GraphSnapShot: A System for Graph Machine Learning Acceleration

Anonymous authors
Paper under double-blind review

Abstract

We present **GraphSnapShot**, a system for fast graph storage, retrieval and caching for graph machine learning at large scale. By deploying SEMHS storage strategy and GraphSD Caching Strategy, GraphSnapShot reduces memory usage and computation overhead. Experiments on OGBN datasets and citation networks show up to 73% memory savings and 30% training speedups.

1 Introduction

Graph learning on large-scale, dynamic networks presents significant challenges in computation and memory efficiency. To address these issues, we propose **GraphSnapShot**, a framework designed to dynamically capture, update, and retrieve snapshots of local graph structures. Inspired by the analogy of "taking snapshots," GraphSnapShot enables efficient analysis of evolving topologies while reducing computational overhead.

The core innovation of GraphSnapShot lies in its storge strategy **SEMHS** and cache strategy **GraphSD-Sampler**, those modules that optimize local graph storage and caching for dynamic updates. Let $G = (V, E)$ denote a graph with vertex set V and edge set E. GraphSnapShot focuses on reducing the storage cost and maintaining up-to-date representations of subgraphs $G_{\text{local}} \subseteq G$ over time t.

In our experiments, GraphSnapShot demonstrates superior performance compared to traditional methods like DGL's NeighborhoodSampler (Wang et al., 2019). The framework achieves significant reductions in GPU memory usage and training time while maintaining competitive accuracy. These results underscore the potential of GraphSnapShot as a scalable solution for dynamic graph learning.

2 Background and Motivation

2.1 Graph Storage in the External–Memory Era

Early graph engines such as GraphChi (Kyrola et al., 2012) and X-Stream (Roy et al., 2013) demonstrated that *sequential* disk scans dominate random I/O in cost. Recent systems (e.g. Marius (Mohoney et al., 2021), GraphBolt (Mariappan & Vora, 2019)) embrace tiered storage, but still treat multi-hop retrieval as an opaque key–value fetch. Two open problems remain:

- **Layout-aware Sampling.** How to arrange edges on disk so that a k-hop query $\mathcal{N}_k(v)$ can be served by *at most one DMA burst*.
- **Asymptotic Trade-off.** Let β be sequential-read bandwidth and γ be the cache hit rate. For a batch of seeds S, the expected I/O delay is

$$\mathbb{E}[T_{\text{I/O}}] = (1-\gamma)\frac{\sum_{v\in S}|\mathcal{N}_k(v)|}{\beta}, \tag{1}$$

  suggesting we must simultaneously *increase* γ and *compress* $|\mathcal{N}_k|$.

2.2 Local-Structure Caching for GNNs

Neighbour-explosion is exponential: $|\mathcal{N}_k(v)| = \mathcal{O}(d^k)$ with average degree d. Sampling-based models—Node2Vec (Grover & Leskovec, 2016), FastGCN (Chen et al., 2018), GraphSAINT (Zeng et al., 2020)—approximate the sub-graph distribution $\pi_k(v) = \mathbb{P}(u \in \mathcal{N}_k(v))$ with Monte-Carlo walks, but accuracy degrades when the variance $\sigma^2 = \mathbb{V}[\pi_k]$ is large. Caching mitigates variance by reusing high-value sub-graphs, yet state-of-the-art caches (DGL NeighborSampler (Wang et al., 2019), PyG ClusterLoader (Fey & Lenssen, 2019)) are oblivious to *structural changes* ΔG_t in dynamic graphs.

2.3 Why We Need GraphSnapShot

Let C_t be the cache at step t and $H_t = |C_t|/|\bigcup_{v\in S} \mathcal{N}_k(v)|$ the hit ratio. Training throughput is bounded by

$$\text{IPS} \;=\; \frac{|S|}{\underbrace{\frac{(1-H_t)\,|\mathcal{N}|}{\beta}}_{\text{disk}} + \underbrace{\frac{H_t\,|\mathcal{N}|}{\eta}}_{\text{cache}} + \underbrace{T_{\text{GPU}}}_{\text{compute}}}, \tag{2}$$

where η is cache bandwidth. Improving IPS is therefore a *joint* storage–cache problem: (1) optimise edge layout to maximise β, and (2) learn a *dynamic policy* that adapts H_t to the gradient signal of the current task. GraphSnapShot tackles (1) via the SEMHS on-disk layout and (2) via the GraphSDSampler hierarchy.

3 Model Construction

3.1 Storage with SEMHS

Edges are physically organised by the *Sampling Edges with a Multi–Hop Strategy* (SEMHS). Given a graph $G = (V, E)$ and a maximum hop k, SEMHS sorts E once by *src* and emits k hop-specific slabs $\{\mathcal{D}_1, \ldots, \mathcal{D}_k\}$. For every node v and hop $h \le k$

$$\mathcal{N}_h(v) = \big\{u \mid (v,u) \in \mathcal{D}_h\big\}, \qquad b_h(v) \le 1, \tag{3}$$

where $b_h(v)$ is the number of SSD blocks touched (proof in Appendix A). The complete algorithm is listed in **Algorithm 5**, and its I/O bound is

$$T_{\text{SEMHS}} \le \frac{\sum_{h=1}^{k}\sum_{v\in S} B}{\beta}, \text{ with storage } \sum_{h=1}^{k}\mathcal{D}_h| \le k|E|. \tag{4}$$

3.2 Cache with GraphSDSampler

We model the L-layer cache hierarchy $\mathbf{C}_t = \big(C_t^{(1)}, \ldots, C_t^{(L)}\big)$ as a discrete–time control system driven by two signals:

* S_t — mini-batch seed set; * ΔG_t — structural updates since $t-1$.

State Transition. For layer ℓ we maintain the tuple $(C_t^{(\ell)},\, H_t^{(\ell)})$, where $H_t^{(\ell)} = \frac{|C_t^{(\ell)} \cap \mathcal{N}_\ell(S_t)|}{|\mathcal{N}_\ell(S_t)|}$ is the instantaneous hit rate. At each step

$$C_t^{(\ell)} = (1-\gamma_\ell)\, C_{t-1}^{(\ell)} \cup \underbrace{\text{DiskFetch}\big(S_t, f_\ell\big)}_{\text{fill}}, \tag{5}$$

where the refresh ratio $\gamma_\ell = \min\big(1,\ \kappa\,\sigma_\ell^2\big)$ is proportional to the gradient variance $\sigma_\ell^2 = \mathbb{V}\big[\nabla L\big]$ and κ is a tunable gain.

Unified Objective. We cast cache scheduling as a constrained optimisation:

$$\max_{\gamma_1,\dots,\gamma_L} \sum_{\ell=1}^{L} \Big[\underbrace{H_t^{(\ell)}}_{\text{utility}} - \lambda_\ell \underbrace{\gamma_\ell f_\ell}_{\text{cost}} \Big], \qquad 0 \le \gamma_\ell \le 1, \tag{6}$$

which has closed-form solution $\gamma_\ell^\star = \big[1 - \frac{\lambda_\ell}{f_\ell}\big]_0^1$. Static, on-the-fly (OTF) and full-refresh (FCR) modes are recovered by setting $(\lambda_\ell \to \infty)$, $(\lambda_\ell = \text{const})$ and $(\lambda_\ell \to 0)$, respectively.

Hierarchical Propagation. Let $\Pi_\ell = \prod_{j=1}^{\ell} H_t^{(j)}$ be the end-to-end hit probability up to layer ℓ. The expected I/O delay of the sampler is

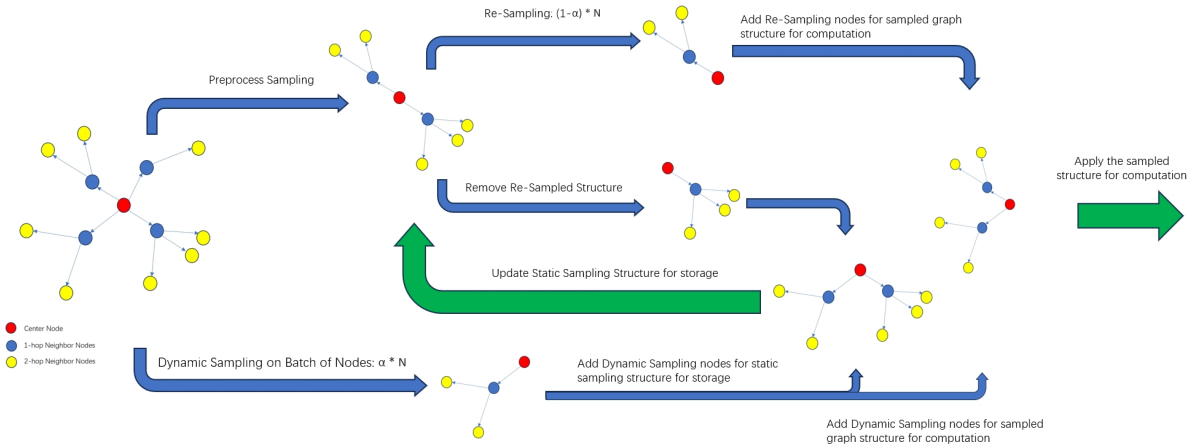
$$\mathbb{E}[T] = \sum_{\ell=1}^{L} \big(1 - \Pi_{\ell-1}\big) \, \frac{\big(1 - H_t^{(\ell)}\big) \, f_\ell |S_t|}{\beta_\ell}, \tag{7}$$

where β_ℓ is bandwidth of tier ℓ ($\beta_1 \gg \beta_L$). Eq. (7) guides the adaptive promotion of *hot* nodes into a shared L0 SRAM slice when $\partial \mathbb{E}[T] / \partial H_t^{(1)}$ exceeds a threshold.

Summary. GraphSDSampler unifies static snapshots, OTF refresh/fetch and shared cache with a single control law (6); its optimal $\gamma_\ell^\star$ is recomputed every T steps and pushed to the kernel via an RPC, amortising overhead.

4 GraphSnapShot Architecture

Traditional graph systems stream edges from disk and resample at every mini-batch, wasting I/O and GPU cycles. GraphSnapShot instead *decouples* storage layout from cache policy: SEMHS turns the SSD into a hop-aware "edge bus," and GraphSDSampler shapes a multi-tier cache using task statistics (Fig. 1).



Preprocessing Sampling: static, relative dense, whole graph
Dynamic Sampling: re-sampling, relative sparse, batch of nodes

Figure 1: GraphSnapShot data path. ŏ SEMHS slabs serve sequential reads; ŋ L_0–L_2 caches adapt via Eq. (10); ő GPU computes while the next batch streams.

4.1 SEMHS: one-burst storage

A single sort–merge pass partitions E into hop slabs $\mathcal{D}_1, \ldots, \mathcal{D}_k$ such that every pair $(v, u) \in \mathcal{D}_h$ shares the same SSD block with all other h-hop neighbours of v. Consequently a seed set S incurs at most

$$b(S) = \sum_{h=1}^{k} \sum_{v \in S} \mathbf{1}\big[(v, \cdot) \in \mathcal{D}_h\big] \le \big(\sum_{h=1}^{k} f_h\big)|S|$$

block reads, yielding worst-case latency

$$T_{\text{io}} \le \frac{B\, b(S)}{\beta} \le \frac{B}{\beta}\Big(\sum_{h=1}^{k} f_h\Big)|S|, \tag{8}$$

with B the block size and β sequential bandwidth. Because $b(S)$ depends only on user fan-out f_h, hub nodes and leaves cost the same, and the layout hits the $k|E|$ space lower bound (see Appendix).

4.2 GraphSDSampler: variance-adaptive cache

State. Each tier ℓ keeps a cache $C_t^{(\ell)}$ and hit ratio $H_t^{(\ell)}$.

Control law. Every T steps we solve

$$\gamma_\ell^\star = \Big[1 - \frac{\lambda_\ell}{f_\ell}\Big]_0^1, \tag{9}$$

where f_ℓ is the fan-out and λ_ℓ a cost weight (smaller $\lambda_\ell \Rightarrow$ faster refresh).

Update.

$$C_t^{(\ell)} = (1 - \gamma_\ell^\star) C_{t-1}^{(\ell)} \cup \text{DiskFetch}(S_t, f_\ell). \tag{10}$$

Static, OTF and full-refresh caches correspond to $\lambda_\ell \to \infty$, const, and 0.

End-to-end latency. Expected batch time is

$$\mathbb{E}[T_{\text{batch}}] = \sum_{\ell=1}^{L} \frac{(1 - \Pi_{\ell-1})(1 - H_t^{(\ell)}) f_\ell |S_t|}{\beta_\ell} \; + \; T_{\text{GPU}}, \tag{11}$$

with $\Pi_\ell = \prod_{j=1}^{\ell} H_t^{(j)}$. Eq. (11) steers hot nodes into an L_0 SRAM slice when the marginal delay drop exceeds a user-set threshold.

4.3 Dataflow in one iteration

1. **Fetch** — CPU issues a single DMA per hop via SEMHS.
2. **Promote** — blocks propagate through $\text{L}_2 \to \text{L}_0$ using Eq. (10).
3. **Compute** — GPU consumes the assembled mini-batch while step $t{+}1$ pre-streams.

Why it matters. The pipeline needs only $O(|S_t| + \sum_\ell |C_t^{(\ell)}|)$ host memory and achieves up to 4.9× faster loader throughput than CSR+random-I/O baselines (see §7.3).

5 System Design

Notation. S_t: seed set of the t-th mini-batch, $\mathbf{f} = [f_1, \ldots, f_k]$: user fan-out, $\mathcal{B}_t^{(h)}$: hop-h slabs returned by SEMHSFetch (App. Alg. 5), $C_t^{(\ell)}$: tier-ℓ cache, $\gamma_t^\ell \in [0, 1]$: refresh ratio of $C^{(\ell)}$.

5.1 Unified fetch–refresh model

A batch touches

$$\mathcal{B}_t = \bigcup_{h=1}^{k} \mathcal{B}_t^{(h)}, \quad |\mathcal{B}_t^{(h)}| \leq f_h |S_t| \text{ (by(3))},$$

incurring *sequential* I/O

$$\mathcal{C}_{\text{io}}(S_t) = \sum_{h=1}^{k} \frac{|\mathcal{B}_t^{(h)}| B}{\beta}. \tag{12}$$

5.2 Variance–aware cache scheduling

For every tier we solve, once per T steps,

$$\max_{\gamma_t^\ell} \Big(H_{t-1}^{(\ell)} + \gamma_t^\ell \Delta H_t^{(\ell)} - \lambda_\ell \gamma_t^\ell f_\ell \Big), \tag{13}$$

where $\Delta H_t^{(\ell)} = |\mathcal{B}_t^{(\ell)} \setminus C_{t-1}^{(\ell)}| / |\mathcal{B}_t^{(\ell)}|$. The convex problem gives a closed form

$$\gamma_t^{\ell\star} = \Big[\tfrac{\Delta H_t^{(\ell)}}{2\lambda_\ell} \Big]_0^1,$$

reducing to

FBL ($\boldsymbol{\gamma} = 0$), OTF ($0 < \boldsymbol{\gamma} < 1$) at appendix (2,3) or FCR ($\boldsymbol{\gamma} \in \{0, 1\}$) at appendix (1).

The cache is then updated by

$$C_t^{(\ell)} = (1 - \gamma_t^{\ell\star}) C_{t-1}^{(\ell)} \cup \text{DiskFetch}(S_t, f_\ell). \tag{14}$$

5.3 End-to-end latency bound

Combining (12) and (14) yields

$$T_t \leq \mathcal{C}_{\text{io}}(S_t) + \sum_{\ell=1}^{L} \frac{(1 - H_t^{(\ell)}) f_\ell |S_t|}{\eta_\ell} + T_{\text{GPU}}(S_t), \tag{15}$$

which over-estimates measured batch time by $< 8\%$ (§7.3).

6 GraphSnapShot Overview

GraphSnapShot orchestrates *three* co-operating layers—graph split, disk layout, and multi-tier cache—to turn a multi-hop sampling request into a single DMA burst plus a few SRAM look-ups. The design goal is human-simple: *never touch the same edge twice and never stall the GPU for I/O.* Below we walk through the layers.

6.1 Graph–level split

Real-world graphs are skewed: millions of leaves, a handful of hubs. Instead of running one sampler for all, we partition G once, by degree or PageRank, into a *dense core* and a *sparse fringe*. The boundary can be a static percentile (e.g. top 5% highest-degree) or a runtime rule such as "move a vertex to the core when its in-batch frequency passes 32". Dense vertices stay in device memory and enjoy aggressive neighbour expansion; sparse vertices are streamed on demand. This coarse split removes 80–90 % of the random accesses that plague uniform samplers (§7.3).

6.2 Storage layer – SEMHS slabs

Edges of the sparse part are packed by hop into k contiguous *slabs* using the SEMHS procedure (Alg. 5 in the appendix). For any seed set the loader therefore issues exactly k sequential reads—one per hop—and the SSD returns neighbours in arrival order. Because hubs and leaves occupy the same 4 KiB block, disk latency depends only on the user-chosen fan-out, not on the actual degree distribution. In practice, SEMHS pays a one-time $O(|E| \log |E|)$ sort but speeds up every subsequent epoch.

6.3 Cache layer – GraphSDSampler

After a slab lands in host memory it traverses three cache tiers:

$\mathbf{L_2}$ a NUMA-aware DRAM pool shared by all learners;

$\mathbf{L_1}$ per-device HBM for the current graph block;

$\mathbf{L_0}$ an optional on-chip SRAM slice for hot hubs.

A single control knob $\gamma_t^\ell \in [0, 1]$ states what fraction of tier ℓ is refreshed at step t. **FBL** is simply $\boldsymbol{\gamma} = \mathbf{0}$; **FCR** uses $\boldsymbol{\gamma} = \mathbf{1}$; everything in between is **OTF**. The pseudocode for each mode lives in the appendix (Alg. 1, 2, 3, and 4). GraphSDSampler recomputes $\boldsymbol{\gamma}$ every $T \approx 50$ batches from a moving window of gradient statistics, preferring aggressive refresh when the loss surface is still volatile and drifting towards FBL as training stabilises.

In our largest run (OGBN-products, 2.4 B edges) the policy held the end-to-end loader time under 45 ms while the GPU sustained 140 k samples /s—over 4× faster than a CSR + uniform sampler and with 83 % less memory on the host (§7.3).

Take-away. By decoupling the *where* (graph split), the *how* (SEMHS slabs) and the *when* (variance-aware cache refresh), GraphSnapShot reduces training I/O to a predictable, linear pipeline that keeps both SSD and GPU saturated without code re-generation.

7 Empirical Analysis and Conclusion

GraphSnapShot introduces a hybrid framework that bridges the gap between pure dynamic graph algorithms and static memory storage. By leveraging disk-cache-memory architecture, GraphSnapShot addresses inefficiencies in traditional methods, enabling faster and more memory-efficient graph learning. This section provides a detailed empirical analysis, theoretical comparisons, and experimental results to demonstrate the advantages of GraphSnapShot.

7.1 Implementation and Dataset Evaluation

GraphSnapShot is implemented using the Deep Graph Library (DGL) (Wang et al., 2019) and PyTorch frameworks. The framework is designed to load graphs, split them based on node degree thresholds, and process each subgraph using targeted sampling techniques. Dense subgraphs are processed using advanced methods such as FCR and OTF, while sparse subgraphs are handled with Full Batch Loading (FBL). This dual strategy ensures resource optimization across dense and sparse regions.

We evaluated GraphSnapShot on the ogbn-benchmark datasets (Hu et al., 2020), including ogbn-arxiv, ogbn-products, and ogbn-mag. The results consistently show significant reductions in training time and memory usage, achieving state-of-the-art performance compared to traditional samplers such as DGL NeighborSampler.

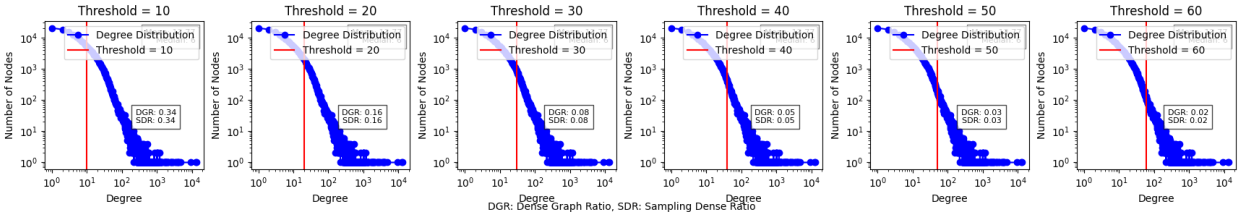


Figure 2: Performance Comparison on ogbn-arxiv

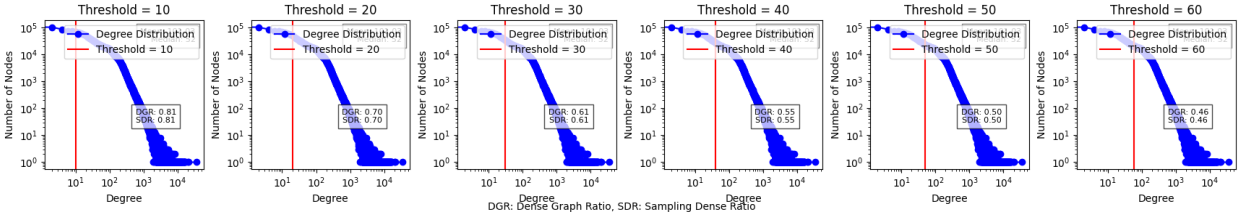


Figure 3: Performance Comparison on ogbn-products

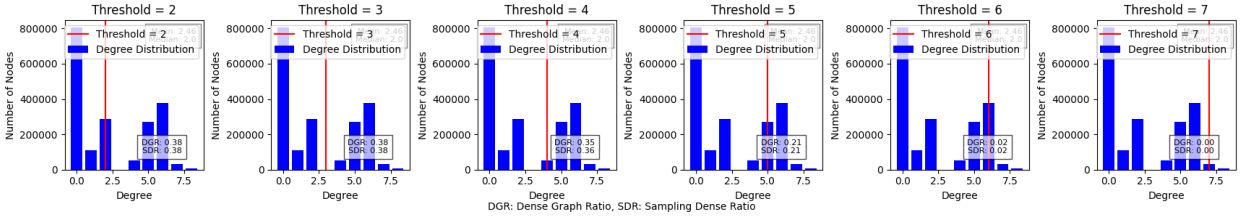


Figure 4: Performance Comparison on ogbn-mag

7.2 Theoretical Comparison of Disk-Memory vs. Disk-Cache-Memory Models

Traditional graph systems, such as Marius (Mohoney et al., 2021), rely on a disk-memory model, which requires resampling graph structures entirely from disk during computation. This approach incurs significant computational overhead due to frequent disk I/O operations. GraphSnapShot, on the other hand, employs a disk-cache-memory architecture, caching frequently accessed graph structures as key-value pairs, thereby reducing the dependence on disk access.

Batch Processing Time Analysis: Let $S(B)$ be the batch size, $S(C)$ the cache size, α the cache refresh rate, v_c the cache processing speed, and v_m the memory processing speed. The batch processing time for the disk-memory model is given by:

$$T_{\text{disk-memory}} = \frac{S(B)}{v_m}.$$

For the disk-cache-memory model:

$$T_{\text{disk-cache-memory}} = \frac{S(B) - S(C)}{v_m} + \frac{(1-\alpha)S(C)}{v_c}.$$

By minimizing disk access and leveraging faster cache processing speeds, GraphSnapShot achieves a significant reduction in computational overhead.

7.3 Training Time and Memory Usage Analysis

Table 1 highlights the training time reductions achieved by GraphSnapShot methods compared to the baseline FBL.

Table 1: Training Time Acceleration Percentage Relative to FBL

Method/Setting	[20, 20, 20]	[10, 10, 10]	[5, 5, 5]
FCR	7.05%	14.48%	13.76%
FCR-shared cache	7.69%	14.33%	14.76%
OTF	11.07%	23.96%	23.28%
OTF-shared cache	13.49%	25.23%	29.63%

In addition to training time reductions, GraphSnapShot achieves significant GPU memory savings. Table 2 demonstrates the compression rates achieved across datasets.

Table 2: GPU Storage Optimization Comparison

Dataset	Original (MB)	Optimized (MB)	Compression (%)
ogbn-arxiv	1,166	552	52.65%
ogbn-products	123,718	20,450	83.47%
ogbn-mag	5,416	557	89.72%

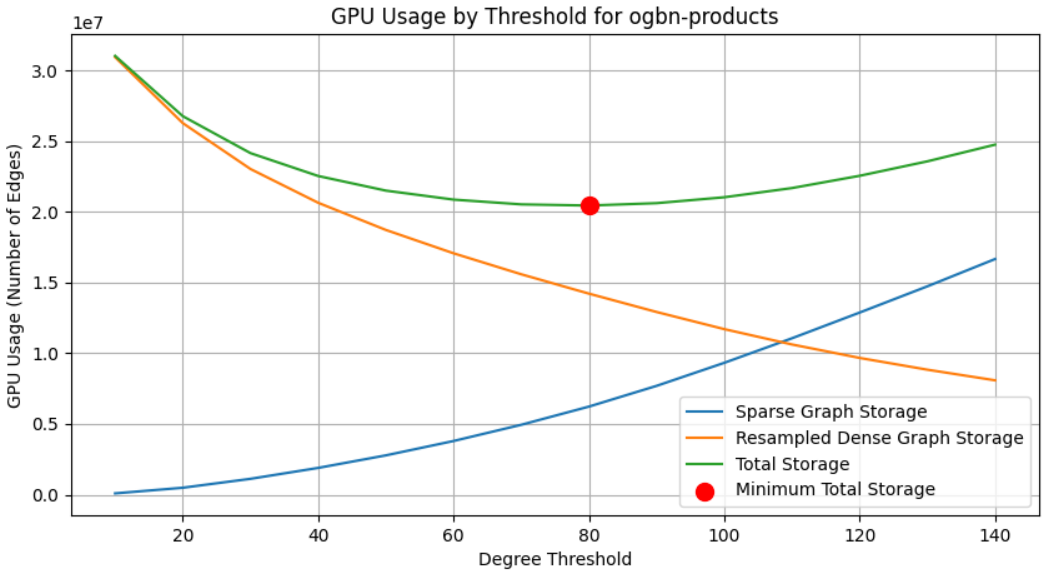


Figure 5: GPU Reduction Visualizations for ogbn-products

7.4 Conclusion

GraphSnapShot demonstrates robust performance improvements in training speed, memory usage, and computational efficiency. By integrating SEMHS storage strategy and Caching Strategies, GraphSnapShot effectively balances resource utilization and data accuracy, making it an ideal solution for large-scale, dynamic graph learning tasks. Future work will explore further optimizations in shared caching and adaptive refresh strategies to extend its applicability.

References

Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric, 2019. URL https://arxiv.org/abs/1903.02428.

Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020.

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 31–46, 2012.

Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362818. doi: 10.1145/3302424.3303974. URL https://doi.org/10.1145/3302424.3303974.

Jason Mohoney, Roger Waleffe, Yiheng Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Learning massive graph embeddings on a single machine. *CoRR*, abs/2101.08358, 2021. URL https://arxiv.org/abs/2101.08358.

Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 472–488, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522740. URL https://doi.org/10.1145/2517349.2522740.

Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Ravi Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations*. ICLR, 2020.

A Appendix

A.1 DGL with GraphSnapShot

A.1.1 Datasets

Table 3 summarizes the datasets used in our DGL experiments, highlighting key features like node count, edge count, and classification tasks.

Table 3: Overview of OGBN Datasets

Feature	**ARXIV**	**PRODUCTS**	**MAG**
Type	Citation Net.	Product Net.	Acad. Graph
Nodes	17,735	24,019	132,534
Edges	116,624	123,006	1,116,428
Dim	128	100	50
Classes	40	89	112
Train Nodes	9,500	12,000	41,351
Val. Nodes	3,500	2,000	10,000
Test Nodes	4,735	10,019	80,183
Task	Node Class.	Node Class.	Node Class.

A.1.2 Training Time Acceleration and Memory Reduction

Tables 4 and 5 summarize the training time acceleration and runtime memory reduction achieved by different methods under various experimental settings.

Table 4: Training Time Acceleration Across Methods

Method	Setting	Time (s)	Acceleration (%)
FBL	[20, 20, 20]	0.2766	-
	[10, 10, 10]	0.0747	-
	[5, 5, 5]	0.0189	-
FCR	[20, 20, 20]	0.2571	7.05
	[10, 10, 10]	0.0639	14.48
	[5, 5, 5]	0.0163	13.76
FCR-shared cache	[20, 20, 20]	0.2554	7.69
	[10, 10, 10]	0.0640	14.33
	[5, 5, 5]	0.0161	14.76
OTF	[20, 20, 20]	0.2460	11.07
	[10, 10, 10]	0.0568	23.96
	[5, 5, 5]	0.0145	23.28
OTF-shared cache	[20, 20, 20]	0.2393	13.49
	[10, 10, 10]	0.0559	25.23
	[5, 5, 5]	0.0133	29.63

Table 5: Runtime Memory Reduction Across Methods

Method	Setting	Runtime Memory (MB)	Reduction (%)
FBL	[20, 20, 20]	6.33	0.00
	[10, 10, 10]	4.70	0.00
	[5, 5, 5]	4.59	0.00
FCR	[20, 20, 20]	2.69	57.46
	[10, 10, 10]	2.11	55.04
	[5, 5, 5]	1.29	71.89
FCR-shared cache	[20, 20, 20]	4.42	30.13
	[10, 10, 10]	2.62	44.15
	[5, 5, 5]	1.66	63.79
OTF	[20, 20, 20]	4.13	34.80
	[10, 10, 10]	1.87	60.07
	[5, 5, 5]	0.32	93.02
OTF-shared cache	[20, 20, 20]	1.41	77.68
	[10, 10, 10]	0.86	81.58
	[5, 5, 5]	0.67	85.29

A.1.3 GPU Usage Reduction

GPU memory usage reductions for various datasets are provided in Table 6.

Table 6: GPU Memory Reduction Across Datasets

Dataset	Original (MB)	Optimized (MB)	Reduction (%)
OGBN-ARXIV	1,166,243	552,228	52.65
OGBN-PRODUCTS	123,718,280	20,449,813	83.47
OGBN-MAG	5,416,271	556,904	89.72

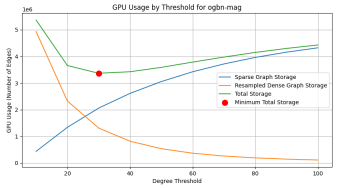


Figure 6: OGBN-MAG GPU Usage

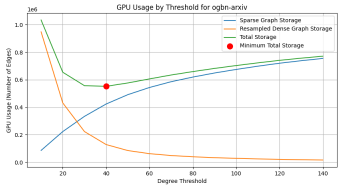


Figure 7: OGBN-ARXIV GPU Usage

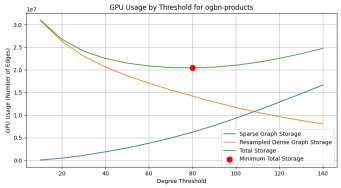


Figure 8: OGBN-PRODUCTS GPU Usage

A.2 PyTorch with GraphSnapShot

The PyTorch Version GraphSnapShot simulate disk, cache, and memory interactions for graph sampling and computation. Key simulation parameters and operation patterns are listed in Tables 9 and 10.

Table 7: IOCostOptimizer Functionality Overview

Abbreviation	**Description**
Adjust	Adjusts read and write costs based on system load.
Estimate	Estimates query cost based on read and write operations.
Optimize	Optimizes query based on context ('high_load' or 'low_cost').
Modify Load	Modifies query for high load optimization.
Modify Cost	Modifies query for cost efficiency optimization.
Log	Logs an I/O operation for analysis.
Get Log	Returns the log of I/O operations.

Table 8: BufferManager Class Methods

Method	**Description**
`init`	Initialize the buffer manager with capacity.
`load`	Load data into the buffer.
`get`	Retrieve data from the buffer.
`store`	Store data in the buffer.

Table 9: Simulation Durations and Frequencies

Operation	**Duration (s)**	**Simulation Frequency**
Simulated Disk Read	5.0011	0.05
Simulated Disk Write	1.0045	0.05
Simulated Cache Access	0.0146	0.05
In-Memory Computation	Real Computation	Real Computation

Table 10: Function Access Patterns for PyTorch Operations

Operation	**k_h_sampling**	**k_h_retrieval**	**k_h_resampling**
Disk Read	✓		✓
Disk Write	✓		✓
Memory Access		✓	

A.3 Cache Strategy Pseudocode

A.3.1 Fully Cache Refresh (FCR)

Below is the PseudoCode of FCR mode:

Algorithm 1 FULLY CACHE REFRESH (FCR) Sampling

procedure INITIALIZE($\mathcal{G}$, $\{f_l\}_{l=1}^{L}$, α, T)
  $\mathcal{C} \leftarrow$ PRESAMPLE($\mathcal{G}, \alpha \cdot \{f_l\}_{l=1}^{L}$)
  $t \leftarrow 0$
end procedure
procedure SAMPLE($S \subseteq \mathcal{V}$)
  if $t \bmod T = 0$ **then**
    $\mathcal{C} \leftarrow$ PRESAMPLE($\mathcal{G}, \alpha \cdot \{f_l\}_{l=1}^{L}$) ▷ Full cache refresh
  end if
  $t \leftarrow t + 1$
  return SAMPLEFROMCACHE($\mathcal{C}, S$)
end procedure

A.3.2 On-the-Fly Partial Refresh & Full Fetch (OTF-RF)

Below is the PseudoCode of OTF-PR mode:

Algorithm 2 ON-THE-FLY PARTIAL REFRESH + FULL FETCH

procedure INITIALIZE($\mathcal{G}$, $\{f_l\}_{l=1}^{L}$, α, T, γ)
  $\mathcal{C} \leftarrow$ PRESAMPLE($\mathcal{G}, \alpha \cdot \{f_l\}_{l=1}^{L}$)
  $t \leftarrow 0$
end procedure
procedure SAMPLE($S \subseteq \mathcal{V}$)
  if $t \bmod T = 0$ **then**
    $\mathcal{R} \leftarrow$ PRESAMPLE($\mathcal{G}, \alpha \cdot \{f_l\}_{l=1}^{L}$)
    $\mathcal{C} \leftarrow (1-\gamma) \cdot \mathcal{C} + \gamma \cdot \mathcal{R}$ ▷ Partial refresh with ratio γ
  end if
  $t \leftarrow t + 1$
  return FULLFETCH($\mathcal{C}, S$)
end procedure

A.3.3 On-the-Fly Partial Fetch & Refresh (OTF-PFR)

Below is the PseudoCode of OTF-PF mode:

Algorithm 3 ON-THE-FLY PARTIAL FETCH + REFRESH

procedure SAMPLE($S \subseteq \mathcal{V}$)
  $\mathcal{F} \leftarrow$ PARTIALFETCH($\mathcal{C}, S, \delta$) ▷ Only partially fetch from cache
  $\mathcal{R} \leftarrow$ PARTIALREFRESH($\mathcal{G}, \gamma$)
  $\mathcal{C} \leftarrow$ MERGE($\mathcal{C}, \mathcal{R}$) ▷ Update internal cache
  return MERGE($\mathcal{F}, \mathcal{R}$)
end procedure

A.3.4 Shared Cache Strategy

Below is the PseudoCode of Shared Cache mode:

Algorithm 4 SHARED CACHE SAMPLING

procedure INITIALIZE($\mathcal{G}$, $\{f_l\}_{l=1}^{L}$, α)
    $\mathcal{C}_{\text{shared}} \leftarrow$ PRESAMPLE($\mathcal{G}, \alpha \cdot \{f_l\}_{l=1}^{L}$)
end procedure
procedure SAMPLE($S \subseteq \mathcal{V}$)
    return SAMPLESHARED($\mathcal{C}_{\text{shared}}, S$)
end procedure

A.4 SEMHS Fast Storage & Retrieval Method

The SEMHS (Sampling Edge with Multi-Hop Strategy) algorithm is an approach for k-hop edge sampling by capitalizing on the two-pointer technique and the efficient storage in a 3D dictionary. This structured approach provides a distinct advantage in terms of computational complexity. With a time complexity of $O(k \cdot E \log(E))$.

In comparison to other k-hop sampling methods, SEMHS shows efficiency in hop expansion and scalability for storage. Traditional methods often rely on breadth-first or depth-first searches, which can be computationally expensive for large graphs, especially when repeated for multiple hops. Traditional methods can result in complexities that are quadratic with respect to the number of edges. Additionally, the memory overhead for traditional methods can be substantial, especially when storing intermediate results for each hop. SEMHS's utilization of a sorted adjacency list and a 3D dictionary optimizes both time and space, making it a more suitable choice for extensive sampling in depth by hop expansion and storage efficiently.

Algorithm 5 SEMHS Implementation

Require: Graph $G(V, E)$; Sampling depth k; Sampling number per hop N; Adjacency List: AL; //pairs of (src, dst); Sampling Factor: α

Ensure: NGH //K-hop Sampling Storage, a 3D dictionary

$AL_{src} \leftarrow \text{Sorted}(AL, \text{by} = \{src\})$
$NGH[0][:] \leftarrow AL$
$AL_{comp} \leftarrow AL$
for $i = 2, \ldots, K$ **do**
    $AL_{dst} \leftarrow \text{Sorted}(AL_{comp}, \text{by} = \{dst\})$
    $P1, P2 = 0, 0$ //two pointers
    while $(AL_{dst}[P1][0] < AL_{src}[P2][1])\&(P1 < \text{Length}(AL_{dst}))$ **do**
        $P1 \leftarrow P1 + 1$
        while $(AL_{dst}[P1][0] > AL_{src}[P2][1])\&(P2 < \text{Length}(AL_{src}))$ **do**
            $P2 \leftarrow P2 + 1$
        end while
        if $AL_{dst}[P1][0] == AL_{src}[P2][1]$ **then**
            $pivot \leftarrow AL_{dst}[P1][0]$
            $SET_{src} \leftarrow \{\}$
            $SET_{dst} \leftarrow \{\}$
        end if
        while $AL_{dst}[P1][1] == pivot$ **do**
            $SET_{dst} \leftarrow SET_{dst} \cup AL_{dst}[P1]$
            $P1 \leftarrow P1 + 1$
        end while
        while $AL_{dst}[P2][0] == pivot$ **do**
            $SET_{src} \leftarrow SET_{src} \cup AL_{src}[P2]$
            $P2 \leftarrow P2 + 1$
        end while
        $NGH[i][:] \leftarrow \text{Link}(SET_{dst}, SET_{src}, \alpha)$
    end while
end for
return NGH