In situ neighborhood sampling for large-scale GNN training

Yuhang Song
Boston University
yuhangs@bu.edu

Po Hao Chen
Brown University
pch@brown.edu

Yuchen Lu
Boston University
luyc@bu.edu

Naima Abrar
Boston University
anaima@bu.edu

Vasiliki Kalavri
Boston University
vkalavri@bu.edu

ABSTRACT

Graph Neural Network (GNN) training algorithms commonly perform neighborhood sampling to construct fixed-size mini-batches for weight aggregation on GPUs. State-of-the-art disk-based GNN frameworks compute sampling on the CPU, transferring edge partitions from disk to memory for every mini-batch. We argue that this design incurs significant waste of PCIe bandwidth, as entire neighborhoods are transferred to main memory only to be discarded after sampling. In this paper, we make the first step towards an inherently different approach that harnesses near-storage compute technology to achieve efficient large-scale GNN training. We target a single machine with one or more SmartSSD devices and develop a high-throughput, epoch-wide sampling FPGA kernel that enables pipelining across epochs. When compared to a baseline random-access sampling kernel, our solution achieves up to 4.26× lower sampling time per epoch.

CCS CONCEPTS

• **Computing methodologies** → *Machine learning*; • **Hardware** → *Emerging technologies.*

KEYWORDS

graph neural networks; GNN training; SmartSSD

ACM Reference Format:
Yuhang Song, Po Hao Chen, Yuchen Lu, Naima Abrar, and Vasiliki Kalavri. 2024. In situ neighborhood sampling for large-scale GNN training. In *20th International Workshop on Data Management on New Hardware (DaMoN '24), June 10, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3662010.3663443

1 INTRODUCTION

Graph Neural Networks (GNNs) have recently emerged as a powerful approach to performing machine learning tasks on graph-structured data. GNNs have demonstrated excellent performance on a variety of prediction and classification scenarios, such as recommendation, fraud detection, and code summarization [10, 13, 19, 29, 31, 33]. In these application domains, graphs are often massive, consisting of billions of edges and rich vertex attributes [4, 6, 7]. As a result, the working set size required for large-scale GNN training

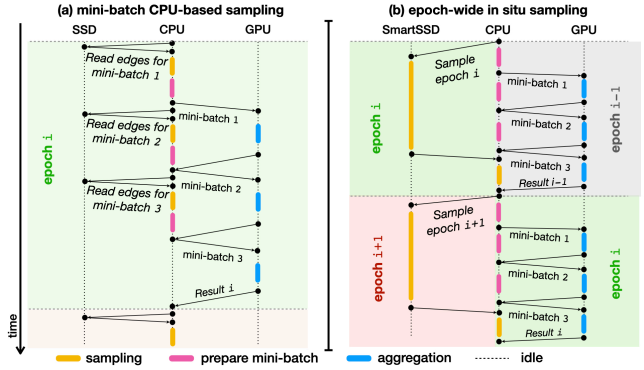


Figure 1: High-level overview of our proposed epoch-wide in situ sampling. Existing disk-based GNN training frameworks perform per mini-batch sampling on the CPU. This approach incurs multiple I/O operations per epoch to load edges in memory. In contrast, we propose offloading the sampling stage to an FPGA-equipped SSD. Our design enables overlapping the sampling computation for epoch i+1 with the mini-batch aggregation for epoch i.

may exceed the capacity of a single machine's main memory. To address this scalability challenge, three approaches have been proposed. Distributed in-memory GNN systems [9, 27, 28, 32, 34, 35] partition the graph across multiple machines that communicate over the network to perform training. On the other hand, scale-up designs target high-end servers with multiple GPUs [18, 30]. Finally, disk-based GNN systems split the graph into partitions that are gradually transferred to CPU memory during training [21, 26].

To make GNNs friendly to GPU architectures, many training algorithms employ *neighborhood sampling* to construct fixed-size *mini-batches* for weight aggregation [24]. State-of-the-art disk-based GNN frameworks [21, 26] perform sampling on the CPU, as shown in Figure 1(a). Edge partitions are transferred from disk to memory for each mini-batch. We argue that this design (i) incurs redundant I/O, as *entire neighborhoods are loaded in memory only to be discarded after sampling*, and (ii) may lead to low GPU utilization, as the CPU is busy with sampling and I/O.

To quantify the I/O redundancy of CPU-based sampling when edge partitions are loaded from secondary storage, we ran experiments with GraphSAGE models trained on the Papers100M [3] and Yahoo [5] datasets. We measured the average sample size per epoch and we found that only between 0.6% to 18% of the total edges are retained during sampling. Thus, repeatedly transferring all edges

DaMoN '24, June 10, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0667-7/24/06
https://doi.org/10.1145/3662010.3663443

to main memory incurs a significant waste of PCIe bandwidth. Additionally, we monitored the GPU utilization of a state-of-the-art disk-based GNN system [26] for five epochs of a 3-layer model. Indeed, we observed that that the GPU is underutilized (close to 0%) during 60% of the training duration.

Our proposal: Epoch-wide in situ neighborhood sampling. In this paper, we make the first step towards an inherently different approach that harnesses near-storage compute technology to achieve efficient large-scale GNN training. In particular, we target a single machine with one or more SmartSSD [16] devices. Our key insight is to move sampling closer to the storage by leveraging the platform's onboard FPGA to effectively alleviate the bottleneck on the data path to the host. To this end, we develop a high-throughput sampling FPGA kernel that enables pipelining *across epochs*.

Figure 1(b) shows how our design can exploit the intra-device PCIe switch to overlap sampling on the SmartSSD with training on the GPU. While the CPU and GPU are occupied with mini-batch preparation and aggregation for epoch `i`, the SmartSSD can perform sampling on epoch `i+1`. We believe that in-situ sampling will result in more efficient end-to-end training by reducing the amount of I/O and more effectively saturating the GPU resources.

Interestingly, our preliminary results (§ 3) contradict a previous claim that the SmartSSD architecture is unsuitable for this workload [17]. Indeed, we show that if the sampling kernel is implemented naively, the SSD → FPGA transfer time dominates. Our design effectively alleviates this bottleneck thanks to the key observation that *sampling computations across mini-batches (and epochs) have no data dependencies*. The only data dependency occurs between layers of the same epoch. As a result, we can perform sampling for a layer of an entire epoch at once and avoid triggering the FPGA kernel per mini-batch. When compared to the baseline random-access sampler [17], our epoch-wide sampler achieves up to 4.26× lower end-to-end sampling time.

We have made our code and experiments publicly available[1].

2 IN SITU NEIGHBORHOOD SAMPLING

In this section, we first provide the necessary background on GNN neighborhood sampling and our target hardware platform (§2.1). We then describe a baseline random-access sampling kernel and analyze its limitations (§2.2). Finally, we introduce our proposed epoch-wide kernel (§2.3) and contrast it with the baseline.

2.1 Preliminaries

Neighborhood sampling in GraphSAGE. In this work, we focus on mini-batch training of GraphSAGE GNN models [12]. GraphSAGE is an inductive model that learns representations of graph nodes as functions of their neighborhoods. Training proceeds in epochs that correspond to a full pass over the training nodes of the graph. At the beginning of an epoch, training nodes are divided into small batches. GraphSAGE follows a message-passing architecture, where a node iteratively gathers and aggregates information from its neighbors. A model may have one or more layers, reaching to the k-hop neighborhood of the training nodes. For each mini-batch, GraphSAGE samples multi-hop neighborhoods of the target nodes uniformly at random. It then retrieves the features of these neighbors and iteratively aggregates their features. At each layer, a target node's features are combined with those of the sampled neighbors to produce updated features, while the sampled neighbors serve as the target nodes for the next layer. As iterations proceed, the features capture the characteristics of the target nodes' neighborhoods. Finally, the training nodes of the mini-batch are sent to the GPU for training along with the aggregated feature vectors.

[1] https://github.com/CASP-Systems-BU/damon24-gnn-in-situ-sampling/

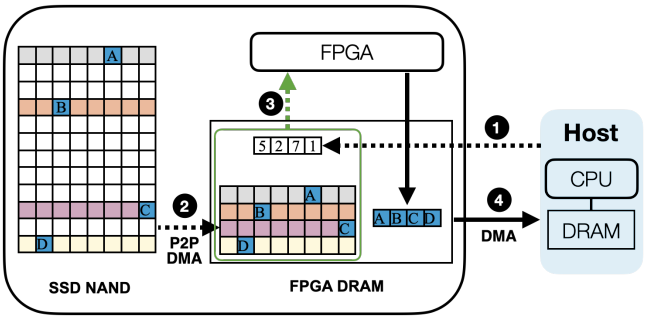


Figure 2: Illustration of the random-access sampler. (1) The host generates random offsets to sample from and sends them to the FPGA. (2) The corresponding blocks are transferred to the FPGA DRAM with a P2P read. (3) The FPGA extracts the neighbors from the blocks. (4) Finally, the FPGA sends the sampled results to the host using DMA.

Hardware platform. A SmartSSD [16] can offload computations to an embedded FPGA that has a direct communication channel to a NAND array and the host. The host can issue read/write requests to the SSD controller and computation requests to the FPGA. The SmartSSD has a PCIe 3.0 connection to the host and an on-board PCIe 3.0 switch that provides an internal data path between the SSD NAND and the FPGA DRAM.

2.2 Limitations of a random-access kernel

To highlight the challenges of developing a high-performing sampling kernel, we first analyze the limitations of the random-access solution introduced as a baseline in SmartSAGE [17]. This baseline kernel is a straightforward adaptation of mini-batch sampling on the SmartSSD. Figure 2 illustrates its operation. The main idea is to transfer the necessary neighbors from the SmartSSD NAND to FPGA DRAM and then to the host. For each mini-batch, the host generates a set of random offsets and triggers the FPGA kernel to collect the corresponding data. This naive approach incurs a significant data transfer overhead due to the fact that the minimum transfer block size for P2P communication from the SSD NAND to the FPGA memory is 512*B*. As a result, a separate read operation per neighbor (4*B*) results in 128× space amplification. Further, since sampling is performed per mini-batch, the kernel is called thousands of times per epoch, adding to the total overhead.

We implement two optimizations to improve the performance of this basic kernel. First, we implement batch-wise sample deduplication to reduce the amount of data transferred to the host. Since training nodes may have common neighbors, some nodes may appear repeatedly in the sample. We perform deduplication at the

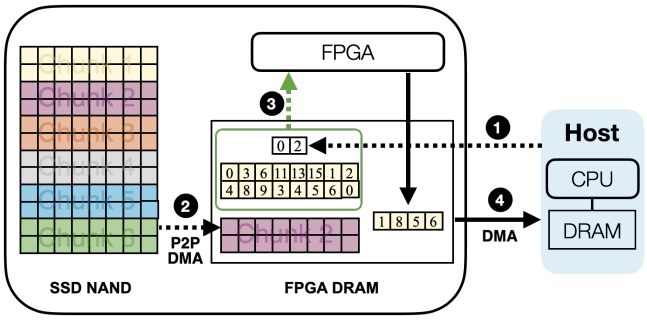


Figure 3: Illustration of the epoch-wide sampler. (1) The host sends the set of target nodes to the FPGA. (2) The corresponding chunks are read from the SSD NAND to the FPGA DRAM. (3) The FPGA performs random sampling and (4) uses DMA to copy the results back to the host.

host to only retain unique nodes as targets for the next layer. The second optimization addresses the case where multiple sampled nodes belong to same $512B$ block. While the naive implementation issues a separate read of that block for each node, we use buffer offsets to enable extracting multiple samples from the same block.

2.3 Epoch-wide sampling kernel

Our proposed design is motivated by two key observations: (i) the set of training nodes that serve as the target nodes for the first layer of sampling is fixed and known before the training starts, and (ii) computing a sample for mini-batch $j+1$ does not depend on the result of mini-batch j. The only data dependency occurs between layers of the same mini-batch. Therefore, we can sample neighbors of each layer for all mini-batches of an epoch in parallel. The set of training nodes are randomly shuffled before each epoch starts, thus, sampling for multiple epochs can also be performed independently. Next, we describe how we leverage these insights to achieve high-throughput sampling and avoid redundant I/O. Figure 3 provides an overview of the epoch-wide sampler.

Data organization. We perform a lightweight preprocessing step to organize edges into fixed-size sorted indexed chunks. As shown in Figure 4, each chunk has a *header* followed by the neighborhood data. The header includes the ID of the first node (`src`), the number of nodes in this chunk (`cnt`), and a list of pointers to the starting positions of each node's neighborhood (`offsets`). Chunks are optionally padded in the end to respect the $512B$ alignment requirement. Packing multiple neighborhoods in a single chunk allows drawing multiple samples per layer with a single read and reduces redundant data transfers. For the results we present in this paper, we set the chunk size to $512MB$, which leads to a $1.2\times$ space increase, in the worst-case. We leave the task of carefully tuning the chunk size as future work.

Host program. The host program orchestrates the kernel calls and reconstructs the mini-batch subgraphs after sampling. For each epoch, the host collects the target nodes of the current layer, sorts them, and looks up their corresponding chunk IDs. It then initiates the P2P read to load the necessary chunks to the FPGA and triggers the sampling kernel. Even though the host currently waits for all

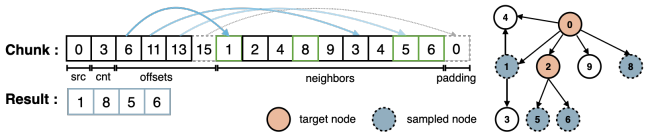


Figure 4: an example chunk that contains the neighbors of 3 nodes starting from node ID 0.

layers of an epoch to complete before triggering sampling for the next one, our methods are designed to operate in an asynchronous manner. This design allows the host to initiate sampling for the next epoch before the current one has finished, and to trigger sampling kernels on multiple SmartSSD devices concurrently.

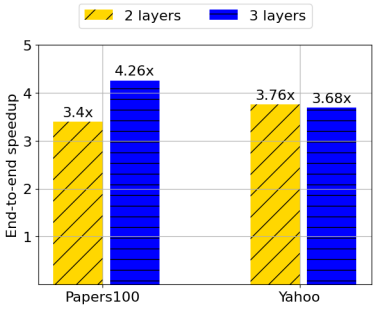
FPGA kernel. The epoch-wide kernel organizes the FPGA memory into (i) an input buffer where chunk data are loaded, (ii) a result buffer to store resulting samples, and (iii) an array that contains the sorted target node IDs for the current layer. For each chunk, the kernel iterates over the target nodes and computes their neighborhood offset boundaries. If the degree is less than the sample size, all neighbors are copied to the result buffer, and the remaining positions are filled with special dummy values. Otherwise, the kernel draws a uniform sample of offsets and extracts the neighbors from the resulting locations.

Example. We show an example chunk in Figure 4. The chunk contains the neighborhoods of 3 nodes, starting from node ID 0. This information is followed by a list of offsets (6, 11, 13, 15) and the neighbors. The zero padding in the end is necessary to make the chunk size a multiple of 16 in this example. Given target nodes 0 and 2 with a fanout of 2, the FPGA kernel can simultaneously draw samples for both nodes. By looking up the offsets in the header, it determines that the neighbors of target nodes 0 and 2 are stored in the ranges $[6, 11)$ and $[13, 15)$, respectively. It then draws random samples from these ranges according to the fanout and stores them in the result buffer.

Discussion. The random-access kernel performs the sampling task on the host by generating random offsets. In this case, the SmartSSD only acts as an accelerator to retrieve data from the SSD to the host. This approach does not fully exploit the FPGA's parallelization or the internal high-bandwidth PCIe switch. In contrast, the epoch-wide kernel computes the samples on the FPGA in parallel and fully utilizes the PCIe bandwidth by streaming the entire edge file, chunk by chunk, to the FPGA memory.

3 PRELIMINARY RESULTS

We use two GraphSAGE configurations for our experiments: a 2-layer model with fanout $\{25, 10\}$ and a 3-layer model with fanout $\{20, 15, 10\}$. Our SmartSSD system consists of a $4TB$ Samsung SSD connected to a Xilinx `KU15P Kintex UltraScale` FPGA through PCIe `Gen3 x4`. In all experiments, we set the unrolling factor to 32. The host CPU is an `AMD EPYC (Milan) 7713P 64C/128T 2.0Ghz` with $252GB$ of DRAM. We use two real-world graphs that comprise billions of edges. Their characteristics are shown in Table 1. The raw edge file size of the Papers100M graph is $25GB$ and the Yahoo edge file is $67GB$. Our preprocessing phase compresses the graph structure to $6.5GB$ and $30GB$ of data, respectively. While the edges



(a) Execution time of the epoch-wide over the random-access kernel

(b) Execution time breakdown of the sampling kernels for the 2-layer GNN

(c) Execution time breakdown of the sampling kernels for the 3-layer GNN

Figure 5: Preliminary evaluation results on two real-world graphs using GraphSAGE.

Dataset	Vertices	Edges	Training Nodes
Papers100M [3]	111×10^6	1.6×10^9	1.2×10^6
Yahoo [5]	1.4×10^9	6.6×10^9	1.4×10^6

Table 1: Graph datasets we use in our evaluation.

Dataset	Fanout	Sample Size (GB)	Savings (GB)
Papers100M	{20, 15, 10}	0.76	5.74
	{25, 10}	0.19	6.31
Yahoo	{20, 15, 10}	1.17	28.83
	{25, 10}	0.32	29.68

Table 2: Average sample size and bandwidth saving per epoch

of both datasets can comfortably fit in the main memory of our system, we emphasize that our goal is to overlap the sampling computation on the SmartSSD with mini-batch preparation on the CPU and weight aggregation on the GPU. End-to-end training will require access to the full graph, including the feature vectors, which can exceed the available memory. At the time of writing, evaluating our sampling kernel on even larger datasets is work in progress.

Bandwidth savings. To quantify the bandwidth savings of our solution, we perform training for five epochs. We measure the final sample size per epoch, computed as the sum of the sample sizes per layer. Table 2 shows the results. We report the average sample size per epoch and the corresponding bandwidth savings compared to existing disk-based approaches that perform sampling on the CPU. For the 3-layer model on Yahoo, the final sample size is only $1.17GB$, resulting in $28.83GB$ of bandwidth and memory savings per epoch. Considering that models need to be trained for $10 - 100$ epochs to achieve good accuracy, the total savings can be enormous.

Sampling performance. Next, we measure the average per-epoch speedup of the epoch-wide sampling kernel compared to the random-access kernel. Figure 5a shows the results. The epoch-wide kernel achieves between 3.4× and 4.26× better performance over the baseline. To understand the overheads that our proposed solution addresses, we plot the breakdown of the execution time in Figure 5b and Figure 5c. For both models, the random-access kernel spends most of its time on data transfer, which indicates that the SSD → FPGA communication becomes a bottleneck. In contrast, the epoch-wide sampler spends most of the execution time on the FPGA, performing the sampling computation.

4 RELATED WORK

Various recent works show that sampling can become a bottleneck during GNN training [11, 14, 20, 21]. A promising direction is to accelerate this phase of GNN training by performing the sampling computation on GPUs [11, 14]. However, for larger-than-memory graphs, transferring edges to the GPU without direct access technology, like GPUDirect [2], would require an expensive intermediate copy. The benefits of in-storage computing extend beyond GNNs to other large-scale ML and data analytics workloads [8, 15, 23, 25]. NeSSA [22] is a recent example that uses a SmartSSD-GPU system to accelerate DNN training.

5 CONCLUSION AND FUTURE WORK

In this paper, we show that near-storage computational devices offer an opportunity for high-performance and cost-effective large-scale GNN training. By pushing the sampling computation closer to the storage, we can reduce data transfer overheads and offload CPU cycles to enable potential overlapped execution. Our proposed epoch-wide kernel leverages the fact that sampling computations across mini-batches are independent to perform per-layer sampling of an entire epoch with a single kernel call. Our preliminary evaluation demonstrates significant benefits over the random-access baseline and bandwidth savings over existing disk-based approaches.

As an immediate next step, we plan to integrate our sampling kernel with DGL [1] and evaluate its benefits over end-to-end training. We will also extend the host program to enable triggering sampling kernels on multiple SmartSSD devices in parallel. Finally, we are working on implementing sampling kernels for alternative GNN algorithms and extend support to layer-wise sampling.

6 ACKNOWLEDGEMENTS

This work was partially supported by the National Science Foundation under Grant No. 2237193.

REFERENCES

[1] Deep graph library. https://www.dgl.ai. Last access: March 2024.

[2] Nvidia gpudirect storage. https://docs.nvidia.com/gpudirect-storage/configuration-guide/index.html. Last access: March 2024.

[3] Open graph benchmark. https://ogb.stanford.edu. Last access: March 2024.

[4] Size of the world wide web. https://www.worldwidewebsize.com. Last access: March 2024.

[5] Webscope graph and social data. https://webscope.sandbox.yahoo.com/catalog.php?datatype=g. Last access: March 2024.

[6] An introduction to brain networks. In Alex Fornito, Andrew Zalesky, and Edward T. Bullmore, editors, *Fundamentals of Brain Network Analysis*, pages 1–35. Academic Press, San Diego, 2016.

[7] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[8] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R Ganger. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 91–102, 2013.

[9] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 551–568, 2021.

[10] Hongyang Gao, Zhengyang Wang, and Shuiwang Ji. Large-scale learnable graph convolutional networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1416–1424, 2018.

[11] Ping Gong, Renjie Liu, Zunyao Mao, Zhenkun Cai, Xiao Yan, Cheng Li, Minjie Wang, and Zhuozhao Li. gsampler: General and efficient gpu-based graph sampling for graph learning. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 562–578, New York, NY, USA, 2023. Association for Computing Machinery.

[12] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[13] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive sampling towards fast graph representation learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 4558–4567. 2018.

[14] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating graph sampling for graph machine learning using gpus. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 311–326, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Kimberly Keeton, David A Patterson, and Joseph M Hellerstein. A case for intelligent disks (idisks). *Acm Sigmod Record*, 27(3):42–52, 1998.

[16] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. Smartssd: Fpga accelerated near-storage data analytics on ssd. *IEEE Computer architecture letters*, 19(2):110–113, 2020.

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

[18] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 401–415, 2020.

[19] Yao Ma, Suhang Wang, Charu C Aggarwal, and Jiliang Tang. Graph convolutional networks with eigenpooling. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 723–731, 2019.

[20] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-mei Hwu. Accelerating sampling and aggregation operations in gnn frameworks with gpu initiated direct storage accesses. *arXiv preprint arXiv:2306.16384*, 2023.

[21] Yeonhong Park, Sunhong Min, and Jae W Lee. Ginex: Ssd-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching. *arXiv preprint arXiv:2208.09151*, 2022.

[22] Neha Prakriya, Yu Yang, Baharan Mirzasoleiman, Cho-Jui Hsieh, and Jason Cong. Nessa: Near-storage data selection for accelerated machine learning training. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '23, page 8–15, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Erik Reidel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. 1998.

[24] Marco Serafini and Hui Guan. Scalable graph neural network training: The case for sampling. *ACM SIGOPS Operating Systems Review*, 55(1):68–76, 2021.

[25] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 119–132, 2013.

[26] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. Mariusgnn: Resource-efficient out-of-core training of graph neural networks. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 144–161, 2023.

[27] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. Flexgraph: a flexible and efficient distributed framework for gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 67–82, 2021.

[28] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. Neutronstar: Distributed gnn training with hybrid dependency management. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1301–1315, 2022.

[29] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[30] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: a factored system for sample-based gnn training over gpus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 417–434, 2022.

[31] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pages 974–983, New York, NY, USA, 2018. ACM.

[32] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. Agl: A scalable system for industrial-purpose graph machine learning. *Proceedings of the VLDB Endowment*, 13(12).

[33] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[34] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. Bytegnn: Efficient graph neural network training at large scale. *VLDB*, 15(6):1228–1242, 2022.

[35] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4582–4591, 2022.