RingSampler: GNN sampling on large-scale graphs with io_uring

Qixuan Chen
taliac@bu.edu
Boston University

Yuhang Song
yuhangs@bu.edu
Boston University

Melissa Martinez
melimtz@bu.edu
Boston University

Vasiliki Kalavri
vkalavri@bu.edu
Boston University

ABSTRACT

Neighborhood sampling is a critical computation step in graph learning with Graph Neural Networks (GNNs), often accounting for the majority of the training time. To mitigate this bottleneck and scale training to very large graphs, existing approaches offload the sampling computation to GPUs or computational storage, such as SmartSSDs. Given the ubiquity of multi-core CPUs and high-throughput SSDs, we investigate a simpler design that performs CPU-based sampling, making GPU resources fully available to the aggregation stage of training instead. We propose RingSampler, a new GNN sampling system that leverages `io_uring` to support efficient training of billion-edge graphs on a single machine. RingSampler parallelizes sampling by transparently assigning mini-batches to threads and effectively overlapping computation with I/O operations. Our results show that RingSampler significantly outperforms SmartSSD-based sampling on large graphs and is competitive with GPU-accelerated approaches on graphs that fit in main memory.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Information systems** → **Computing platforms**.

KEYWORDS

Graph Neural Networks; io_uring; Neighborhood sampling

ACM Reference Format:
Qixuan Chen, Yuhang Song, Melissa Martinez, and Vasiliki Kalavri. 2025. RingSampler: GNN sampling on large-scale graphs with io_uring. In *17th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '25), July 10–11, 2025, Boston, MA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3736548.3737829

HotStorage '25, July 10–11, 2025, Boston, MA, USA

ACM ISBN 979-8-4007-1947-9/2025/07…$15.00
https://doi.org/10.1145/3736548.3737829

1 INTRODUCTION

Graph Neural Networks (GNNs) [38] have been extensively used in machine learning tasks that require learning from graph-structured data. GNNs capture relationships and dependencies between entities in a network, making them suitable for applications such as as node classification, link prediction, and recommendation systems [11, 19, 36, 37].

Neighborhood sampling [2, 10, 13] can be a significant bottleneck in graph learning. Empirical studies have shown that it often accounts for more than 50% of the overall training time, and in some cases, exceeds 90% of the end-to-end execution time. This overhead is primarily due to the irregular memory access patterns and poor data locality inherent in the sampling process, which introduce substantial delays during data preparation. [9, 15, 17, 24, 25] As a result, various sampling acceleration approaches have been proposed. Nextdoor [15] and gSampler [9] offload sampling to GPUs, however, they are restricted by the capacity of GPU memory. Even for small graphs, the sampling computation competes for GPU resources with the aggregation stage, potentially increasing end-to-end training time. Ginex [25] and MariusGNN [30] support training on large graphs by loading edge partitions from SSDs to memory but they suffer from unnecessary I/O [29]. To avoid the data transfer overhead, FlashGNN [23], SmartSAGE [20], and BeaconGNN [32] push the sampling computation to the SSD controller.

In this paper, we propose RingSampler, a new GNN sampling system that leverages `io_uring` to effectively address the sampling bottleneck and fully utilize modern multi-core CPUs and SSDs. RingSampler supports neighborhood sampling on larger-than memory graphs, without requiring any specialized hardware. We make the following contributions:

- We design and implement RingSampler, an `io_uring`-based GNN sampling system tailored for multi-core CPUs

and SSDs. RingSampler leverages an index-based sampling mechanism to avoid unnecessary data movement, retrieving only the sampled neighbors from disk.
- We implement a multi-threaded, asynchronous sampling engine that fully utilizes multi-core CPUs and achieves non-blocking execution by overlapping I/O preparation and completion.
- We conduct a comprehensive experimental evaluation, across multiple datasets and sampling systems, including in-memory, GPU-based and SmartSSD-based baselines.

When compared to out-of-core systems, RingSampler exhibits significant benefits, achieving up to 52× lower sampling time per epoch. Our results show that RingSampler is competitive even against in-memory and GPU-accelerated approaches, while it can also be suitable for on-demand sampling in near-real-time GNN serving settings. We have released our code and experiments for public access[1].

2 BACKGROUND AND RELATED WORK

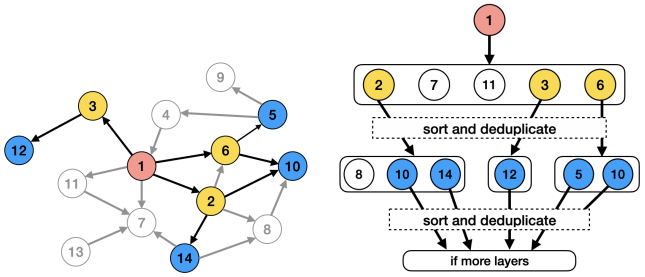
In this section, we provide background on GNN sampling and discuss existing GNN sampling approaches and their limitations, which motivate us to design RingSampler.

2.1 GNN Sampling

To keep the learning cost controllable and make GNNs friendly to GPU architectures, many GNN algorithms perform sampling to construct fixed-size *mini-batches* for training and inference [18, 28, 33]. Sampling bounds the computation complexity and avoids load imbalance in graphs with skewed degree distribution. Most GNN methods perform sampling either *per-node*, so that each vertex selects a subset of its neighbors at each layer [3, 5, 6], or *per-layer*, so that multiple vertices are selected simultaneously for each GNN layer in a single step [1, 14, 39]. Subgraph-based sampling techniques have also been proposed, though these require an expensive pre-processing step to partition the graph into clusters [4].

In this work, we focus on the GraphSAGE [10] node-wise sampling model. GraphSAGE training proceeds in epochs and within each epoch, nodes are divided into mini-batches. Nodes selected for training within a mini-batch are called *target* nodes. GraphSAGE recursively samples neighbors of target nodes, uniformly at random, up to k hops (layers), to form subgraphs for weight aggregation. The sample size per layer is defined by the *fanout* parameter.

Figure 1 shows an example of the sampling process on a 2-layer GraphSAGE model with a `fanout` of {3, 2} for the first and second layer, respectively. Given target node 1, we randomly select neighbors {2, 3, 6}, which serve as target nodes of the next layer. Next, we randomly select up to two



(a) Input graph (b) Workflow overview

Figure 1: GraphSAGE 2-hop sampling example

neighbors per target node, generating the sample {10, 14, 12, 5, 10}. The list of sampled nodes is dedupliacted in between layers, so that target nodes are unique.

2.2 Existing GNN sampling systems

We divide existing systems into three categories, based on where the sampling computation is executed, and we analyze the strengths and limitations of each approach.

2.2.1 Out-of-core CPU-based systems. To handle large-scale graphs without distributing tasks across multiple machines, CPU-based systems such as MariusGNN [30], Ginex [25] and GNNDDrive [16], leverage SSDs for graph storage and perform sampling on a single machine. They acknowledge the overhead introduced by frequent data movement between storage layers and propose solutions to address this issue.

Due to the nature of GNNs, a node may get repeatedly sampled across different layers. MariusGNN mitigates this redundancy by reusing previously sampled neighbors across layers. However, this reuse compromises the randomness of sampling and may affect model accuracy. On the other hand, Ginex constructs a neighbor cache during offline preprocessing, storing the neighbors of important nodes in memory. During training, only nodes not in the cache are fetched from the SSD. GNNDrive employs storage-efficient APIs and asynchronous I/O but it does so only during the feature retrieval stage, while it performs neighborhood sampling in memory.

While the above approaches reduce data movement, they still load the full neighborhood of each target node into main memory during sampling. Since only a subset of neighbors is actually used, this leads to unnecessary I/O.

2.2.2 GPU-based systems. Leveraging the massive parallelism of modern GPUs, systems such as DGL [31] (GPU mode), Nextdoor [15] and gSampler [9] demonstrate significantly better performance than CPU-based sampling methods. The typical workflow of GPU-based sampling approaches involves three steps: (i) transferring the graph data from main memory to GPU memory, (ii) performing the sampling algorithm and, (iii) copying the sample back to the CPU for

[1] https://github.com/CASP-Systems-BU/RingSampler

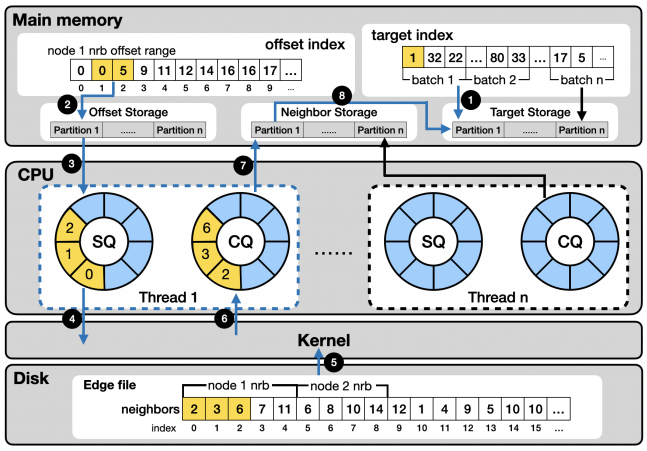


Figure 2: RingSampler overview

downstream tasks. Additionally, techniques such as CGC compression [35] minimize transfer costs by compressing graph representations. Yet, GPU-based systems require the graph structure to fit either in GPU or CPU memory, rendering them unsuitable for training very large graphs. Moreover, this design creates contention for GPU resources, leading to inefficient resource utilization. For instance, in our experiments with DGL in GPU mode, we observed that DGL's GPU pipeline performs both sampling and feature access on the GPU, resulting in nearly zero CPU usage during training.

2.2.3 In situ SSD-based systems. SSDs have been widely used in disk-based and GPU-based systems as additional storage. However, frequent data movement between SSDs and computation memory can become a significant bottleneck, prompting the development of systems that perform sampling directly on SSDs. Notable SSD-based systems include SmartSAGE [20], FlashGNN [23], and the SmartSSD-based sampling system [29]. Despite their advantages, in situ sampling systems rely on specific hardware characteristics, making them challenging to adopt broadly. Further, due to the limited compute capabilities of storage devices, they do not perform as well as other sampling approaches, as we demonstrate in Section 4.2.

3 GNN SAMPLING WITH IO_URING

To address the limitations of existing GNN sampling systems, we propose RingSampler, a CPU-based system that leverages high-bandwidth SSDs and `io_uring` to perform efficient sampling on larger-than-memory graphs on a single machine. In this section, we describe how RingSampler avoids loading entire node neighborhoods in memory by constructing an in-memory offset index and how it effectively parallelizes sampling across mini-batches to utilize the available I/O bandwidth and achieve non-blocking execution.

3.1 System overview

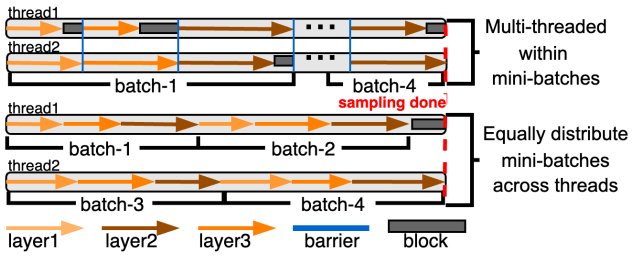
Data Preprocessing. Since RingSampler is designed to store graph data on SSDs and perform sampling on the CPU, efficient data access is critical. To leverage `io_uring`'s efficient random I/O, we design a hybrid data structure consisting of two in-memory index structures and one on-disk edge file. We use an in-memory target index array to store target nodes. Given n threads, we divide the array into n batches, with each thread independently processing its assigned batch to ensure a balanced workload. The edge file is constructed by sorting all edges based on their source nodes, then storing only the destination nodes as a flat list of integers. This organization ensures that all neighbors of a given source node are stored contiguously on disk. Alongside this, we build an in-memory offset index, where the neighbors of node x reside in the range [offset_index[x], offset_index[$x+1$]) in the edge file. This allows sampling to operate directly on index ranges, as each neighbor can be uniquely identified by its offset index rather than by its actual value.

To support efficient parallel processing, RingSampler allocates three thread-local workspaces for intermediate storage of offsets, neighbors, and target nodes. Each workspace is partitioned such that threads access only their assigned regions, avoiding contention. As a result, memory usage scales with the number of threads but remains independent of the total number of graph edges.

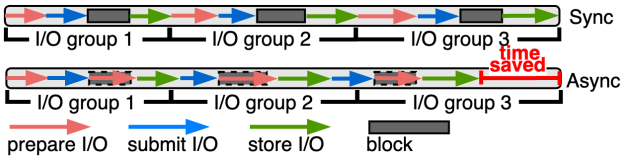
Sampling. Figure 2 illustrates the workflow of RingSampler, using the graph of Figure 1a to show how it performs sampling for target node {1}. To sample neighbors for node {1} ①, RingSampler looks up its neighborhood range $[0, 5)$ in the offset index ②. It then randomly selects *fanout* (3) offsets from this range (in this case {0, 1, 2}) as the sampled neighbors ③. Using these sampled offsets, RingSampler submits `io_uring` read requests to fetch only the corresponding entries—{2, 3, 6}—from the edge file and load them to memory ④ ⑤ ⑥. The sampled neighbor values are then stored ⑦, and a deduplication step is performed to obtain the unique set of nodes that will be used as the target nodes for the next sampling layer ⑧.

By using offset-based sampling, RingSampler avoids fetching full neighbor lists from disk (e.g., neighbors 7 and 11 are skipped and not loaded). In real-world datasets, a node's neighborhood can contain hundred of thousands of neighbors [27]. Thus, this strategy significantly reduces disk I/O and eliminates unnecessary data loading, without modifying the sampling algorithm or compromising accuracy.

Eliminating thread synchronization. Figure 3a shows an example of a multi-threaded sampling approach, where threads collaboratively process mini-batches, similar to MariusGNN. However, due to layer dependencies in GraphSAGE



(a) Parallelism strategy comparison



(b) Synchronous vs. Asynchronous I/O Pipeline

Figure 3: Parallel and asynchronous design

sampling, this design forces threads to wait for one another before proceeding to the next mini-batch, causing frequent synchronization and poor CPU utilization.

To eliminate this synchronization overhead, RingSampler adopts a parallel design where mini-batches are evenly distributed across threads. Since mini-batches are mutually independent, threads can process their assigned batches without coordination. To further enable parallelism, each thread is assigned a dedicated pair of `io_uring` ring buffers, Submission Queue (SQ) for storing I/O requests and Completion Queue (CQ) for storing I/O results.

Overlapping computation and I/O. Beyond inter-thread synchronization, blocking can also occur because of I/O. `io_uring` allows batching a group of I/O operations in a single system call, with the group size determined by the buffer capacity, commonly referred to as the `Queue Depth`. The lifecycle of processing an I/O groups involves preparing the I/O operation load on SQ, submitting the request, and retrieving the result, corresponding to steps ③, ④, and ⑤ in Figure 2, respectively. In a synchronous pipeline, as shown in Figure 3b, a significant portion of CPU resources is wasted while waiting for the kernel to complete I/O operations.

To address this inefficiency, we leverage `io_uring`'s completion polling mode, which enables the CQ to continuously poll for I/O results from the kernel without issuing additional system calls. This allows us to design an asynchronous pipeline, shown in Figure 3b. While CQ is collecting I/O results for group-1, we simultaneously prepare I/O requests for group-2 and load them into SQ. By the time group-2's I/O requests are ready, group-1's results are already available in the CQ, ready for processing and group-2's I/O requests can be submitted immediately. By overlapping I/O completion waiting time with I/O preparation, RingSampler reduces idle CPU time and maximizes throughput, effectively minimizing intra-thread blocking.

Table 1: Graphs used in the evaluation. The size columns correspond to the edge list, in raw text and binary format.

Graph	|V|	|E|	Raw Size (GB)	Bin Size (GB)
ogbn-papers [12]	111M	1.6B	24.1	6.8
Friendster [21]	65M	3.6B	30.1	13.5
Yahoo [27]	1.4B	6.6B	66.9	35.3
Synthetic [26]	134M	8.2B	140.8	31.7

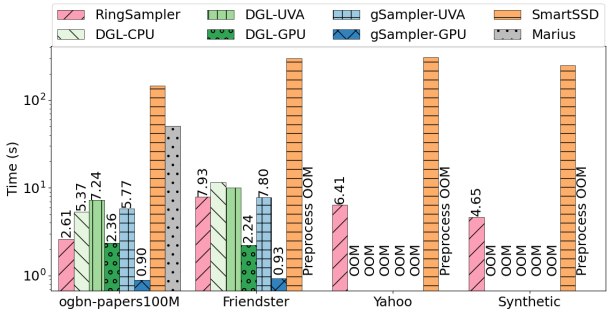


Figure 4: Performance comparison across in-memory, out-of-core, and GPU-based sampling systems.

4 EXPERIMENTAL EVALUATION

We compare RingSampler 's performance with various baselines and evaluate its scalability under memory constraints and increasing sampling depth, as well as its suitability for real-time inference workloads.

4.1 Experimental setting

We conduct all experiments on a machine equipped with an AMD EPYC (Milan) 7713P 2.0Ghz with 64 CPU cores, 256 GB of DRAM, one NVIDIA A100 80 GB GPU, and a 4 TB Samsung SmartSSD. The system runs Ubuntu 20.04, PyTorch 2.3.1, DGL v2.3.0, and CUDA 12.1.

We use four real-world and synthetic graph datasets of varying sizes, shown in Table 1. The size columns correspond to the edge list only, in raw text and binary format, as node features are not used in sampling. Unless otherwise stated, our default configuration uses a GraphSAGE model with 3 layers, fanout of {20, 15, 10}, a mini-batch size of 1024, and we run RingSampler with 64 threads. `io_uring` is set to completion polling mode and the ring size is 512.

Baselines. We compare RingSampler with baselines from all categories of GNN sampling approaches (cf.§ 2.2): MariusGNN [30], smartSSD [29], DGL, as an in-memory baseline, and gSampler [9]. `DGL-CPU` refers to a configuration that performs sampling on the CPU with graph data in main memory;

DGL-UVA and gSampler-UVA store data in CPU memory and use Unified Virtual Addressing (UVA) to transfer data to the GPU for sampling; and finally, DGL-GPU and gSampler-GPU store and sample the graph in GPU memory.

4.2 Overall sampling performance

In the first experiment, we compare RingSampler's performance to that of baselines, across all datasets of Table 1. We measure the execution time of the sampling phase per epoch of training and plot average results across five epochs. Figure 4 shows the results, where *OOM* signifies that the experiment failed with an out-of-memory error.

We observe that only RingSampler and SmartSDD can successfully complete the sampling computation on the large graphs. Although Marius is designed to handle larger-than-memory graphs, it fails on these datasets with an out-of-memory error encountered during its pre-processing phase. The FPGA-based SmartSSD system suffers from significant overhead caused by transferring data from the SSD to FPGA memory. Moreover, due to the limited computational power of the FPGA compared to the CPU, its sampling performance is 30× to 60× lower than that of RingSampler.

For small graphs, such as ogbn-papers100M and Friendster, DGL-GPU and gSampler-GPU outperform all other systems, as they can compute sampling entirely in the GPU. Interestingly, though, RingSampler's performance is competitive with that of DGL-GPU, while it is significantly faster than all in-memory DGL sampling approaches.

These results demonstrate that RingSampler effectively utilizes CPU and SSD resources to achieve superior performance on very large graphs, without relying on specialized hardware or GPU acceleration. Further, RingSampler performs as well as in-memory solutions on small graphs.

4.3 Comparison with out-of-core systems

In the next set of experiments, we take a closer look at the performance of sampling approaches that support larger-than-memory graphs. We first evaluate systems under memory constraints and then investigate how sampling performance is affected as we increase the GNN layers. We use the ogbn-papers dataset for all experiments in this section.

Performance under memory constraints. In this experiment, we evaluate the performance of sampling systems under memory constraints. In particular, we use cgroup to limit the available memory during the experiment and measure sampling duration per epoch. Figure 5 shows the results.

Overall, RingSampler outperforms Marius by up to 18.5× and the SmartSSD-based sampling approach by up to 53×. With only 4 GB of memory, RingSampler completes one sampling epoch of a billion-edge graph in 20.22s. Additionally, RingSampler is the only system that can successfully sample the graph under a 4 GB limit. This is because RingSampler

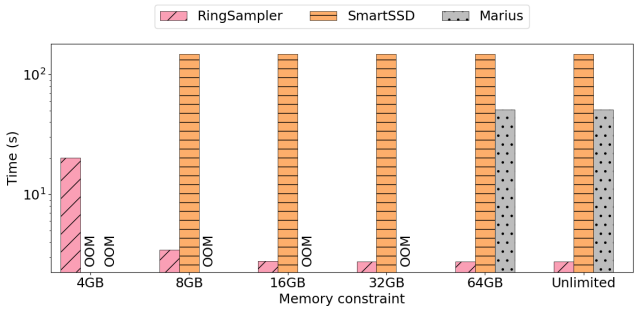


Figure 5: Sampling performance of out-of-core systems under memory constraints (ogbn-papers)

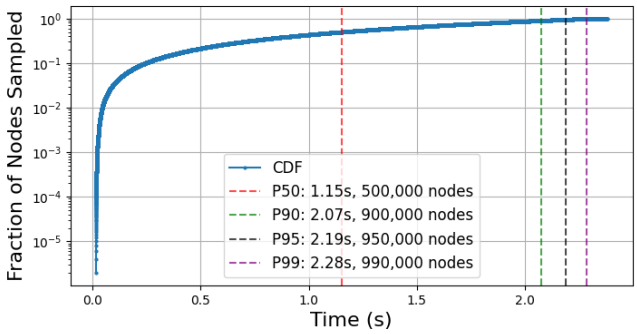


Figure 6: Latency CDF of per-request inference sampling in RingSampler, on a workload of 1 million target nodes

maintains only essential metadata in memory, such as the target index, offset index, and temporary storage for sampled neighbors. The space complexity of these auxilliary structures depends only on the number of nodes and is independent of the number of edges. For example, RingSampler requires roughly the same amount of memory to perform sampling on the Sythetic dataset, too, despite it having 5× more edges than ogbn-papers. In contrast, the SmartSSD approach requires at least 8 GB to store its host-side in-memory data structures. Marius has higher memory requirements than the other systems, as it uses in-memory partitions for both sampling and feature retrieval. To reduce I/O, Marius partitions graph data and loads selected partitions into memory for sampling. As a result, it provides a tunable trade-off between memory usage, sampling performance, and sampling quality. Using fewer partitions improves performance by reusing memory-resident partitions, but this comes at the cost of reduced sampling randomness, which can affect training accuracy [22].

4.4 On-demand sampling performance

In this section, we evaluate the performance of RingSampler in the context of GNN inference and demonstrate its potential

for real-time, on-demand sampling. To simulate a streaming inference scenario, we select 1 million target nodes from the `ogbn-papers` dataset. We set the mini-batch size to 1, simulating a scenario where individual sampling requests arrive to the system by concurrent clients. We keep all other configuration parameters to their default values.

To assess latency and throughput, we log the timestamps at which each node's sampling is completed. Figure 6 plots a latency CDF of the sampling requests. We observe that RingSampler maintains low and predictable latency even under sustained load. For example, 50% of sampling requests complete within 1.15s, and 90% within 2.07s. The narrow gap between the median and tail latencies demonstrates RingSampler 's ability to handle real-time sampling requests efficiently and consistently, making it well-suited for latency-sensitive inference tasks on large-scale graphs. However, a smart caching strategy would be needed to further improve responsiveness, making RingSampler fully inference-ready.

5 DISCUSSION

Rationale for using io_uring. We chose `io_uring` over alternatives such as `libaio` [8] and SPDK [34] based on insights from a recent systematic study [7], which comprehensively evaluates all three APIs. `io_uring` offers superior support for batched and asynchronous I/O via its ring-based interface, improving efficiency over `libaio`. Although SPDK offers better performance, it relies on user-space drivers and specialized configurations, limiting portability. In contrast, `io_uring` strikes the best balance among performance, portability, and ease of use—making it ideal for our system.

End-to-end implementation. RingSampler can be seamlessly integrated into existing GNN training frameworks, such as DGL, with minimal modifications. DGL uses a DataLoader to perform sampling and generate subgraphs, which are then passed to the GPU for feature retrieval and model training. To integrate RingSampler, we can implement a custom DataLoader that invokes our CPU-based sampler to prefetch subgraphs asynchronously and yield them as they become ready. These sampled subgraphs can then be passed directly to DGL's GPU-accelerated feature retrieval pipeline. This design enables an efficient end-to-end system that decouples sampling and feature access across CPU and GPU, allowing both to operate in parallel.

Limitations. RingSampler currently supports only node-wise GNN sampling, but we are planning to extend it to layer-wise sampling [1, 40] too. In addition, the full potential of `io_uring` has yet to be fully leveraged. We plan to integrate features such as kernel-side polling mode, which could further reduce I/O latency in future versions of RingSampler. Our approach can also be combined with other on-disk sampling techniques, such as in-situ sampling, to enable heterogeneous execution that leverages both CPU and SSD compute capabilities, enhancing overall system efficiency.

6 CONCLUSION

We presented RingSampler, a novel CPU-based sampling system built on `io_uring`, that addresses out-of-core training challenges in large-scale GNNs. RingSampler minimizes unnecessary data movement from disk to CPU memory and exhibits excellent sampling performance through a parallel and asynchronous design. Unlike prior approaches that require loading graph data into memory for sampling, RingSampler is the first to combine disk and CPU by storing graph data on disk and using `io_uring` for efficient random access. This enables in-memory-like sampling while supporting graphs larger than memory.

7 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This work was supported by the National Science Foundation under Grant No. 2237193.

REFERENCES

[1] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rytstxWAW

[2] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, 942–950. https://proceedings.mlr.press/v80/chen18p.html

[3] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *International Conference on Machine Learning*. PMLR, 942–950.

[4] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 257–266.

[5] Weilin Cong, Rana Forsati, Mahmut Kandemir, and Mehrdad Mahdavi. 2020. Minimal variance sampling with provable guarantees for fast training of graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1393–1403.

[6] Hanjun Dai, Zornitsa Kozareva, Bo Dai, Alex Smola, and Le Song. 2018. Learning steady-states of iterative algorithms over graphs. In *International conference on machine learning*. PMLR, 1106–1114.

[7] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '22)*. Association for Computing Machinery, New York, NY, USA, 120–127. https://doi.org/10.1145/3534056.3534945

[8] Daniel Ehrenberg. [n.d.]. Littledan/Linux-Aio: How to use the linux AIO feature. https://github.com/littledan/linux-aio?tab=readme-ov-

file

[9] Ping Gong, Renjie Liu, Zunyao Mao, Zhenkun Cai, Xiao Yan, Cheng Li, Minjie Wang, and Zhuozhao Li. 2023. gSampler: General and Efficient GPU-based Graph Sampling for Graph Learning. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 562–578. https://doi.org/10.1145/3600006.3613168

[10] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *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. https://proceedings.neurips.cc/paper_files/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf

[11] William L. Hamilton, Rex Ying, and Jure Leskovec. 2018. Inductive Representation Learning on Large Graphs. arXiv:cs.SI/1706.02216 https://arxiv.org/abs/1706.02216

[12] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[13] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling Towards Fast Graph Representation Learning. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/01eee509ee2f68dc6014898c309e86bf-Paper.pdf

[14] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling Towards Fast Graph Representation Learning. 4558–4567.

[15] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/3447786.3456244

[16] Qisheng Jiang, Lei Jia, and Chundong Wang. 2024. GNNDrive: Reducing Memory Contention and I/O Congestion for Disk-based GNN Training. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 650–659. https://doi.org/10.1145/3673038.3673063

[17] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao B. Schardl, Charles E. Leiserson, and Jie Chen. 2022. Accelerating Training and Inference of Graph Neural Networks with Fast Sampling and Pipelining. arXiv:cs.LG/2110.08450 https://arxiv.org/abs/2110.08450

[18] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. arXiv:cs.LG/1609.04836 https://arxiv.org/abs/1609.04836

[19] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. arXiv:cs.LG/1609.02907 https://arxiv.org/abs/1609.02907

[20] Yunjae Lee, Jinha Chung, and Minsoo Rhu. 2022. SmartSAGE: training large-scale graph neural networks using in-storage processing architectures. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 932–945. https://doi.org/10.1145/3470496.3527391

[21] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[22] Renjie Liu, Yichuan Wang, Xiao Yan, Haitian Jiang, Zhenkun Cai, Minjie Wang, Bo Tang, and Jinyang Li. 2025. DiskGNN: Bridging I/O Efficiency and Model Accuracy for Out-of-Core GNN Training. *Proc. ACM Manag. Data* 3, 1, Article 34 (Feb. 2025), 27 pages. https://doi.org/10.1145/3709738

[23] Fuping Niu, Jianhui Yue, Jiangqiu Shen, Xiaofei Liao, and Hai Jin. 2024. FlashGNN: An In-SSD Accelerator for GNN Training. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 361–378. https://doi.org/10.1109/HPCA57654.2024.00035

[24] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-mei Hwu. 2024. Accelerating Sampling and Aggregation Operations in GNN Frameworks with GPU Initiated Direct Storage Accesses. *Proc. VLDB Endow.* 17, 6 (may 2024), 1227–1240. https://doi.org/10.14778/3648160.3648166

[25] Yeonhong Park, Sunhong Min, and Jae W. Lee. 2022. Ginex: SSD-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2626–2639. https://doi.org/10.14778/3551793.3551819

[26] RapidsAtHKUST. 2019. Graph500KroneckerGraphGenerator. https://github.com/RapidsAtHKUST/Graph500KroneckerGraphGenerator.

[27] Yahoo Research. 2002. WebScope Graph and Social Data. https://webscope.sandbox.yahoo.com/catalog.php?datatype=g. Last access: March 2025.

[28] Marco Serafini and Hui Guan. 2021. Scalable Graph Neural Network Training: The Case for Sampling. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 68–76.

[29] Yuhang Song, Po Hao Chen, Yuchen Lu, Naima Abrar, and Vasiliki Kalavri. 2024. In situ neighborhood sampling for large-scale GNN training. In *Proceedings of the 20th International Workshop on Data Management on New Hardware* (Santiago, AA, Chile) *(DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, Article 11, 5 pages. https://doi.org/10.1145/3662010.3663443

[30] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2023. MariusGNN: Resource-Efficient Out-of-Core Training of Graph Neural Networks. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 144–161. https://doi.org/10.1145/3552326.3567501

[31] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[32] Yuyue Wang, Xiurui Pan, Yuda An, Jie Zhang, and Glenn Reinman. 2024. BeaconGNN: Large-Scale GNN Acceleration with Out-of-Order Streaming In-Storage Computing. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 330–344. https://doi.org/10.1109/HPCA57654.2024.00033

[33] D. Randall Wilson and Tony R. Martinez. 2003. The general inefficiency of batch training for gradient descent learning. *Neural Netw.* 16, 10 (dec 2003), 1429–1451. https://doi.org/10.1016/S0893-6080(03)00138-2

[34] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 154–161. https://doi.org/10.1109/CloudCom.2017.14

[35] Hongbo Yin, Yingxia Shao, Xupeng Miao, Yawen Li, and Bin Cui. 2022. Scalable Graph Sampling on GPUs with Compressed Graph *(CIKM '22)*. Association for Computing Machinery, New York, NY, USA, 2383–2392. https://doi.org/10.1145/3511808.3557443

[36] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural

Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 974–983. https://doi.org/10.1145/3219819.3219890

[37] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. arXiv:cs.LG/1802.09691 https://arxiv.org/abs/1802.09691

[38] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81. https://doi.org/10.1016/j.aiopen.2021.01.001

[39] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-dependent importance sampling for training deep and large graph convolutional networks. *Advances in neural information processing systems* 32 (2019).

[40] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-Dependent Importance Sampling for Training Deep and Large Graph Convolutional Networks. arXiv:cs.LG/1911.07323 https://arxiv.org/abs/1911.07323

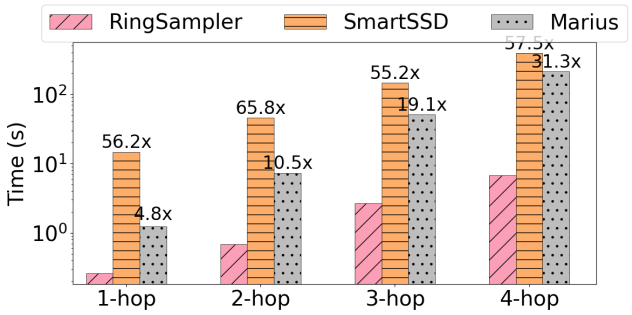


Figure 7: Sampling performance of out-of-core systems as the number of GNN layers increases (ogbn-papers)

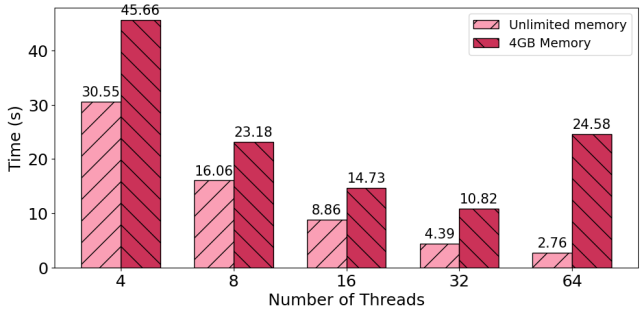


Figure 8: Scalability of RingSampler sampling over one epoch with varying number of threads

A ADDITIONAL EXPERIMENTAL RESULTS

A.1 Effect of sampling layers

We also evaluate how sampling performance is affected, as we increase the number of GNN layers, without any memory restrictions. In the GraphSAGE algorithm, the number of sampled neighbors increases exponentially with each additional layer. To assess scalability, we run experiments with different fanout configurations: [20], [20, 15], [20, 15, 10], and [20, 15, 10, 5], corresponding to 1-hop through 4-hop sampling.

In Figure 7, RingSampler consistently outperforms `SmartSSD` and `Marius`, presenting both lower overall sampling time, and a slower growth rate compared to the other disk-based systems. Specifically, RingSampler achieves over $55\times$ speedup over `SmartSSD` across all four hops, and outperforms `Marius` by an increasingly wider margin—from $4.8\times$ at 1-hop to $31.3\times$ at 4-hop. While `SmartSSD` exhibits a growth rate similar to RingSampler as the number of layers increases, its baseline performance remains substantially lower. In contrast, `Marius` suffers from a steep rise in sampling time as the number of layers grows. These results demonstrate that RingSampler not only delivers the best performance across all layer depths but also offers superior scalability, making it adaptable to varying sampling requirements.

A.2 RingSampler scalability

To evaluate the scalability and effectiveness of RingSampler 's multi-threaded design, we measure sampling performance per epoch while varying the number of threads, keeping all other configurations at their default values. As shown in Figure 8, when memory is not constrained, the sampling time decreases almost linearly with the number of threads, up to the maximum number of available CPU cores. This indicates that RingSampler efficiently utilizes multi-core CPU resources, with minimal contention or synchronization overhead.

Interestingly, when memory is constrained to 4GB (the minimum required for RingSampler to run with 64 threads), we observe that performance actually peaks at 32 threads. This is because memory usage in RingSampler scales with the number of threads. At 32 threads, there is still sufficient memory available to cache neighbor data, reducing I/O overhead. However, with 64 threads, nearly all available memory is consumed by the two index structures and the three workspaces, leaving little room for caching and resulting in more frequent disk reads.

This result also indicates that the minimum memory requirement of RingSampler can be further reduced when using fewer threads, without sacrificing performance. These findings highlight RingSampler 's flexibility and scalability, making it a practical solution for resource-constrained environments as well as large-scale graph sampling tasks.