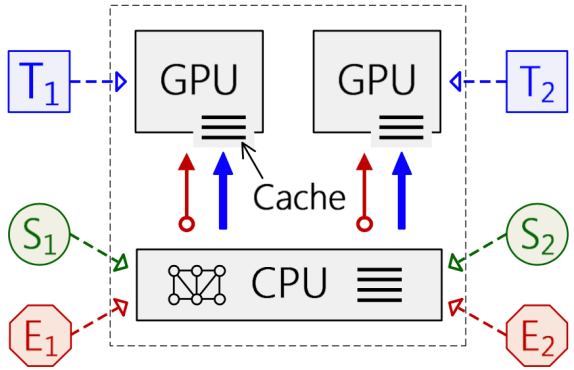


GCN w/ 3-hop sampling on OGB-Papers100M

GPU-based GNN Training

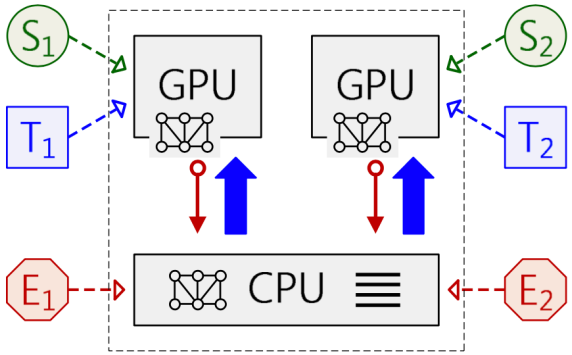
GPUs have been widely exploited to accelerate GNN training

- Train: almost
- Extract: PaGraph[SOCC'20]
- Sample: DGL, NextDoor[EuroSys'21]
- **Both ?**



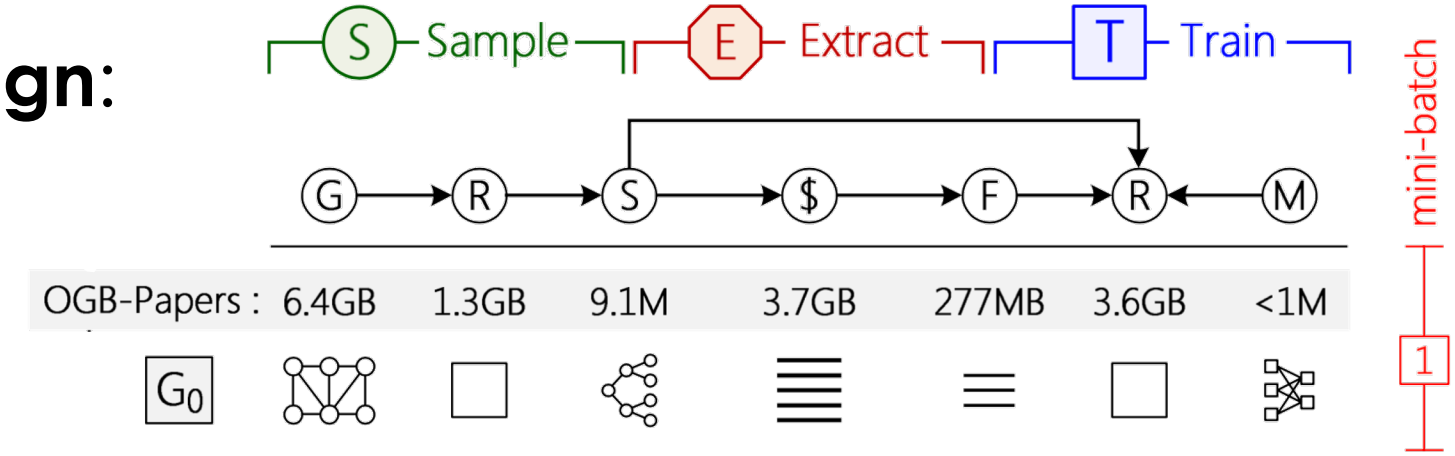
GNN Systems	Sample	Extract	Train	TOT
T_{SOTA}	2.93	5.55	4.00	12.50
w/ GPU-based Caching	2.88	1.73	4.00	8.62
w/ GPU-based Sampling	0.70	5.46	4.01	10.21
w/ Both	**0.70**	**3.62**	4.00	8.37

GCN w/ 3-hop sampling on OGB-Papers100M



Analysis

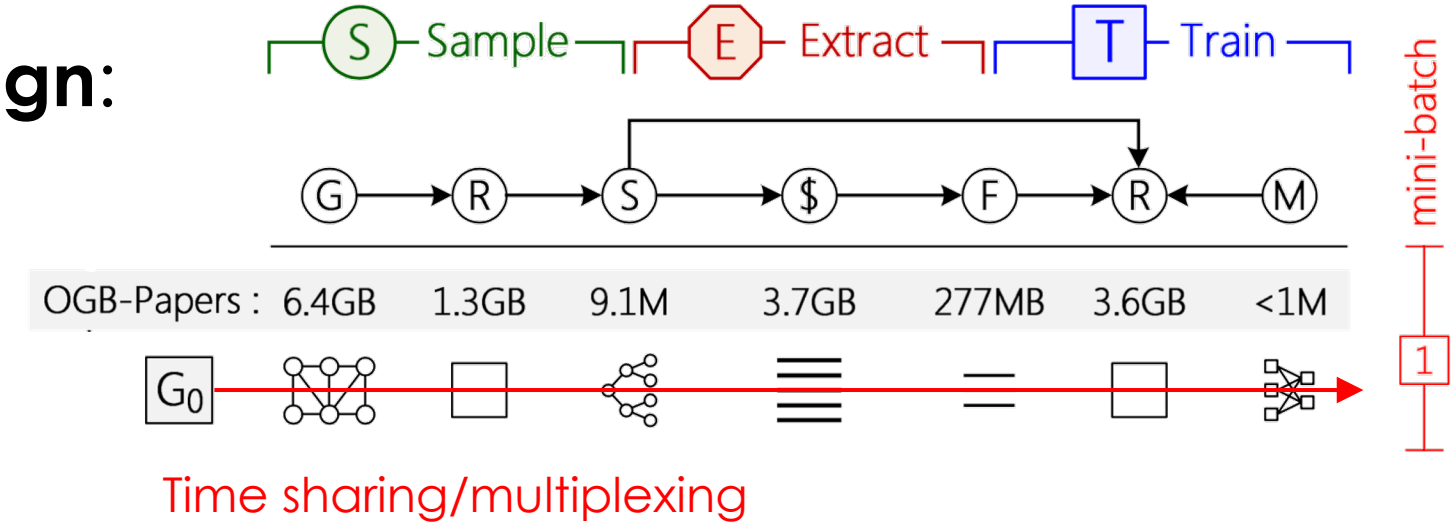
Traditional Design:



Ⓖ Graph topo Ⓢ Samples Ⓢ Cache Ⓕ Features Ⓜ Model Ⓡ Runtime state

Analysis

Traditional Design:



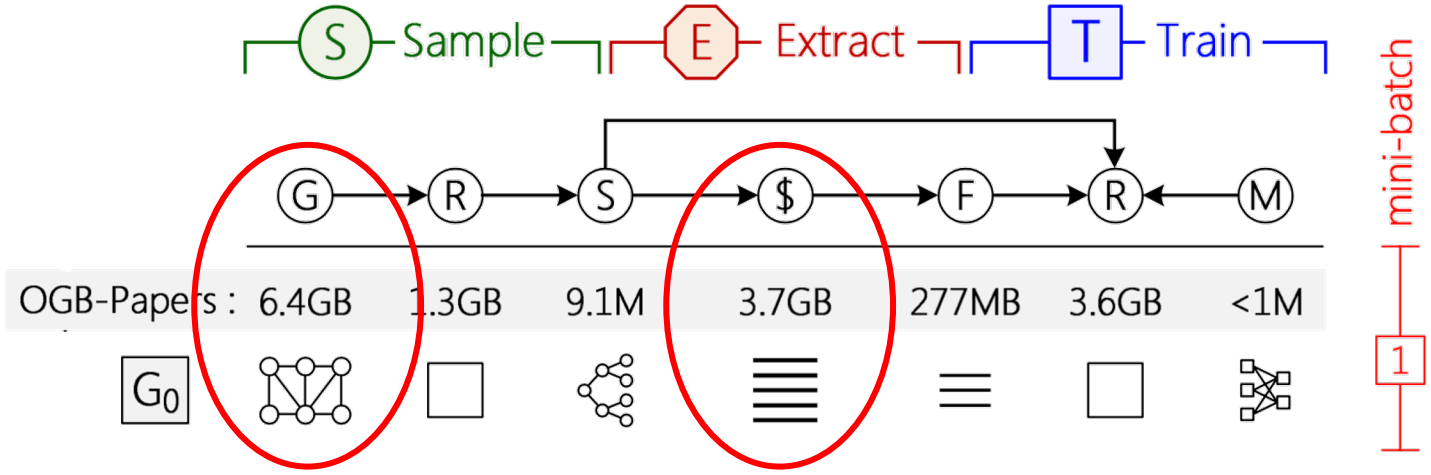
Time sharing/multiplexing

Ⓖ Graph topo Ⓢ Samples Ⓢ Cache Ⓕ Features Ⓜ Model Ⓡ Runtime state

Analysis

Problems

- Capacity



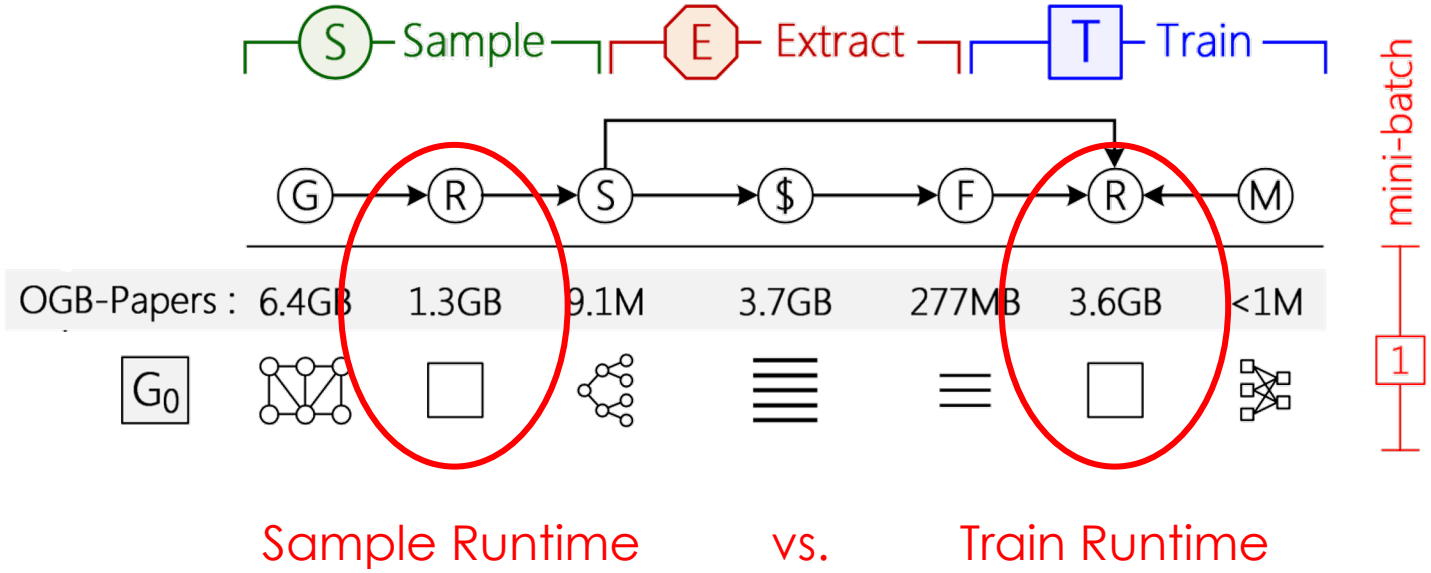
Graph Topo vs. Feature cache

Ⓖ Graph topo Ⓢ Samples Ⓒ Cache Ⓕ Features Ⓜ Model Ⓡ Runtime state

Analysis

Problems

- Capacity

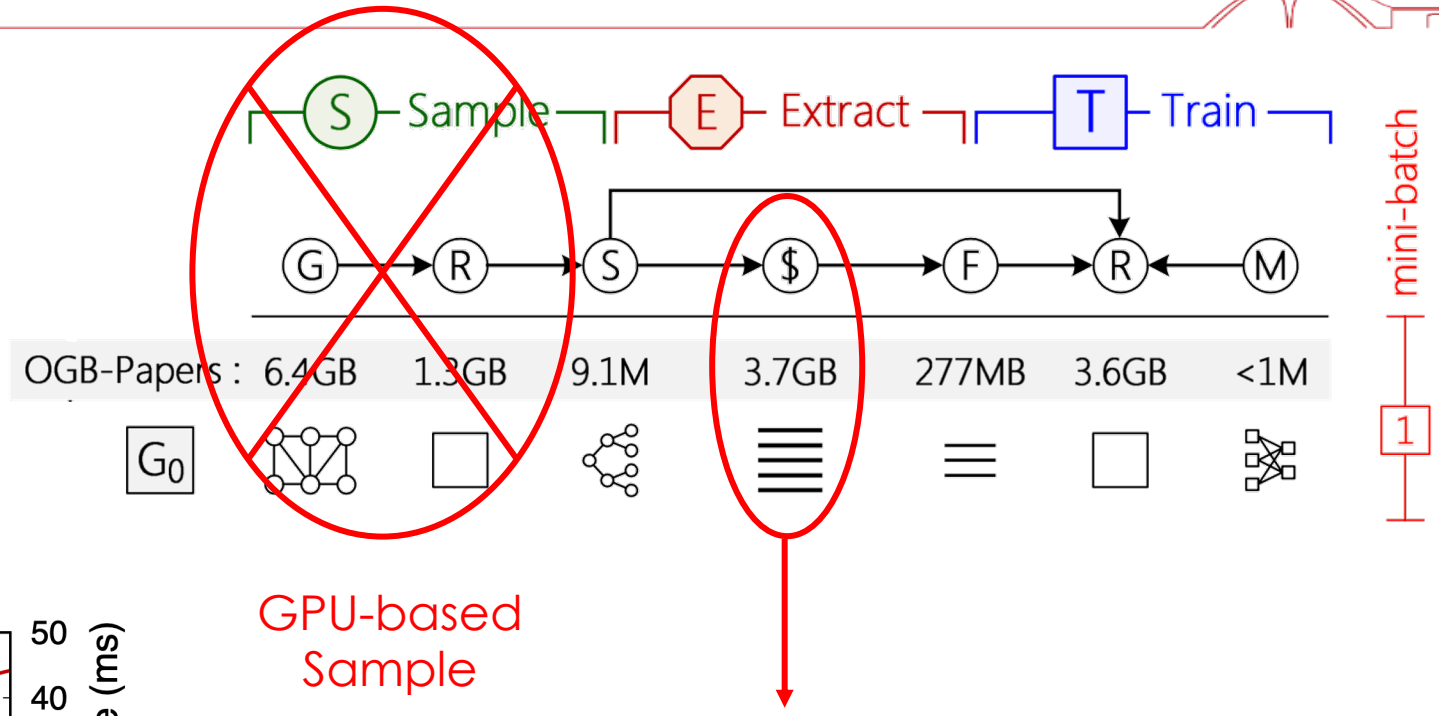


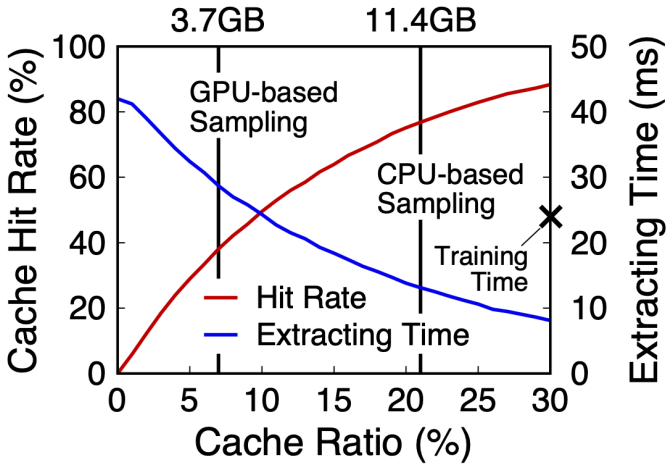
Ⓖ Graph topo Ⓢ Samples Ⓢ Cache Ⓕ Features Ⓜ Model Ⓡ Runtime state

Analysis

Problems

- Capacity





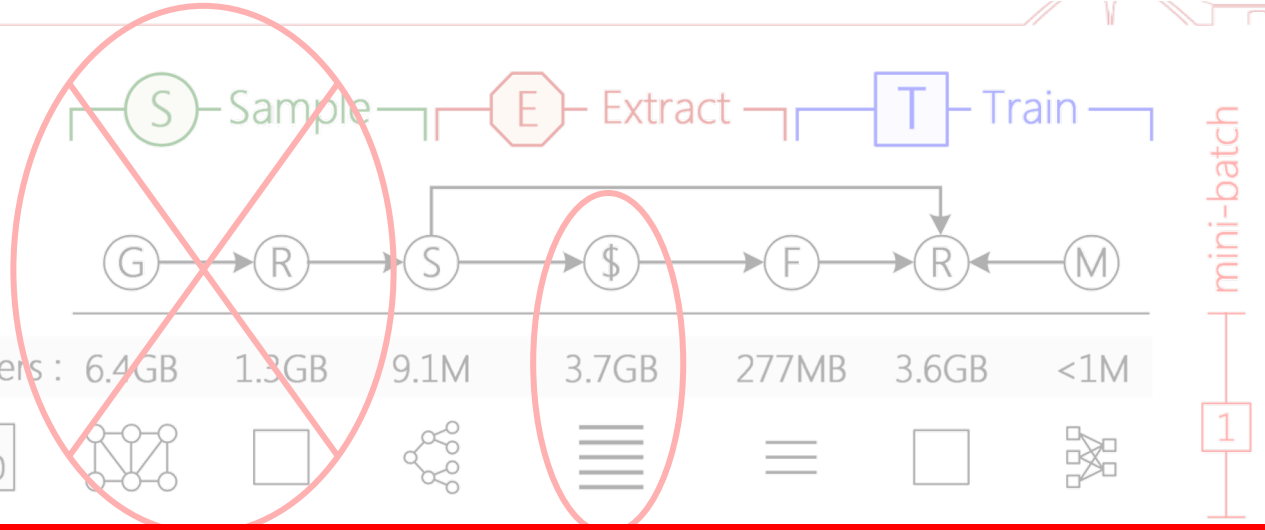
OGB-Papers100M

Ⓖ Graph topo Ⓢ Samples $ Cache Ⓕ Features Ⓜ Model Ⓡ Runtime state

Analysis

Problems

- Capacity



1. How to eliminate contention on GPU memory between different stages of the SET model

OGB-Papers100M

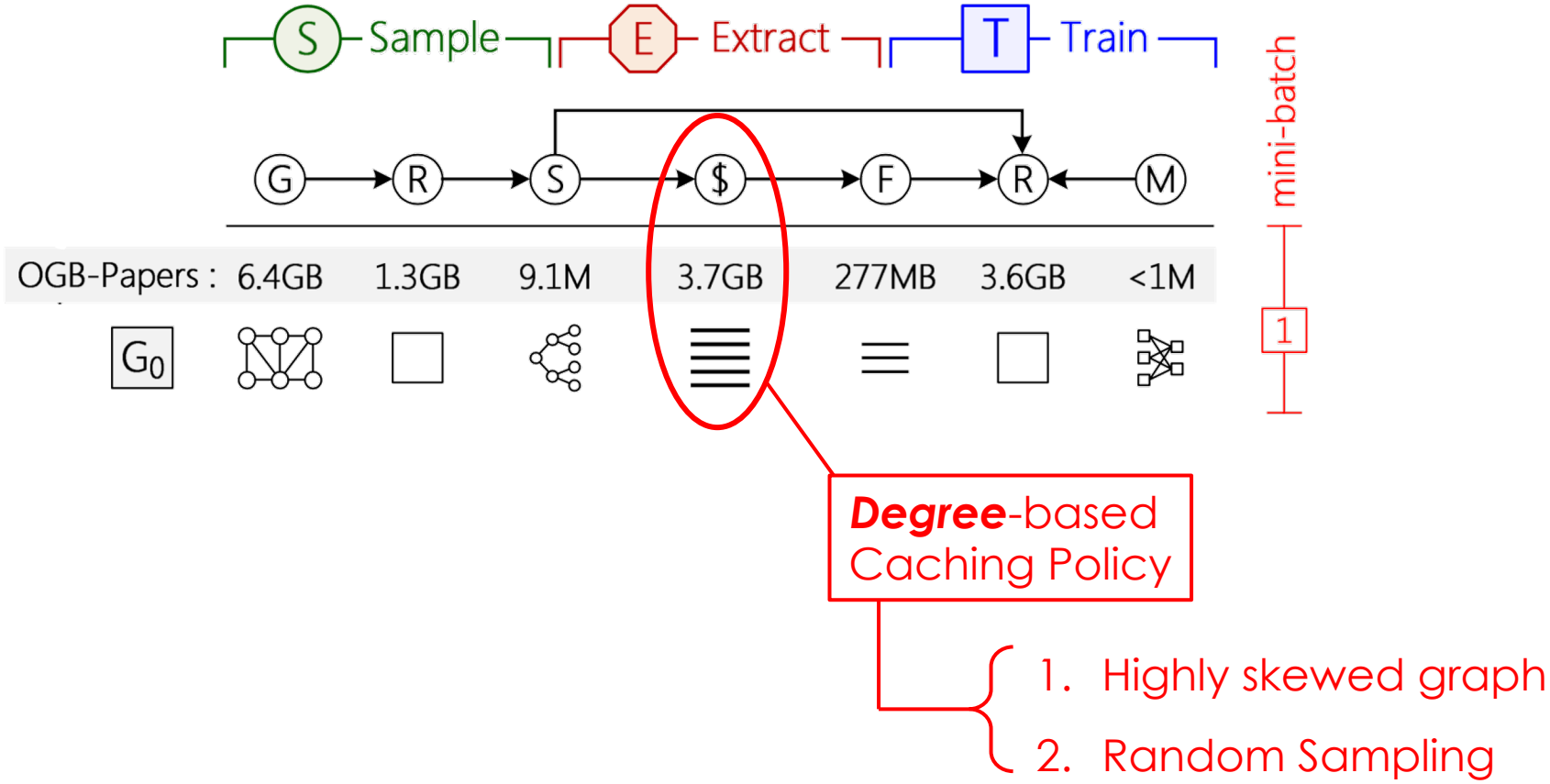
11.4 GB

G Graph topo S Samples $ Cache F Features M Model R Runtime state

Analysis

Problems

- Capacity
- Efficiency

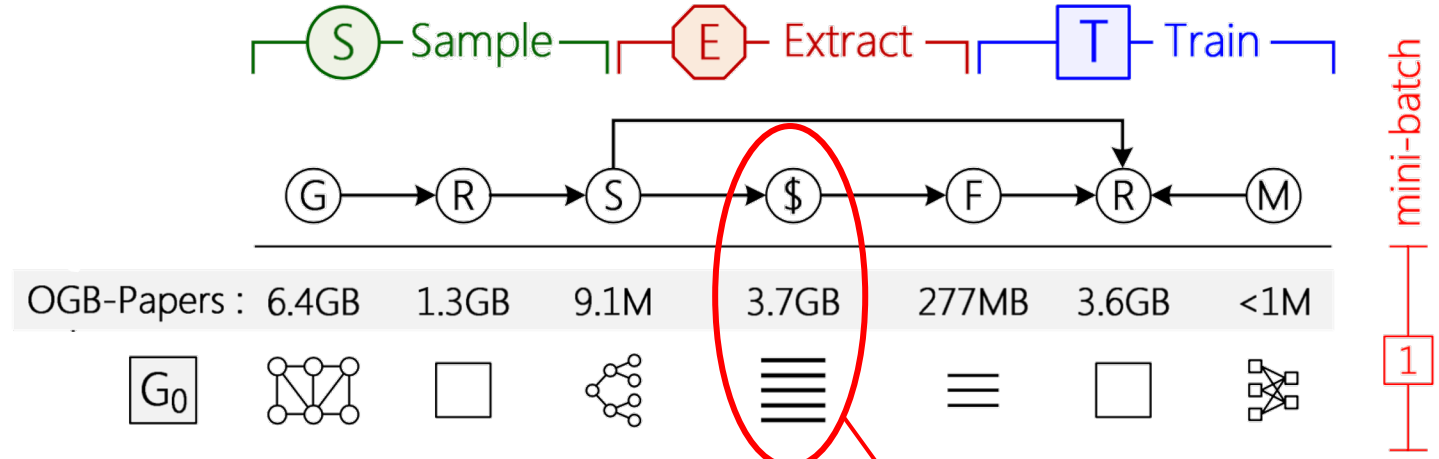


(G) Graph topo (S) Samples ($) Cache (F) Features (M) Model (R) Runtime state

Analysis

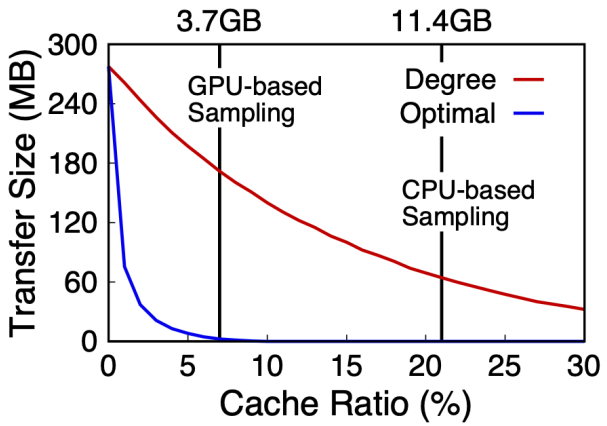
Problems

- Capacity
- Efficiency

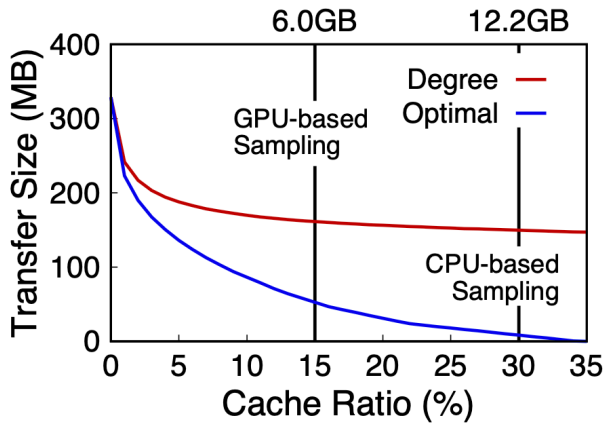


Degree-based Caching Policy

1. Highly skewed graph
2. Random Sampling



OGB-Papers100M



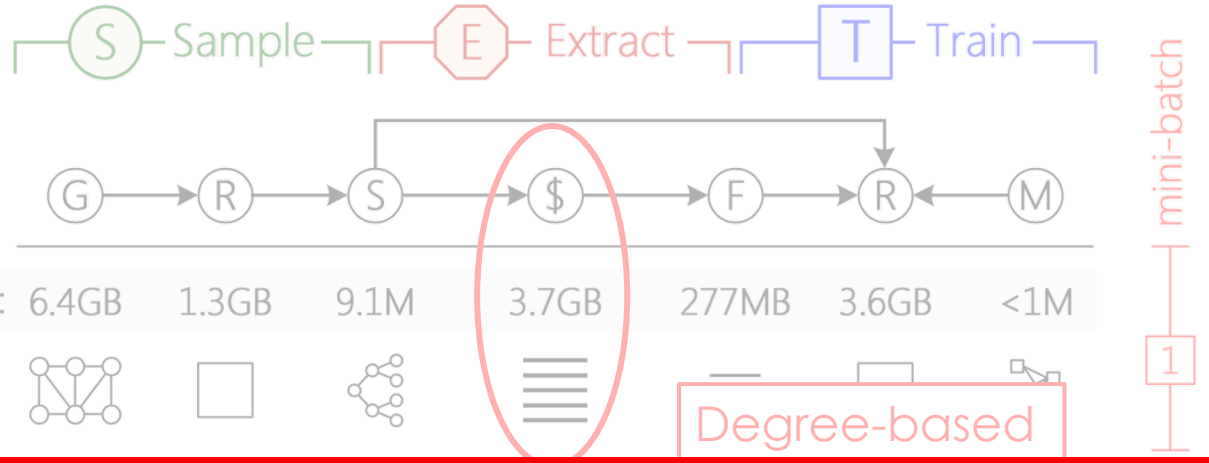
Weighted sampling

Ⓖ Graph topo Ⓢ Samples $ Cache Ⓕ Features Ⓜ Model Ⓡ Runtime state

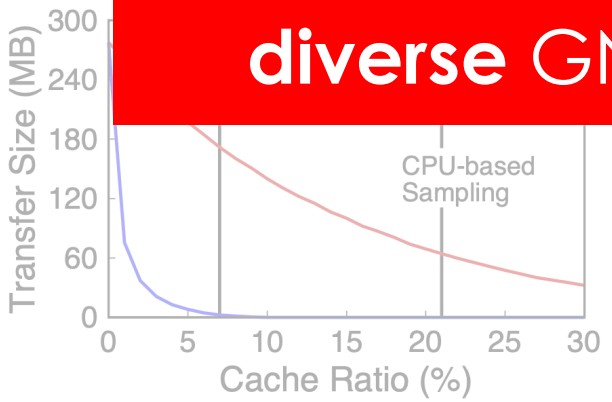
Analysis

Problems

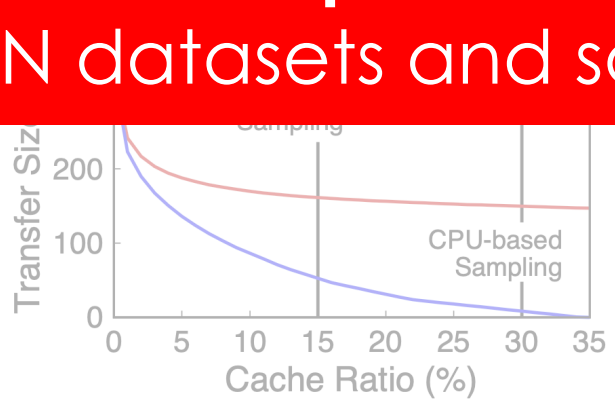
- Capacity
- Efficiency



2. How to achieve optimal cache efficiency for diverse GNN datasets and sampling algorithms



OGB-Papers100M



Weighted sampling

graph

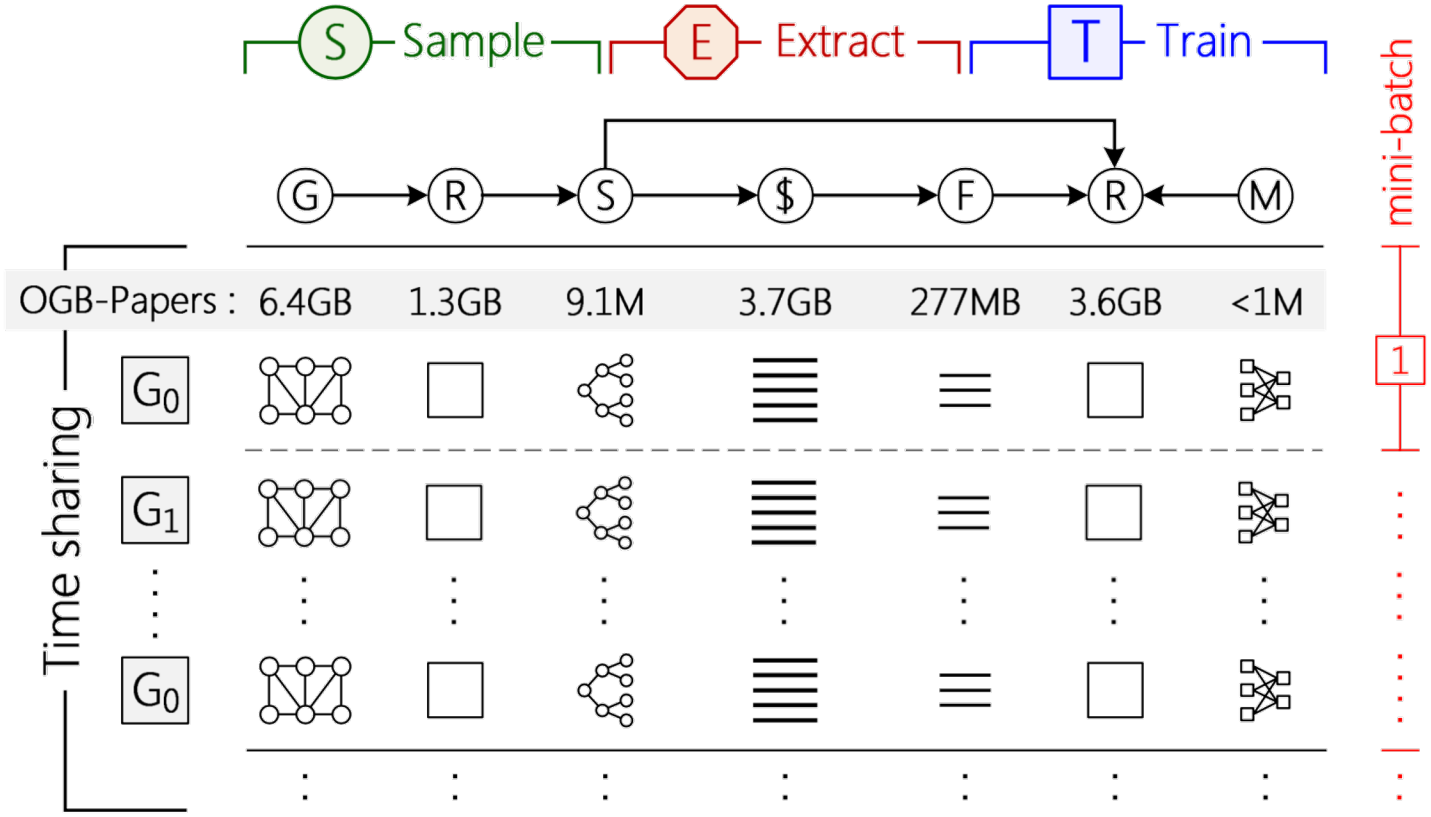
2. Random Sampling

G Graph topo S Samples $ Cache F Features M Model R Runtime state

General Idea

Time sharing

w/ multi-GPUs

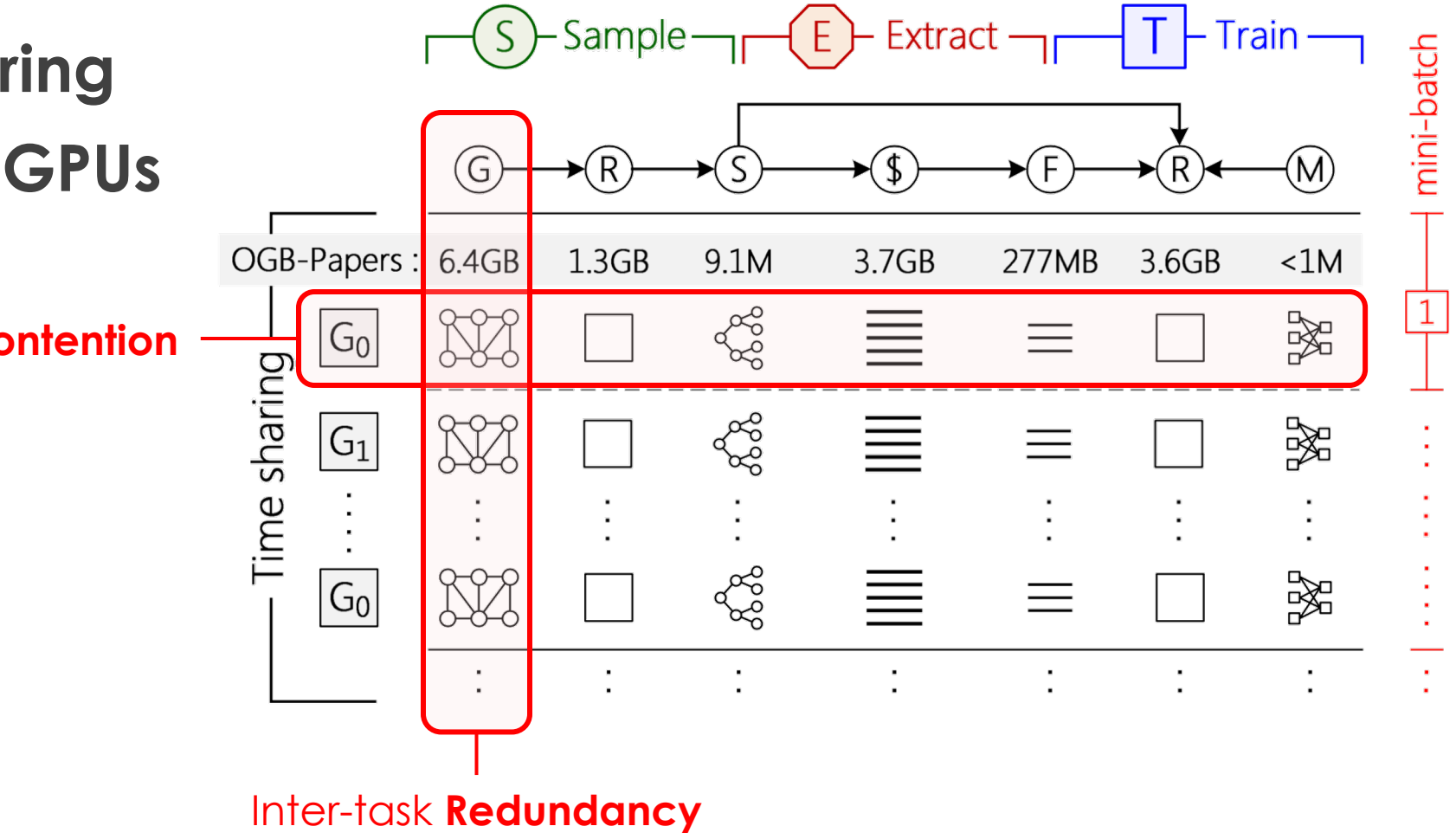


G Graph topo S Samples $ Cache F Features M Model R Runtime state

General Idea

Time sharing

w/ multi-GPUs



G Graph topo S Samples $ Cache F Features M Model R Runtime state

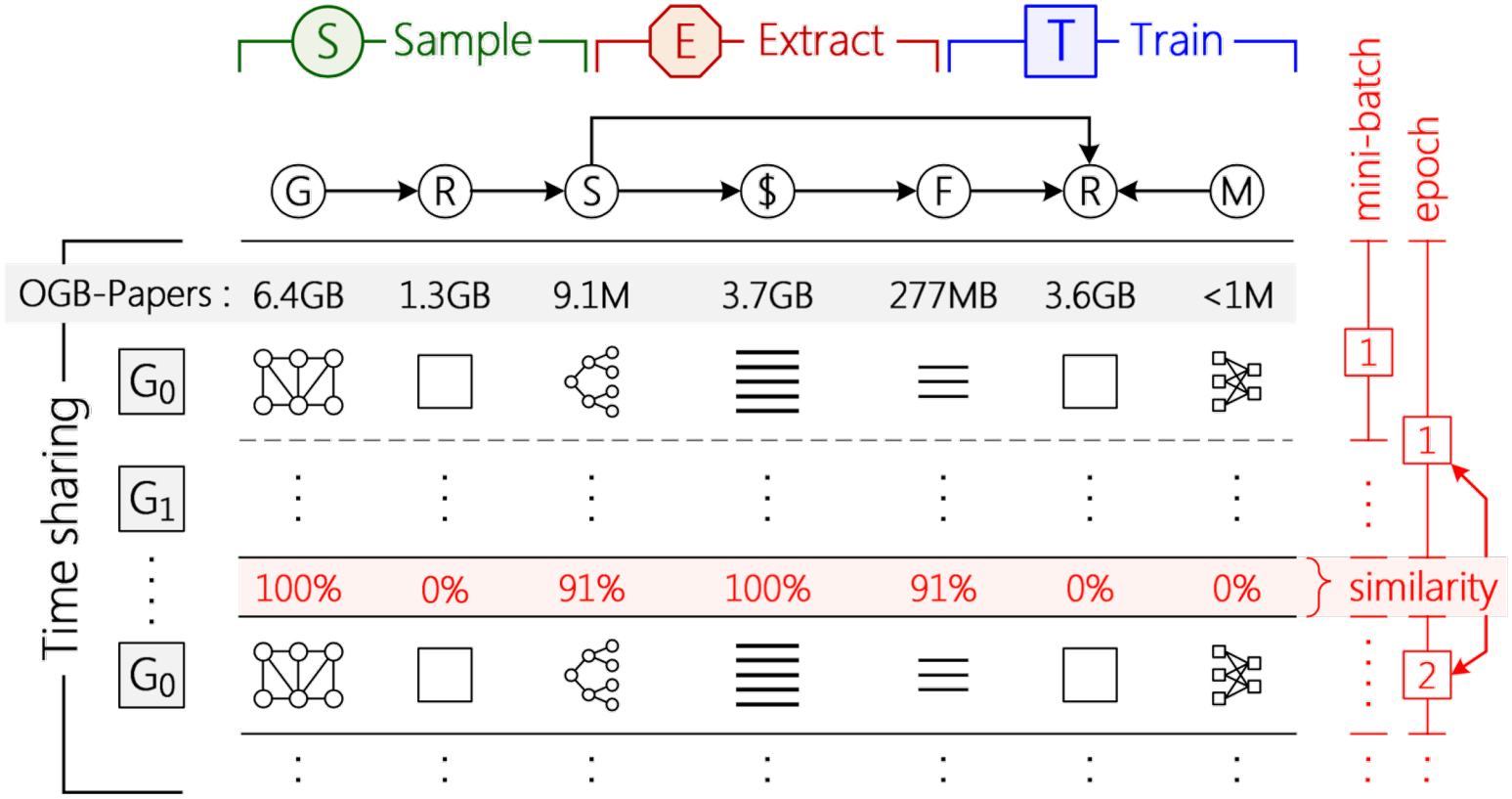
General Idea

Observation:

HIGH

cross-GPU

similarity



G Graph topo S Samples $ Cache F Features M Model R Runtime state

General Idea

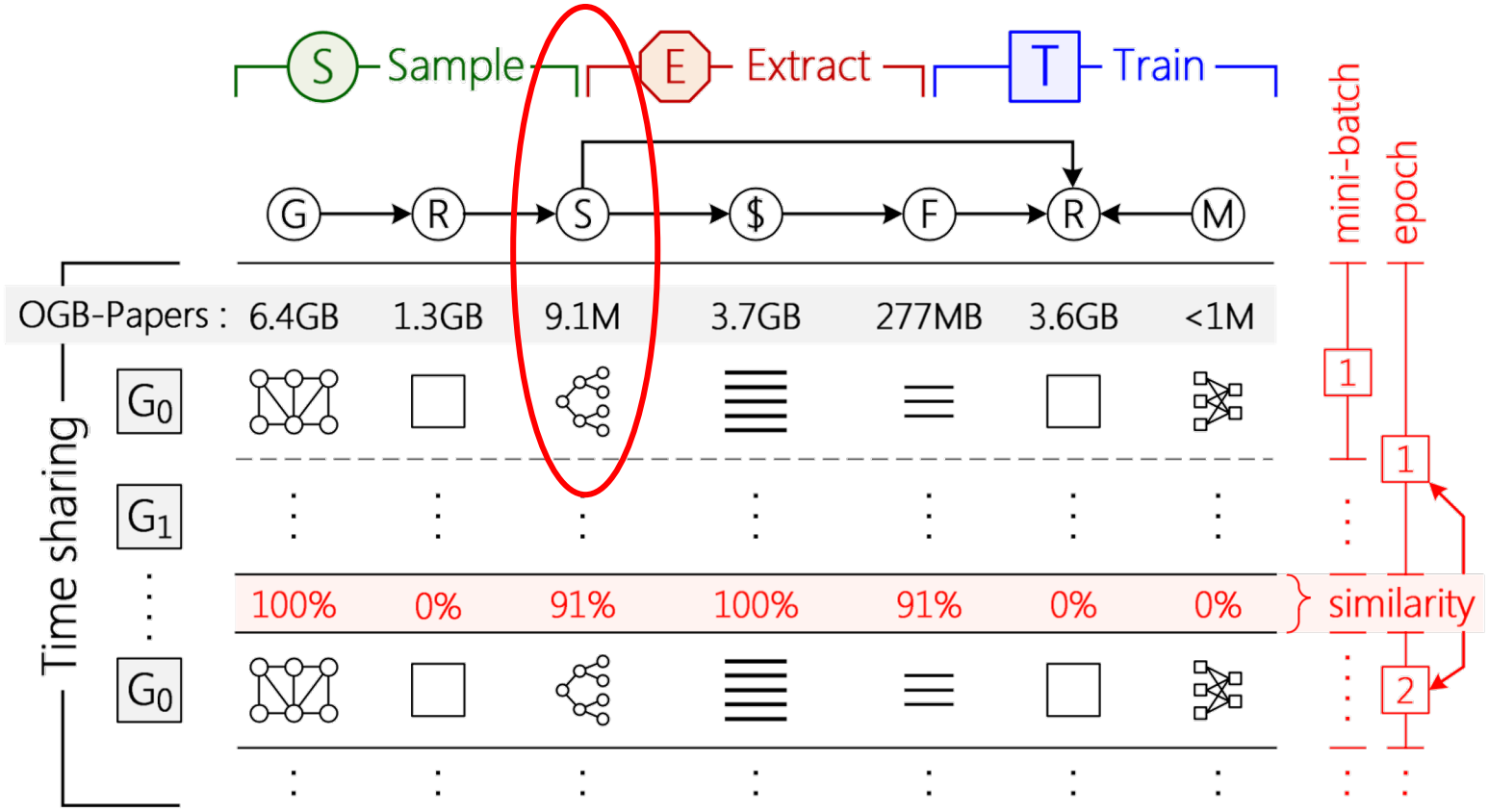
Observation:

HIGH

cross-GPU similarity

LOW

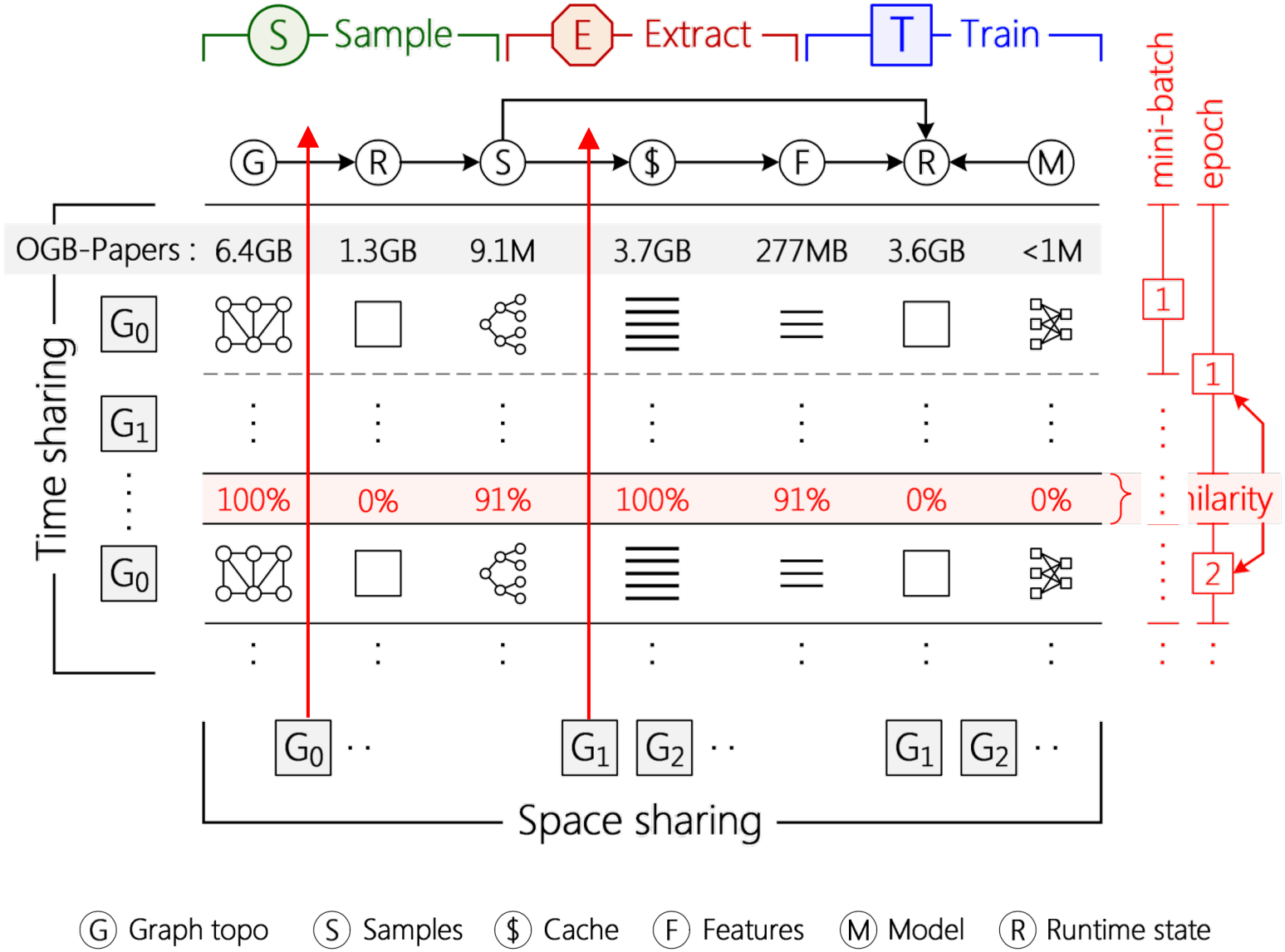
cross-stage data sharing



Ⓖ Graph topo Ⓢ Samples $ Cache Ⓕ Features Ⓜ Model Ⓡ Runtime state

General Idea

Space sharing w/ multi-GPUs



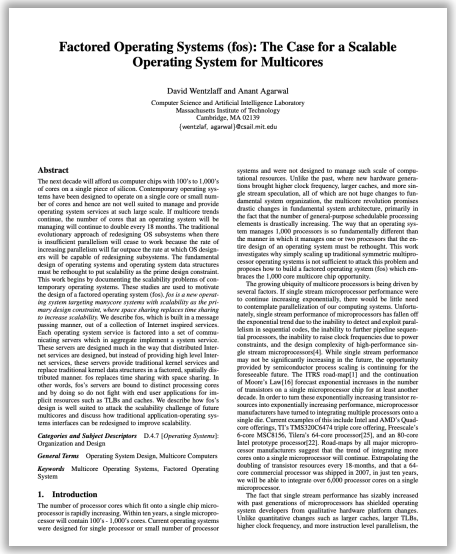
Our Approach

A factored design

- Inspired by the ***factored operating system*** (*fos*[ACM OSR'09])

Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores

David Wentzlaff and Anant Agarwal @MIT



Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores

David Wentzlaff, Anant Agarwal
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
{wentzlaf, agarwal}@csail.mit.edu

Abstract

The next decade will afford us computer chips with 100's to 1,000's of cores on a single piece of silicon. Contemporary operating systems have been designed to operate on a single core or small number of cores and hence are not well suited to manage and provide operating system services at such large scale. If multicore trends continue, the number of cores that an operating system will be managing will continue to double every 18 months. The traditional evolutionary approach of redesigning OS subsystems when there is insufficient parallelism will cease to work because the rate of increasing parallelism will far outpace the rate at which OS designers will be capable of redesigning subsystems. The fundamental design of operating systems and operating system data structures must be rethought to put scalability as the prime design constraint. This work begins by documenting the scalability problems of contemporary operating systems. These studies are used to motivate the design of a factored operating system (fos). *fos is a new operating system targeting manycore systems with scalability as the primary design constraint, where space sharing replaces time sharing to increase scalability.* We describe fos, which is built in a message passing manner, out of a collection of Internet inspired services. Each operating system service is factored into a set of communicating servers which in aggregate implement a system service. These servers are designed much in the way that distributed Internet services are designed, but instead of providing high level Internet services, these servers provide traditional kernel services and replace traditional kernel data structures in a factored, spatially distributed manner. fos replaces time sharing with space sharing. In other words, fos's servers are bound to distinct processing cores and by doing so do not fight with end user applications for implicit resources such as TLBs and caches. We describe how fos's design is well suited to attack the scalability challenge of future multicores and discuss how traditional application-operating system interfaces can be redesigned to improve scalability.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design

General Terms Operating System Design, Multicore Computers

Keywords Multicore Operating Systems, Factored Operating System

1. Introduction

The number of processor cores which fit onto a single chip microprocessor is rapidly increasing. Within ten years, a single microprocessor will contain 100's - 1,000's cores. Current operating systems were designed for single processor or small number of processor systems and were not designed to manage such scale of computational resources. Unlike the past, where new hardware generations brought higher clock frequency, larger caches, and more single stream speculation, all of which are not huge changes to fundamental system organization, the multicore revolution promises drastic changes in fundamental system architecture, primarily in the fact that the number of general-purpose schedulable processing elements is drastically increasing. The way that an operating system manages 1,000 processors is so fundamentally different than the manner in which it manages one or two processors that the entire design of an operating system must be rethought. This work investigates why simply scaling up traditional symmetric multiprocessor operating systems is not sufficient to attack this problem and proposes how to build a factored operating system (fos) which embraces the 1,000 core multicore chip opportunity.

The growing ubiquity of multicore processors is being driven by several factors. If single stream microprocessor performance were to continue increasing exponentially, there would be little need to contemplate parallelization of our computing systems. Unfortunately, single stream performance of microprocessors has fallen off the exponential trend due to the inability to detect and exploit parallelism in sequential codes, the inability to further pipeline sequential processors, the inability to raise clock frequencies due to power constraints, and the design complexity of high-performance single stream microprocessors[4]. While single stream performance may not be significantly increasing in the future, the opportunity provided by semiconductor process scaling is continuing for the foreseeable future. The ITRS road-map[1] and the continuation of Moore's Law[16] forecast exponential increases in the number of transistors on a single microprocessor chip for at least another decade. In order to turn these exponentially increasing transistor resources into exponentially increasing performance, microprocessor manufacturers have turned to integrating multiple processors onto a single die. Current examples of this include Intel and AMD's Quad-core offerings, TI's TMS320C6474 triple core offering, Freescale's 6-core MSC8156, Tilera's 64-core processor[25], and an 80-core Intel prototype processor[22]. Road-maps by all major microprocessor manufacturers suggest that the trend of integrating more cores onto a single microprocessor will continue. Extrapolating the doubling of transistor resources every 18-months, and that a 64-core commercial processor was shipped in 2007, in just ten years, we will be able to integrate over 6,000 processor cores on a single microprocessor.

The fact that single stream performance has sizably increased with past generations of microprocessors has shielded operating system developers from qualitative hardware platform changes. Unlike quantitative changes such as larger caches, larger TLBs, higher clock frequency, and more instruction level parallelism, the

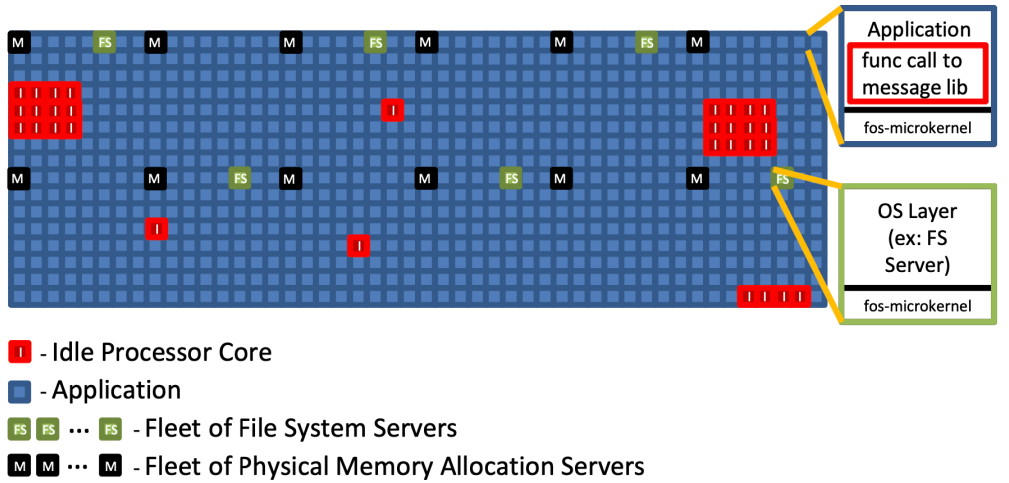


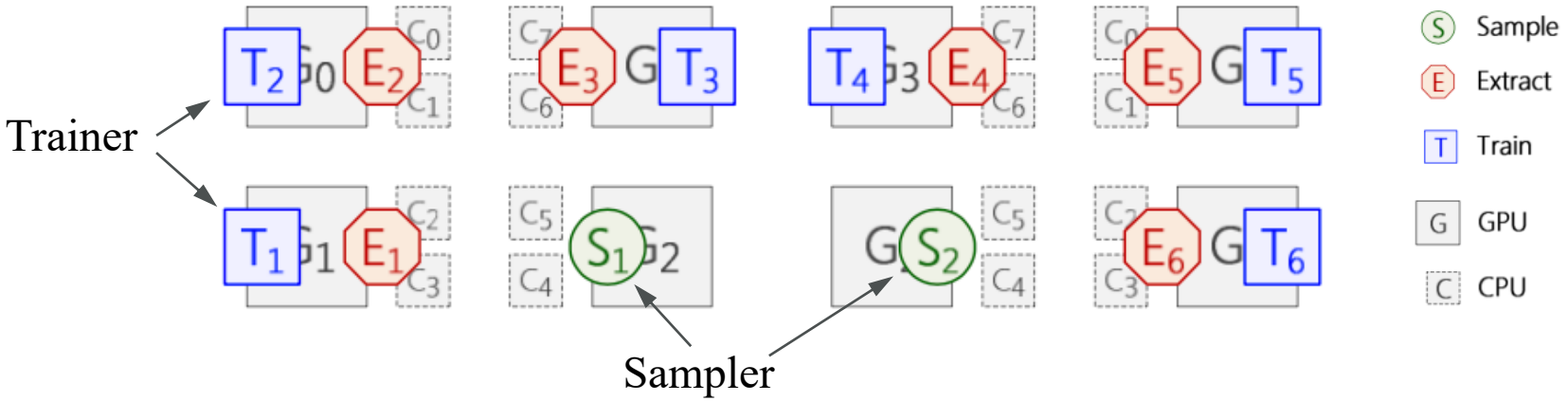
Figure 4. OS and application clients executing on the fos-microkernel

Our Approach

A factored design

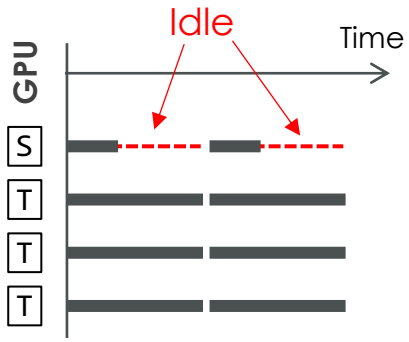
- Inspired by the ***factored operating system*** (*fos*[ACM OSR'09])
- **GNNLab**: a factored system for sample-based GNN training
  - Perform each stage on ***dedicate processors*** (GPUs and/or CPUs)

An example of GNNLab on an 8-GPU machine (2 Samplers & 6 Trainers)



Our Approach

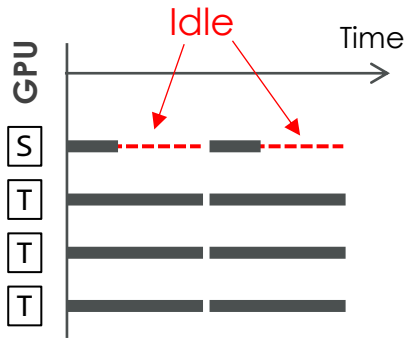
Fundamental Challenge: load imbalance



- ***Coarse-grained*** (stage-level) workload partitioning
- ***Limited GPUs***: normally <= 8, even just 1
- ***Diverse*** datasets and workloads
  - Sampling vs. Training: e.g., GCN = 1 : 1, while PinSAGE = 1 : 10

Our Approach

Fundamental Challenge: load imbalance



- ***Coarse-grained*** (stage-level) workload partitioning
- ***Limited GPUs***: normally <= 8, even just 1
- ***Diverse*** datasets and workloads
  - Sampling vs. Training: e.g., GCN = 1 : 1, while PinSAGE = 1 : 10

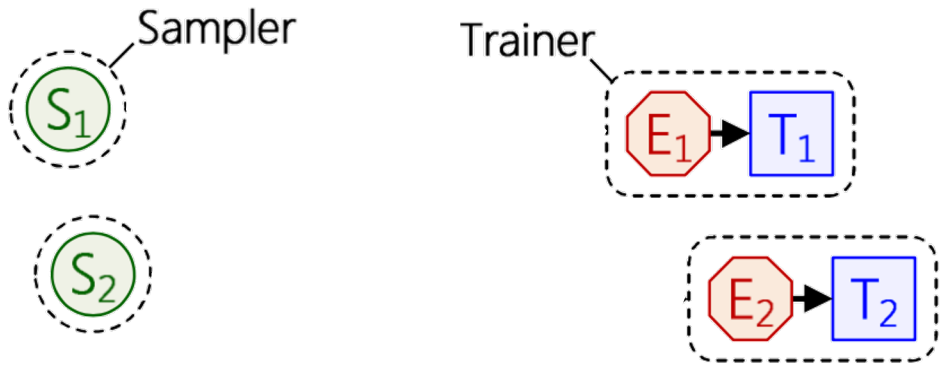
GOALs of GNNLab

1. Make stages work together **efficiently**
2. Assign GPUs to different stages **flexibly**

Execution Engine

Architecture

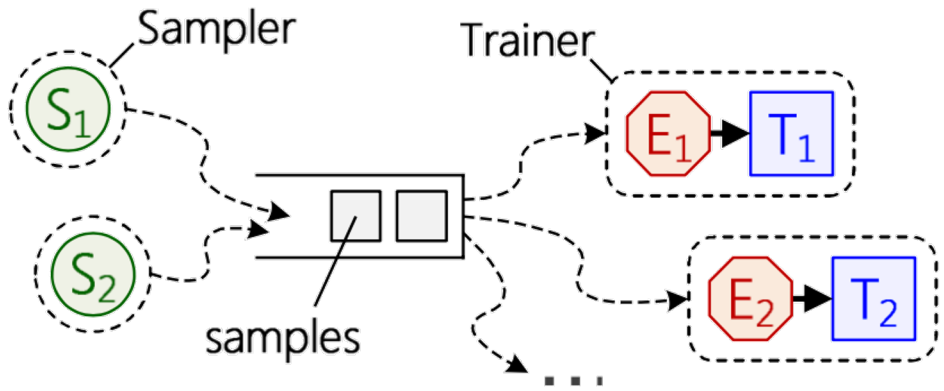
- Two executors **@GPUs**
  - **Sampler**: Sample stage
  - **Trainer**: Extract & Train stage



Execution Engine

Architecture

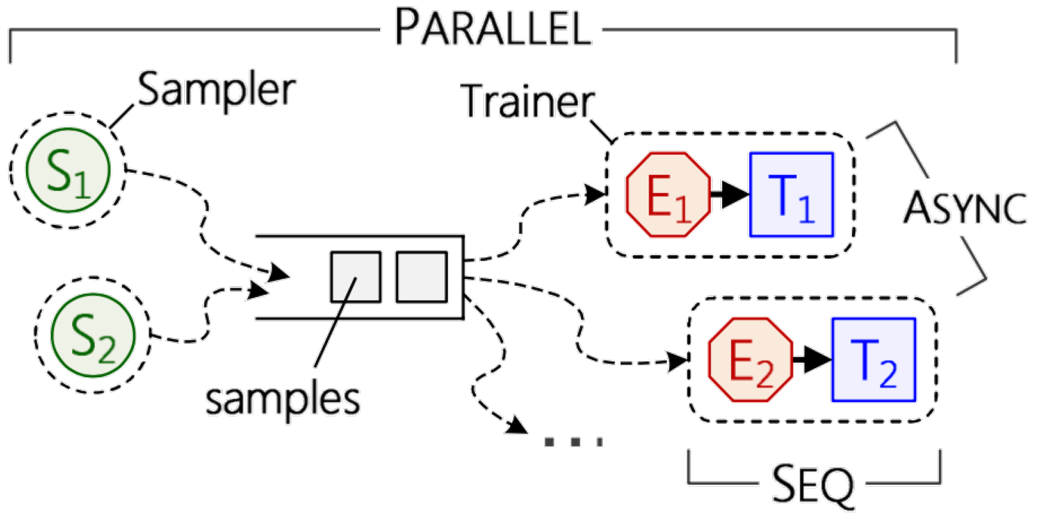
- Two executors **@GPUs**
  - **Sampler**: Sample stage
  - **Trainer**: Extract & Train stage
- A ***global*** queue **@CPUs**



Execution Engine

Architecture

- Two executors **@GPUs**
  - **Sampler**: Sample stage
  - **Trainer**: Extract & Train stage
- A ***global*** queue **@CPUs**
- Execution flow
  - Inter-executor: **Parallel**
  - Intra-executor: **Sequential** w/ ***pipelining***
  - Gradient updates w/ bounded staleness



Flexible Scheduling

GPU allocation scheme

- OB: performance of executors on GPU is ***predictable***
- N_g: the number of GPUs
- N_s (resp. N_t): #GPUs allocated to Samplers (resp. Trainers)
- T_s (resp. T_t): the processing time of Sampler (resp. Trainer)

$$N_s = \left\lceil \frac{N_g}{K+1} \right\rceil$$

$$K = \frac{T_t}{T_s}$$

Flexible Scheduling

GPU allocation scheme

- OB: performance of executors on GPU is ***predictable***
- N_g: the number of GPUs
- N_s (resp. N_t): #GPUs allocated to Samplers (resp. Trainers)
- T_s (resp. T_t): the processing time of Sampler (resp. Trainer)
- ***Prefer to*** allocate GPUs to Samplers
  - ❖ *temporarily switching from Sampler to Trainer is efficient*
- Dynamic executor ***switching*** [see our paper]

Prefer to Sampler

$$N_s = \left\lceil \frac{N_g}{K+1} \right\rceil$$

$$K = \frac{T_t}{T_s}$$

GPU-based Feature Caching

A general caching scheme

- Hotness metric h_v and cache ratio α

1. Store and sort vertices according to their h_v
2. Load features of *top-ranked* $\alpha|V|$ vertices w.r.t. h_v into GPU cache

For example: PaGraph[SOCC'20]

h_v = *out-degree of each vertex*

GPU-based Feature Caching

Pre-sampling based caching policy (*PreSC*)

- OB: Cross-task (epoch) ***similarity*** of access footprint

Sampling alogrithms	**PR**	**TW**	**PA**	**UK**
3-hop random	73.97	78.89	91.29	77.46
Random walks	78.16	72.68	87.14	64.40
3-hop weighted	77.69	66.64	89.57	72.96

GPU-based Feature Caching

Pre-sampling based caching policy (*PreSC*)

- OB: Cross-task (epoch) ***similarity*** of access footprint

Sampling alogrithms	**PR**	**TW**	**PA**	**UK**
3-hop random	73.97	78.89	91.29	77.46
Random walks	78.16	72.68	87.14	64.40
3-hop weighted	77.69	66.64	89.57	72.96

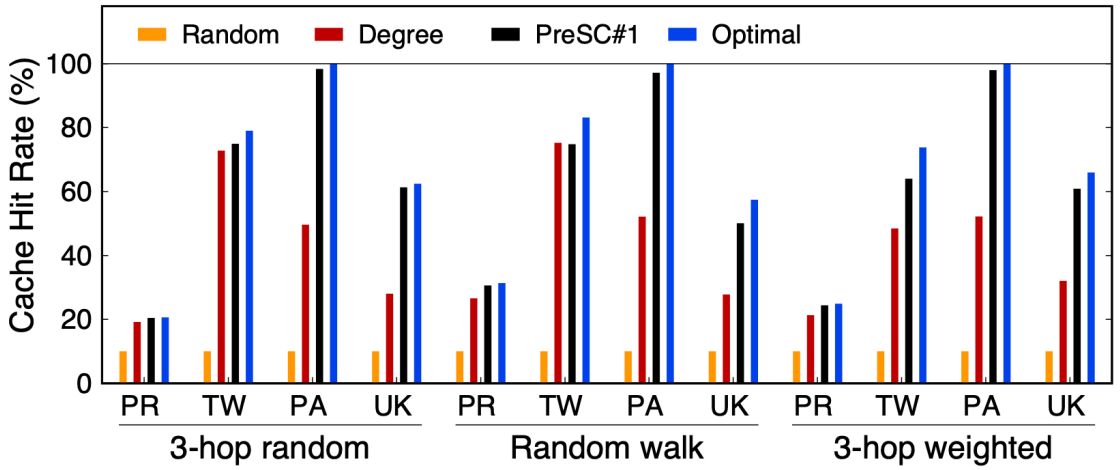
- General idea: ***pre-sample*** a few rounds to estimate vertex ***hotness***
  1. Conducts **K** sampling stages (normally 1) for the training set
  2. Record ***visit count*** of the sampled vertices
  3. Use average count as the hotness metric h_v

GPU-based Feature Caching

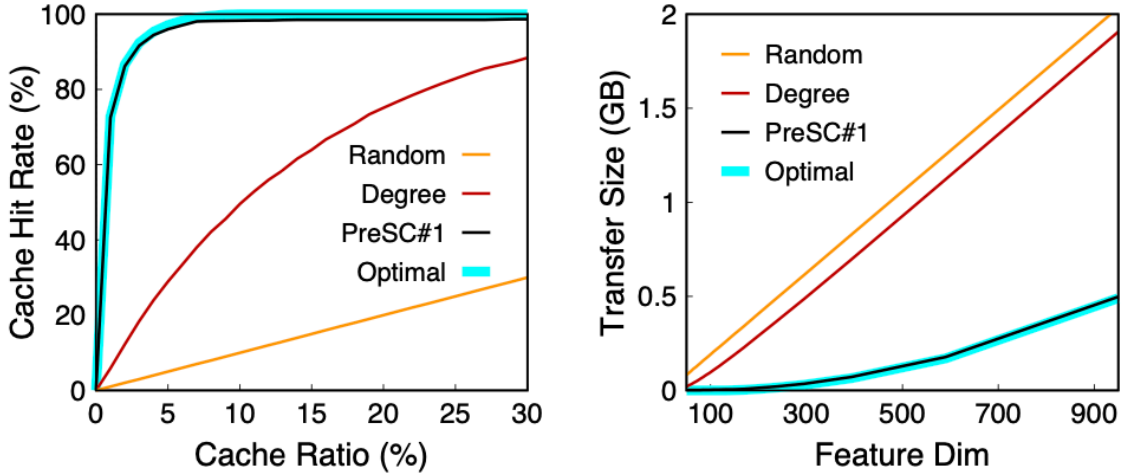
Pre-sampling based caching policy (*PreSC*)

- ***Efficiency***: close to Optimal, vs. Degree avg 1.5× (up to 2.2×)
- ***Robustness***: stable for all 12 cases

Cache Ratio=10%



Paper100M with 3-hop random



Evaluation

Testbed

- 8 NVDIA V100 GPU w/ 16GB memory
- Intel Xeon 2×24 CPU

Datasets

Dataset	#Vertex	#Edge	Dim.	#TS	Vol_G	Vol_F
PR [5]	2.4M	124M	100	197K	481MB	934MB
TW [33]	41.7M	1.5B	256	417K	5.6GB	40GB
PA [4]	111M	1.6B	128	1.2M	6.4GB	53GB
UK [9]	77.7M	3.0B	256	1.0M	11.3GB	74GB

PR can be loaded into a single GPU

GNNs

- GCN (3-hop rand ngb)
- GraphSAGE (2-hop rand ngb)
- PinSAGE (rand walks)

Baselines

System	Design	Sample	Extract	Train
PyG	N/A	CPU	No cache	GPU
DGL	Time S.	GPU	No cache	GPU
T_{SOTA}	Time S.	GPU w/ Opt.	Cache w/ Degree	GPU
GNNLab	Space S.	GPU w/ Opt.	Cache w/ PreSC	GPU

Overall Performance

- **MERITS**:

  (**A1**) space sharing design

  (**A2**) pre-sampling policy

  (**A3**) efficient sampling impl.

- vs. PyG: 10.2×~74.3×
- vs. DGL: 2.4×~. 9.1×
  due to (**A1**)~(**A3**)
- vs. T_{SOTA}: 1.6×~3.8×, e.f. PR
  due to (**A1**) and (**A2**)

Our flexible scheduling scheme already provides ***optimal GPU allocations*** in an 8-GPU machine

GNN Model	Dataset	PyG	DGL	T_{SOTA}	GNNLab
GCN	PR	11.91	1.33	0.22	0.33 (2S)
	TW	12.15	3.86	1.80	0.47 (2S)
	PA	14.82	4.56	2.46	0.84 (2S)
	UK	15.04	OOM	OOM	1.47 (2S)
GraphSAGE	PR	8.17	0.79	0.07	0.11 (4S)
	TW	8.18	1.81	0.35	0.20 (2S)
	PA	9.68	2.47	0.85	0.30 (2S)
	UK	9.86	OOM	2.01	0.61 (1S)
PinSAGE	PR	×	0.94	0.30	0.40 (1S)
	TW	×	2.50	0.98	0.58 (1S)
	PA	×	2.97	1.65	1.05 (1S)
	UK	×	OOM	OOM	1.81 (1S)

Performance Breakdown

S, **E**, and **T** represent Sample, Extract, and Train stages. **G**, **M**, and **C** represent graph sampling, marking cached vertices, and copying samples to host memory in Sample stage, respectively. **R%** and **H%** represent the cache ratio of features and the cache hit rate.

GNN	Dataset	DGL			T_{SOTA}			GNNLab		
		S	E	T	S = G + M	E (R%, H%)	T	S = G + M + C	E (R%, H%)	T
GCN	PR	0.35	2.81	1.22	0.30 = 0.29 + 0.01	0.04 (100, 100)	1.18	0.39 = 0.29 + 0.01 + 0.09	0.15 (100, 100)	1.18
	TW	0.74	9.44	1.48	0.29 = 0.26 + 0.03	3.68 (1, 29)	1.53	0.37 = 0.26 + 0.03 + 0.08	0.76 (25, 89)	1.51
	PA	1.20	10.70	4.00	0.79 = 0.70 + 0.10	3.64 (7, 38)	4.00	0.96 = 0.68 + 0.10 + 0.18	0.49 (21, 99)	3.82
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.56 = 0.39 + 0.03 + 0.14	3.06 (14, 70)	3.09
GSG	PR	0.13	1.92	0.23	0.16 = 0.15 + 0.01	0.03 (100, 100)	0.25	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	0.24
	TW	0.38	4.65	0.44	0.12 = 0.11 + 0.01	0.62 (15, 77)	0.44	0.16 = 0.11 + 0.01 + 0.03	0.41 (32, 89)	0.43
	PA	0.56	6.06	1.25	0.38 = 0.33 + 0.06	1.42 (11, 56)	1.18	0.46 = 0.31 + 0.06 + 0.08	0.28 (25, 99)	1.15
	UK	OOM	OOM	OOM	0.19 = 0.19 + 0.00	4.49 (0, 0)	1.08	0.26 = 0.18 + 0.02 + 0.06	1.39 (18, 72)	1.01
PSG	PR	0.40	1.64	1.75	0.16 = 0.16 + 0.01	0.03 (100, 100)	1.74	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	1.72
	TW	0.72	5.22	2.59	0.23 = 0.22 + 0.02	1.12 (4, 60)	2.60	0.28 = 0.21 + 0.02 + 0.05	0.51 (26, 86)	2.52
	PA	1.86	4.85	5.78	0.54 = 0.49 + 0.05	1.68 (6, 37)	6.09	0.61 = 0.47 + 0.04 + 0.09	0.33 (22, 97)	6.01
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.65 = 0.49 + 0.03 + 0.13	3.37 (13, 57)	7.00

Performance Breakdown

S, **E**, and **T** represent Sample, Extract, and Train stages. **G**, **M**, and **C** represent graph sampling, marking cached vertices, and copying samples to host memory in Sample stage, respectively. **R%** and **H%** represent the cache ratio of features and the cache hit rate.

GNN	Dataset	DGL			T_{SOTA}			GNNLab		
		S	E	T	S = G + M	E (R%, H%)	T	S = G + M + C	E (R%, H%)	T
GCN	PR	0.35	2.81	1.22	0.30 = 0.29 + 0.01	0.04 (100, 100)	1.18	0.39 = 0.29 + 0.01 + 0.09	0.15 (100, 100)	1.18
	TW	0.74	9.44	1.48	0.29 = 0.26 + 0.03	3.68 (1, 29)	1.53	0.37 = 0.26 + 0.03 + 0.08	0.76 (25, 89)	1.51
	PA	1.20	10.70	4.00	0.79 = 0.70 + 0.10	3.64 (7, 38)	4.00	0.96 = 0.68 + 0.10 + 0.18	0.49 (21, 99)	3.82
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.56 = 0.39 + 0.03 + 0.14	3.06 (14, 70)	3.09
GSG	PR	0.13	1.92	0.23	0.16 = 0.15 + 0.01	0.03 (100, 100)	0.25	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	0.24
	TW	0.38	4.65	0.44	0.12 = 0.11 + 0.01	0.62 (15, 77)	0.44	0.16 = 0.11 + 0.01 + 0.03	0.41 (32, 89)	0.43
	PA	0.56	6.06	1.25	0.38 = 0.33 + 0.06	1.42 (11, 56)	1.18	0.46 = 0.31 + 0.06 + 0.08	0.28 (25, 99)	1.15
	UK	OOM	OOM	OOM	0.19 = 0.19 + 0.00	4.49 (0, 0)	1.08	0.26 = 0.18 + 0.02 + 0.06	1.39 (18, 72)	1.01
PSG	PR	0.40	1.64	1.75	0.16 = 0.16 + 0.01	0.03 (100, 100)	1.74	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	1.72
	TW	0.72	5.22	2.59	0.23 = 0.22 + 0.02	1.12 (4, 60)	2.60	0.28 = 0.21 + 0.02 + 0.05	0.51 (26, 86)	2.52
	PA	1.86	4.85	5.78	0.54 = 0.49 + 0.05	1.68 (6, 37)	6.09	0.61 = 0.47 + 0.04 + 0.09	0.33 (22, 97)	6.01
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.65 = 0.49 + 0.03 + 0.13	3.37 (13, 57)	7.00

Performance Breakdown

S, **E**, and **T** represent Sample, Extract, and Train stages. **G**, **M**, and **C** represent graph sampling, marking cached vertices, and copying samples to host memory in Sample stage, respectively. **R%** and **H%** represent the cache ratio of features and the cache hit rate.

GNN	Dataset	DGL			T_{SOTA}			GNNLab		
		S	E	T	S = G + M	E (R%, H%)	T	S = G + M + C	E (R%, H%)	T
GCN	PR	0.35	2.81	1.22	0.30 = 0.29 + 0.01	0.04 (100, 100)	1.18	0.39 = 0.29 + 0.01 + 0.09	0.15 (100, 100)	1.18
	TW	0.74	9.44	1.48	0.29 = 0.26 + 0.03	3.68 (1, 29)	1.53	0.37 = 0.26 + 0.03 + 0.08	0.76 (25, 89)	1.51
	PA	1.20	10.70	4.00	0.79 = 0.70 + 0.10	3.64 (7, 38)	4.00	0.96 = 0.68 + 0.10 + 0.18	0.49 (21, 99)	3.82
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.56 = 0.39 + 0.03 + 0.14	3.06 (14, 70)	3.09
GSG	PR	0.13	1.92	0.23	0.16 = 0.15 + 0.01	0.03 (100, 100)	0.25	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	0.24
	TW	0.38	4.65	0.44	0.12 = 0.11 + 0.01	0.62 (15, 77)	0.44	0.16 = 0.11 + 0.01 + 0.03	0.41 (32, 89)	0.43
	PA	0.56	6.06	1.25	0.38 = 0.33 + 0.06	1.42 (11, 56)	1.18	0.46 = 0.31 + 0.06 + 0.08	0.28 (25, 99)	1.15
	UK	OOM	OOM	OOM	0.19 = 0.19 + 0.00	4.49 (0, 0)	1.08	0.26 = 0.18 + 0.02 + 0.06	1.39 (18, 72)	1.01
PSG	PR	0.40	1.64	1.75	0.16 = 0.16 + 0.01	0.03 (100, 100)	1.74	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	1.72
	TW	0.72	5.22	2.59	0.23 = 0.22 + 0.02	1.12 (4, 60)	2.60	0.28 = 0.21 + 0.02 + 0.05	0.51 (26, 86)	2.52
	PA	1.86	4.85	5.78	0.54 = 0.49 + 0.05	1.68 (6, 37)	6.09	0.61 = 0.47 + 0.04 + 0.09	0.33 (22, 97)	6.01
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.65 = 0.49 + 0.03 + 0.13	3.37 (13, 57)	7.00

Performance Breakdown

S, **E**, and **T** represent Sample, Extract, and Train stages. **G**, **M**, and **C** represent graph sampling, marking cached vertices, and copying samples to host memory in Sample stage, respectively. **R%** and **H%** represent the cache ratio of features and the cache hit rate.

GNN	Dataset	DGL			T_{SOTA}			GNNLab		
		S	E	T	S = G + M	E (R%, H%)	T	S = G + M + C	E (R%, H%)	T
GCN	PR	0.35	2.81	1.22	0.30 = 0.29 + 0.01	0.04 (100, 100)	1.18	0.39 = 0.29 + 0.01 + 0.09	0.15 (100, 100)	1.18
	TW	0.74	9.44	1.48	0.29 = 0.26 + 0.03	3.68 (1, 29)	1.53	0.37 = 0.26 + 0.03 + 0.08	0.76 (25, 89)	1.51
	PA	1.20	10.70	4.00	0.79 = 0.70 + 0.10	3.64 (7, 38)	4.00	0.96 = 0.68 + 0.10 + 0.18	0.49 (21, 99)	3.82
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.56 = 0.39 + 0.03 + 0.14	3.06 (14, 70)	3.09
GSG	PR	0.13	1.92	0.23	0.16 = 0.15 + 0.01	0.03 (100, 100)	0.25	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	0.24
	TW	0.38	4.65	0.44	0.12 = 0.11 + 0.01	0.62 (15, 77)	0.44	0.16 = 0.11 + 0.01 + 0.03	0.41 (32, 89)	0.43
	PA	0.56	6.06	1.25	0.38 = 0.33 + 0.06	1.42 (11, 56)	1.18	0.46 = 0.31 + 0.06 + 0.08	0.28 (25, 99)	1.15
	UK	OOM	OOM	OOM	0.19 = 0.19 + 0.00	4.49 (0, 0)	1.08	0.26 = 0.18 + 0.02 + 0.06	1.39 (18, 72)	1.01
PSG	PR	0.40	1.64	1.75	0.16 = 0.16 + 0.01	0.03 (100, 100)	1.74	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	1.72
	TW	0.72	5.22	2.59	0.23 = 0.22 + 0.02	1.12 (4, 60)	2.60	0.28 = 0.21 + 0.02 + 0.05	0.51 (26, 86)	2.52
	PA	1.86	4.85	5.78	0.54 = 0.49 + 0.05	1.68 (6, 37)	6.09	0.61 = 0.47 + 0.04 + 0.09	0.33 (22, 97)	6.01
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.65 = 0.49 + 0.03 + 0.13	3.37 (13, 57)	7.00

Performance Breakdown

S, **E**, and **T** represent Sample, Extract, and Train stages. **G**, **M**, and **C** represent graph sampling, marking cached vertices, and copying samples to host memory in Sample stage, respectively. **R%** and **H%** represent the cache ratio of features and the cache hit rate.

GNN	Dataset	DGL			T_{SOTA}			GNNLab		
		S	E	T	S = G + M	E (R%, H%)	T	S = G + M + C	E (R%, H%)	T
GCN	PR	0.35	2.81	1.22	0.30 = 0.29 + 0.01	0.04 (100, 100)	1.18	0.39 = 0.29 + 0.01 + 0.09	0.15 (100, 100)	1.18
	TW	0.74	9.44	1.48	0.29 = 0.26 + 0.03	3.68 (1, 29)	1.53	0.37 = 0.26 + 0.03 + 0.08	0.76 (25, 89)	1.51
	PA	1.20	10.70	4.00	0.79 = 0.70 + 0.10	3.64 (7, 38)	4.00	0.96 = 0.68 + 0.10 + 0.18	0.49 (21, 99)	3.82
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.56 = 0.39 + 0.03 + 0.14	3.06 (14, 70)	3.09
GSG	PR	0.13	1.92	0.23	0.16 = 0.15 + 0.01	0.03 (100, 100)	0.25	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	0.24
	TW	0.38	4.65	0.44	0.12 = 0.11 + 0.01	0.62 (15, 77)	0.44	0.16 = 0.11 + 0.01 + 0.03	0.41 (32, 89)	0.43
	PA	0.56	6.06	1.25	0.38 = 0.33 + 0.06	1.42 (11, 56)	1.18	0.46 = 0.31 + 0.06 + 0.08	0.28 (25, 99)	1.15
	UK	OOM	OOM	OOM	0.19 = 0.19 + 0.00	4.49 (0, 0)	1.08	0.26 = 0.18 + 0.02 + 0.06	1.39 (18, 72)	1.01
PSG	PR	0.40	1.64	1.75	0.16 = 0.16 + 0.01	0.03 (100, 100)	1.74	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	1.72
	TW	0.72	5.22	2.59	0.23 = 0.22 + 0.02	1.12 (4, 60)	2.60	0.28 = 0.21 + 0.02 + 0.05	0.51 (26, 86)	2.52
	PA	1.86	4.85	5.78	0.54 = 0.49 + 0.05	1.68 (6, 37)	6.09	0.61 = 0.47 + 0.04 + 0.09	0.33 (22, 97)	6.01
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.65 = 0.49 + 0.03 + 0.13	3.37 (13, 57)	7.00

Performance Breakdown

S, **E**, and **T** represent Sample, Extract, and Train stages. **G**, **M**, and **C** represent graph sampling, marking cached vertices, and copying samples to host memory in Sample stage, respectively. **R%** and **H%** represent the cache ratio of features and the cache hit rate.

GNN	Dataset	DGL			T_{SOTA}			GNNLab		
		S	E	T	S = G + M	E (R%, H%)	T	S = G + M + C	E (R%, H%)	T
GCN	PR	0.35	2.81	1.22	0.30 = 0.29 + 0.01	0.04 (100, 100)	1.18	0.39 = 0.29 + 0.01 + 0.09	0.15 (100, 100)	1.18
	TW	0.74	9.44	1.48	0.29 = 0.26 + 0.03	3.68 (1, 29)	1.53	0.37 = 0.26 + 0.03 + 0.08	0.76 (25, 89)	1.51
	PA	1.20	10.70	4.00	0.79 = 0.70 + 0.10	3.64 (7, 38)	4.00	0.96 = 0.68 + 0.10 + 0.18	0.49 (21, 99)	3.82
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.56 = 0.39 + 0.03 + 0.14	3.06 (14, 70)	3.09
GSG	PR	0.13	1.92	0.23	0.16 = 0.15 + 0.01	0.03 (100, 100)	0.25	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	0.24
	TW	0.38	4.65	0.44	0.12 = 0.11 + 0.01	0.62 (15, 77)	0.44	0.16 = 0.11 + 0.01 + 0.03	0.41 (32, 89)	0.43
	PA	0.56	6.06	1.25	0.38 = 0.33 + 0.06	1.42 (11, 56)	1.18	0.46 = 0.31 + 0.06 + 0.08	0.28 (25, 99)	1.15
	UK	OOM	OOM	OOM	0.19 = 0.19 + 0.00	4.49 (0, 0)	1.08	0.26 = 0.18 + 0.02 + 0.06	1.39 (18, 72)	1.01
PSG	PR	0.40	1.64	1.75	0.16 = 0.16 + 0.01	0.03 (100, 100)	1.74	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	1.72
	TW	0.72	5.22	2.59	0.23 = 0.22 + 0.02	1.12 (4, 60)	2.60	0.28 = 0.21 + 0.02 + 0.05	0.51 (26, 86)	2.52
	PA	1.86	4.85	5.78	0.54 = 0.49 + 0.05	1.68 (6, 37)	6.09	0.61 = 0.47 + 0.04 + 0.09	0.33 (22, 97)	6.01
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.65 = 0.49 + 0.03 + 0.13	3.37 (13, 57)	7.00

Performance Breakdown

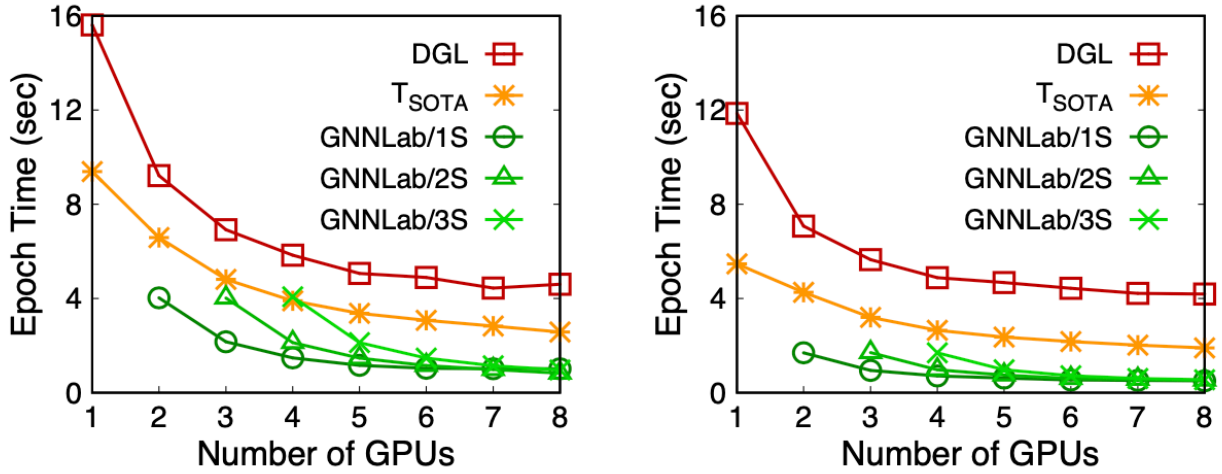
S, **E**, and **T** represent Sample, Extract, and Train stages. **G**, **M**, and **C** represent graph sampling, marking cached vertices, and copying samples to host memory in Sample stage, respectively. **R%** and **H%** represent the cache ratio of features and the cache hit rate.

GNN	Dataset	DGL			T_{SOTA}			GNNLab		
		S	E	T	S = G + M	E (R%, H%)	T	S = G + M + C	E (R%, H%)	T
GCN	PR	0.35	2.81	1.22	0.30 = 0.29 + 0.01	0.04 (100, 100)	1.18	0.39 = 0.29 + 0.01 + 0.09	0.15 (100, 100)	1.18
	TW	0.74	9.44	1.48	0.29 = 0.26 + 0.03	3.68 (1, 29)	1.53	0.37 = 0.26 + 0.03 + 0.08	0.76 (25, 89)	1.51
	PA	1.20	10.70	4.00	0.79 = 0.70 + 0.10	3.64 (7, 38)	4.00	0.96 = 0.68 + 0.10 + 0.18	0.49 (21, 99)	3.82
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.56 = 0.39 + 0.03 + 0.14	3.06 (14, 70)	3.09
GSG	PR	0.13	1.92	0.23	0.16 = 0.15 + 0.01	0.03 (100, 100)	0.25	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	0.24
	TW	0.38	4.65	0.44	0.12 = 0.11 + 0.01	0.62 (15, 77)	0.44	0.16 = 0.11 + 0.01 + 0.03	0.41 (32, 89)	0.43
	PA	0.56	6.06	1.25	0.38 = 0.33 + 0.06	1.42 (11, 56)	1.18	0.46 = 0.31 + 0.06 + 0.08	0.28 (25, 99)	1.15
	UK	OOM	OOM	OOM	0.19 = 0.19 + 0.00	4.49 (0, 0)	1.08	0.26 = 0.18 + 0.02 + 0.06	1.39 (18, 72)	1.01
PSG	PR	0.40	1.64	1.75	0.16 = 0.16 + 0.01	0.03 (100, 100)	1.74	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	1.72
	TW	0.72	5.22	2.59	0.23 = 0.22 + 0.02	1.12 (4, 60)	2.60	0.28 = 0.21 + 0.02 + 0.05	0.51 (26, 86)	2.52
	PA	1.86	4.85	5.78	0.54 = 0.49 + 0.05	1.68 (6, 37)	6.09	0.61 = 0.47 + 0.04 + 0.09	0.33 (22, 97)	6.01
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.65 = 0.49 + 0.03 + 0.13	3.37 (13, 57)	7.00

Scalability

- DGL and T_{SOTA}
  - Time sharing design
  - More work on CPUs
  - CPUs stop growing

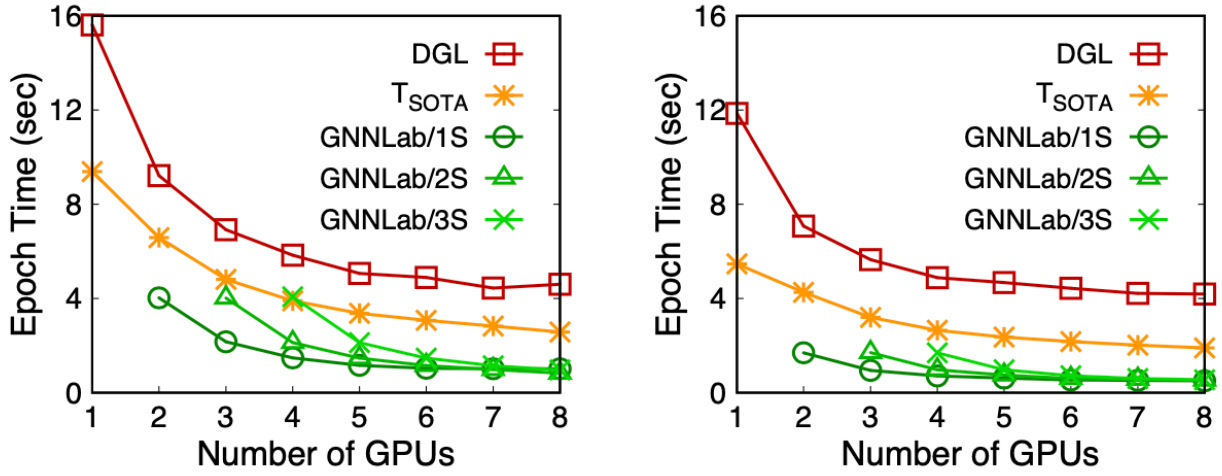
GCN on PA and TW

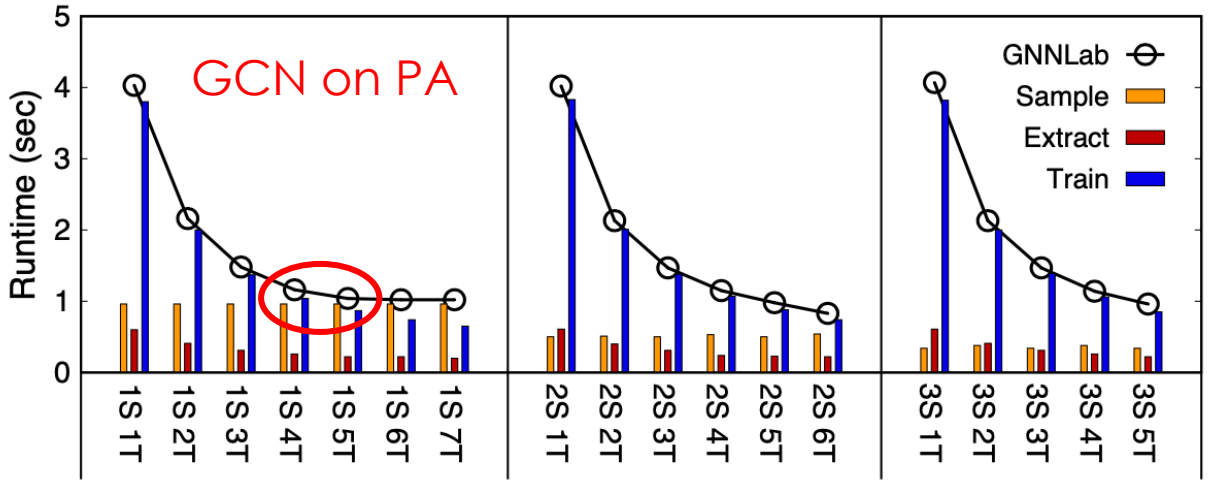


Scalability

- DGL and T_{SOTA}
  - Time sharing design
  - More work on CPUs
  - CPUs stop growing
- GNNLab
  - Good parallelism
  - Bottleneck may change
  - 1 Sampler is **not enough**
  - 3 Sampler is **too much**

GCN on PA and TW

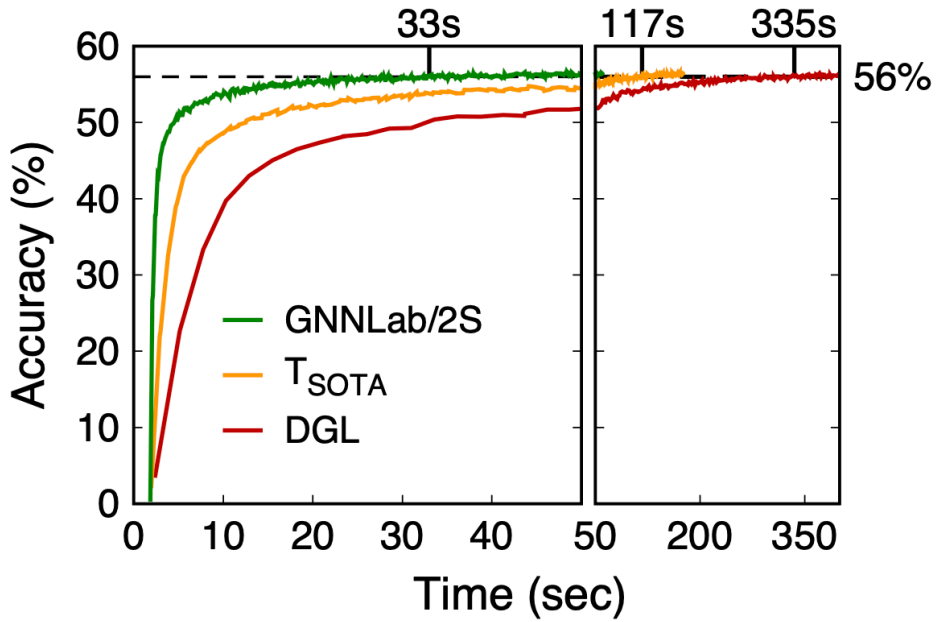




Training Convergence

Case: GraphSAGE on Papers100M

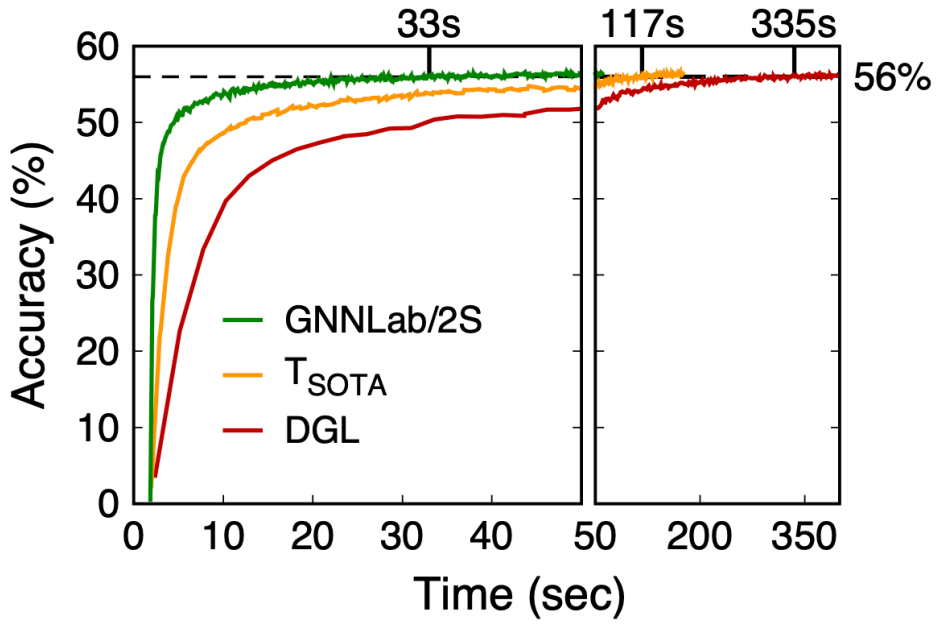
- Converge to same accuracy targets
- GNNLab outperforms
  DGL by 10.2× and T_{SOTA} by 3.5×



Training Convergence

Case: GraphSAGE on Papers100M

- Converge to same accuracy targets
- GNNLab outperforms
  DGL by 10.2× and T_{SOTA} by 3.5×
  1. Faster training (per epoch)
     - vs. DGL by 8.2× and T_{SOTA} by 2.8×



Training Convergence

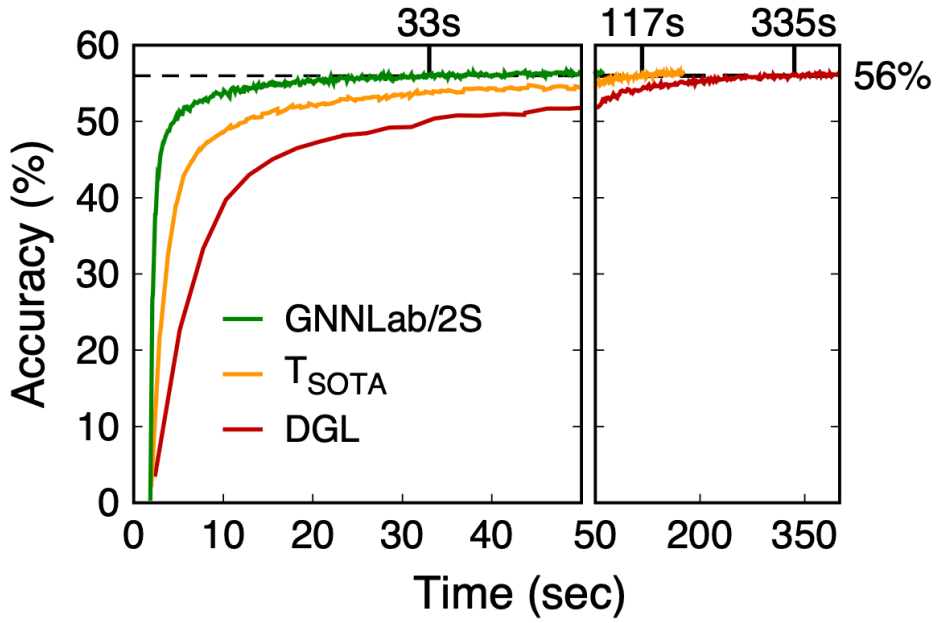
Case: GraphSAGE on Papers100M

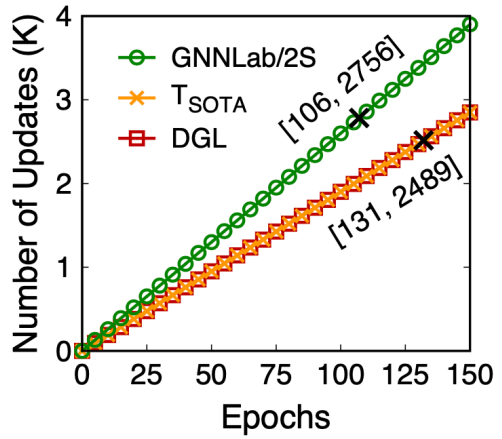
- Converge to same accuracy targets
- GNNLab outperforms

  DGL by 10.2× and T_{SOTA} by 3.5×

  1. Faster training (per epoch)
     - vs. DGL by 8.2× and T_{SOTA} by 2.8×
  2. Fewer epochs, reduced by 1.24×
     - GNNLab: 106 (6 GPU workers for training)
     - DGL/ T_{SOTA} : 131 (8 GPU workers for training)

The more GPUs allocated for model training, the fewer gradient updates per training epoch, and more epochs are required to achieve the same expected accuracy.





Conclusion & Thanks

GNNLab: a factored system for sample-based GNN training

- ► Replace time sharing with ***space sharing*** design
- ► ***Flexible*** architecture and scheduling for load balance
- ► A new ***efficient*** and ***robust*** caching policy

GNNLab: A Factored System for Sample-based GNN Training over GPUs

Jianbang Yang[† 1] Dahai Tang[† 3,4] Xiaoniu Song[1,2] Lei Wang[4] Qiang Yin[5]
Rong Chen[‡ 1,2] Wenyuan Yu[4] Jingren Zhou[4]

[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
[2]Shanghai AI Laboratory [3]Hunan University [4]Alibaba Group [5]BASICS, Shanghai Jiao Tong University

Abstract

We propose GNNLab, a sample-based GNN training system in a single machine multi-GPU setup. GNNLab adopts a factored design for multiple GPUs, where each GPU is dedicated to the task of graph sampling or model training. It accelerates both tasks by eliminating GPU memory contention. To balance GPU workloads, GNNLab applies a global queue to bridge GPUs asynchronously and adopts a simple yet effective method to adaptively allocate GPUs for different tasks. GNNLab further leverages temporarily switching to avoid idle waiting on GPUs. Furthermore, GNNLab proposes a new pre-sampling based caching policy that takes both sampling algorithms and GNN datasets into account, and shows an efficient and robust caching performance. Evaluations on three representative GNN models and four real-life graphs show that GNNLab outperforms the state-of-the-art GNN systems DGL and PyG by up to 9.1× (from 2.4×) and 74.3× (from 10.2×), respectively. In addition, our pre-sampling based caching policy achieves 90% – 99% of the optimal cache hit rate in all experiments.

Keywords: graph neural networks, sample-based GNN training, caching policy

ACM Reference Format:
Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: A Factored System for Sample-based GNN Training over GPUs. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3492321.3519557

† Jianbang Yang and Dahai Tang contributed equally to this work.
‡ Rong Chen is the corresponding author (rongchen@sjtu.edu.cn).

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04…$15.00
https://doi.org/10.1145/3492321.3519557

1 Introduction

Graph neural networks (GNNs) are an emerging family of neural networks that operate on graph-structured data [56], and have demonstrated convincing performance on many applications in recommendation systems [28, 57], molecule analysis [18], social network mining [32, 54, 60], fraud detection on e-commerce platforms [? ?], to name a few. GNN models aim to learn a low-dimensional feature representation (i.e., embedding) for each vertex in a graph, which can be further fed into various downstream graph-related tasks like vertex classification [24, 48], link prediction [45, 60] and graph clustering [?]. Different from traditional deep learning models, in GNN models, each vertex *recursively* updates its feature by aggregating features from its neighbors in the input graph [20], causing both implementation and performance challenges for existing tensor-oriented deep learning frameworks like TensorFlow [7], PyTorch [41] and MXNet [14]. Many GNN systems [2, 16, 19, 30, 36, 37, 47, 50, 51, 55, 64] have been developed in the recent past to address these challenges.

Many real-life graphs are large-scale and associated with rich vertex attributes (i.e., features), sometimes with highly skewed power-law degree distributions [8, 13, 21]. Under such conditions, it is inefficient, even unfeasible, to *simultaneously* consider all neighbors within L hops for each training vertex when training a GNN model with L layers [34, 64]. A practical solution to this problem is *sample-based* GNN training, which adopts various graph sampling algorithms to sample a fixed size of neighbors within L hops for each training vertex. Each training vertex and its corresponding sampled neighbors constitute a sample, and all samples are processed in a *mini-batch* manner [24, 57, 62]. In this way, the computation of each mini-batch can be largely reduced. The whole training process conducts such *iteration* (i.e., epoch) multiple times until the GNN model converges to expected accuracy [19, 59, 63].

Recently, GPUs have been widely exploited to accelerate GNN training [26, 30]. A typical scenario is a single machine equipped with *multiple* GPUs. Since large-scale graphs (including both topological and feature data) exceed limited GPU memory capacity (i.e., 32GB or smaller), most of the existing GNN systems store graph topological and feature

GNNLab will be published in **EuroSys 2022**

Artifact Evaluation: https://github.com/SJTU-IPADS/fgnn-artifacts

Open source: https://github.com/SJTU-IPADS/gnnlab (available soon)

AI needs Systems Research

Questions

Space sharing advance

- Fine-grained: inside a GPU (MPS and MIG)
- Decouple GPU CU and GPU memory

How about other GNNs and sampling algorithms?

- ClusterGCN and Shallow Subgraph Samplers

How to contribute to DGL?