Efficient Systems for Large-Scale Graph Representation Learning

by

Tianhao Huang

B.Eng., Tsinghua University, 2018
S.M., Massachusetts Institute of Technology, 2021

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by: Tianhao Huang
Department of Electrical Engineering and Computer Science
May 16, 2025

Certified by: Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

THESIS COMMITTEE

Thesis Supervisor

Srinivas Devadas
Edwin Sibley Webster Professor of Electrical Engineering and Computer Science
Department of Electrical Engineering and Computer Science, MIT

Thesis Readers

Julian Shun
Associate Professor of Electrical Engineering and Computer Science
Department of Electrical Engineering and Computer Science, MIT

Tushar Krishna
Associate Professor of Electrical and Computer Engineering
School of Electrical and Computer Engineering, Georgia Tech

Efficient Systems for Large-Scale Graph Representation Learning

by

Tianhao Huang

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2025 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Graph representation learning has gained significant traction in critical domains including finance, social networks, and transportation systems due to its successful application to graph-structured data. Graph neural networks (GNNs), which integrate the power of deep learning with graph structures, have emerged as the leading methods in this field, delivering superior performance across diverse graph related tasks. However, training graph neural networks on large-scale datasets encounters scalability challenges on current system architectures. First, the sparse, non-localized structures of real-world graphs lead to inefficiencies in data sampling and movement. This characteristic heavily stresses system input/output (I/O), particularly burdening the peripheral buses during the sampling phase of GNN training. Second, the suboptimal mapping of training procedure to GPU kernels leads to compute inefficiencies, including substantial kernel orchestration overhead and redundant computations.

Addressing these challenges requires a comprehensive, full-stack optimization approach that fully leverages hardware capabilities. This thesis presents two complementary works to achieve the goal. The first work, HANOI, unblocks the data loading bottleneck in out-of-core GNN training by co-designing the sampling algorithms to align with the hierarchical memory organization of commodity hardware. HANOI drastically reduces I/O traffic to external storage, delivering up to 4.2× speedup over strong baselines with negligible impacts on the model quality. Notably, HANOI is able to obtain competitive performance close to in-memory training with only a fraction of memory requirements. Building on this foundation, the second work, JOESTAR, introduces a unified framework for optimized GNN training on GPUs. JOESTAR adapts the multistage sampling approach from HANOI to in-memory training which frees CPUs from heavy data loading workloads. JOESTAR also identifies novel kernel fusion opportunities and formulates better execution schedules by jointly considering the sampling and compute stages. Combined with compiler infrastructure in PyTorch, JOESTAR achieves state-of-the-art GNN training throughputs for billion-edge graph datasets on a single GPU.

Thesis supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Seven years ago, a seemingly ordinary email notification lit up my phone, quietly marking one of the most consequential moments of my life. At the time, I had only heard others describe the PhD as a path that is both grinding and rewarding, but I had little sense of just how profoundly enriching and bittersweet the journey would be. Now, reaching its final stretch, I find myself resonating with those words in ways I couldn't have anticipated. More than anything, I've come to realize that pursuing a PhD, not for the academic credential itself, but for the learning, growth, and purpose it fosters, has been one of the greatest privileges of my life. I am truly grateful to everyone who helped make it possible.

Of all the people who have supported me, I owe my deepest gratitude first and foremost to my advisor, Arvind, for his unwavering support, patience, and trust throughout this journey. From the very beginning, he gave me an extraordinary degree of freedom to explore research topics that excited me, while also offering steady guidance whenever I needed it most. It took me longer than it should have to fully appreciate the generosity and rarity of such mentorship. I had hoped for the chance to continue learning from him, now that I feel more capable and clear about the direction I want to pursue. His unexpected passing is a profound loss. I will carry forward his values, vision, and the spirit of inquiry he exemplified with deep respect and gratitude. I am also sincerely thankful to his wife, Gita, for the warmth and hospitality she extended to us students. The home gatherings she hosted were not only moments of rest and joy, but also reflections of the kindness and openness that defined their shared life.

I extend my gratitude to my thesis committee, Professors Srini Devadas, Julian Shun and Tushar Krishna, for their support and guidance. Srini generously stepped in as my thesis supervisor when I was at a complete loss after Arvind's passing. Without hesitation, he helped organize the committee and provided steady check-ins that pulled me back from procrastination. His selfless support was instrumental in helping me see this journey through. I'm also thankful to Julian and Tushar for readily joining the committee, even on short notice. Julian had also served on my research qualification committee, and both he and Tushar offered valuable encouragement in the final stretch of my PhD. I would also like to thank Professor Leslie Kolodziejski for her unwavering kindness. She worked on my financial support and offered heartfelt check-ins after Arvind's passing. It allowed me to focus on finishing my thesis without the added stress of funding concerns.

Next, I would like to thank my mentors and collaborators: Xuhao Chen, Shuotao Xu, Thomas Bourgeat, Jie Chen (IBM Research), Jay Lorch (Microsoft Research), and Oscar

Pinto (Samsung). Working alongside Xuhao in the same office for the past four years was a constant source of inspiration. Our long debates over ideas, paper discussions, and shared frustrations and breakthroughs are memories I will always cherish. I wish him all the best in his pursuit of a faculty career. A special thank you to Shuotao, who brought me to Redmond for a summer internship. His generosity, openness, and steady support helped me through one of the most difficult periods of my life. Although I ultimately took a different career path, I hope we'll have the chance to work together again. And Thomas – I feel incredibly lucky to have crossed paths with him, and equally regretful not to have learned more from him before he left for EPFL. The conversations we shared remain some of the most intellectually invigorating and joyful moments of my time at MIT.

To my labmates at CSAIL and all members of the Computation Structures Group: Jiazheng Liu, Thomas Bourgeat, Chanwoo Chung, Shuotao Xu, Andy Wright, and Sizhuo Zhang. Thank you for the camaraderie, insightful discussions, shared meals, and spontaneous life chats that made the lab feel like home. Special thanks to Jiazheng Liu and Jianming Tong for being steadfast friends along the way. I would not have stayed remotely in shape if it weren't for Jiazheng's relentless enthusiasm for climbing walls, both literal and metaphorical.

To all my friends and old-timers: Samantha Ying, Ziyan He, Hiro and Tomo Ogasawara, Yujun Lin, Minseok Jung, Christos Zarkos, Shixin Song, Yuheng Yang, Han Cai, Yifan Yang, Muhua Xu, Peidong Wang, Yucan Wu, Zhixin Liao, Paul Mure, Yucheng Yang, Yulu Tang, Jilan Lin, Haotao Wang, Qing Xiao, Zheng Zhan, and Irene Wang – thank you. Whether it was late nights in Stata, lazy weekend brunches at Ashdown, exotic bites in Newbury, or getting lost in Seattle's wild trails, those memories are the ones I return to whenever I need strength and renewal.

Last but not least, to my parents, cousins, and all my family. Thank you for your constant love, encouragement, and belief in me, even when I doubted myself. Your quiet strength and steady support have carried me through the hardest moments of this journey. I could not have made it here without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are a versatile and powerful tool for modeling relationships between entities in various domains. Extracting valuable insights from graph data is of long-standing interest to both academic research and industry [1–3]. Traditionally, analytics over graph data are done through classic algorithms such as graph traversals, PageRank [4], subgraph pattern mining [2, 5], etc. Although these algorithms are well studied and have excellent explainability, they are rather limited in their predictive performance and hard to generalize because of their nature: they are non-learning based methods. For the last decade, graph representational learning has emerged as a popular field to address the difficulty of classic graph algorithms. Graph representation learning [6, 7] aims to learn low-dimensional node embeddings that encode the neighborhood structural information and other input features. The generated embeddings are then fed to task-specific decoders for diverse graph learning tasks, e.g., node classification, link prediction or graph prediction [6]. More recently, the combination of deep learning with graph data further sparked a new class of graph representation learning methods: graph neural networks (GNNs). GNNs [8–18] strongly outperform conventional encoding methods [19–22] in graph representation learning for both accuracy and generality, thus enjoying wide adoption in real-world use cases, including social networks, recommendation systems, traffic modeling, weather forecast, and more [23–28].

Similar to machine learning models in other domains, GNNs are trained iteratively by making predictions on the input graph data (i.e., forward passes) and updating model parameters with backpropagation [29] (i.e., backward passes). GNNs integrate the structural information into node embeddings through the *message passing* mechanism [8, 18]. In a message passing layer, input node embeddings are propagated along edges, aggregated in destination nodes and passed to the next layer for further transformation, usually by a Multi-Layer Perceptron (MLP) or another message passing layer. In the majority of GNN frameworks including this thesis, graph data are represented in sparse adjacency matrices

for graph structures and dense matrices for embeddings. Message passing layers are lowered to a common set of kernels between sparse and dense matrices [30, 31], the most notable of which being sparse dense matrix multiplication (SpMM) with different reduction operations (e.g., `MEAN`, `MAX`, or `SUM`) and sampled dense-dense matrix multiplication (SDDMM) with different pairwise operations (e.g., `ADD` or `MUL`). These kernels are well-studied topics on modern GPUs [32]. Thus, in common practice, the model computation part of GNN training is entirely offloaded to GPUs and enjoys significant performance speedups over CPUs.

As GNNs are being applied to real-world datasets, several system-level scalability challenges arise due to the unique characteristics of graph data and message passing layers. The first challenge emerges in graph sampling and I/O. To limit the memory footprint, Stochastic Gradient Descent (SGD) becomes essential in large-scale GNN training, which entails mini-batching and sub-sampling of input graph data on CPUs and then transferring the sampled subgraphs to GPUs for model computation. However, since many graph datasets are highly non-localized (the Small-World Phenomenon [1]), the neighborhood size of a node can grow *exponentially* with the number of hops. The data-intensive sampling step performed by CPUs and bus transfer to GPUs constitute a significant portion of overhead in the training pipeline. This bottleneck in sampling and I/O, summarized as "data loading" in the thesis, is further exacerbated by the trend of applying GNNs to increasingly larger graph datasets. For example, the largest datasets in leading graph learning benchmarks [33, 34] require 200GB–1TB of space when fully loaded, far exceeding the memory capacity of GPUs and even many CPU servers at the time of writing. This either renders training infeasible on single-node GNN frameworks due to out-of-memory (OOM) errors or makes it extremely slow as the system spends most time waiting for I/O from the external storage [35, 36].

Apart from the data loading bottleneck, sampling based GNN training also suffers from poor execution efficiency on GPUs. While key computation kernels such as general matrix multiplication (GEMM) and SpMM accelerate well on GPUs, the overall GPU utilization still remains suboptimal. The main cause of the under-utilization is attributed to the kernel invocation patterns in GNN sampling and model compute. On GPUs, sampling based GNN training systems tend to spawn a swarm of short-lived kernels with low execution latencies. The orchestration work of these fine-grained lightweight kernels (e.g., graph sampling, graph transpose) and frequent host synchronizations expose the kernel launch overhead directly to the training pipeline, leading to low GPU activity and thus poor resource utilization. These factors greatly limit GPU-centric GNN training from reaching its full potential.

In this thesis, we seek to address challenges above in a holistic approach through algorithm-system co-design and joint optimizations of GNN sampling and compute. In particular, to solve the most pressing data loading bottleneck, we leverage a novel *multistage sampling*

paradigm for GNN training. Multistage sampling separates conventional monolithic, single-stage sampling into multiple stages that fit well into the hierarchical memory organizations of machine learning hardware and promote active data reuse. We propose graph sampling methods with careful algorithm-system co-design, taking both hardware characteristics at each memory tier and model accuracy into account. Two systems with different hardware configurations are built to demonstrate the advantages of this approach. In the first system HANOI, we consider an out-of-core training setup where graph dataset sizes exceed even the host memory. HANOI directly works with the graph data stored externally on cheap but slow media such as solid state drives (SSDs) or hard disks. The key enabler for efficient data loading is the first-stage GNN sampling between external storage and host memory, which only samples coarse-grained fragments of graph structures and features in an I/O-friendly manner. Unlike prior works that achieve I/O efficiency at the cost of model accuracy [37], HANOI provides a robust remedy to minimize the impacts of rigid sampling on model quality. Enhanced with deep pipelining across multiple sampling and compute stages, HANOI is able to decouple I/O from the critical path completely.

The second system, JOESTAR, targets more common in-memory training scenarios. JOESTAR presents a unified framework to jointly optimize GNN sampling and compute on GPUs. JOESTAR utilizes multistage sampling to divide the heavyweight data loading between CPUs and GPUs, where the majority of sampling work is offloaded to more powerful GPUs. Thus, CPUs and PCIe buses are freed from the data loading bottleneck. JOESTAR differentiates from existing GNN compilers [38–41] in its novel integration of graph sampling and other structural operations within the model compilation process. This unified perspective of GNN compilation leads to discovery of new optimization opportunities, such as elimination of redundant graph structure operations interleaved with model computation and reduction of data movement through novel cross-stage operator fusions. Moreover, JOESTAR conducts profile-guided optimization to determine the best data formats and orders of expensive matrix multiplications. The inclusion of sampling, forward and backward passes makes cost estimation much more accurate than heuristic approaches in prior works [30, 40].

1.1 Thesis Contributions

This thesis systematically advances large-scale graph representation learning by developing efficient techniques for accessible hardware infrastructure. Through in-depth profiling, it identifies data loading as the most pressing performance bottleneck, followed by inefficient mapping of GNN training on GPUs, which represents a drastic shift from machine learning workloads in other domains. This thesis then demonstrates how to tackle the challenges

above with one novel sampling paradigm and two GNN training systems. In particular, the contributions of this thesis entail the following parts.

A multistage sampling perspective to GNN training:

1. First-principle performance analysis of multistage sampling compared to single-stage with caching, which clarifies the source of speedups.
2. Statistical analysis of model convergence behavior with multistage sampling, which identifies potential impacts on model accuracy.

An I/O-efficient out-of-core GNN system:

1. Design and implementation of HANOI, a fast and accurate out-of-core GNN training system which achieves excellent I/O efficiency.
2. Accuracy-aware aspects in the sampler design of HANOI to minimize loss of model accuracy from multistage sampling.
3. An end-to-end system evaluation on gigantic datasets that shows HANOI almost entirely hides the I/O bottleneck with negligible impacts on the model quality.

A compute-efficient GPU-centric GNN system:

1. JOESTAR, an in-memory system that leverages multistage sampling to make GNN training GPU-centric, even for datasets that exceeds the size of GPU memory.
2. A unified compilation framework for end-to-end GNN training on GPUs, including graph sampling, graph structural operations and GNN model computation.
3. Novel optimization passes that jointly consider all training stages for better runtime performance and a comprehensive evaluation that demonstrates state-of-the-art training performance of JOESTAR.

1.2 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 familiarizes readers with basic operations in the construction of graph neural networks followed by state-of-the-art practice in large-scale GNN training. Next, we highlight system-level challenges in further scaling up GNN over massive graph data.

Particularly, we summarize the data loading bottleneck and inefficient GPU runtime as the major causes of low compute utilization, which motivates the solutions in the following chapters.

- Chapter 3 discusses the fundamental idea of multistage sampling which is employed throughout the thesis to address the most severe data loading bottleneck in GNN training. In this chapter, we analyze the effects of multistage sampling in both I/O costs and model convergence. Several metrics are drawn from the analysis to guide the design of multistage samplers in real systems.

- Chapter 4 introduces out-of-core GNN training as a cost-efficient alternative to scale up GNN over massive graph data. We then describe the design of HANOI, an out-of-core GNN system which instantiates multistage sampling with hierarchical pipelining to break out of the data loading bottleneck. The algorithm-system co-design of first-stage samplers proposed by HANOI are explained in great detail, including the GNN-aware graph partitioner and hub node augmentation. They play a key role for HANOI to achieve a good balance of training efficiency and model quality. This chapter concludes with a comprehensive evaluation of HANOI including training throughputs, model performance as well as the sensitivity analysis under different memory and I/O constraints.

- Chapter 5 presents JOESTAR, the second GNN system in this thesis focusing on efficient GPU-centric in-memory training. JOESTAR has different instantiations of multistage sampling, which makes GNN training GPU-bound even for datasets beyond the capacity of GPU memory. Next, motivated by a detailed kernel profiling on GPUs, we describe the unified GNN compilation framework in JOESTAR, including intermediate representations (IRs) for key GNN operations, optimization passes and low-level kernel implementations. Towards the end of this chapter, we compare the training throughputs and resource utilization of JOESTAR with competitive in-memory training baselines.

- Finally, we conclude the thesis and briefly discuss future research directions in Chapter 6.

Chapter 2

Background and Motivations

2.1 GNN Preliminaries

This work mainly considers message-passing GNNs which are state-of-art methods designed for representation learning on graphs [6]. The graph input comes in the form of $G(\mathcal{V}, \mathcal{E}, \mathbf{H})$, where $\mathcal{V}$ and $\mathcal{E}$ are the node set and edge set of G with $\mathbf{H}$ as the input node feature embeddings attached with $\mathcal{V}$. GNNs use message passing to iteratively update the features and explicitly learn from the graph structure. GNNs also utilize the expressive power of neural networks in feature transformation. Output node embeddings generated by GNNs are considered to contain rich information of the neighborhood, which are used in various downstream tasks in a decoding manner. In this thesis, we are mainly concerned with training GNN models as encoders, since it constitutes a common backbone in graph learning.

Let $\sigma(\cdot)$ be a non-linear activation function, $\mathcal{N}(v)$ denote the node set of first-hop neighborhood of v in G and $\boldsymbol{h}_v^k$ be the intermediate embedding of v at the k-th hidden layer. Formally speaking, an L-layer GNN can be formulated as below ($0 \leq k < L$):

$$\boldsymbol{z}_v^{(k+1)} = f\big(\boldsymbol{h}_v^{(k)}, \texttt{AGG}\big\{g(\boldsymbol{h}_v^{(k)}, \boldsymbol{h}_u^{(k)}) | u \in \mathcal{N}(v)\big\}\big) \tag{2.1}$$

$$\boldsymbol{h}_v^{(k+1)} = \sigma(\boldsymbol{z}_v^{(k+1)}), \boldsymbol{h}_v^{(0)} = \mathbf{H}_v \tag{2.2}$$

where `AGG` is an operator for aggregating incoming neighbor features after message passing, f is a function (usually learnable, e.g., a linear layer) to update the aggregated features, and the optional g applies transformation to features before message passing. $\boldsymbol{z}_v^{(L)}$ is usually taken as the final encoded representation for node v, which is fed to task-specific decoders. Mainstream GNN architectures generally follow this framework but differ in the choices of g for message passing [9, 10, 17], the aggregation function `AGG` [9, 11, 12] and the feature

update function f. For example, in GraphSAGE [9] with the `MEAN` aggregator, f applies two independent linear transformations to the embedding of source node v and the aggregated embedding while g does nothing, so Eq. (2.1) is written as:

$$\boldsymbol{z}_v^{(k+1)} = \boldsymbol{h}_v^{(k)}\mathbf{W}_1 + \frac{1}{|\mathcal{N}(v)|}\sum_{u\in\mathcal{N}(v)} \boldsymbol{h}_u^{(k)}\mathbf{W}_2 \tag{2.3}$$

$\mathbf{W}_1, \mathbf{W}_2$ are two learnable weight matrices at Layer k. In GAT [10], function g employs the attention mechanism to obtain an attention coefficient for each (v, u) pair, which is later normalized as scalar weights in message passing:

$$e_{v,u} = \sigma(\boldsymbol{h}_v\mathbf{W}_{attn}{\mathbf{a_1}}^T + \boldsymbol{h}_u\mathbf{W}_{attn}{\mathbf{a_2}}^T) \tag{2.4}$$

Computationally, the graph structures are represented as sparse adjacency matrices. In this thesis, we mainly consider three formats to express the sparse matrices, namely the coordinate list (COO), compressed sparse row (CSR) and compressed sparse column (CSC) formats. Node and edge data attached to the graph are stored as dense tensors indexed by the assigned node or edge IDs. The message passing and aggregation steps are then expressed with matrix kernels such as GEMM, SpMM and SDDMM. For example, a GraphSAGE layer (Eq. (2.3)) is mapped to

$$\mathbf{Z} = \mathbf{H}\cdot\mathbf{W}_1 + \mathbf{D}^{-1}\mathbf{A}\cdot\mathbf{H}\cdot\mathbf{W}_2 \tag{2.5}$$

where $\mathbf{D}$, $\mathbf{A}$ are the degree matrix and adjacency matrix of G respectively, in the CSC format. Here $\mathbf{A}\cdot\mathbf{H}$ invokes SpMM. Correspondingly, the attention mechanism in GAT is mapped to

$$\begin{gathered}\boldsymbol{h}_1 = \mathbf{H}\mathbf{W}_{attn}{\mathbf{a_1}}^T, \boldsymbol{h}_2 = \mathbf{H}\mathbf{W}_{attn}{\mathbf{a_2}}^T \\ \mathbf{E} = (\boldsymbol{h}_1 \oplus \boldsymbol{h}_2^T)\odot\mathbb{I}[\mathbf{A}]\end{gathered} \tag{2.6}$$

where Eq. (2.6) invokes a generalized SDDMM using non-zeroes in the adjacency matrix $\mathbf{A}$ as a sparsification mask ($\oplus$ means addition is used as the pairwise operator in the generalized SDDMM). Likewise, the backward passes of GNN training are expressed in the same set of operations plus transposition of the sparse matrices. For SpMM ($\mathbf{X} = \mathbf{A}\mathbf{H}$), the gradients

are computed as

$$\frac{\partial L}{\partial \mathbf{H}} = \mathbf{A}^T \frac{\partial L}{\partial \mathbf{X}} \quad \text{(SpMM)} \tag{2.7}$$

$$\frac{\partial L}{\partial \mathbf{A}} = (\frac{\partial L}{\partial \mathbf{X}} \mathbf{H}^T) \odot \mathbb{I}[\mathbf{A}] \quad \text{(SDDMM)} \tag{2.8}$$

The gradients of vanilla SDDMM ($\mathbf{E} = (\mathbf{H}_1 \mathbf{H}_2^T) \odot \mathbb{I}[\mathbf{A}]$):

$$\frac{\partial L}{\partial \mathbf{H}_1} = \frac{\partial L}{\partial \mathbf{E}} \mathbf{H}_2 \quad \text{(SpMM)} \tag{2.9}$$

$$\frac{\partial L}{\partial \mathbf{H}_2} = \left(\frac{\partial L}{\partial \mathbf{E}}\right)^T \mathbf{H}_1 \quad \text{(SpMM)} \tag{2.10}$$

For a more exhaustive description of GNN-specific operators, we refer readers to prior works on GNN frameworks with automatic differentiation [30, 31]. Note that in the above discussions, we largely omit common operations in dense neural networks such as linear layers, non-linear activations, dropouts, batch normalization, etc. These layers, lowered to dense matrix and vector kernels, can also represent substantial computation overhead in GNN training.

2.2 Mini-Batch Training of GNNs

Traditionally, GNNs are trained in a full-batch fashion and accelerated on GPUs [8, 42, 43]. Full-batch training requires data including G, input feature $\mathbf{H}$, hidden features and model parameters to be resident on the GPU memory. Although GNN models are usually small compared to other deep neural networks, the input and intermediate feature data can take up significant storage space, making full-batch training on GPUs impractical as we consider realistic datasets with significantly larger $|\mathcal{V}|$ and $|\mathcal{E}|$. Thus, mini-batch training of GNNs [9] has become a standard approach for larger graph datasets. Therein, the host CPU regularly samples mini-batches of training data from G and $\mathbf{H}$, and then transfers them to the GPU through peripheral buses, while the GPU now only holds model states and performs model training on the sampled mini-batch. Mini-batch training greatly alleviates memory pressure on the GPU side since the sampled subgraphs are expected to be much smaller than G. Besides, it also speeds up model convergence through Stochastic Gradient Descent (SGD) compared to vanilla gradient descent in full-batch training [44].

However, different from other domains, sampling mini-batches in GNNs is trickier because nodes are interconnected with edges so cannot be taken independently. Based on Eq. (2.1), the output embedding of a node at the L-th layer depends on its L-hop neighborhood

which can grow exponentially with L on real-world graphs with low locality. This well-known problem of neighborhood explosion has motivated the proposal of various sampling techniques to reduce mini-batch sizes by sub-sampling the L-hop neighborhood of training example. Notable proposals include node-wise sampling [9, 45], layer-wise sampling [46, 47] and subgraph sampling [48–50]. This sub-sampling of graph structures essentially results in an approximation of the forward computation in GNNs. Among them, neighbor sampling (NS) [9] is the most widely-used sampling algorithm in large-scale graph datasets due to its robustness. It enforces upper bounds for the number of sampled neighbors per node at each layer to a fixed fan-out parameter, usually much lower than the maximum degree of the graph to restrict the sizes of sampled subgraphs. Therefore, the current practice of large-scale GNN training consists of multiple stages below (Fig. 2.1):

1. Graph sampling: extracting and sub-sampling L-hop neighbors from the selected mini-batch of training examples. Neighbor sampling (NS) is a common choice for the sampling algorithm.
2. Feature gathering: gathering required features of sampled subgraph from input features $\mathbf{H}$ on the host.
3. Bus transfer: transferring the mini-batch data from CPU memory to GPU memory through peripheral buses such as PCIe.
4. Model computation: GPU performs compute-intensive forward and backward passes to update GNN model parameters.

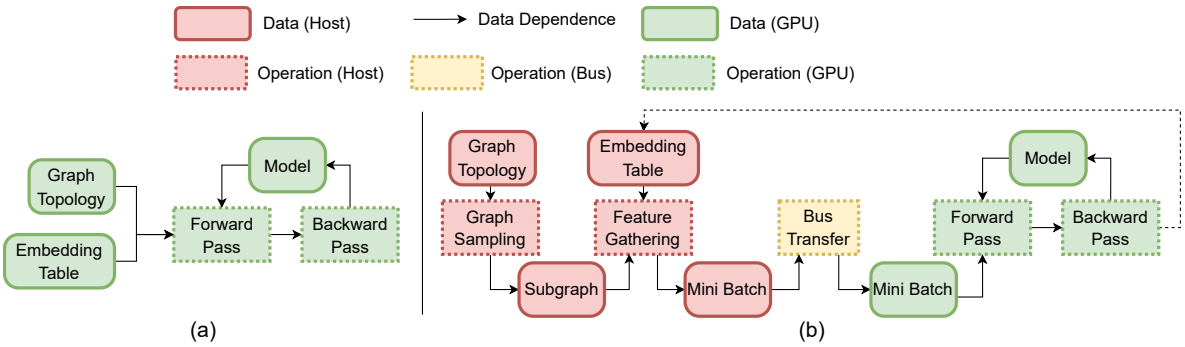


Figure 2.1: Full-batch (a) and mini-batch (b) training pipelines for GNNs

2.3 Challenges of Scaling up Mini-Batch GNN Training

There has been extensive research on accelerating sparse computation patterns of GNNs on GPUs [38–40, 42, 43, 51–54], CPUs [27, 55, 56] and even hardware accelerators [57–61].

However, most of them focus on full-batch GNN computation which is inherently difficult to scale to real-world large graphs. Mini-batch GNN training, on the other hand, demonstrates a drastically different workload profile from full-batch. It requires a fresh perspective to understand its performance characteristics and bottlenecks. In particular, the inefficiencies of mini-batch GNN training systems are two-fold.

2.3.1 The Data Loading Bottleneck

Compared to full-batch GNN computation, mini-batch GNN training systems are heavily stressed in the data loading step, i.e., graph sampling, feature gathering and bus transfer (Fig. 2.1). In existing systems, these stages are carried out on the host CPU due to GPU memory constraints. The main cause is the widening gap between bandwidths of CPUs, PCIe and GPUs, as shown in Fig. 2.2.. As performance scaling of CPUs and PCIe continues to lag behind GPUs for the past decade and foreseeable future, we expect current CPU-heavy data loading designs will continue to be the main performance obstacle to large-scale GNN training.

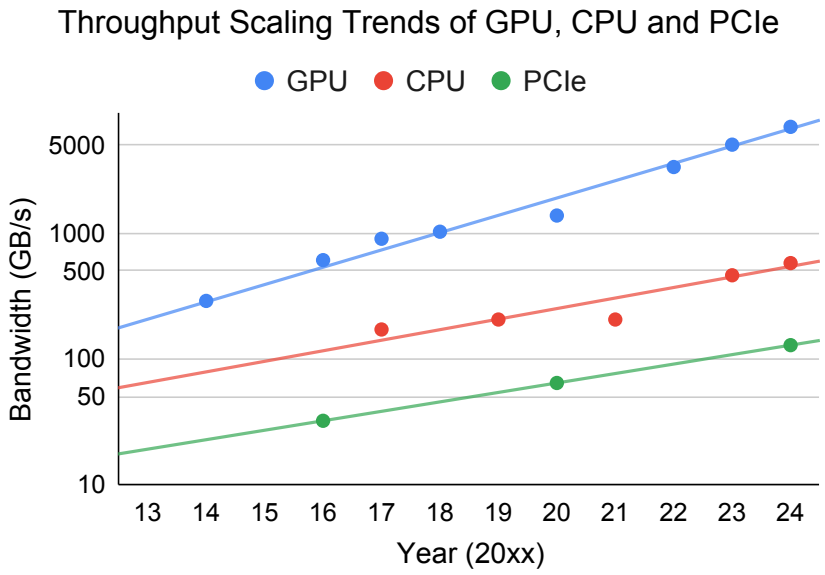


Figure 2.2: Memory throughput scaling trends of top- of-line GPU, CPU models and PCIe (x16) for the past decade.

The data loading bottleneck becomes more pressing as we consider the faster growth of datasets than the memory. One goal of this thesis is to push the boundary of mini-batch training paradigm *beyond the in-memory setting*. This aligns with the growing trend of applying GNNs to massive industry-scale [24–27, 62] graphs. Some benchmark datasets [33, 34] designed to mimic real-world applications have also expanded beyond the memory capacity

of a typical server machine. To handle data in such scales, people adopt distributed training or out-of-core training which utilizes the external storage as the active data source. Both methods are frequently bounded by I/O, either in the form of network or storage, as they are usually one or two orders of magnitude slower than the memory bandwidth.

We highlight these concerns by collecting the runtime distribution of PyG [31], perhaps the most popular GNN training framework, with results illustrated in Fig. 2.3. Neighbor sampling with a fanout of 10,10,10 and the GraphSAGE model with a hidden dimension of 256 are used for all experiments. For small datasets (`arxiv`, `products`, details of datasets shown later) that fit within host memory, while data preparation operations are relatively efficient, they still constitute a substantial portion of total runtime due to the inherent performance disparities between CPUs, PCIe and GPUs. As the GPU becomes more powerful in the future, we expect the CPU and PCIe bottlenecks to be even more significant. For larger datasets (`papers`, `mag240m-c`) that exceed host memory capacity, we employ memory mapping (`mmap`) to virtually extend the trainer's address space to prevent out-of-memory errors. However, this straightforward out-of-core solution introduces substantial overhead: the frequent page-in and page-out operations during graph sampling and feature gathering operations dominate the execution time, accounting for over 80% of the total training time. As a result, the I/O bottleneck on secondary storage leaves both CPU and GPU heavily underutilized.

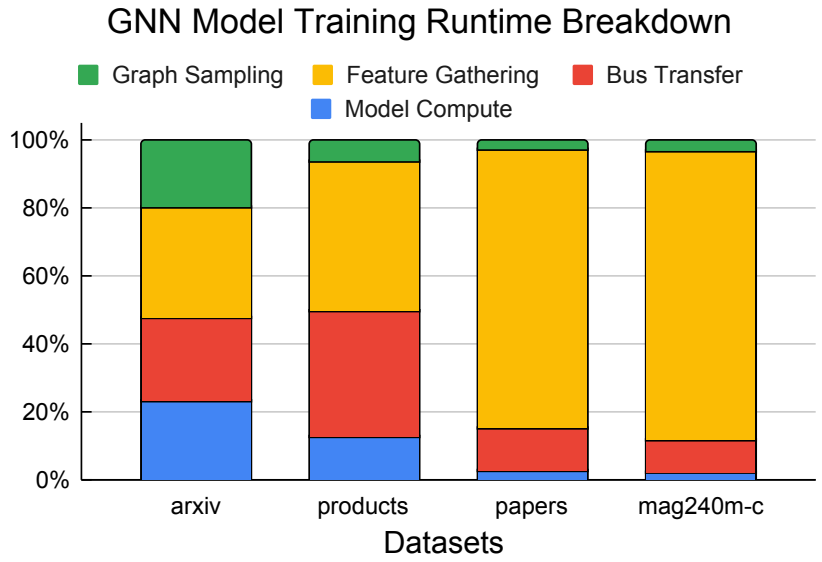


Figure 2.3: Training runtime breakdown for two small and two large datasets on a machine with 64GB host memory and one RTX 4090 GPU for model compute.

2.3.2 Low Compute Utilization from Inefficient GPU Runtime

Mini-batch GNN training is also prone to inefficient kernel mapping and orchestration on GPUs due to its unique input characteristics. Subgraphs from mini-batch sampling are highly dynamic data, as the shape and topology of them keep changing. This trait renders GNN kernel optimizations designed for full-graph computation largely ineffective. Frameworks that rely on computationally-expensive preprocessing [42, 43] are particularly ill-suited for handling dynamic mini-batch graphs. Moreover, reduced input data sizes from mini-batch sampling makes it difficult to fully utilize the massive parallelism provided by modern GPUs. Most kernels, as a result, are short-lived, exposing the kernel launching and host synchronization overheads. To demonstrate this, we profile the training process of a typical GCN [8] model with a batch size of 1000 for the `products` dataset. It turns out the GPU is busy for only less than 35% of time after excluding the data loading time. In contrast, training the ResNet50 [63] model yields higher than 91% GPU busy time.

In addition to inefficient kernel orchestration on GPUs, the widely-adopted neighbor sampling method [9] generates subgraphs that are not only much sparser but also subgraphs that iteratively shrink in sizes with each layer. For these highly sparse graphs, an interesting observation is that dense matrix and element-wise kernels can outweigh sparse kernels in terms of latency. Prior works that only focus on optimizing sparse GNN kernels [38, 42, 43, 53] offer limited advantages in this scenario.

Chapter 3

Addressing the Data Loading Bottleneck with Multistage Sampling

This chapter describes the method used throughout the thesis to address the most significant performance issue of GNN training systems: the data loading bottleneck. Motivated by the frictions between the traits of GNN sampling and modern machine learning hardware, the thesis introduces multistage sampling co-designed with hierarchical memory organization as the main solution in reducing I/O and promoting data reuse.

3.1 A Multistage Sampling Perspective

As described in Chapter 2, existing mini-batch GNN training systems adopt a *flat* sampling architecture, where mini-batches are repeatedly sampled from the complete graph data stored in larger but slower memory tiers. This monolithic approach leads to several performance limitations. The irregular data accesses inherent in sampling operations cause poor I/O performance at lower memory tiers. Transferring mini-batches across multiple memory hierarchies is easily bottlenecked by the peripheral bus interface. Although there are works proposing to build caching layers to reduce costly I/O [36, 64–67], the fundamental mismatch between fine-grained sampling and certain memory tiers (particularly secondary storage like hard disks) will still cause significant performance degradation. Besides, the separation between GNN sampling and model computation hinders optimizations such as gather-compute kernel fusion, which is critical to alleviate the data movement overhead in GNN training.

To overcome these challenges, this thesis proposes a multistage sampling scheme for GNN training, where data sampled in earlier stages serves as the source in later stages. To illustrate this approach, consider a two-stage sampling implementation. The data loading process runs in two nested loops (Algorithm 1). The inner loop (Line 3) reuses the previously-sampled

data $\mathcal{G}', \mathbf{H}'$ to generate mini-batches, thereby converting frequent data traffic on M1 into infrequent bulk data transfers from M1 to M2 (Line 2).

Algorithm 1 GNN training with two-stage mini-batch sampling

Require: Graph $\mathcal{G}$ with feature data $\mathbf{H}$; GNN samplers for the two stages `SAMPLE1&GATHER1`, `SAMPLE2&GATHER2`; memory tiers M1 and M2 with decreasing capacities but increasingly faster I/O. Initially, $\mathcal{G}$ and $\mathbf{H}$ are stored in M1.

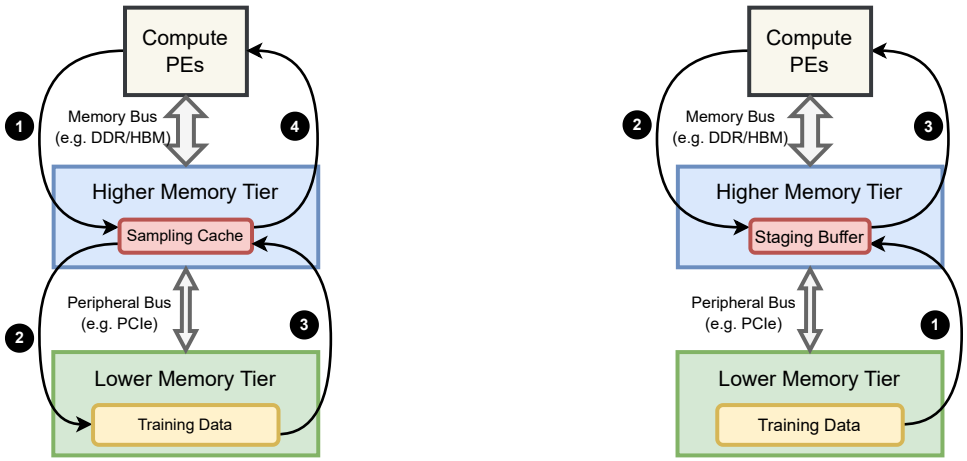
1: **for** $\mathcal{G}'$ in `SAMPLE1`($\mathcal{G}$) **do**
2: $\quad\mathbf{H}' \leftarrow$ `GATHER1`($\mathcal{G}'$, $\mathbf{H}$) ▷ Gather sampled features from M1 and load to M2
3: $\quad$**for** $\mathcal{G}''$ in `SAMPLE2`($\mathcal{G}'$) **do**
4: $\quad\quad\mathbf{H}'' \leftarrow$ `GATHER2`($\mathcal{G}''$, $\mathbf{H}'$) ▷ Gather sampled features from M2 for final training
5: $\quad\quad$Train the GNN model with $\mathcal{G}''$, $\mathbf{H}''$
6: $\quad\quad$Update $\mathbf{H}'$ ▷ Write updated features to M2 (model dependent)
7: $\quad$**end for**
8: $\quad$Update $\mathbf{H}$ ▷ Write updated features from M2 to M1 (model dependent)
9: **end for**

Note that Algorithm 1 puts no hard constraints on the choices of GNN samplers so far. Different combinations of samplers would likely result in drastically different training performance and model quality. Therefore, we aim to answer these two questions in the following sections:

- When does multistage sampling benefit model training performance?
- How does the cascading of samplers affect model convergence?

3.2 Training Performance: I/O Cost Analysis

The runtime of large-scale GNN training is dominated by I/O incurred by mini-batch sampling and data loading (Fig. 2.3). To answer the first question, we compare I/O costs in the conventional single-stage sampling approach with the proposed multistage sampling approach. The discussion is based on an abstracted system architecture for generality shown in Fig. 3.1. The lower memory tier provides larger capacities than the higher memory tier and keeps the complete training data, albeit at the cost of slower I/O interface. We denote I/O bandwidths of low/high memory tiers as BW_l and BW_h. This memory organization is representative of modern machine learning hardware. For example, the lower and higher memory tiers could correspond to host memory and GPU memory, where the compute PEs are GPU cores. For out-of-core settings where the training data is larger than the host memory, the two memory tiers represent the secondary storage and host memory, while the compute PEs become CPU cores. Fig. 3.1a shows data path in single-stage sampling, where data accesses to the lower memory tier (② and ③) are in the critical path of every



(a) Single-stage GNN sampling

(b) Multistage GNN sampling

Figure 3.1: Data access paths in GNN training with (a) the flat single-stage sampling scheme and (b) the hierarchical two-stage sampling scheme.

mini-batch generation step. ① performs mini-batch and graph sampling. ② requests data from the lower memory tier on cache misses. ③ returns the requested data and updates the sampling cache. ④ finalizes the sampled data for training. Fig. 3.1b illustrates data paths in multistage sampling. Accesses the lower memory tier (①) are decoupled from the frequent paths of mini-batch generation (② ③). ① performs large-batch sampling and loads the data into the staging buffer. ② ③ perform multiple mini-batch sampling iterations using data from only the staging buffer.

Given a subset of training examples $\mathcal{B}$, we characterize the working set size of a graph sampling algorithm $\mathcal{S}$ as $\mathcal{C}_{\mathcal{S}}(\mathcal{B})$. As we will show later, $\mathcal{C}_{\mathcal{S}}(\mathcal{B})$ plays a critical role in the effectiveness of multistage sampling. Sampling cache widely exists in state-of-art systems based on flat single-stage sampling (Fig. 3.1a). We use the average cache miss ratio β to characterize its utility, which is dependent on multiple factors such as cache policies, capacity and input data. The average I/O cost per iteration of Fig. 3.1a is then

$$\text{Cost}_a = \frac{1-\beta}{BW_h}\mathcal{C}_{\mathcal{S}}(\mathcal{B}) + \frac{\beta}{BW_l}\mathcal{C}_{\mathcal{S}}(\mathcal{B}). \tag{3.1}$$

For multistage sampling (Fig. 3.1b), we take the following procedure to ensure the same batch size as single-stage sampling: the first-stage sampler, denoted as $\mathcal{S}1$, produces a large batch $\mathcal{B}_K$ with K times more training examples than $\mathcal{B}$. The second-stage sampler, $\mathcal{S}2$, produces mini-batches in K consecutive iterations from $\mathcal{B}_K$. The average I/O cost per iteration for

Fig. 3.1b is

$$\text{Cost}_b = \frac{1}{BW_h}\mathcal{C}_{\mathcal{S}2}(\mathcal{B}) + \frac{1}{BW_l \cdot K}\mathcal{C}_{\mathcal{S}1}(\mathcal{B}_K) \leq \frac{1}{BW_h}\mathcal{C}_{\mathcal{S}1}(\mathcal{B}_K) + \frac{1}{BW_l \cdot K}\mathcal{C}_{\mathcal{S}1}(\mathcal{B}_K). \tag{3.2}$$

Since there is usually a substantial performance gap between BW_l and BW_h (e.g., PCIe is two orders of magnitude slower than GPU memory), we ignore the insignificant first term. Hence, multistage sampling achieves an I/O cost reduction of

$$\frac{\text{Cost}_b}{\text{Cost}_a} \lesssim \frac{\mathcal{C}_{\mathcal{S}1}(\mathcal{B}_K)}{\beta K \mathcal{C}_{\mathcal{S}}(\mathcal{B})} = \frac{1}{\beta}\frac{\mathcal{C}_{\mathcal{S}1}(\mathcal{B}_K)/|\mathcal{B}_K|}{\mathcal{C}_{\mathcal{S}}(\mathcal{B})/|\mathcal{B}|}, \text{ when } BW_h \gg \frac{BW_l}{\beta}. \tag{3.3}$$

Conclusion. We define $\mathcal{C}_{\mathcal{S}}(\mathcal{B})/|\mathcal{B}|$ as the normalized working set size for the sampler $\mathcal{S}$ and $\mathcal{B}$. The normalized working set size of first-stage sampling is the deciding factor in overall I/O cost reduction, even though the general form in Eq. (3.3) does not guarantee multistage sampling achieves I/O savings compared to single-stage. It is thus crucial to pick an I/O-efficient sampler at the lower memory tier for realistic performance advantages. We also remark that the sampler choice at the higher memory tier becomes less relevant to overall I/O costs if caching fails to bridge the performance gap between memory tiers.

3.3 Model Convergence: Statistical Analysis

In this subsection we provide a sketch of theoretical analysis to describe the model convergence behavior with multistage sampling, which we will leverage to understand existing strategies and also improve them. The analysis is based on the decomposition of gradient estimation errors in GNN training. The errors have been widely known to directly influence the convergence behavior in stochastic gradient training [44, 45, 47, 68]. We adopt notations introduced in Chapter 2, while new ones are listed in Table 3.1. To provide a gist of analysis, we consider the vanilla SGD-style training algorithm, where the parameters are updated as $\theta_{k+1} = \theta_k - \eta \cdot \tilde{g}_{\mathcal{B}}(\theta_k)$. We begin with the standard assumption on the smoothness of the objective function:

Assumption 3.3.1. *$F(\cdot)$ is continuously differentiable and its gradient $\nabla F(\cdot)$ is L-Lipschitz continuous, i.e.,*

$$\nabla F(\theta_1) - \nabla F(\theta_2) \leq L\|\theta_1 - \theta_2\|_2, \forall \theta_1, \theta_2.$$

It then follows from standard stochastic gradient analysis (referring to LazyGCN [68] up to

$\theta, \theta_k, \theta_*$	Model parameters. θ_k refers to the parameters at the k-th iteration, while θ_* denotes the optimal parameters.
$\mathcal{B}, \mathcal{B}_1, \mathcal{B}_2$, etc.	A sampled mini-batch of training examples.
$f_v, \tilde{f}_v$	f is the exact objective function for a single training sample v, while $\tilde{f}$ denotes the approximate objective function under graph sampling.
$F, \tilde{F}$	Full-batch versions of f_v and $\tilde{f}_v$, e.g., $F(\theta) = \frac{1}{\lvert\mathcal{V}\rvert}\sum_{v\in\mathcal{V}} f_v(\theta)$. Thus, F is the objective function in original full-batch GNN training.
$F_\mathcal{B}, \tilde{F}_\mathcal{B}$	Mini-batch version of f_v and $\tilde{f}_v$, e.g., $F_\mathcal{B}(\theta) = \frac{1}{\lvert\mathcal{B}\rvert}\sum_{v\in\mathcal{B}} f_v(\theta)$.
$\nabla F(\theta_k), g(\theta_k)$	$\nabla F(\theta_k)$ is the full gradient on the model parameters at the k-th iteration. $g(\theta_k)$ denotes the gradient estimator for the corresponding exact or approximate objective function, e.g., $\tilde{g}_\mathcal{B}(\theta_k) = \nabla\tilde{F}_\mathcal{B}(\theta_k)$.

Table 3.1: Summary of Notations

Equation 34 for details) that with a proper choice of the step size $0 < \eta \leq \frac{1}{L}$, we can yield

$$\frac{1}{T}\sum_{k=1}^{T}\mathbb{E}\left[\|\nabla F(\theta_k)\|_2^2\right] \leq \frac{2}{\eta}\frac{\mathbb{E}\left[F(\theta_1)\right] - \mathbb{E}\left[F(\theta_*)\right]}{T} + 2\eta L\Delta_T \xrightarrow{T\to\infty} 2\eta L\Delta_T \tag{3.4}$$

where $\Delta_T = \frac{1}{T}\sum_{k=1}^{T}\mathbb{E}\left[\|\tilde{g}_\mathcal{B}(\theta_k) - \nabla F(\theta_k)\|_2^2\right]$. Real-world implementations of GNN training usually adopt diminishing learning rates so the right-hand side of Eq. (3.4) eventually converges to 0, regardless of the asymptotic behavior of Δ_T. Nonetheless, it is clear that the average gradient estimation error Δ_T plays a critical role in the upper bound of model convergence rates.

Decomposition of the error term. Under the condition of unbiased mini-batch sampling ($\mathbb{E}\left[g_\mathcal{B}(\theta_k) - \nabla F(\theta_k)\right] = 0$) which commonly holds, and assuming the independence of sampling training examples and graph structures, the error decomposition is straightforward:

$$\begin{aligned}
&\mathbb{E}\left[\|\tilde{g}_\mathcal{B}(\theta_k) - \nabla F(\theta_k)\|_2^2\right] \\
&= \mathbb{E}\left[\|\tilde{g}_\mathcal{B}(\theta_k) - g_\mathcal{B}(\theta_k)\|_2^2\right] + \mathbb{E}\left[\|g_\mathcal{B}(\theta_k) - \nabla F(\theta_k)\|_2^2\right] + 2\mathbb{E}\langle\tilde{g}_\mathcal{B}(\theta_k) - g_\mathcal{B}(\theta_k), g_\mathcal{B}(\theta_k) - \nabla F(\theta_k)\rangle \\
&= \mathbb{E}\left[\|\tilde{g}_\mathcal{B}(\theta_k) - g_\mathcal{B}(\theta_k)\|_2^2\right] + \mathbb{E}\left[\|g_\mathcal{B}(\theta_k) - \nabla F(\theta_k)\|_2^2\right] = \mathbb{E}\left[\|\tilde{g}_\mathcal{B}(\theta_k) - g_\mathcal{B}(\theta_k)\|_2^2\right] + \mathbb{V}\left[g_\mathcal{B}(\theta_k)\right].
\end{aligned} \tag{3.5}$$

Therefore, the error term $\|\tilde{g}_\mathcal{B}(\theta_k) - \nabla F(\theta_k)\|_2^2$ in Δ_T consists of two sources:

- *sampling variance* due to mini-batching of training examples ($\mathbb{V}\left[g_\mathcal{B}(\theta_k)\right]$)
- *sampling bias* introduced by graph sampling and non-linear layers ($\mathbb{E}\left[\|\tilde{g}_\mathcal{B} - g_\mathcal{B}\|_2^2\right]$).

Similar tricks can be used to decompose each error term in multistage sampling. The estimation and approximation procedures are sketched below. Therein, $\mathcal{B}_1$ and $\mathcal{B}_2$ denote the sampled sets of training examples at two stages, where $\mathcal{B}_1$ is sampled from $\mathcal{V}$ and $\mathcal{B}_2$ from $\mathcal{B}_1$, both without bias. $\hat{F}$ represents the objective function after the second-stage graph sampling as a further approximation of $\tilde{F}$.

Single-stage sampling: $F(\theta) \xRightarrow[\mathcal{B}\sim\mathcal{V}]{\text{estimate}} F_{\mathcal{B}}(\theta) \xRightarrow{\text{approx}} \tilde{F}_{\mathcal{B}}(\theta)$.

Multistage sampling: $F(\theta) \xRightarrow[\mathcal{B}_1\sim\mathcal{V}]{\text{estimate}} F_{\mathcal{B}_1}(\theta) \xRightarrow[\mathcal{B}_2\sim\mathcal{B}_1]{\text{estimate}} F_{\mathcal{B}_2}(\theta) \xRightarrow{\text{approx}} \tilde{F}_{\mathcal{B}_2}(\theta) \xRightarrow{\text{approx}} \hat{F}_{\mathcal{B}_2}(\theta)$.

Mini-batch induced variance. Following the decomposition scheme, we can break down the variance induced by mini-batch sampling into a sum of variances introduced at each stage, provided that sampling is always unbiased (each example shares the same probability of being selected):

$$\mathbb{V}\left[g_{\mathcal{B}_n}(\theta_k)\right] = \mathbb{V}\left[g_{\mathcal{B}_1}(\theta_k)\right] + \mathbb{E}\left[\|g_{\mathcal{B}_2} - g_{\mathcal{B}_1}\|_2^2\right] + \cdots + \mathbb{E}\left[\|g_{\mathcal{B}_n} - g_{\mathcal{B}_{n-1}}\|_2^2\right]. \tag{3.6}$$

To lower the variance, it is thus crucial to avoid sampling examples of very close gradients at each sampling stage. Most GNN training systems by default conduct sampling with random shuffling. This works well in maintaining the diversity of training examples and reduces the variance in proportion to the batch size: $\mathbb{V}_{\mathcal{B}\sim\mathcal{V}}\left[g_{\mathcal{B}}(\theta_k)\right] = \frac{1}{|\mathcal{B}|}\mathbb{V}_{v\sim\mathcal{V}}\left[g_v(\theta_k)\right]$. However, in more restricted settings, random shuffling suffers from low execution efficiency due to fine-grained random I/O access patterns [69]. Striking a good balance between sampling variance and efficiency will be key to stable and fast model training.

Graph sampling induced bias. The existence of non-linear layers in GNNs induces non-zero output bias even though graph sampling itself is unbiased:

$$\mathbb{E}_{\tilde{\mathcal{N}}\sim\mathcal{N}}\left[\sigma(\sum_{v\in\tilde{\mathcal{N}}}\mathbf{x}_v)\right] \neq \mathbb{E}_{\tilde{\mathcal{N}}\sim\mathcal{N}}\left[\sum_{v\in\tilde{\mathcal{N}}}\sigma(\mathbf{x}_v)\right],\ \tilde{\mathcal{N}} \text{ is uniformly sampled from } \mathcal{N},\ \sigma \text{ is non-linear.}$$

Thus, we derive an upper bound using the Cauchy–Schwarz inequality and seek methods to minimize the bound instead (two-stage sampling shown here):

$$\frac{1}{2}\mathbb{E}\left[\|\hat{g}_{\mathcal{B}_2} - \nabla F(\theta_k)\|_2^2\right] \leq \mathbb{E}\left[\|\tilde{g}_{\mathcal{B}_2} - \nabla F(\theta_k)\|_2^2\right] + \mathbb{E}\left[\|\hat{g}_{\mathcal{B}_2} - \tilde{g}_{\mathcal{B}_2}\|_2^2\right]. \tag{3.7}$$

Similarly to mini-batch sampling variance, the approximation errors of graph sampling are also additive for each stage. While the additive errors are likely to impede model convergence, the decoupling of sampling strategies unblocks the opportunities of choosing the best-suited

algorithms based on hardware characteristics at each stage. Co-designing the sampling algorithm benefits from much faster training runtime, potentially offsetting the effect of slower model convergence.

Conclusion. This section presents a general analysis of the gradient errors in GNN training with multistage sampling. Key conclusions, given in Eqs. (3.5) to (3.7), can be summarized as the following takeaway message: model convergence with multistage sampling is bounded by *the combination of gradient estimation variance and approximation error at each stage.*

Leveraging the insights gathered from Sections 3.2 and 3.3, this thesis now describes two systems for large-scale GNN training. They are rooted in two slightly different hardware setups, but both are widely accessible to machine learning practitioners.

Chapter 4

HANOI: Fast and Accurate Out-of-Core GNN Training

4.1 Introduction

Scaling the training of graph neural networks on massive graphs will open up many new real-world applications. The practice of single-node, in-memory GNN training sets a scalability limit when datasets become much larger than DRAM of a single machine, which is the central issue to be discussed in this chapter. Two standard techniques to mitigate the capacity issue are the use of a distributed-memory cluster and secondary storage (e.g., local or remote SSD, HDD). Much recent work has focused on distributed GNN training [27, 62, 70–75], which has been shown to be bounded by network I/O frequently [27, 62, 70, 74, 76]. Distributed hardware also imposes a natural cost barrier for many individuals and organizations. In this chapter, we consider a cheaper alternative, *out-of-core* GNN training on a single machine. Our target is the ordinary servers employed by the cloud providers or individual practitioners for machine learning. Such machines usually do not have a gigantic amount of main memory, but are equipped with abundant secondary storage, which can store the full dataset in formats that facilitate pre-processing and training.

Nonetheless, achieving effective out-of-core GNN training is nontrivial if we just trade network I/O in distributed training for storage I/O in out-of-core training. Modern storage technology such as NVMe SSDs, though very fast by storage standards, remains an order of magnitude slower than DRAM. The fine-grained and irregular access patterns during mini-batching exhibit an inherent mismatch with the block-addressable interface of SSDs, causing read amplifications and further reducing the effective bandwidth available for training (Fig. 4.1). Thus, simply treating the secondary storage as a memory extension (e.g., with

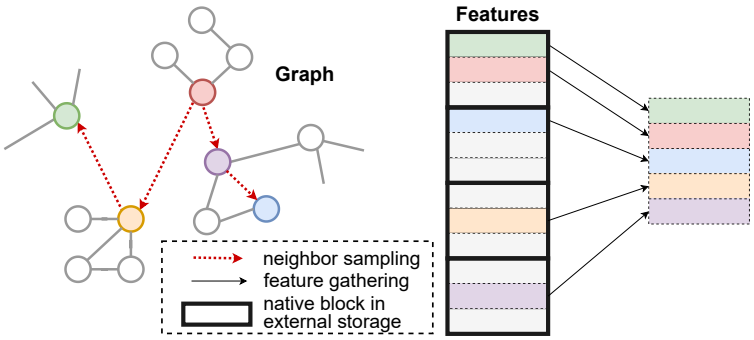


Figure 4.1: Mini-batch sampling in GNNs over external data. Feature gathering incurs fine-grained random accesses, causing under-utilization of IO bandwidth.

`mmap`) is a non-solution; we measured an order of magnitude slowdown of training throughput on a machine equipped with a fast local SSD (Section 4.4.2).

In this thesis, we adopt a drastically different approach by co-designing the sampling scheme in GNN training to accommodate the secondary storage. The central idea in our approach is to separate GNN sampling into two stages, one coarse-grained and one fine-grained, so that the storage I/O is decoupled from the data intensive sampling stage in the existing GNN training pipeline. The coarse-grained "macro-batch" sampling stage precedes the current mini-batch sampling step. While the macro-batches are directly drawn from the secondary storage and requires bulk I/O, it resides in the memory, serving as the staging area for the subsequent mini-batch preparation steps. The small and irregular data accesses incurred by mini-batch sampling and feature gathering are then confined entirely within the fast main memory. We show that decoupled I/O accompanied with extensive pipelining and data reuse techniques could fully overlap the I/O overhead.

However, the introduction of macro-batches inevitably changes the construction of mini-batches, which might have undesirable effects on the model training. To guarantee both runtime performance and model quality, we propose an accuracy-aware partition-based sampling algorithm combining the insights from a comprehensive accuracy study. First, we design a lightweight but highly efficient GNN-aware graph partitioner (GAP) that simultaneously balances labels in each partition and mitigates neighbor loss. During macro-batch sampling, a small portion of hub nodes are permanently pinned inside the memory and participate in the mini-batch sampling steps. The augmentation of hub nodes contributes to relatively isolated nodes and partitions, which helps recover the model accuracy at a low space and computation cost.

We realize the system and algorithmic techniques above into Hanoi, a GNN training system capable of handling graph datasets much larger than the available memory budget.

Compared to prior out-of-core methods, HANOI provides a much better trade-off point of model quality and training time, achieving almost identical model accuracy with comparable runtime performance as in-memory training, while prior methods either fall short in model quality or exhibit substantial slowdown (Table 4.1). We show that HANOI achieves comparable training throughputs with in-memory training while saving up to 85% of main memory budgets. Particularly, we show much improved accuracy over MariusGNN [37], an existing out-of-core GNN training system sharing a similar two-stage batching design, but lacking the accuracy aware components that makes HANOI truly attractive.

Table 4.1: Comparison of model accuracy and runtime between in-memory training baseline and recent out-of-core solutions. Worst results in each column are emphasized in red.

Method	**ΔAccuracy(%)**	**Runtime**	**Memory(GB)**
In-Memory [9]	0	1×	407
MariusGNN [37]	-3.56	up to 1.08×	64
Ginex [36]	0	up to 73.1×	64
This work	-0.14	up to 1.67×	64

In summary, this work makes the following contributions:

- We study a practical but under-explored scenario for training GNNs using secondary storage as a cost-effective approach to scale up GNNs to massive graph datasets.
- An extensive empirical analysis of partition-based sampling that identifies requirements to minimize the accuracy loss of GNN models. Based on this, we propose accuracy-aware sampling techniques including GNN-aware graph partitioning and hub nodes augmentation.
- An efficient implementation of HANOI with easy-to-use interfaces that scales single-node GNN training to larger-than-memory datasets without sacrificing accuracy.
- A detailed evaluation that demonstrates accuracy, throughput and model accuracy benefits of HANOI.

4.2 System Design of HANOI

This section describes the system architecture of HANOI. HANOI is designed to train GNNs on massive graph learning datasets with few assumptions on the hardware requirements. Particularly, unlike cache-based solutions [36, 77] requiring high-end NVMe SSD arrays for good performance, it works equally well on machines with fast local SSDs or slower

networked storage. HANOI also handles graph data preprocessing with limited memory budgets. Previous works [37, 78], though capable of training the model in an out-of-core fashion, usually demand a high-memory fat node for data preprocessing, which compromises the original purpose of an out-of-core system.

4.2.1 I/O Decoupling with Multistage Sampling

The most significant challenge to out-of-core GNN training is the inherent mismatch between the fine-grained random data accesses and the block interface of secondary storage. When exposing the storage interface directly to the fine-grained I/O, it is common to observe substantial performance from read amplification and high I/O contention. Although approaches such as Ginex and GIDS [36, 77] propose to design better caching policies to reduce I/O pressure, their mitigation does not solve the root issue. Therefore, they are still bounded by I/O with limited speedups and depend on expensive SSD RAID arrays to achieve satisfiable performance.

While in HANOI, the main idea of leveraging the multistage sampling paradigm is able to *decouple* the coarse-grained storage I/O from the fine-grained sampling stage. When mapped to the abstraction in Fig. 3.1, the lower and higher memory tier correspond to secondary storage and host memory, respectively. To achieve substantial I/O savings, we co-design samplers at the secondary storage tier ($\mathcal{S}_1$) to (1) generate much less I/O traffic ($\mathcal{C}_{\mathcal{S}1}(\mathcal{B}_K)/K \ll \mathcal{C}_{\mathcal{S}}(\mathcal{B})$) and (2) in more storage-friendly patterns (fully utilizing the storage bandwidth with minimal read amplification).

Specifically, HANOI adopts a combination of partition-based subgraph sampling and hub nodes augmentation to implement $\mathcal{S}_1$, which we denote as macro-batch sampling below. First, HANOI partitions and reorders graph structures and features into coarse-grained blocks in a preprocessing step, similarly to prior out-of-core systems on graph analytics [79–81]. The partitions are stored contiguously in the secondary storage for ease of access. During preprocessing, HANOI also selects a subset of nodes as hub nodes and permanently pins the associated graph structure and feature data in host memory. For each training epoch, graph partitions are randomly shuffled to guarantee randomness in stochastic training. A designated number of graph partitions with their associated feature partitions are periodically selected, loaded in host memory and augmented by the hub nodes to construct the macro-batch. Therefore, the normalized working set of macro-batch sampling in HANOI is bounded by the ratio of non-training nodes to training nodes in each partition, which stays relatively fixed under different batch sizes. We compare the normalized working set sizes of $\mathcal{S}_1$ and the neighbor sampling method in Fig. 4.2, evaluated on the `papers` dataset and using a fanout

of 10,10,10 for neighbor sampling. Clearly, the first-stage sampling in HANOI is superior to neighbor sampling unless the number of training examples becomes excessively large (>100K). Such large batch sizes are impractical since the working set would easily exceed GPU memory. For more realistic settings when the batch size is around 1000, the normalized working set size in HANOI is 81.4% less than neighbor sampling. Taking the more storage-friendly I/O access patterns of HANOI into account, HANOI achieves even more I/O cost savings effectively compared to neighbor sampling.

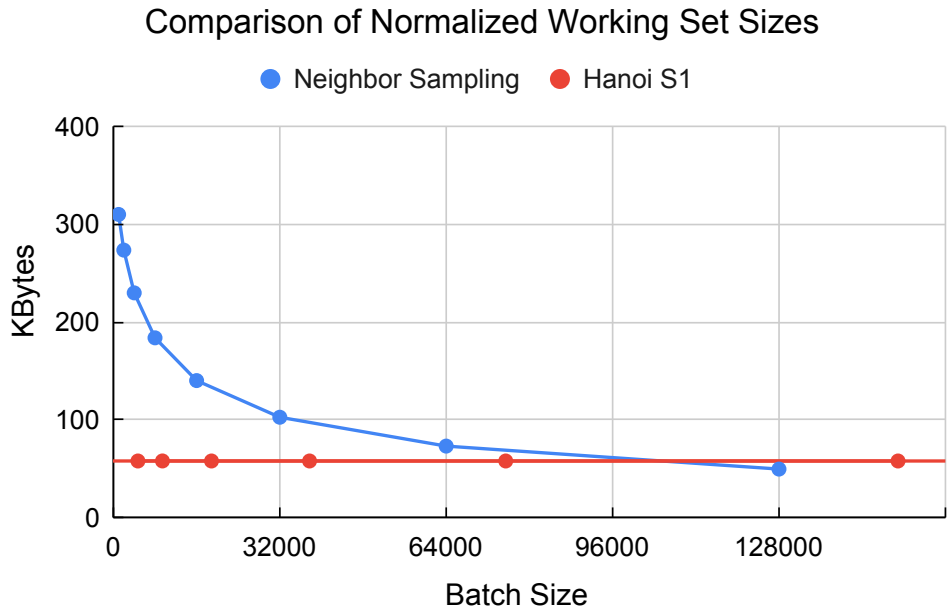


Figure 4.2: Comparing the normalized working set sizes of the first-stage sampler $\mathcal{S}_1$ in HANOI with neighbor sampling.

4.2.2 Hierarchical Pipelining

In this section, we describe deep and hierarchical pipelines of the complex stages in HANOI to utilize all hardware components simultaneously. Pipelining in HANOI is enabled in different granularity, at both the macro-batch level and mini-batch level. I/O and compute are not only pipelined in each sampling stage but also across stages, where the mini-batch pipeline is nested within each step of the larger macro-batch pipeline.

First, we use *double buffering* in the CPU memory to overlap coarse-grained I/O from secondary storage and necessary computation allocated to CPU to construct from fragments of data a complete macro-batch graph. Storage I/O is immediately issued for the next macro-batch before macro-batch construction completes. Meanwhile, we issue the nested

mini-batch pipeline for the current macro-batch step. The pipeline consists of mini-batch sampling on CPU, bus transfer and computation on GPU. It is fired once the dependent macro-batch graph becomes available. We summarize the pipeline organization in Fig. 4.3.

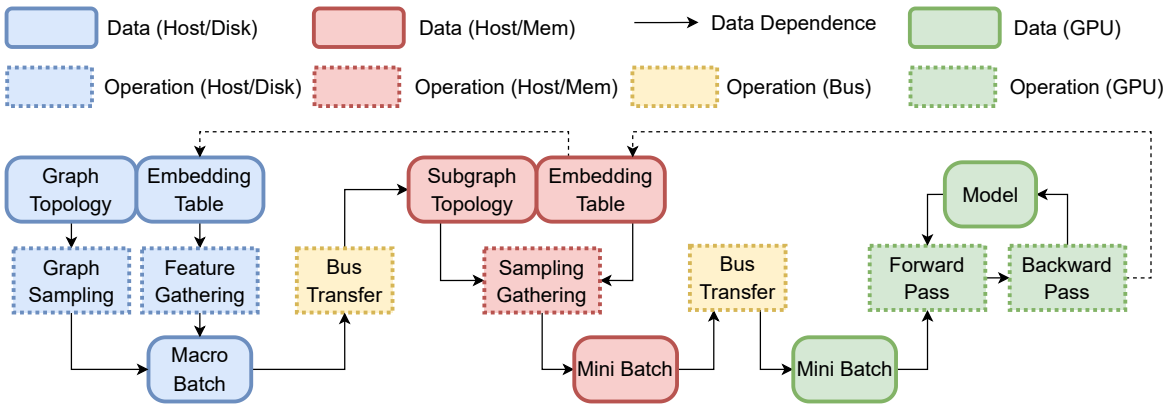


Figure 4.3: Proposed GNN macro-batch and mini-batch training pipeline in HANOI

Moreover, inspired by *data echoing* [82], which reuses a mini-batch to accelerate training of convolutional neural networks and transformer models, HANOI takes advantage of *cost-free* macro-batch reuse to further overlap I/O in GNN training. Our idea is to repeatedly use the current macro-batch for training while waiting for storage I/O to finish of the next macro-batch. Macro-batch reuse is beneficial when the latency of I/O stage in the macro-batch pipeline is longer than the combination of in-memory graph construction and the nested mini-batch pipeline. It essentially harvest the otherwise idle CPU and GPU power to perform more iterations of model updates at no extra cost. We define "reuse factor", the number of passes for which the nested mini-batch pipeline has iterated over the macro-batch data. To prevent models from overfitting into a small subset of data, we judiciously limit reuse factors in HANOI to 4, as shown by prior works [68, 82] to have no negative effects on the final model quality.

4.2.3 Overall System Architecture

Finally, we illustrate the overall HANOI system architecture in Fig. 4.4. To improve I/O efficiency, the graph structure and feature data are divided into coarse-grained blocks, each of which is laid out contiguously in the secondary storage. For the first-stage macro-batch sampling, we randomly shuffle the partitions and select a few of them each time to move to the main memory, constructing the macro-batch graph and features. Then, in the second-stage mini-batch sampling, since data are resident in relatively fast host memory, we perform the fine-grained neighbor sampling algorithm as usual. Different stages of sampling, I/O and

compute are pipelined to guarantee an ideal overlapping of secondary storage I/O, mini-batch sampling, PCIe transfer and GPU model compute.

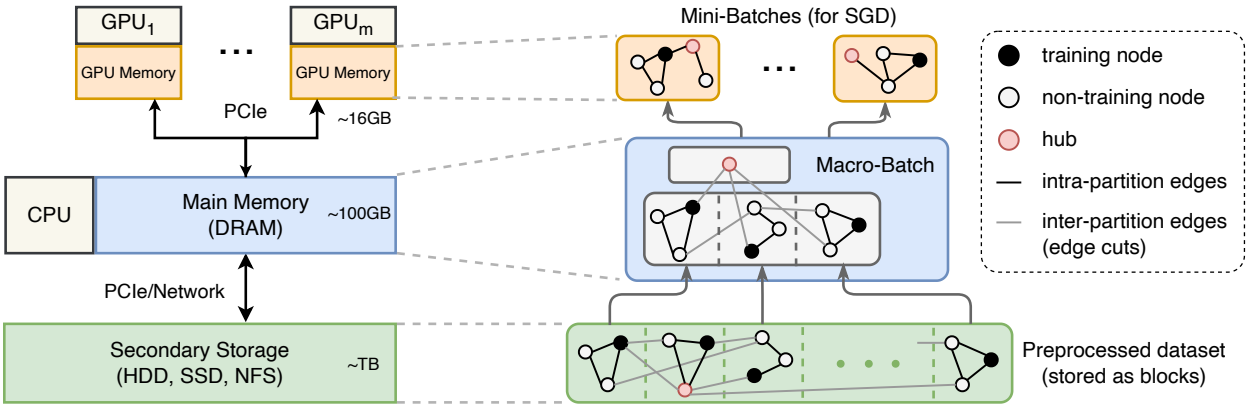


Figure 4.4: The overall algorithm system co-design of HANOI.

4.3 Accurate Macro-Batch Sampling in HANOI

The introduction of the macro-batch stage into the GNN training pipeline permanently changes how the mini-batches are sampled from the original datasets. As a result, it might cause unexpected repercussions to the model quality, exemplified by Table 4.2, as the simple random partition-based method by MariusGNN causes noticeable accuracy drops.

Table 4.2: Accuracy drops of MariusGNN compared to the in-memory baseline. This evaluation uses a 3-layer GraphSAGE model, neighbor sampling with fanouts of 15,10,5 and a batch size of 1000.

Buffer/storage ratio	`arxiv`	`flickr`	`ogbn-papers`	`mag240m-c`
1/8	-2.5	-2.7	-1.2	-1.0
1/16	-3.5	-3.4	-1.7	-1.8

In this section, we counteract the undesired effects by dissecting the macro-batch sampling procedure with empirical experiments. We argue that neighbor loss incurred by inadvertently sparse macro-batches and biased distribution of the training examples play significant roles in the degrading model quality (Section 4.3.1). Based on the findings, we propose two mitigation techniques, namely GNN-aware graph partitioning (GAP, Section 4.3.2) and hub nodes pinning (Section 4.3.3). GAP balances between edge cuts and training bias of graph partitions by imposing a novel loss objective in the underlying partitioning algorithm, while hub nodes enables HANOI to go beyond partition-based batching: pinning a small portion of hub nodes in the macro-batch graph turns out sufficient to restore the model accuracy almost entirely.

Table 4.3: *Over-sparsification* caused by the macro-batch sampling in MariusGNN. "Subgraph $(1/n)$" means the in-memory subgraph created by MariusGNN by sampling $1/n$ of all nodes.

Datasets	`arxiv`	`flickr`	`papers`	`mag240m-c`
No Sampling	14.4	9.9	41.4	19.1
Subgraph(1/8)	2.2	1.3	19.4	4.8
Subgraph(1/16)	1.0	0.7	17.9	3.8

4.3.1 Avoiding Pitfalls in Macro-Batch Sampling

Neighborhood Loss

Since GNNs learn the output node embeddings from the input features and its neighborhood structure *jointly*, it is critical to retain rich graph topology information during GNN sampling. However, from the statistics in Table 4.3 we observe that in MariusGNN, most of the neighbors are thrown away in the excessively sparse macro-batch subgraph. This is understandable since MariusGNN adopts a structure oblivious method, i.e., random partition-based batching, to construct the macro-batch. For cases where the original graphs are already sparse (e.g., `arxiv`, `flickr` in Table 4.2) and the generated macro-batch subgraph has only one or two edges per node, the accuracy degradation has been the most severe.

On the other hand, retaining the complete neighborhood structure for a large amount of nodes is infeasible due to neighborhood explosion. Thus, instead of including the entire multi-hop neighborhood, we conduct a sensitivity analysis to determine how to select the most influential nodes for given training examples. As illustrated in Fig. 4.5, nodes with high importance should be retained, while those of low priorities could be discarded safely.

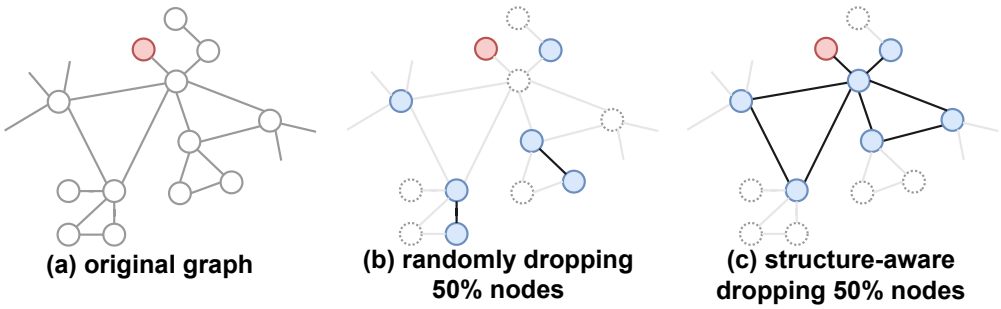


Figure 4.5: A comparison of neighborhood selection strategies. Dashed nodes are dropped, while blue ones remain. Edges between selected nodes are kept.

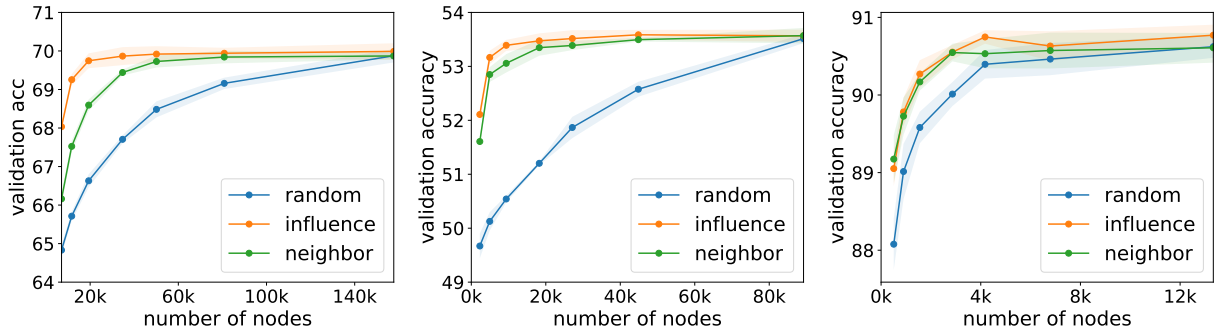
Motivated by the observation, we conduct experiments with three representative GNN datasets (`arxiv` [83], `flickr` [49] and `amazon-cobuy` [84]) to study the influence of node selection on model quality. The protocol below is followed to train a three-layer GraphSAGE model until convergence:

- Given a batch of randomly sampled training nodes $\mathcal{B}$, the L-hop neighborhood is extracted from the original graph.
- We compute a score for each node in the neighborhood based on a scoring function and discard the bottom-k% nodes.
- The subgraph induced by the remaining nodes and $\mathcal{B}$ is used as the mini-batch to update the GNN model.

We test three types of scoring functions, each representing a different neighborhood selection (or dropping) policy. 1) `random`: random scoring function, which is oblivious to the graph structure and simulates MariusGNN. 2) `neighbor`: 1-hop neighbors of $\mathcal{B}$ are assigned a score of 1, while others get random scores in $[0, 1)$. This strategy favors direct neighbors of $\mathcal{B}$ while prunes non-direct neighbors randomly. 3) `influence`: inspired by a previous theory on GNNs [13], we use the landing probability of reverse L-step lazy random walks (RLRW) as the scoring function:

$$\text{RLRW}_{\mathcal{B}}(v) = \sum_{u \in \mathcal{B}} \text{RLRW}_u(v) = \tilde{A}^L \mathbf{e}_{\mathcal{B}} \tag{4.1}$$

where $\tilde{A}$ is the transition matrix of RLRW and $\mathbf{e}_{\mathcal{B}}$ is a multi-hot vector with ones representing nodes in $\mathcal{B}$. If the theory holds, this function computes the influence of node v to output embeddings directly.



(a) `arxiv` (b) `flickr` (c) `amazon-cobuy`

Figure 4.6: Sensitivity of GNN accuracy to neighborhood loss under different policies. X-axis means the number of nodes left after discarding the bottom ranked nodes.

We perform a sweep of k from 0 to 100 and visualize the results in Fig. 4.6. Among others, we have three major observations:

- GNN accuracy generally drops with higher neighborhood loss but different policies affect the sensitivity differently.
- The `random` policy, agnostic to the graph structure, leads to the worst degradation of

model quality as more nodes are discarded. This curve explains the accuracy losses of MariusGNN, since its random partition-based macro-batches discard neighbors in a similar randomized fashion.

- With `neighbor` and `influence` policies, GNNs are much more *resilient* to neighborhood loss. In the `influence` policy, dropping out 80–90% of the low-ranked nodes does not cause visible accuracy loss. Meanwhile, the simpler `neighbor` policy performs surprisingly well, almost overlapping with the more sophisticated approach when up to 60% nodes are discarded.

These small-scale experiments deliver a strong signal of how to alleviate the model quality issues due to neighborhood loss. By prioritizing the existence of important neighbors in the macro-batch, out-of-core GNNs can be made more resilient and accurate.

Partitioning-induced bias

Graph partitioning is an essential step to yield high I/O efficiency in first-stage sampling of HANOI (Section 4.2.1). The conclusion from the last subsection on avoiding neighborhood loss suggests *clustering* nodes when performing the graph partitioning. However, we find that graph clustering alone could be harmful to GNN training. Graph partitioning has another side-effect of co-locating nodes in the same group, limiting the randomness of training examples to be sampled. Due to network homophily [85], clustering-based graph partitioners tend to put together similar nodes, creating highly skewed node partitions. The skewed node distribution, or bias, in each individual partition could lead to highly variant mini-batches and stagnate the model convergence [48].

We echo the observation in our empirical analysis. The bias of training examples is usually represented with their label distributions. As such, we apply a state-of-art min-cut graph partitioner METIS [86] to `arxiv` and validate the impacts of clustering bias to the model accuracy. Not surprisingly, the partitioner in MariusGNN is not plagued by the clustering effects due to its random nature, although it still affects the accuracy in another way (excessive neighborhood loss).

In summary, due to the complex interplay between the neighborhood structures and node similarities, the ideal macro-batch sampling algorithm should be able to accommodate two seemingly contradictory objectives: minimizing neighborhood loss while ensuring a balanced distribution of training labels. In the following, we show how it could be achieved by a novel GNN-aware graph partitioning algorithm along with other enhancements.

4.3.2 GNN-Aware Graph Partitioning

As the first step to circumvent the accuracy pitfalls in macro-batch sampling, we design a GNN-aware graph partitioner (GAP) to meet the unique combination of requirements. Specificially, we aim to achieve with GAP two objectives, 1) co-locating training nodes and important neighbors in the same partition, and 2) balancing the label distribution of training examples for each partition. Moreover, the runtime and space complexity should scale well for the massive graphs and relatively large partition numbers k to ensure sufficient randomness of the macro-batch.

Based on a streaming graph partitioning algorithm FENNEL [87], GAP combines both requirements in its basic design. GAP is lightweight and fast, running in $O(|E| + |V| \log k)$ time and $O(|V| + kL)$ space, where k denotes the partition number and L keeps the types of labels. We made several non-trivial improvements to FENNEL in terms of functionality and runtime, as explained below.

Extended Functionality

FENNEL treats k-way min-cut balanced graph partitioning as an objective optimization problem. Given a partial k-way partitioning $\mathcal{P} = (V_1, \ldots, V_k)$, FENNEL computes the partition $P(v)$ of a new node v based on the maximization of score function $\delta g(v, V_i)$:

$$P(v) = \arg\max_{i \in \{1,\ldots,k\}} \delta g(v, V_i); \;\; \delta g(v, V_i) = \underbrace{|N(v) \cap V_i|}_{1} - \underbrace{\delta c(|V_i|)}_{2}.$$

Term 1 counts the neighbors of v in Partition i; while Term 2 is a monotonically increasing function on the partition size for the purpose of balancing, which could be interpreted as the marginal cost of adding v to V_i.

Vanilla FENNEL has no support for balancing node labels in each partition. In order to achieve label balancing, we separate the monolithic balancing function $\delta c(|V_i|)$ into multiple independent label-wise components $\delta c(|V_i^{(\ell)}|/\mu_\ell)$ for Label $\ell = 0, \ldots, L-1$, where μ_ℓ is the ratio of nodes labeled ℓ in V. To decide the partition of node v with Label ℓ, we consider the label-wise score function instead:

$$\delta g^{(\ell)}(v, V_i) = |N(v) \cap V_i| - \delta c^{(\ell)}(|V_i^{(\ell)}|/\mu_\ell). \tag{4.2}$$

We list the algorithm details in Algorithm 2.

[1]Instead of a standard heap, our implementation uses the ordered set data structure where `update` can be performed with one `remove` and one `insert` in $O(\log k)$.

Improved Runtime Complexity

While many prior works hold a common opinion that FENNEL has a runtime complexity of $O(|E| + |V|k)$ [88, 89], we improve the algorithm to $O(|E| + |V| \log k)$, making it scale to much larger k. Note that a reasonably large k is desired from Section 4.3.1 for combating skewness in mini-batches.

The complexity improvement is based on the idea of *neighbor-guided search* and maintaining balancing scores in a max-heap. To get the partitioning assignment for a node, prior implementations usually examine all k partitions which takes $O(k)$. Our key observation is that for sparse graphs with a majority of nodes having less than k neighbors, it is sufficient to look at a subset of k partitions. It is done by comparing two candidate choices, the partition with the highest balancing score (Line 17) and the result of neighbor-guided search (Line 18 – 25). Retrieving these two candidates costs $O(\log k)$ and $O(\deg(v))$, respectively. Looping over all nodes in V yields a runtime complexity of $O(|E| + |V| \log k)$. Regarding the space overhead, our partitioner inherits the advantage from FENNEL as a streaming partitioner, requiring almost no extra memory for storing the graph topology. Overall, it only takes $O(|V| + kL)$ space, where $O(|V|)$ is essential for keeping the partitioning assignments of nodes in V, and an extra $O(kL)$ is required by the data structures keeping the label-wise balancing scores.

In summary, combining all these improvements, we successfully design a graph partitioner that suits the demands of HANOI. Our partitioner takes only **8.4min** to partition `ogbn-papers`, a billion-edge graph into 1024 buckets within a **64GB** memory budget, while the graph partitioner [86] commonly adopted in other GNN systems take a peak memory of **600GB** and runs for **∼1h 20min** for 64-way partitioning. We conduct a more comprehensive performance evaluation of our partitioner in Section 4.4.2.

4.3.3 Hub Nodes Augmentation

Our graph partitioner is designed to explicitly mitigate neighbor loss, but not necessarily prioritize important neighbors. For example, if a target node is at the boundary of a community, it may be forced to lose important neighbors already assigned to another partition. However, directly incorporating the important score during partitioning would again make the partitioner too expensive and impractical, since it requires us to compute scores for all $|V|^2$ node pairs. Therefore, we partition the graph without explicitly considering the priority, but add this consideration after partitioning to compensate potential loss.

Our approach is based on a key observation in real-world graph data: the existence of heavy-hitters. Those highly connected nodes are *globally influential* hubs to potentially many

training nodes. Then, our idea is to identify the hubs as a preprocessing step and then buffer the data of pivots in the memory permanently as a prioritization of the highly influential nodes. Hubs are thus always available when we form a macro-batch.

We select hubs in the following steps after graph partitioning. For each partition, we compute the influence score for each node in the graph with respect to the partition, i.e., the accumulated score over each training node in the partition with Eq. (4.1). This is an indication of how influential each node is to each partition. We then sum up the scores over all partitions for every node, and choose the top-k scored nodes as the pivots. As this is a preprocessing step that is performed only once for a given graph, the cost is amortized over the training process.

4.4 Evaluation

4.4.1 Experimental Methodology

Evaluation Hardware. We use two machines to conduct our experiments. Machine 1 (MA1) is a typical workstation with fast external storage, while Machine 2 (MA2) is a compute node in the cloud with slower external storage. We will show that our system can perform well on both machines, even though the speed of external storage varies.

- MA1: Intel Core i9-10920X CPU (24 vCPUs), NVIDIA RTX 4090 GPU, up to 64GB memory and local SSDs up to 4.8GBps I/O bandwidth. Machine 1 is optimized for random IO, delivering ∼1 million IOPS with 4KB native block sizes.
- MA2: Intel Xeon Silver 4215 CPU (16 vCPUs), NVIDIA Titan Xp GPU, up to 256GB memory and network storage up to 2GBps I/O bandwidth. Machine 2 does not handle random I/O as well as Machine 1, with a much larger native block size of 128KB.

Datasets. We use large benchmark datasets listed in Table 4.4. Mag240M and OGB datasets come from the Open Graph Benchmark [33]. Mag240M-C is the subgraph extracted from Mag240M that contains only paper citation edges. All graphs are made undirected if they are originally not. The data statistics after the transformation are listed in Table 4.4. On each dataset, the task is to predict node labels. Although the small dataset `arxiv` easily fits into the main memory in our setting, it is included in the evaluation to demonstrate the competitive model quality.

GNN models and hyper-parameters. We evaluate two representative GNN architectures, GraphSAGE [9] and GAT [10]. We use a universal setting of 3 layers, a hidden dimension of

Table 4.4: Summary of evaluated datasets in HANOI. Graphs are made *undirected* to achieve state-of-the-art accuracy for GNN models.

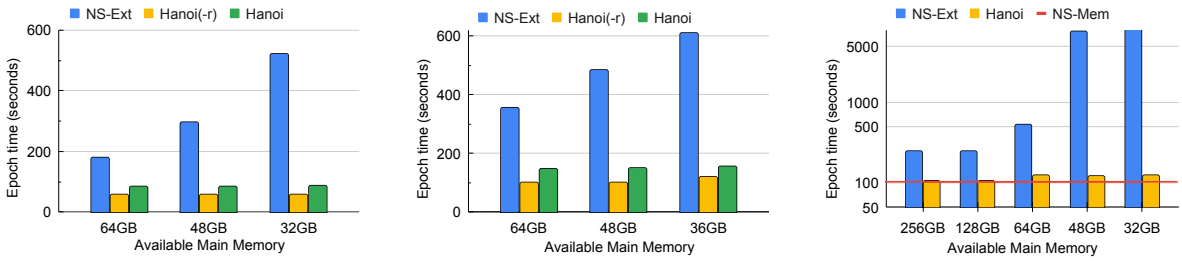
Dataset	# Nodes	# Edges	Feature	Train Ratio	Size
`arxiv`	169K	2.3M	128	0.54	123MB
`flickr`	89K	0.9M	500	0.50	186MB
`products`	2.5M	123.7M	100	0.08	2.8GB
`papers`	111M	3.2B	128	0.01	103GB
`mag240m-c`	122M	2.6B	768	0.01	214GB
`mag240m`	244M	3.5B	768	0.0045	407GB

256, a learning rate of 1e-3 and a dropout of 0.5. Neighbor sampling [9] is adopted as the mini-batch sampling method since it is the de facto approach in large-scale GNN training. We set the batch size to 1000 and a fan-out of (15,10,5) as in previous works [36, 37, 70, 71, 90].

4.4.2 Runtime Performance

We evaluate training speed to show that our solution incurs marginal slowdown compared to in-memory training (`NS-Mem`), but is dramatically faster than a straightforward solution that simply uses the memory as a cache of the external storage (`NS-Ext`).

Per-Epoch Training Time



(a) `papers` on MA1. (b) `mag240m-c` on MA1. (c) `mag240m` on MA2.

Figure 4.7: Per-Epoch Training Time of different implemntations under varying memory budgets. `NS-Mem` is not shown on MA1 as it fails to run due to OOM.

Fig. 4.7a compares the training time per epoch for `papers` on MA1, with various memory budgets from 32GB to 64GB. We observe that HANOI is significantly faster than the `NS-Ext` method, thanks to the sequential accesses enabled by our partitioning and the coarse-grain batching strategy. More importantly, the speedup is larger when given a smaller memory

budget, where our method runs slightly slower when decreasing the memory capacity, but `NS-Ext` suffers a significant slowdown. This means that our method is resilient to the memory capacity and can be scalable to larger datasets. By comparing `Hanoi(-r)` (batch reuse disabled) and `Hanoi`, we also see that enabling batch reuse does not increase training time proportionally, because it does not add any extra I/O time. We observe the same trend for `mag240m-c`, as shown in Fig. 4.7b.

We also evaluate the training speed on MA2 with up to 256GB memory, but a slower storage than that of MA1. Fig. 4.7c shows the epoch training time of `mag240m-c` on MA2. Note that in this figure the y-axis is in log-scale. As the external storage on MA2 is attached via the network, `NS-Ext` becomes extremely slow, e.g., up to 73.1× slower than HANOI which only incurs sequential I/O traffic. However, `Hanoi` is still only slightly slower than `NS-Mem`. This is because HANOI can better utilize the I/O bandwidth, and also the batch reuse can help hide I/O overhead as the I/O time is overlapped with the computation time.

I/O Bandwidth Utilization

To understand the training speedups in detail, we compare the I/O performance of our method with the `mmap` method in Fig. 4.8. We can see that `mmap` system struggles to saturate the I/O bandwidth while constantly incurring I/O traffic, a typical signal that the system is *thrashing*. On the other hand, HANOI exhibits much higher peak I/O utilization than `mmap`. The intermittent I/O bandwidth peaks in HANOI are separated with gaps, where the current macro-batch graph is being constructed or waiting for the previous macro-batch to be fully consumed.

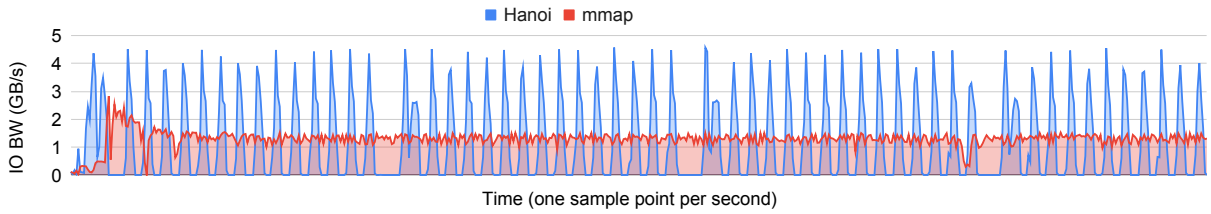


Figure 4.8: Comparing the I/O Bandwidth utilization of HANOI with the `mmap`-based solution on MA1 under 48GB memory budget. Each data point is sampled every second over a 10-minute time period.

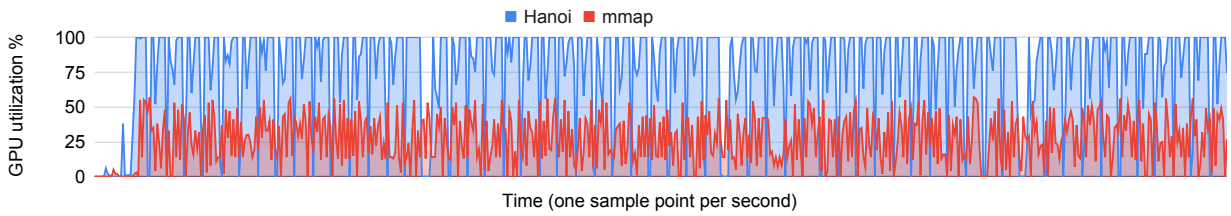


Figure 4.9: Comparing the GPU utilization of HANOI with the `mmap`-based solution on MA1 under 48GB memory budget. Each data point is sampled every second over a 10-minute time period.

GPU Utilization

The other aspect to understand the efficiency of GNN training is to profile the GPU utilization. In Fig. 4.9 we compare the GPU utilization of HANOI with the `mmap` method. It shows clearly that our method achieves much better GPU utilization than `mmap`. The key difference is that when using `mmap`, the GPU spends a lot of time waiting for random I/O, making itself heavily underutilized, whereas HANOI systematically hides the I/O overhead with careful decoupling and pipelining, shifting the system bottleneck from disk I/O to the GPU side.

Graph Partitioning Performance

Finally, we compare the execution time of our graph partitioner GAP with FENNEL in Fig. 4.10. We observe that our partitioner consistently overperforms Fennel, while the performance gap increases rapidly as the number of partitions k increases. This improvement on time complexity is crucial for scaling up to large datasets, as more partitions may be needed by larger datasets to provide enough randomness in our macro-batching strategy.

4.4.3 Model Accuracy

We perform a comprehensive evaluation on GNN model accuracy, to show that our batching strategy is practical and accurate.

Model Convergence

We demonstrate the convergence rate in wall clock time for a wide range of datasets in Fig. 4.11 and Fig. 4.12 to get a detailed look at the training efficiency of HANOI. Again, we use in-memory training with neighbor sampling as the accuracy baseline (`NS-Mem`) and compare with various methods.

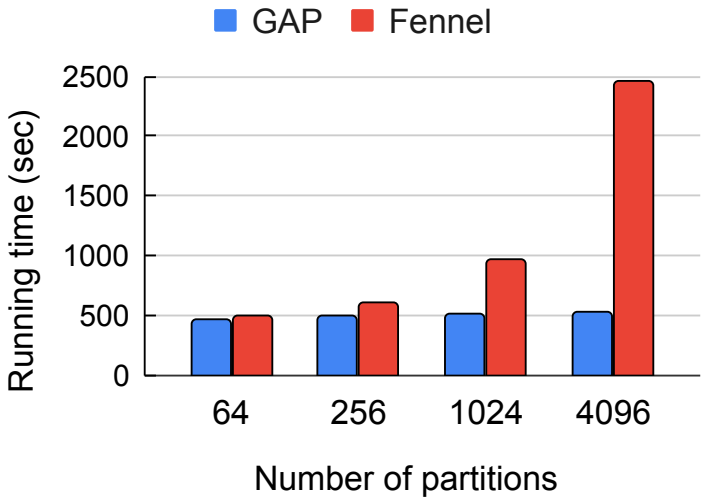
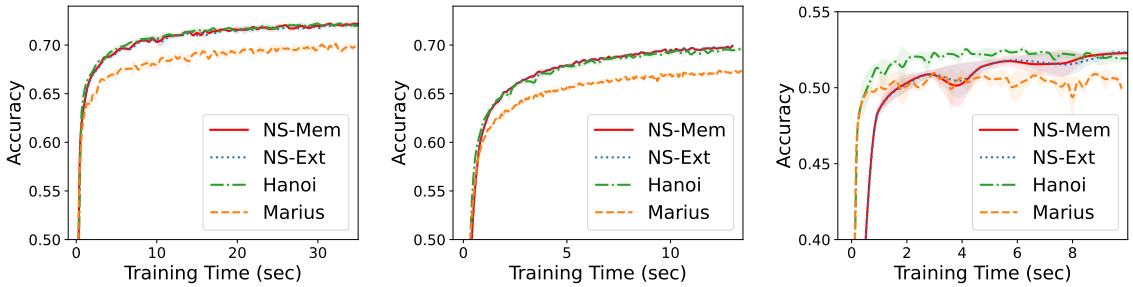


Figure 4.10: Comparing the execution time of GAP partitioner with FENNEL under different number of partitions.

- `NS-Ext`: Similar to Section 4.4.2, `NS-Ext` utilizes `mmap` with over-provisioned I/O threads to serve as a decent out-of-core GNN training baseline.
- `Marius`: We ported MariusGNN [37] into our framework and report it as `Marius` in the figure. Our system yields better training performance than the original implementation in MariusGNN due to better I/O handling and hierarchical pipelining.
- `Hanoi`: Our method with accuracy-aware macro-batch sampling techniques, including macro-batch reuse.

All configurations are run on MA1 in this section except `NS-Mem`, which is run on MA2 with sufficient memory capacity for in-memory training.

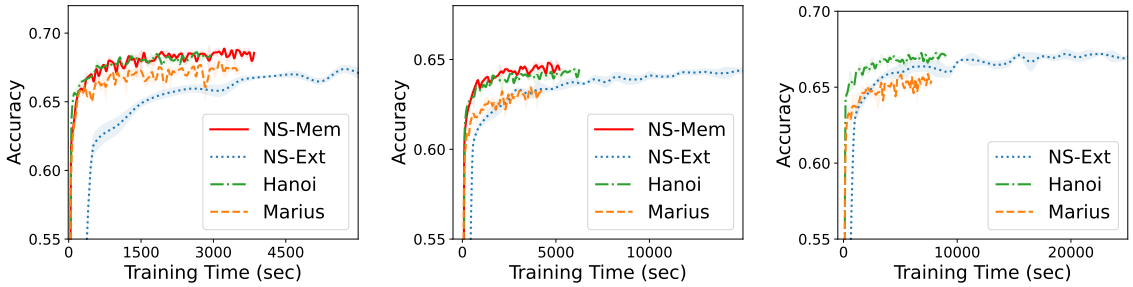


(a) `arxiv` (b) `arxiv` (c) `arxiv`

Figure 4.11: Model convergence in wall-clock time (small datasets).

For smaller graph datasets in Fig. 4.11, HANOI is able to at least match the model

convergence rate of `NS-Mem`, the in-memory training baseline without multistage sampling. In particular, HANOI converges the fastest for `flickr`, likely because the edge cuts introduced by macro-batch sampling act as a positive regularization for the training process. Since small datasets can be entirely cached in host memory, there is no visible difference between `NS-Mem` and `NS-Ext`. However, `Marius` suffers from much slower convergence rates and fails to reach the same accuracy levels as other solutions.



(a) `papers` (b) `mag240m-c` (c) `mag240m`

Figure 4.12: Model convergence in wall-clock time (large datasets).

Fig. 4.12 compares the convergence rates for the three largest datasets. Notably, `NS-Mem` fails to run for `mag240m` since it requires more than 500GB of runtime memory during training - no machines accessible by us are equipped with sufficient DRAM for this experiment. However, with `Hanoi`, we are able to train an accurate GNN model with only 64GB of memory on MA1, which reaches the target accuracy level more than 5× faster than `NS-Ext`. Besides, for `papers` and `mag240m-c`, `Hanoi` is able to catch up with `NS-Mem` at significant faster convergence rates than `Marius` and `NS-Ext`.

End-to-End Model Accuracy

We compare end-to-end overall accuracy of our system (`ours`) with the original in-memory solution (`base`) and the out-of-core solution used in Marius (`marius`). Note that `base` is run with enough memory to hold all data in memory, so that there is no accuracy loss. However, `ours` and `marius` are out-of-core solutions, which may affect model accuracy. For each pair of model and dataset, we train it until convergence and report the test accuracy.

Fig. 4.13a and Fig. 4.13b illustrate the model accuracy of GraphSAGE and GAT respectively. Across all datasets and models, we observe that `ours` matches the accuracy of `base`, while `marius` suffers a constant accuracy drop. The accuracy gap between `base` and `marius` varies from 1.7% to 3.1%, which is significant in the context of GNN training. This is expected as `marius` randomly includes neighbor nodes in their batches, while `ours` picks neighbors according to their affinity to the training samples.

4.5 Related Work

The common practice in GNN training is to use a GNN training framework, e.g., DGL [30] and PyG [31]. However, they only support in-memory training, i.e., they can not support datasets larger than the memory capacity. There are a large volume of research efforts to scale up GNN training, including both algorithmic and system approaches. In the following, we discuss the mainstream scaling-up approaches and relevant optimization techniques.

Graph sampling for GNNs. Graph sampling has been widely used to solve the neighbor explosion problem and thus scale up GNNs. Sampling can reduce memory footprint, CPU-GPU communication cost, and computation of neighborhood aggregation. Current sampling schemes in GNNs fall into several categories, each differing in the structure of the sampled data, the size and number of samples, the size of the input nodes to the sampled batch, etc. There are node-wise neighbor sampling [9, 45, 91], layer-wise sampling [46, 47], and subgraph sampling [48–50]. The first-stage sampling in HANOI can be viewed as subgraph sampling, while the second stage is mainly experimented with node-wise neighbor sampling proposed by [9], as it generalizes better to a wide range of graph datasets with competitive model accuracy. However, our approach can be adapted to other schemes as well, which is left as future work.

Distributed GNN Training. To scale up ML training to massive datasets, a frequently studied approach is distributed-memory training. There are also a large number of distributed GNN training systems [27, 42, 51, 52, 55, 62, 70, 72–75, 92]. However, distributed GNN training often requires frequent and voluminous inter-machine communication, including features, graph structures, models, and gradients, causing substantial challenges in parallelization, synchronization, and pipelining.

Out-of-core graph processing. For traditional graph analytics tasks there exist many out-of-core systems such as Graphchi [93], Chaos [79], COST [94], Mosaic [95], among others [81, 96]. This line of research is valuable and inspirational though not directly applicable to build efficient out-of-core GNN training systems. Marius [78] and PyTorch-BigGraph [97] are out-of-core training systems built for "shallow" graph embedding which does not learn node embeddings from its neighborhood strucure. MariusGNN [37] represents the most recent efforts in building out-of-core training systems for GNNs. It achieves high training throughputs through careful management of in-memory buffers, but does not study how the system design affects accuracy in detail. Our research fills in the gap of both fast and accurate GNN training in out-of-core settings.

Caching. Caching is a widely-studied optimization technique to improve data transfer in GNNs. GNS [67], PaGraph [66] and GNNLab [98] all employ GPU memory as a cache of

main memory and keep likely-reusable data for fast accesses. Ginex [36] is an recent work that studies using the main memory as a cache of storage that demonstrate good speedups. It achieves near optimal cache hit rate and improves speed by 1.5-3× over a naive `mmap` based system. Nevertheless, none of the approaches fundamentally shifts the bottleneck of disk-to-memory data transfer. They do not eliminate fine-grained random accesses or consider the spatial locality, which are critical for accessing data stored in the secondary storage.

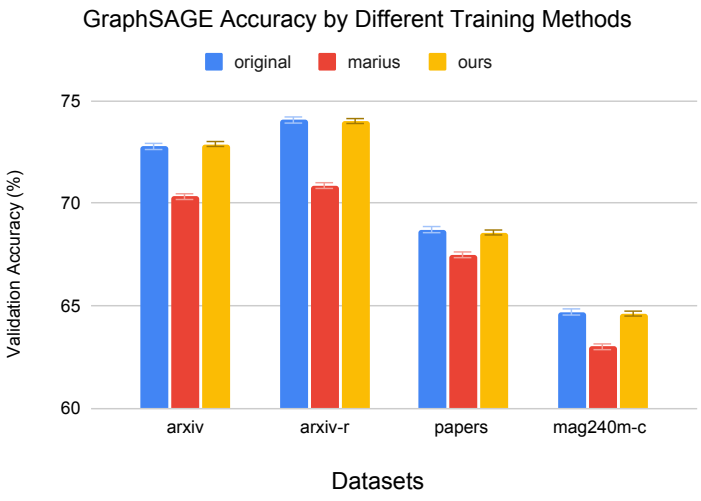
Other optimizations. In addition to caching, there are many other efforts on improving GNN training throughputs by using performance engineering techniques [38, 43, 53, 54, 90, 99, 100]. Notably, FeatGraph [38] does tiling, a typical software optimization to improve cache performance. SALIENT [90] does pipelining to overlap GPU computation and CPU-GPU communication. They are all orthogonal to the scaling-up solutions.

4.6 Conclusions

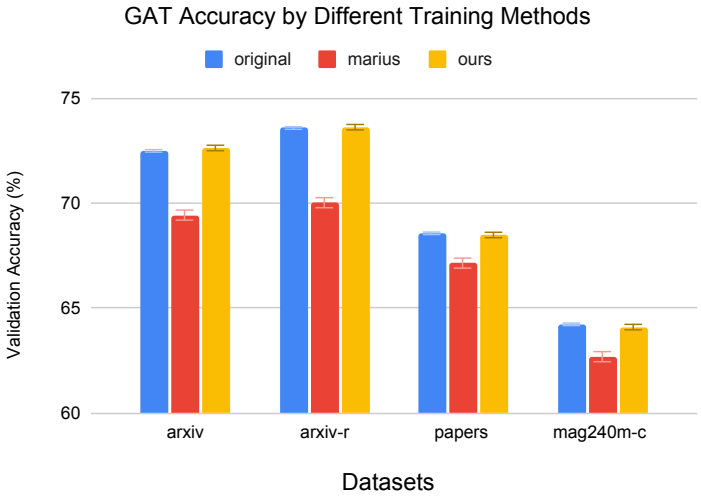
A big limiting factor in the advancement of GNN training is how to scale it to massive data size. In this work we build an out-of-core training system to scale GNNs beyond the memory constraint of a single machine. Based on the literature of graph processing, it is not difficult to design a *fast* out-of-core system, but it is likely at the cost of degrading model accuracy. The key challenge in building such a system is how to achieve high training throughputs without accuracy concerns. Driven by this observation, we carefully design a multistage sampling strategy to decouple I/O from the rest of data intensive training pipeline. The resulting system, Hanoi, is the first to achieve fast training convergence (close to in-memory training) without compromising accuracy.

Algorithm 2 Lightweight Graph Partitioning with Label Balancing

Input: $G(V, E)$, k: #partitions, V_t: training nodes, $\mathbf{y}_t$: training labels (label ids $0, \ldots, L-1$), $\mathcal{P}$: the initial partition assignments for $v \in V$
Output: refined partition assignments $\mathcal{P}$
$n \leftarrow |V|$
Initialize a label array $\mathbf{y}[0, \ldots, n-1]$ for all $v \in V$
$\mathbf{y}[:] \leftarrow L$, then $\mathbf{y}[V_t] \leftarrow \mathbf{y}_t$ ▷ Label non-training nodes with id L
Create $L+1$ arrays $BalanceScores[0, \ldots, L][0, \ldots, k-1]$ ▷ Label-wise balancing scores per partition
Initialize $BalanceScores$ following the second term in Eq. (4.2).
Create $L+1$ max-heaps $Heaps[0, \ldots, L]$ ▷ Keep label-wise balancing scores sorted
for $\ell, p \in \{0, \ldots, L\} \times \{0, \ldots, k-1\}$ **do**
    $Heaps[\ell].insert((BalanceScores[\ell][p], p))$ ▷ Tuples are ordered lexicographically
end for
Initialize an array $NumNbr[k]$ with 0's
Initialize an empty queue $AdjQ$
for $v \in V$ **do**
    $\ell, p_v \leftarrow \mathbf{y}[v], \mathcal{P}[v]$
    Move v out of Partition p_v: update $BalanceScores[\ell][p_v]$, $Heaps[\ell]$ accordingly[1]
    $score_1, p_1 \leftarrow Heaps[\ell].peek()$ ▷ 1st candidate partition
    for $u \in G.in\_neighbor(v)$ **do** ▷ Neighbor-guided search
        $p_u \leftarrow \mathcal{P}[u]$
        if $NumNbr[p_u] == 0$ **then**
            $AdjQ.push(p_u)$ ▷ Book-keep adjacent partitions
        end if
        $++NumNbr[p_u]$ ▷ Count #neighbors per partition
    end for
    $score_2, p_2 \leftarrow \max_{p \in \text{AdjQ}}(NumNbr[p] + BalanceScores[\ell][p])$ ▷ 2nd candidate partition
    if $score_1 > score_2$ **then**
        $p_v \leftarrow p_1$
    else
        $p_v \leftarrow p_2$
    end if
    $\mathcal{P}[v] \leftarrow p_v$ ▷ Completes the assignment for v
    Move v into Partition p_v: update $BalanceScores[\ell][p_v]$, $Heaps[\ell]$ accordingly
    while not $AdjQ.empty()$ **do**
        $NumNbr[AdjQ.pop()] \leftarrow 0$
    end while
end for
return $\mathcal{P}$



(a) GraphSAGE accuracy of different training methods.



(b) GAT accuracy of different training methods.

Figure 4.13: Comparison of end-to-end model accuracy.

Chapter 5

JOESTAR: Joint Optimization for Efficient Sampling-based GNN Training on GPUs

This chapter describes the second major work, JOESTAR, a GPU-centric GNN training framework for large-scale graphs. Compared to the first work HANOI, JOESTAR targets a different hardware configuration by taking a step back and assuming sufficient amounts of host memory are available for GNN training. This setting is arguably more common than out-of-core training, as massive graph data beyond the host memory capacity is not always available or necessary in every domain. Those with time-sensitive requirements for processing massive graph data may opt to obtain high-memory nodes (i.e., machines with large RAM capacity) for in-memory processing when cost is not the primary concern [101]. Under these circumstances, an efficient GNN training solution for in-memory, CPU/GPU cooperative settings that can fully utilize the power of GPU hardware is highly desirable.

The major drawbacks of current in-memory training solutions, as demonstrated in Section 2.3, fall into two aspects: the significant data loading bottleneck and low execution efficiency on the GPU. Unfortunately, existing approaches usually treat them as two separate problems and attempt to address them independently. This isolated view of GNN training hinders cross-stage optimization opportunities, preventing GPU hardware from reaching its full potential. In this work, we take a fundamentally different methodology by jointly considering sampling and model computation in one model optimization workflow. With multistage sampling (Chapter 3), low GPU execution efficiency still remains as the other obstacle to scaling up GNN training. In consequence, the joint optimization of sampling and model computation stages plays a critical role in improving the utility of GPU hardware. It consists of a unified compilation view of GNN training, novel cross-stage operator fusion and profile-guided performance optimizations, greatly alleviating kernel orchestration overheads but also reducing excessive computation and data movements. Eventually, we demonstrate

that JOESTAR achieves state-of-the-art training performance for billion-edge datasets with a single GPU.

5.1 Background and Motivation

5.1.1 Related Works

The popularization of deep learning has made GPUs widely accessible in data centers. Large-scale GNN training is often bottlenecked by CPUs and peripheral buses (e.g. PCIe), with up to 83% of time spent in host-side data preparation and transfer (Fig. 2.3), leaving GPUs heavily underutilized. How to better leverage GPU capabilities for training GNNs with massive graph datasets remains an open question. Among existing frameworks, DGL [30] has the most comprehensive support for GPU-centric mini-batch training with its support of on-GPU graph samplers and Universal Virtual Addressing (UVA [102]), a feature that enables GPUs to directly access data on CPU memory on demand. Recent research has also focused on performance-critical components of GNN training systems, e.g., reducing I/O pressure between CPU and GPU using various caching techniques [64–67, 98], accelerating graph sampling on GPUs [41, 103, 104] and faster GNN computation kernels [103, 105, 106].

Unfortunately, most existing works treat these performance challenges independently. They either focus heavily on the data loading aspect of the problem [64–67, 74, 75, 90, 98] or mainly target the inefficiency of certain GPU operations, for instance, on-GPU sampling [41, 103, 104] or GNN layer implementations [38–40, 43, 53, 54, 105]. The decoupled approach to mini-batch GNN training often fails to yield substantial end-to-end speedups and limits opportunities for cross-stage optimization. A global perspective and a unified optimization methodology are essential for efficient GPU-centric mini-batch GNN training.

5.1.2 Key Operations in Mini-Batch GNN Training

Mini-batch GNN training involves a wide variety of GPU kernels. This diversity arises from the range of operations required for mini-batch sampling, graph structural transformations, and GNN architecture computations. To better understand the long-tail workload characteristics of mini-batch GNN training, we provide a comprehensive categorization of relevant operations, with a focus on those beyond core GNN layer computations.

Operations for Model Computation

GNN model computation consists of two categories: sparse and dense operations. Message passing layers rely on sparse matrix computations such as SpMM and SDDMM, where the sparse component is the adjacency matrix derived from sampled mini-batch graphs. Other standard neural network constructs, including linear layers, activation layers, and residual connections, are implemented as dense matrix computations. Section 2.1 discusses the computation kernels of GNN training in more detail.

Operations for Graph Structural Transformation

Graph structural transformation is a distinct category of operations in GNNs, yet it is often overlooked in prior literature. These transformations entail property queries, mutations and format conversions of target graphs. They are prerequisites to sparse or dense model computations as defined by specific GNN architectures. Typical examples include in/out-degree computation, edge addition/deletion and graph reversal. Although graph structural transformation is not as data intensive as model computation, they contribute non-negligible overheads to overall training due to disproportionally large number of kernels they require. We summarize graph structural transformation as operations over sparse adjacency matrices:

1. Slicing: Extracting specific rows or columns from a sparse matrix..
2. Compaction: Removing empty rows and columns from the input sparse matrix.
3. Transposition: As its name suggests, transposing the sparse matrix.
4. Addition/subtraction: Performing addition or subtraction of one input sparse matrix with another sparse matrix.
5. SpMV: Multiplying the input sparse matrix with a query vector to retrieve information.

Operations for Graph Sampling

Graph sampling is another defining class of operations in GNN training. Prior works [41, 104] uses two sampling operations as the foundation of different graph sampling algorithms. We adopt the abstraction for its simplicity and generality.

1. Individual sampling: `individual_sample(edge_prob, K)`
2. Collective sampling: `collective_sample(node_prob, K)`

Individual sampling samples neighbors for each node independently under the probability distribution `edge_prob` and the number of samples `K`. Collective sampling chooses nodes from the graph following the distribution `node_prob` with a budget `K` and then slices destination nodes of the graphs to sampled nodes. These two primitives serve as building blocks for more complex sampling strategies used in practice.

5.1.3 Challenges of GPU-Centric Mini-Batch GNN Training

As noted in previous chapter, data loading remains the primary bottleneck of in-memory GNN training, even when graph data can fit into the CPU memory. While HANOI addresses this issue for extreme out-of-core configurations via extensive algorithm-system co-design, it is still unclear how best to such an approach to in-memory training. A careful examination of the hardware differences between the two settings is essential. In particular, the byte-addressable DRAM does not suffers from read amplification of fine-grained random I/O. As a result, the benefits of sequentializing data accesses during first-stage sampling become marginal, whereas the extra overhead of macro-batch graph construction can become burdensome. On the other hand, multistage sampling remains appealing in bridging the performance gaps from CPUs and PCIe to GPUs due to potentially reduced sampling work on CPUs and I/O traffic over PCIe. These observations highlight the need of multistage sampling schemes tailored specifically for in-memory training in order to shift more of the intensive sampling work to GPUs.

Moreover, in terms of GPU efficiency, mini-batch GNN training suffers from a unique pattern of hardware under-utilization compared to deep learning in other areas such as image or language modeling. In those domains, training is typically dominated by a small set of compute-intensive kernels (e.g., 2D convolution, large-size GEMM, self attention). In contrast, GNN demonstrates a *long-tail* distribution of kernel types, including sampling operators, graph structural transformations, both sparse and dense model layers. As such, GNN training is neither compute-bound nor by GPU kernel throughputs alone. We show this phenomenon in Table 5.1. "Avg. Compute (us)" indicates the average latency of computation kernels on the GPU and "Avg. Memory (us)" for memory kernels. GCN is trained on the `products` dataset, which fits into the memory of our GPU hardware (RTX 4090), so both sampling and computation happen entirely on the GPU. The table shows that GCN models have significantly more kernel invocations but much lower per-kernel latency compared to ResNet50. This pattern corresponds to lower GPU activity ratios, suggesting that the GPU stays idle for a considerable portion of GCN training time. Note that the GPU activity ratio only indicates whether any part of GPU hardware is active. It does not reflect peak

Table 5.1: GPU kernel statistics of two mainstream models for image classification (ResNet50 [63]) and node classification (GCN [8]). The number in the parenthesis following "GCN" means the batch size used for training.

Models	Kernels/Layer	Avg. Compute (us)	Avg. Memory (us)	Activity
ResNet50	15	121.6	84.6	91%
GCN (1000)	78	17.2	3.4	35%
GCN (8000)	78	72.0	15.3	69%

computation throughputs (e.g. FLOPS utilization, floating point operations per second). Thus, realistic hardware utilization (FLOPS utilization) is likely even lower. On the other hand, improving computation kernels alone would yield limited results due to Amdahl's Law, as there is no positive impact on GPU activity ratios.

In the next two sections, we present the system architecture and optimizations of JOESTAR to systematically address the inefficiencies of GPU-centric mini-batch GNN training.

5.2 System Design of JOESTAR

This section describes the high-level system design of JOESTAR focusing on the multistage sampling schemes and work partitioning between CPUs and GPUs. The goal is to expose overheads in model training on GPUs and reveal a broad space for optimization in GPU-centric training. Notably, even without full-fledged GPU optimizations, our system can already outperform current GNN frameworks [30, 31] by a substantial amount because of alleviated I/O bottlenecks.

Joestar implements multistage sampling by taking into account the memory hierarchy between the host (CPU) memory and GPU memory. The training pipeline resembles that of Hanoi, with a key distinction: second-stage sampling is now co-located with model computation on the GPU (Fig. 5.1). Thus, the pipeline structure is flattened and simplified, consisting of three stages similar to conventional in-memory mini-batch training: CPU (red), bus (yellow) and GPU (green). Double buffering is inserted between pipeline stages for concurrent operations and better utilization of different hardware resources. Given the target mini-batch size $|\mathcal{B}|$, CPUs perform the first-stage sampling in a larger batch size $|\mathcal{B}_L|$. To adapt to a wide range of GPU hardware with varied capacities, $|\mathcal{B}_L|$ is provided to users as a hyper-parameter tuning knob, as it decides the sizes of generated macro-batches. For each macro-batch, the GPU conducts multiple iterations of second-stage sampling, feature gathering and model training until training examples are exhausted. Unlike out-of-core training in HANOI, we do not utilize macro-batch reuse in JOESTAR due to faster first-stage

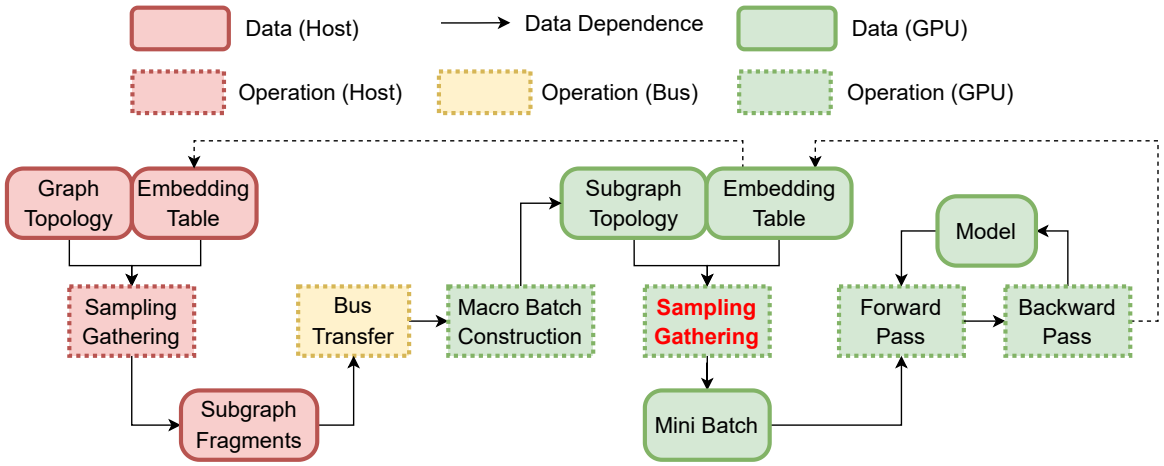
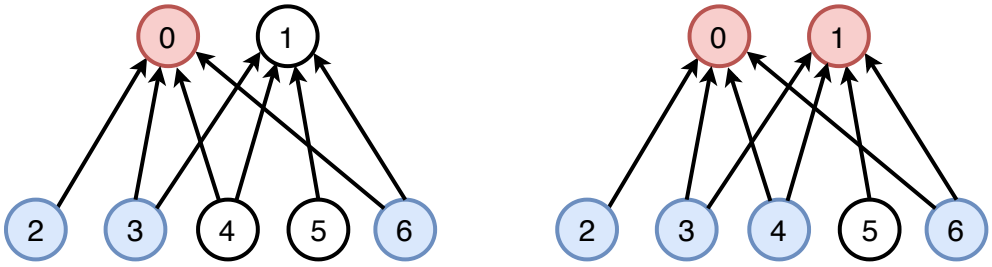


Figure 5.1: Proposed GNN training pipeline in JOESTAR. Heavyweight mini-batch sampling operations (highlighted) are offloaded to the GPU. Macro-batch sampling has low latency on the CPU.

sampling for in-memory data and concerns of overfitting (though not observed empirically).

5.2.1 Multistage Sampling in JOESTAR

JOESTAR mainly targets in-memory training. The main memory (e.g. DRAM) is more flexible and capable than secondary storage in out-of-core training, providing orders of magnitude higher bandwidth and supporting byte-addressable accesses. Thus, the first-stage samplers of JOESTAR require much less hardware-specific adaption from the baseline sampling algorithms. To take advantage of the hardware flexibility, JOESTAR keep the same algorithm in mini-batch sampling unmodified, apply them to a *larger* batch of training examples and extract the sampled subgraph as the macro-batch. We refer to this method as Large-Batch Sampling (LBS). LBS requires no algorithm changes, which ensures strong compatibility with existing implementations. The benefit of LBS comes from the following observation: as $|\mathcal{B}|$ increases, the normalized working set sizes of typical sampling algorithms $\mathcal{C}(\mathcal{B})/|\mathcal{B}|$ monotonically decreases due to the data locality inherent in many real-world graphs. As a result, the overall cost of sampling per epoch becomes lower. We illustrate the observation in Fig. 5.2. This phenomenon is mentioned in several prior works [68, 107] to achieve cost savings in mini-batch sampling. These works leverage the workload reduction of LBS but fail to demonstrate substantial performance speedups against well-optimized baselines based on single-stage sampling. This is primarily due to the lack of key techniques such as intra-batch parallelism and joint optimization of on-GPU sampling and computation which JOESTAR further builds upon LBS. To the best of our knowledge, JOESTAR is the first work to discuss



(a) Batch size = 1, sampling fanout = 3: 3 nodes sampled overall.

(b) Batch size = 2, sampling fanout = 3: 4 nodes sampled overall.

Figure 5.2: Increasing batch sizes in neighbor sampling has a sub-linear scaling of working set sizes. Red nodes are training examples to sample from.

the vast optimization space enabled by LBS in GNN training.

In JOESTAR, second-stage sampling is entirely offloaded to the GPU. To avoid having to implement a new GPU kernel for each different sampling algorithm, JOESTAR chooses a fixed sampling algorithm at this stage. As shown in Section 3.3, the cascading of multiple sampling stages accumulates approximation errors additively in the forward pass of the GNN models. To minimize the error introduced by the second-stage sampling, we use neighbor sampling with very high fan-out parameters (e.g., up to 50 neighbors per node). We notice that outputs of the second-stage sampler are upper-bounded by the macro-batch subgraph. Thus, it is less prone to neighbor explosion than sampling in the original graph, which allows us to adopt the much higher sampling fan-outs. When the macro-batch subgraph is sparse, such as those generated by neighbor sampling of lower fan-outs parameters, the second-stage sampling essentially slices the subgraph and extracts the complete neighborhood structures of training nodes. In this case, no extra graph sampling-induced bias is introduced by the second sampling stage.

Finally, we remark that LBS leads to slight randomness loss by promoting reuse of the same sub-sampled macro-batch subgraph for multiple consecutive mini-batches. If the same node is included in two mini-batches before the macro-batch gets refreshed, its neighbors will be sampled from the same subset of nodes. However, because the macro-batch are updated regularly, the randomness loss is negligible to the final model quality, either empirically or theoretically [68].

5.2.2 Partition-Based Sampling with Historical Embeddings

The decrease in the size of the normalized working set utilized by LBS is most effective for the family of node-wise sampling methods [9, 91]. Due to neighbor explosion, neighbor sampling does not work well with deeper GNN models. Therefore, we propose an alternative first-stage

sampling method to cover models that are not suitable for LBS, which also enables further workload reductions. In the high level, the algorithm resembles partition-based sampling in HANOI but distinguishes itself with several key improvements. We denote the method as PBS (Partition-Based Sampling) and elaborate the algorithm.

Neighbor losses due to edge cuts in graph partitioning are one of the main causes of model accuracy loss (Section 4.3.1). JOESTAR addresses the issue by leveraging historical embeddings [45] of halo nodes. Given a node set $\mathcal{V}$, halo nodes are first-hop neighbors of $\mathcal{V}$ while not included in $\mathcal{V}$ themselves. PBS in JOESTAR (Fig. 5.3) includes halo nodes of sampled graph partitions in the macro-batch to recover from immediate edge cuts due to partitioning. Moreover, PBS retrieves historical embeddings of halo nodes rather than their input node

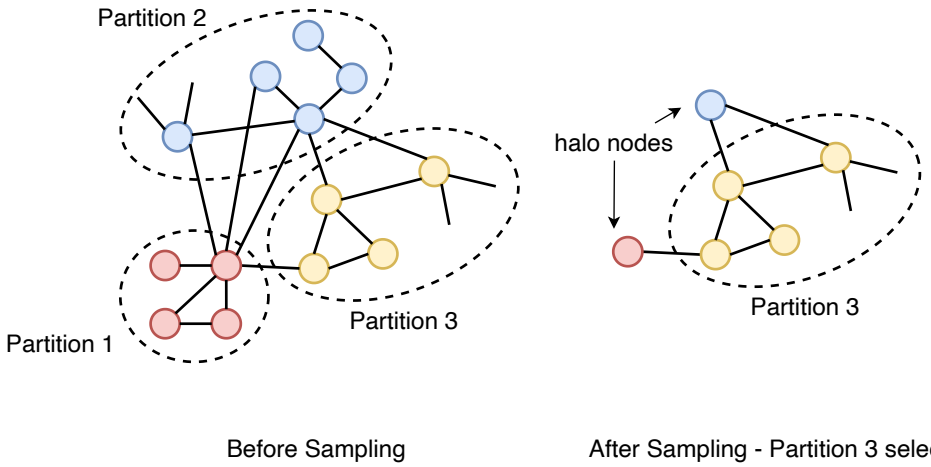


Figure 5.3: PBS in JOESTAR: sampled partitions are selected along with their halo nodes.

features. Historical embeddings refer to the most recent hidden layer activations computed for each node in previous iterations of model training. As model parameters usually change gradually, historical embeddings, though becoming stale with updated model parameters, are still good approximation of accurate hidden activations. By gathering historical embeddings of halo nodes, PBS no longer needs to compute their most up-to-date hidden activations. It effectively stops first-stage samplers from sampling beyond halo nodes, which controls the macro-batch subgraph sizes. PBS keeps a full historical embedding table in the host memory. It is frequently updated by freshly generated hidden activations freshly in the forward passes of GNN models.

The data volume of a macro-batch subgraph in PBS is primarily influenced by the sizes of sampled partitions and the number of halo nodes. Since each halo node induces at least one edge cut of the graph partition, the quality of graph partitioning routine is critical in determining the sizes of macro-batches. Prior works [50, 108] choose min-cut graph

partitioners for this reason. Similarly, we select GAP because of its scalability and capability of maintaining diversity of nodes. Nonetheless, the inclusion of halo nodes still introduces significant data traffic which can be much larger than nodes in the graph partitions. When input features are highly-dimensional or the machine is equipped with legacy PCIe, bus transfer will likely bottleneck the pipeline due to the limited bandwidth of peripheral buses compared to CPU or GPU memory. Thus, more I/O reduction may be desired.

To handle this, JOESTAR proposes ahead-of-time sampling by partially performing steps of second-stage sampling before feature gathering and bus transfer are initiated by the first-stage sampling. The benefits of ahead-of-time sampling comes from the following observation: in large-scale graph datasets, macro-batches constructed by PBS contain a substantial portion of unused data. For those datasets, training nodes constitute only a small portion of all nodes in the graph due to limited labeling (e.g., only 1% are training nodes in `papers`). It is highly likely that the second-stage sampling starting from the small set of nodes will not touch all nodes in a single epoch. Without ahead-of-time sampling, PBS always includes all inner nodes in the partition and halo nodes, even if many of them will not be utilized by the following mini-batch sampling and model training. We demonstrate the mechanism in Fig. 5.4. Ahead-of-time sampling is structure-only and thus lightweight. It does not involve

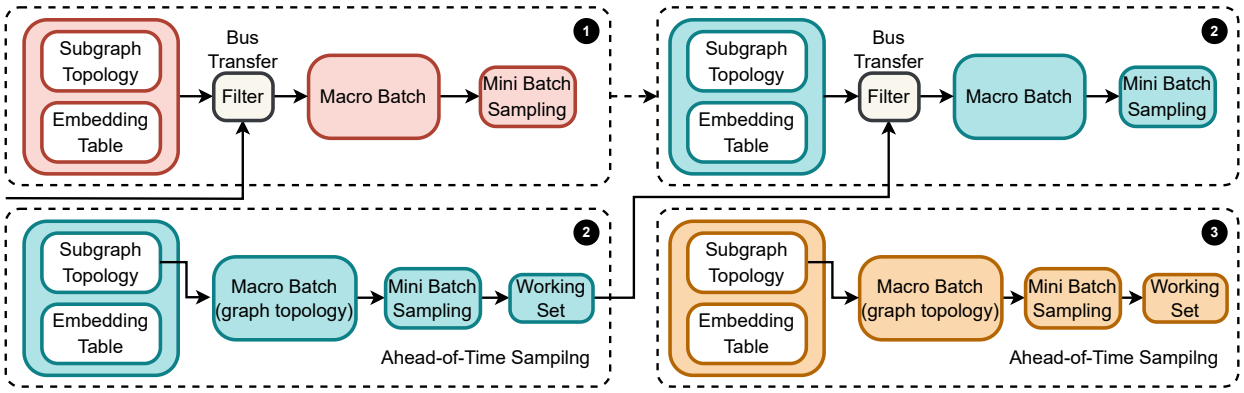


Figure 5.4: Ahead-of-time sampling in JOESTAR decides the working set of the next macro-batch iteration in advance. The macro batch data are then filtered before the bus transfer, reducing the overall I/O traffic.

accesses to large volumes of feature data. The sampling results are used to filter out graph topology input features and historical embeddings in the macro-batch to reduce the overall I/O traffic. Notably, in our experiments ahead-of-time sampling is able to filter out 40% – 50% of data traffic when applied to large datasets `papers` and `mag240m-c`. We provide a more comprehensive evaluation later in Section 5.4.3.

5.2.3 Intra-Batch Parallel Sampling

GPU memory is a scarce resource in large-scale GNN training. For JOESTAR, it puts a tight limit on the number of macro-batches that can be resident on the GPU memory. This leads to a trade-off between macro-batch sizes and sampling parallelism available to CPUs. Increasing macro-batch sizes usually reduces the overall work by CPUs because of graph data locality and hence more overlap in sampled neighborhood structures ($\mathcal{C}(\mathcal{B}_L)/|\mathcal{B}_L| < \mathcal{C}(\mathcal{B})/|\mathcal{B}|$ when $|\mathcal{B}_L| > |\mathcal{B}|$). Meanwhile, excessively large macro-batches restrict inter-batch parallelism due to GPU memory constraints, which significantly lowers sampling thoughputs on CPUs.

JOESTAR addresses the dilemma by proposing intra-batch parallel sampling for the macro-batch stage. Intra-batch parallel sampling takes advantage of the parallelism between nodes and edges within a batch and employ multiple CPU threads to sample a macro-batch collaboratively. Prior works [30, 31, 90] do not favor this type of parallelism because it has suboptimal multi-threaded throughput scaling. However, it is not an issue for JOESTAR, because minimizing macro-batch sampling latencies is prioritized over maximizing sampling throughputs. For each macro-batch, we would like to shorten the CPU stage so that it will be entirely overlapped by bus transfer or GPU stages. Increasing macro-batch sampling throughputs at the cost of worsening latencies on the other hand would stall the GPU, as the mini-batch sampling stage on GPU has to wait for incoming macro-batch data. JOESTAR leverages `parlaylib` [109], a high-performance parallel programming library to implement intra-batch parallelism efficiently.

5.3 Joint Optimization of Sampling and Computation

JOESTAR's multistage sampling scheme enables offloading of second-stage sampling completely to the GPU. By pipelining first-stage sampling on CPUs, bus transfer and on-GPU stages, JOESTAR shifts the primary performance bottleneck from CPUs or bus transfers to GPU execution. Consequently, efficient GPU sampling and GNN model computation are crucial to the overall training performance, which is the focus of this section.

Unfortunately, existing works fail to provide comprehensive solutions to both aspects. Despite substantial research efforts to improve GNN sampler performance and flexibility, most of them treat GNN sampling as an isolated component independent of the rest of GNN training [41, 74, 90, 103]. Similarly, while several studies target GNN kernel optimization [43, 53, 54, 103, 105] and achieve impressive speedups for certain kernels, the impact on end-to-end performance of mini-batch GNN training remains moderate because they do not tackle overheads from sampling or dense tensor operations. This compartmentalized perspective,

while simplifying the problem space, risks overlooking the crucial optimization opportunities at the intersection of sampling and model computation.

To avoid the limitation, JOESTAR adopts a holistic view of the GNN training pipeline, leveraging synergies between sampling and compute. It opens the door to a novel optimization space from the co-location of GNN sampling and computation, even for datasets that are much larger than the GPU memory capacity. Specifically, JOESTAR proposes a compilation-drive workflow to jointly optimize model computation and graph sampling by lowering operations of these two stages into a unified intermediate representations of sparse and dense tensors. A range of optimization opportunities arise to the surface from the abstraction and lowering, for instance, elimination of redundant graph structural computation and novel operator fusions between GNN layers and graph sampling. JOESTAR utilizes PyTorch 2's advanced compilation features [110], such as Dynamo for front-end tracing and Inductor for back-end optimization, in the dynamic mini-batch GNN training workload for the first time. Aggressive kernel fusion and grouping of kernel launching drastically reduce the orchestration overhead that has been significant in mini-batch GNN training. Additionally, JOESTAR employs profile-guided optimizations for better sparse tensor format selection and operator schedules, which represents a methodological advancement beyond current ad-hoc approaches based on heuristics and runtime checks. Fig. 5.5 summarizes the end-to-end workflow of JOESTAR compilation process.

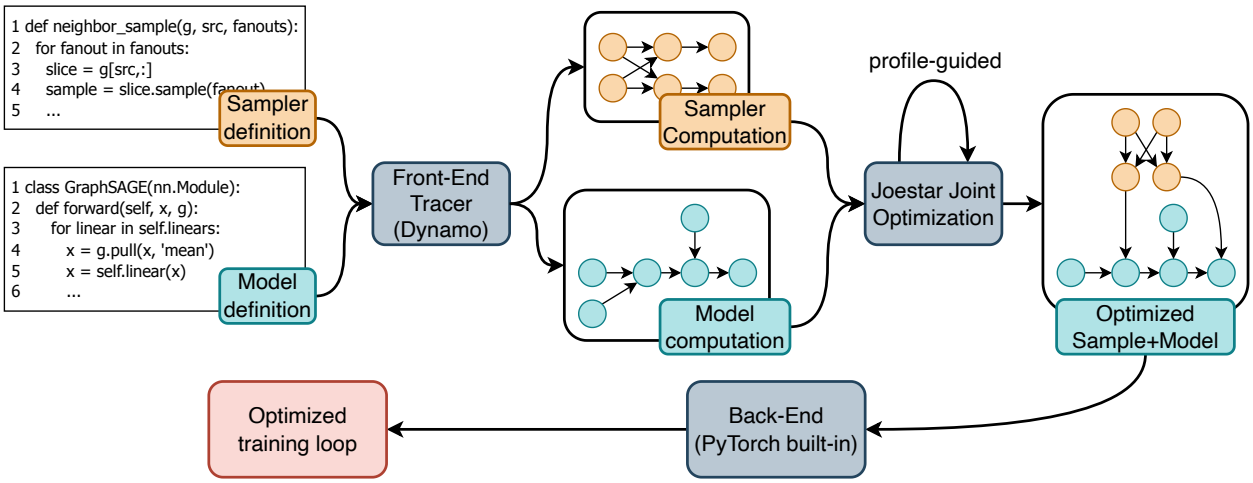


Figure 5.5: Compilation workflow of JOESTAR.

5.3.1 Unified Interfaces of Graph Operations in GNNs

JOESTAR provides multi-level representations to express diverse graph operations of mini-batch GNN training, At the higher level, graphs are treated as a fundamental datatype for which we

introduce common graph query, structural transformation and tensor computation APIs. At the lower level, graphs are described in general sparse tensor formats, i.e., compressed sparse row (CSR), compressed sparse column (CSC), and coordinate list (COO). Correspondingly, the associated graph APIs get translated into compositions of sparse and dense tensor operations, each of which is backed up by actual GPU kernels.

Graph-Level Representation

The graph datatype follows the definition briefly discussed in Section 2.1: a graph structure is given by $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ refers to the node set of G while $\mathcal{E}$ a subset of $\mathcal{V} \times \mathcal{V}$, represents the sparse connections between nodes. Each node $v \in \mathcal{V}$ or edge $e \in \mathcal{E}$ can be attached with one or more attributes, either scalar or multi-dimensional. Attribute data are stored in conventional dense tensors. Nodes and edges are assigned integer IDs which are used to index attribute tensors to retrieve individual data. Following the abstraction of DGL graphs, the node and edge attributes are stored in two containers, *ndata* and *edata*, respectively and indexed by attribute names. G can be bipartite as well. Nodes of a bipartite graph consist of two disjoint sets, $\mathcal{V}_s$ and $\mathcal{V}_d$, where $\mathcal{V}_s$ is the source set and $\mathcal{V}_d$ is the destination set. Similarly, the node IDs are split into two disjoint domains.

We summarize the key graph operations provided by JOESTAR into three groups (Table 5.2): property queries, structural transformations, tensor computation. For tensor computation over graphs we adopt message passing-style interfaces popularized by DGL [30] for node and edge-centric computation. The first category, **property queries**, provides basic information about the graph that is useful in subsequent computations. IDs of nodes and edges, for instance, are often needed in sampling and feature gathering operations. The second category, **structural transformations**, includes operations that transform the graph structure, such as subgraph extraction, augmentation or truncation of graph structures and graph sampling. They produce new graphs from existing ones to be used in the following transformations or GNN model computations. The last category is a core set of **graph computation** operations. These operations follow the same semantics as their namesakes in DGL [30], which enables flexible customization through user-defined functions but also efficient specialization for commonly used ones such as copying messages from source attributes and sum aggregation. One main complaint about the DGL graph interface is the cumbersome syntax when applied to graph sampling [41]. JOESTAR addresses this issue by clearly specifying the default behavior of each operation.

Graph-level interfaces are designed to be an expressive abstraction for developers to implement custom graph samplers and models. Behind the scenes, JOESTAR lowers the abstraction to a set of tensor-level interfaces that are more efficient to implement and optimize.

Table 5.2: Three groups of graph-level operations provided by JOESTAR, using G, G_1, G_2 to denote graphs.

Group	Operation	Description
Property Queries	`num_nodes, num_edges`, etc.	The number of nodes and edges of G.
	`nodes, srcs, dsts, edges`	ID tensors of nodes and edges of G.
Structural Transformations	G`[srcs,:]`, G`[:,dsts]`	Subgraph extraction from G by source and destination node IDs.
	G_1 `op` G_2 (`op:+, -`)	Union and difference of two graphs G_1 and G_2.
	G`.reverse`	Reverse edge directions of G.
	G`.individual_sample(K, probs)` G`.collective_sample(K, probs)`	Sample individually and collectively from G and extract the sampled subgraphs.
Graph Computation	G`.apply_nodes(fn)`	Apply a node-wise function to G. `fn` has access to node attributes.
	G`.apply_edges(fn)`	Apply an edge-wise function to G. `fn` has access to source, destination and edge attributes.
	G`.push(msg_fn, agg_fn)`	Push messages from source nodes to destination nodes. `msg_fn` computes messages and `agg_fn` aggregates messages at destination nodes.
	G`.pull(msg_fn, agg_fn)`	Similar to push but works in the reverse direction. Pull messages from destination nodes to source nodes.

Tensor-Level Interfaces

At the tensor level, the graph datatype is represented as a set of sparse and dense tensors: a sparse tensor A for the graph structure and a collection of dense tensors $\{D_V\}$, $\{D_E\}$ for node and edge attributes. Nonzero elements of A are the edge weights and defaults to 1 if not specified. Currently we do not assume specific patterns of graph topology. Thus, JOESTAR employs three general sparse tensor formats, CSR, CSC and COO, to represent the graph structure. The dense tensors storing node and edge attributes remain in the same formats as underlying tensor frameworks. With the lowering of graph datatype to tensors, we define translations from graph-level operations to tensor-level operations in Table 5.3.

Property Queries. Tensor operations are almost identical to their graph-level counterparts,

Table 5.3: Mapping of graph-level operations to tensor-level operations.

Group	Graph Ops	Lowered Tensor Ops
Property Queries	`num_nodes, num_edges`, etc.	`num_rows, num_cols, num_nnz`
	`nodes, srcs, dsts, edges`	`rows, cols, nnz`
Structural Transformations	G`[srcs,:]`, G`[:,dsts]`	A`[rows,:]`, A`[:,cols]`
	G_1 `op` G_2 (`op:+, -`)	$A_1 + A_2$, $A_1 - A_2$
	G`.reverse()`	A^T
	G`.individual_sample(K, probs)`	A`.rowwise_sample(K, probs).compact()`
	G`.collective_sample(K, probs)`	A`[`A`.row().sample(K, probs),:].compact()`
Graph Computation	G`.apply_nodes(fn)`	Dense tensor operations on D_V
	G`.apply_edges(fn)`	Dense tensor operations on D_E or SDDMM
	G`.push(msg_fn, agg_fn)`	SpMM
	G`.pull(msg_fn, agg_fn)`	SpMM

using row IDs as source nodes and column IDs as destination nodes.

Structural Transformations. The subgraph extraction operation is mapped to a sparse tensor slicing operation, which extracts the rows and columns of A according to the source and destination node IDs. Union and difference operations are translated straightforwardly to sparse tensor addition and subtraction. Graph reversal is implemented as a sparse tensor transpose. To support graph sampling, JOESTAR defines sparse tensor compaction, which removes all-zero rows and columns and remap their IDs to a contiguous range. Hence, individual graph sampling is mapped to a row slicing and row-wise sparse tensor sampling then followed by compaction, while collective sampling is essentially a composition of random number sampling, sparse tensor slicing and compaction. We illustrate the process of individual graph sampling in Fig. 5.6.

Graph Computation. Graph computation is where most of the differences between graph-level and tensor-level operations become evident. The former takes a node- or edge-centric view, while the latter applies the operation on the entire tensors. For `apply_nodes(fn)`, the node-wise function is applied to dense node attribute tensors. It is mapped to corresponding element-wise or row-wise dense tensor operations. For `apply_edges(fn)`, when the user-defined function only accesses edge attributes, it is translated similarly to dense tensor operations. If `fn` involves source and destination node attributes, it is lowered to a sampled dense-dense matrix multiplication (SDDMM) with specicializations. `push(msg_fn, agg_fn)` and `pull(msg_fn, agg_fn)` represent two variants of message passing, one along the edge direction and the other in the reverse direction. They are naturally implemented as SpMM

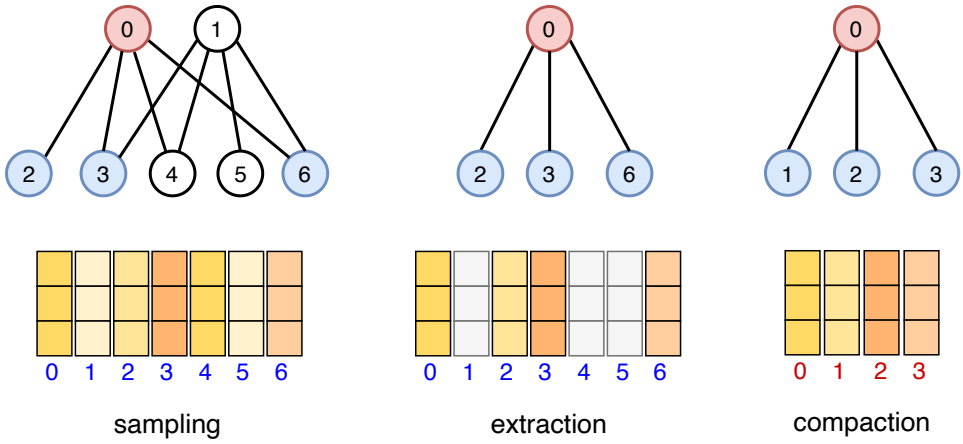


Figure 5.6: Graph sampling in three steps. In GNNs, graph sampling is also associated with feature gathering after compaction.

with specializations. In almost all cases, GNN models do not propagate gradients through the graph structure, so the sparse tensor A is not differentiable. The backward passes of the graph computation operations are thus well defined following common practices in GNN frameworks [30, 31].

Currently, the lowering from graph to tensor operation is carried out manually and then encapsulated as libraries under graph-level APIs. After the lowering, we are able to unify the description of graph sampling and GNN model computations in a single computation dataflow graph (DFG), where each node is a recognized sparse or dense tensor operation and edges are data dependencies between them. Compared with prior solutions [39–41], JOESTAR provides a more comprehensive view of the entire GNN training pipeline, which leads to the optimization opportunities articulated in the following sections.

5.3.2 Compilation Optimizations

Given the unified representation of graph sampling and model computation in a DFG, JOESTAR is able to apply a range of joint compilation optimizations to improve the end-to-end performance. In particular, JOESTAR leverages novel *cross-stage* operator fusion, which fuses part of graph sampling with GNN model computation to eliminate redundant graph structural computation and data movements. To alleviate the kernel launch overhead, JOESTAR also introduces a *kernel invocation grouping* optimization through shape padding and CudaGraphs.

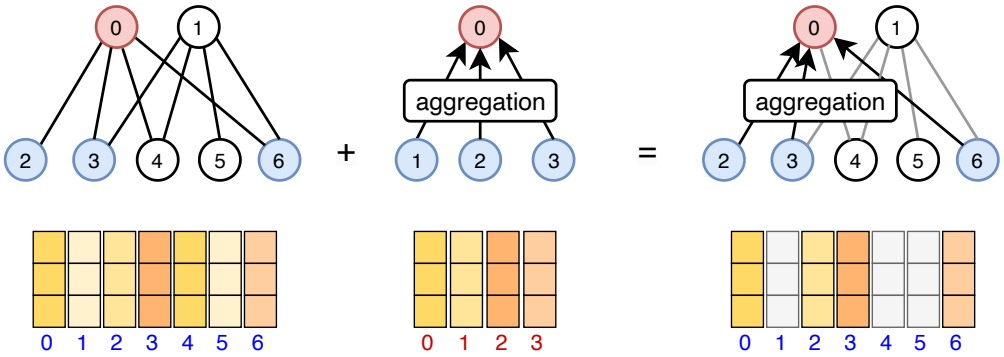


Figure 5.7: Sample-Gather-Aggregation fusion: aggregation implicitly performs a gather operation, making explicit feature gathering unnecessary.

Cross-Stage Operator Fusion

Operator fusion is a well-known optimization technique in deep learning compilers [110–112]. It works by merging multiple operations into a single kernel to reduce the overhead of kernel launching and data movement. For GNN training, most existing works focus on fusing operations within the same stage, i.e., GNN model computation [40, 53, 103, 105] or graph sampling [41], due to an isolated view of the two stages. In JOESTAR, with a unified DFG representation, we are able to perform novel cross-stage operator fusions neglected by prior works. JOESTAR proposes three types of fusions illustrated: *Sample-Gather-Aggregation*, *Gather-Update* and *Sample-Structural* fusion.

In mini-batch GNN training, a typical workflow involves sampling subgraphs, compacting their node IDs, and gathering input features from the original feature tensors. Compaction ensures that the subsequent graph computation can correctly index the gathered features. These steps are followed by the first GNN layer, which either aggregates the input features over the sampled subgraph (e.g., G.`pull`) or updates node features individually (e.g., G.`apply_nodes`). The *Sample-Gather-Aggregation fusion* (Fig. 5.7) is motivated by the observation that the aggregation step implicitly performs a gather operation on the input features, making the intermediate feature gathering redundant. Further, when the subgraph taken by the first GNN layer will not be reused later, we skip the compaction step as well. While this fusion only applies to the first GNN layer, its computation and data movement savings can be significant for node-wise sampling methods, since they recursively expand the sampled subgraphs, making the first layer the most expensive one.

Sample-Gather-Aggregation fusion is feasible only when feature aggregation is the initial operation in the target GNN model. In cases where node feature updates directly follow the sample-gather sequence, JOESTAR enables partial fusion of the sampling and computation stages through Gather-Update fusion, provided the update function operates in a node-wise

manner. A common scenario is when the GNN model applies a linear transformation to the gathered input features. To support this, JOESTAR offers specialized fused kernels, including Gather-Elementwise and Gather-GEMM, to streamline such operations.

Another frequent pattern in GNN training, the interleaving of graph computation and structural transformations, also lends itself well to cross-stage operator fusion. As an example, adding self-loops to sampled subgraphs is a common practice to mitigate numerical issues, such as division by zero. When performed during model computation, it introduces data dependencies that can stall subsequent GNN layers. JOESTAR addresses this by scheduling structural transformations early and fusing them with sampling whenever possible. We provide a fused sampling-and-adding-self-loops kernel to eliminate structural operations in model computation time (Fig. 5.8). Other scenarios that can benefit from Sample-Structural

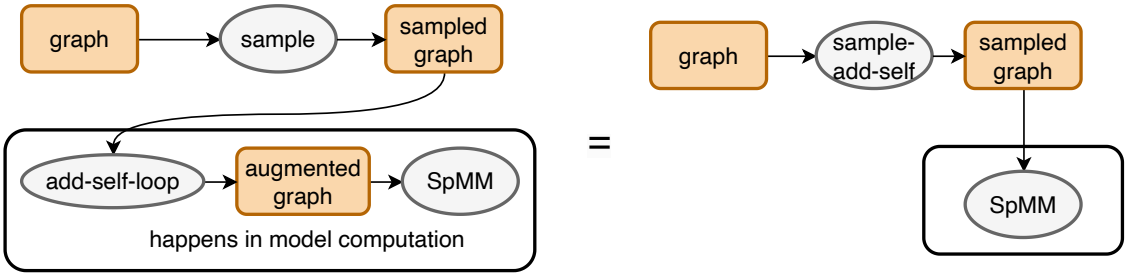


Figure 5.8: An example of Sample-Structural fusion: fusing sampling and add-self-loop kernels.

fusion include node degree calculation and graph reversal.

Lastly, JOESTAR also adopts well-known intra-stage operator fusion optimizations, such as element-wise dense tensor operator fusion, Extract-Select and Edge-Map fusion in graph sampling [41], etc.

Kernel Invocation Grouping

GNN training on GPUs is characterized by a large number of short-lived kernels, which leads to CPU boundedness and low GPU activity ratios (Section 5.1). A viable approach to this challenge is to consolidate multiple kernel invocations into a single kernel launch, thereby reducing the overhead. CUDA's CUDA Graphs [113] offer an effective mechanism to achieve this goal. However, CudaGraphs require static tensor shapes at runtime, which complicates their use in GNN training due to dynamic graph structures and sampling. Existing approaches, therefore, are restricted to full-batch training with static graphs. JOESTAR enables CudaGraphs in highly dynamic mini-batch GNN training through an interesting observation of sampling: although mini-batch graph structures keep changing, their shapes are often similar across iterations due to statistical properties of random sampling. We illustrate the

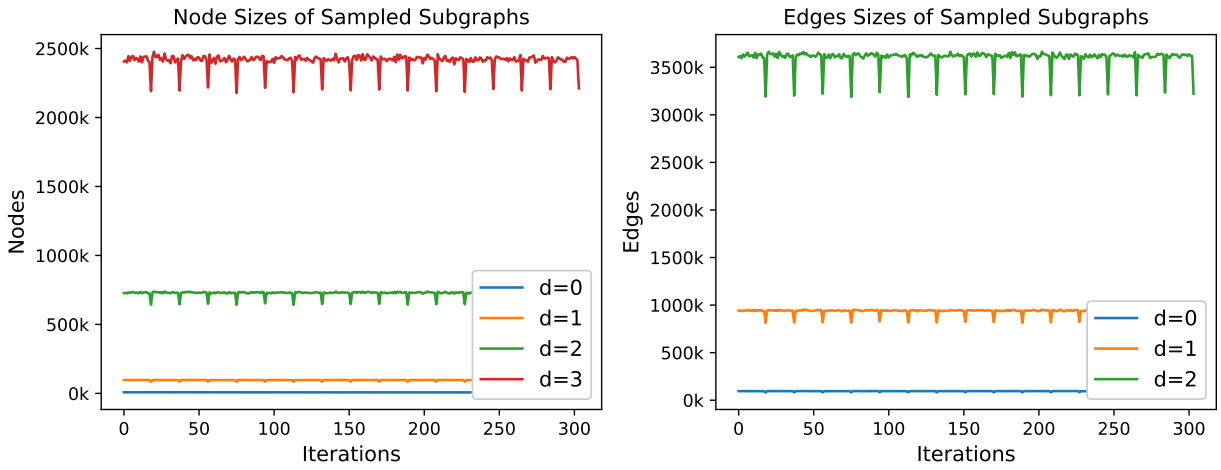


Figure 5.9: Sizes of subgraphs at different hops ($d = 0, 1, 2, 3$) generated by neighbor sampling.

observation in Fig. 5.9. It shows the sizes of sampled subgraphs produced by JOESTAR in terms of the number of sampled nodes and edges for a real-world graph dataset (`papers`) and throughout hundreds of training iterations. Per-hop node and edge statistics stay within a narrow range, with occasional drops due to smaller sizes of last mini-batches in each macro-batch step. Thus, it is feasible to pad sampled subgraphs up to pre-determined fixed sizes without excessive space or computation waste. It enables the usage of CudaGraphs to group all kernel invocations within a training iteration. We careful choose the values filled in the padded regions in CSR/CSC formats so that they do not introduce any extra computation for SpMM and SDDMM kernels. When training starts, JOESTAR dedicates the first epoch as the warm up phase to collect the upper bounds of the sampled subgraph sizes. Padding and CudaGraphs are enabled in following epochs to accelerate GPU computation. We also provide a fallback mechanism to disable CudaGraph execution when the actual sizes of sampled subgraphs exceed the pre-determined padded sizes. The optimizations significantly reduce kernel invocation overhead, decreasing the total number of kernel launches by 60%–70% for a three-layer GraphSAGE model compared to DGL and PyTorch Geometric. Notably, both the forward and backward passes of the model require only a single kernel launch each, with most remaining kernels dedicated to inherently data-dependent graph sampling.

5.3.3 Profile-guided Optimizations

On top of static compilation optimizations, JOESTAR also utilizes profile-guided optimizations to further improve GNN training performance. We opt for profiling-based methods instead of analytical cost modeling for two reasons. First, due to the diversity of GNN workloads, it is difficult to develop a one-size-fits-all cost model that accurately predicts the performance

with different input characteristics, especially in the presence of sparsity and dynamic tensor sizes. Second, the performance of GPU kernels is highly sensitive to the underlying hardware architecture, which makes analytical models hard to generalize. With runtime profiling, we can capture realistic performance and easily adapt to the specific hardware and workload. Profiling is piggybacked on the warm-up phase of GNN training, which we already have in place for collecting the statistics of data shapes.

JOESTAR focuses on two key areas for profile-guided optimizations: sparse tensor format selection and the scheduling of sparse-dense matrix multiplications. For the former, JOESTAR extends beyond existing approaches by considering the end-to-end GNN training pipeline and the impact of sparse tensor formats on both graph sampling and model computation. Notably, it introduces a novel optimization space where sparse tensors can remain in their original CSC or CSR formats without requiring mandatory format conversions. Previous approaches often necessitate such conversions, as the backward pass of SpMM typically expects transposed sparse tensors. By enabling SpMM kernels to accept any of the CSR, CSC or COO formats, JOESTAR eliminates this constraint. To determine the optimal format, JOESTAR employs a brute-force search strategy, evaluating all possible combinations and selecting the best-performing configuration. Only a limited subset of sparse tensor operators, such as SpMM, sparse tensor slicing, and sampling, are significantly influenced by format choices. The performance of most other operators remains largely format-agnostic. This constraint substantially reduces the complexity of the optimization space, making exhaustive profiling both feasible and practical.

The alternation of sparse and dense matrix multiplications in GNN models lead to another optimization opportunity. Mathematically, assuming the input feature matrix is X, the stacking of a message passing layer and another linear layer is then expressed as $Y = AXW$, where Y denotes output features, A is the sparse adjacency matrix from the graph and W is the linear weight matrix. The associativity of matrix multiplication permits two different execution schedules, i.e., $Y = (AX)W$ or $Y = A(XW)$, as demonstrated at the graph level in Fig. 5.10. The two schedules vary in computation costs and are highly dependent on

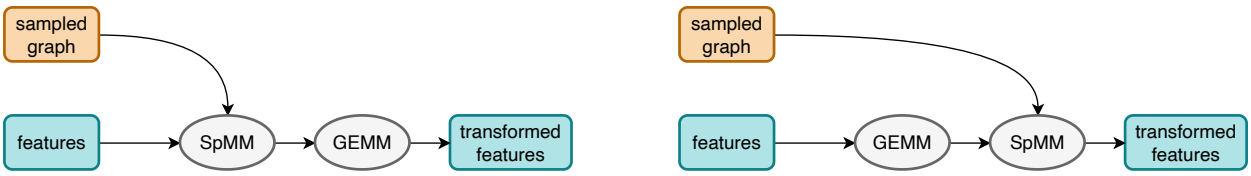


Figure 5.10: It is free to exchange orders of SpMM and GEMM operators in the DFG of GNNs.

the sparsity of A, the input dimension of X and hidden dimension of W. Prior works have proposed heuristics to select the best schedule prior to the execution [60]. In JOESTAR, we

take a more systematic approach by evaluating the costs of two schedules on the end-to-end GNN training pipeline. Remarkably, when SpMM takes precedence over GEMM in the first GNN layer, we observe that the backward pass of this SpMM can be entirely dropped in a dead code elimination (DCE) pass since it is parameter-free. Prior approaches that purely focus on the relative costs of each individual kernel overlook this optimization opportunity. In most GNN models, the sequences of sparse and dense matrix multiplications are usually isolated to each other and do not have nested structures (e.g., $A_1A_2XW_2W_1$). Thus, JOESTAR handles the optimization of each sequence individually: it identifies all occurrences of such consecutive matrix multiplications in the DFG, considers the correlation of schedule decisions between forward and backward passes and selects the best schedule for each one.

5.3.4 Implementation Details

We implement JOESTAR as a GNN library built on PyTorch, leveraging the advanced compilation capabilities introduced in PyTorch 2.0+ [110]. Using TorchDynamo as the compiler frontend, JOESTAR traces Python programs written with graph-level operators into tensor-level dataflow graphs (DFGs) represented as `torch.fx` graphs. These DFGs are then optimized by merging the sampling and model computation stages, applying the joint optimizations outlined in previous sections, and producing a streamlined, unified DFG. The optimized DFG is then processed by PyTorch's backend optimization passes and encapsulated as a Python function.

JOESTAR heavily relies on PyTorch's GPU caching allocator to reduce frequent host-device synchronizations from explicit memory allocations. To address memory fragmentation and prevent out-of-memory errors, JOESTAR fine-tunes the allocation granularity and split sizes of the caching allocator. Additionally, it includes a suite of efficient CUDA kernels for various sparse tensor operations listed in Table 5.3 and fused operators in Section 5.3.2, ensuring robust performance across diverse workloads.

5.4 Evaluation

Our evaluation compares the end-to-end training performance of JOESTAR with existing systems on real-world, large-scale GNN datasets. To identify the source of speedups, we analyze the effects of two key design choices in JOESTAR: multistage sampling and joint optimizations of sampling and model compute. We also evaluate the model quality of JOESTAR to ensure that the performance improvements do not come at the cost of accuracy.

5.4.1 Evaluation Setup

GNN Models and Datasets

Table 5.4: GNN datasets used in the evaluation.

Dataset	# Nodes	# Edges	Feature	Train Ratio	Size
`products`	2.5M	123.7M	100	0.08	2.8GB
`papers`	111M	3.2B	128	0.01	103GB
`mag240m-c`	122M	2.6B	768	0.01	214GB

We choose three representative GNN model architectures for our evaluation: GraphSAGE [9], GAT [10] and GCN [8]. GraphSAGE and GCN follow a similar graph convolutional architecture with a slight difference in aggregation functions. GAT uses a self-attention mechanism in message-passing layers and is more computationally expensive than GraphSAGE and GCN. The experiments are conducted on three large-scale datasets from the Open Graph Benchmark (OGB) [33, 83] in Table 5.4: `products`, `papers` and `mag240m-c`. These graphs are preprocessed into undirected formats by adding reverse edges, which is critical in achieving reasonable model quality. `mag240m-c` is a homogeneous citation subgraph extracted from the full heterogeneous `mag240m` dataset. We leave supporting heterogeneous graphs for future work. Neighbor sampling (NS) is the only sampling method that proves to work well for the scales of datasets we evaluate [64]. Moreover, many baseline systems only support neighbor sampling [37, 64, 75, 90, 108]. Thus, we exclusively employ NS for baselines and also in the first-stage sampling of LBS in JOESTAR. The model hyper-parameters follow the example settings from the OGB repository and stay the same for all three architectures, as shown in Table 5.5. Note that we use a universal batch size of 1000 for all experiments following previous works [67, 75, 90] for fair comparison, since larger batch sizes are easier to parallelize but can lead to slower model convergence [71].

Table 5.5: Model hyper-parameters used in the evaluation.

Dataset	Num. Layers	Hidden Dim.	Fanout	Batch Size
`products`	3	128	(15, 10, 5)	1000
`papers`	3	128	(15, 10, 5)	1000
`mag240m-c`	2	1024	(25, 15)	1000

Testbed Environment and Baselines

We conduct our experiments on two machines. The first machine, **RTX**, is a local workstation equipped with a 12-core (2-way hyper-threaded) Intel CPU and a NVIDIA RTX 4090 GPU

(24GB) memory connected with PCIe 3.0 x16. The second machine, **A10G**, is a cloud server with a 15-core (2-way hyper-threaded) Intel CPU and a NVIDIA A10G GPU (24GB) memory connected with PCIe 4.0 x16. The GPU power of RTX is higher than A10G, providing 1.68x higher memory bandwidth and 2.36x more compute throughputs in FP32 but bounded by half the PCIe bus bandwidth compared to A10G. They are on the two ends of hardware configurations in the ratio of GPU power to PCIe bus bandwidth and thus representative for a spectrum of machine learning systems.

For baseline comparisons, we primarily use DGL [30], a widely adopted framework with robust support for on-GPU GNN training. Additionally, we include SALIENT [90], a state-of-the-art GNN training system that incorporates advanced sampling and pipelining optimizations. The performance results for SALIENT are drawn directly from its original publication. Further details regarding the experimental setup and machine configurations are provided in Section 5.4.2. It is worth noting that we exclude certain recent systems, such as FastGL [103] and GNNLab [98], from our comparisons due to their unusual data preparation practices. Specifically, these systems do not preprocess graph data into undirected formats, which leads to a substantial degradation in model accuracy.

Reporting Metrics

We report the end-to-end training time per epoch, which includes the time taken for data preparation on both the CPU and GPU and model computation during training. The reported time is averaged over 5 epochs after the first epoch to account for the warm-up time due to just-in-time model compilation, profiling passes, memory allocation, etc. Additionally, we evaluate the model's quality upon convergence. In line with standard practices, we report the test accuracy corresponding to the epoch that achieves the highest validation accuracy.

Configurations for Joestar

We evaluate two variants of Joestar: Joestar-I and Joestar-II, which adopt LBS and PBS as the first-stage sampling methods, respectively. Joestar employs a two-stage sampling stragegy, thus requiring an additional set of hyper-parameters for the macro-batches generated by the first-stage sampling. The parameter choices are listed in Table 5.6 for the three datasets. For LBS, the macro-batch size refers to the ratio of training examples in the first-stage sampling to the total number of training examples. The second-stage sampling then takes a fixed number of examples (e.g., 1000) from the macro-batch. For PBS, similarly to Hanoi, we choose the number of partitions (p) used in the preprocessing step and the number of partitions (b) to to construct the macro-batches.

Table 5.6: Additional training hyperparameters for JOESTAR.

Dataset	Macro-Batch Size (PBS)	Macro-Batch Size (b,p) (LBS)
`products`	1/8	(16, 128)
`papers`	1/12	(128, 1536)
`mag240m-c`	1/24	(64, 1536)

5.4.2 End-to-End Performance

The main results for our end-to-end training experiments are shown in Fig. 5.11 and Fig. 5.12. For all runs, the datasets are stored primarily in the CPU memory. GNN sampling takes place on the CPU for DGL and SALIENT, since they do not have multistage sampling. DGL supports UVA sampling, which allows GPU to perform sampling on graph structures and access the feature data stored in the CPU memory directly during feature gathering. However, it requires a copy of graph structure data on the GPU and leads to out-of-memory errors for `papers` and `mag240m-c`, so we disable it for DGL. For SALIENT, we take the reported numbers run on two NVIDIA V100 GPUs to account for the hardware difference between RTX 4090 and V100. The aggregated memory bandwidth of two V100 GPUs is 1.8x higher than that of a single RTX 4090 (2×900GB/s versus 1008GB/s), while their combined computation throughputs in FP32 is 34% of RTX 4090. GNN model computation is known to be heavily memory-bound [103]. Thus, we view the comparison in favor of SALIENT. In addition to the two configurations of JOESTAR described in Section 5.4.1, we also include a variant of JOESTAR-I with model compilation disabled (JOESTAR-I: eager). The eager version of JOESTAR-I runs the model in the same way as SALIENT and serves as a good reference point for other variants of JOESTAR.

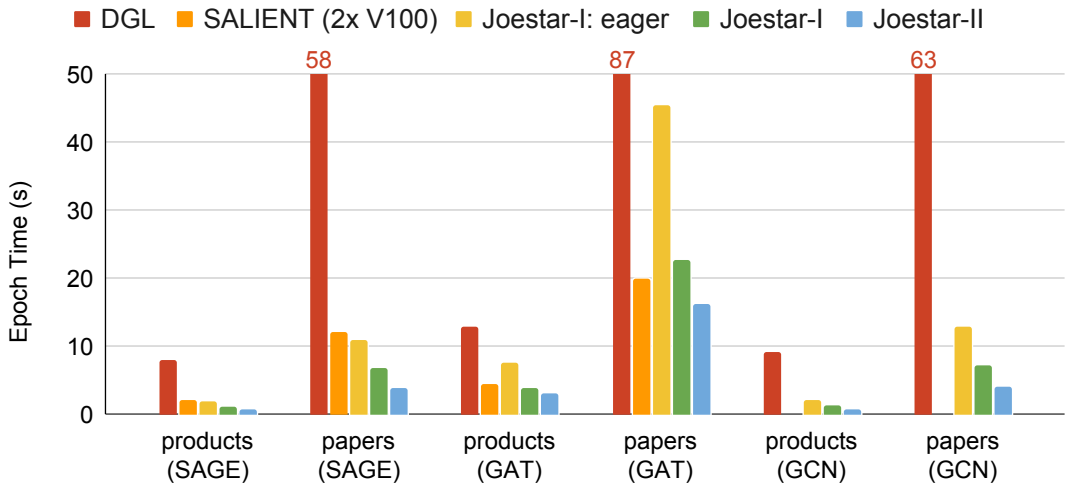


Figure 5.11: End-to-end training performance of JOESTAR and baselines on the RTX machine.

Fig. 5.11 reports the training time on RTX for `products` and `papers` (`mag240m-c` is too large to fit into the host memory of RTX). JOESTAR-I with full-fledged optimizations achieves 8.0×, 3.6× and 8.2× speedups over DGL for GraphSAGE, GAT and GCN models on average. JOESTAR-II further improves the performance, providing 13.1×, 4.8× and 14.3× speedups over DGL. The significant performance is due to the combination of multistage sampling and joint model optimizations. As we can observer, for less computationally expensive models like GraphSAGE and GCN, the performance of JOESTAR-I: eager provides up to 5.3× faster training time than DGL (GraphSAGE on `papers`). The performance gap between DGL and JOESTAR is much less for GAT, since GAT is more GPU-bound and benefits less from the reduction of bus transfer by multistage sampling.

Comparing to SALIENT, JOESTAR-I is on average 1.8× and 1.03× faster for GraphSAGE and GAT, respectively. SALIENT does not support GCN, so we exclude it from the evaluation. SALIENT reduces exposed CPU and bus overheads in GNN training with CPU sampling optimized for throughputs and a sohpisticated pipelining strategy. JOESTAR also squeezes these overheads to almost negligible levels albeit through a different approach of multisage sampling. Consequently, the observed performance differences can primarily be attributed to GPU execution efficiency between SALIENT and JOESTAR. For GPU-bound models such as GAT, SALIENT achieves per-epoch training times that are 44%–59% of those of JOESTAR-I: eager, which aligns closely with the GPU bandwidth disparity between the RTX 4090 and two V100 GPUs. By incorporating joint optimizations, JOESTAR-I achieves a 2.0× speedup over its eager variant, resulting in performance comparable to SALIENT. However, for lighter models like GraphSAGE, SALIENT struggles to fully overlap CPU and bus overheads, allowing JOESTAR-I and JOESTAR-II to consistently outperform it. Notably, JOESTAR-II reduces training time by 66% and 27% for GraphSAGE and GAT, respectively.

We run a similar set of experiments on the A10G machine, as shown in Fig. 5.12. A10G is able to handle the largest `mag240m-c` dataset. Overall, the performance advantage of JOESTAR is slightly less evident than that on RTX, since the baseline is less bottlenecked by the more significant CPU and bus transfers while more bound by GPU execution. JOESTAR-I yields 5.4×, 2.1× and 4.8× speedups over DGL for the three models on average. JOESTAR-II is consistent in reducing the training time on top of JOESTAR-I, providing another 1.5×, 2.0× and 1.6× improvements. When compared against SALIENT++ [75] with two A10G GPUs, a distributed version of SALIENT, JOESTAR-I runs 1.3× and 1.08× faster for `products` and `papers` using only one A10G when training GraphSAGE (the only model evaluated in SALIENT++). Moreover, SALIENT++ reports 7.0s/epoch on `mag240m-c` with **4× A10Gs**, while JOESTAR-II with only **1× A10G** is able to deliver a comparable throughput of 7.8s/epoch.

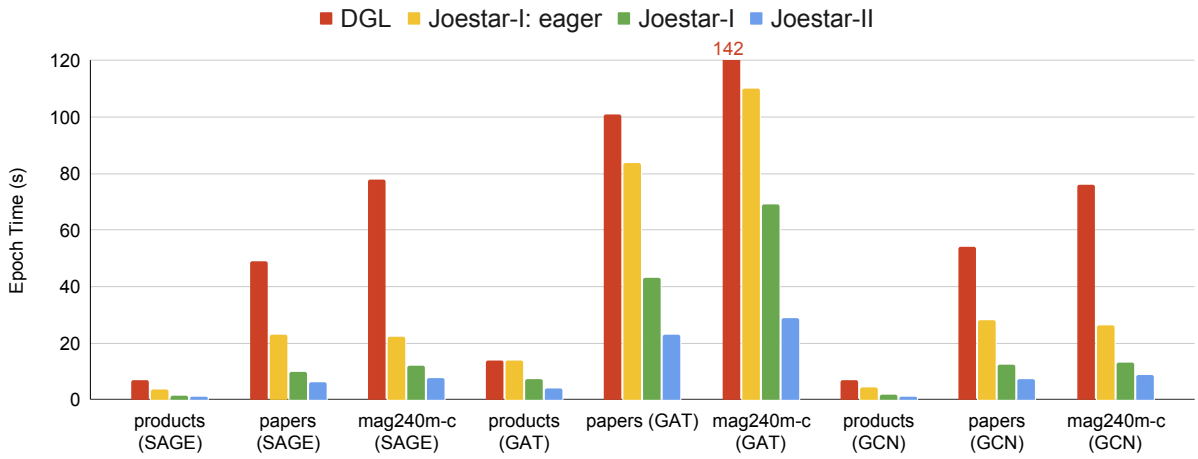


Figure 5.12: End-to-end training performance of JOESTAR and baselines on the A10G machine.

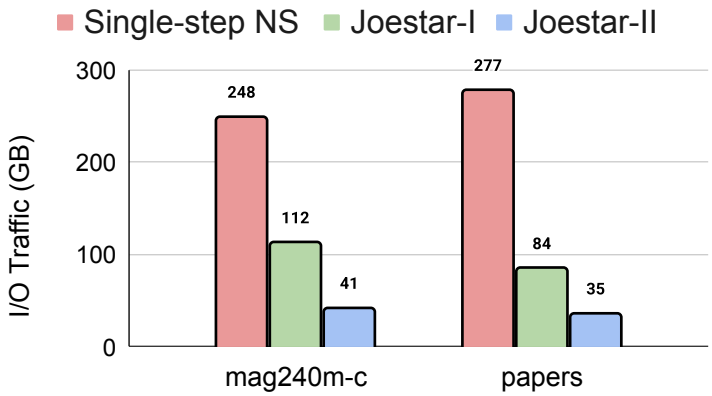


Figure 5.13: I/O traffic reduction of JOESTAR-I and JOESTAR-II.

5.4.3 Effects of Multistage Sampling

To understand the performance benefits of multistage sampling, we profile the I/O traffic in JOESTAR-I and JOESTAR-II incurred by one epoch of training. The results are shown in Fig. 5.13 for `papers` and `mag240m-c`. Single-step NS represents the practice of baseline systems like DGL and SALIENT that do not offload GNN sampling to the GPU. JOESTAR-I reduces the I/O traffic by 2.2× and 3.3× for the two datasets. It partially explains the performance improvements of eager JOESTAR-I over DGL in Fig. 5.11 and Fig. 5.12, while more efficient CPU sampling and stage pipelining contribute to the rest of the performance gain. JOESTAR-II is effetive in further reducing the I/O traffic by 2.7× and 2.4× for the two datasets. Reusing historical embeddings of halo nodes instead of recursively sampling enables the sampling routine to terminate much earlier. PBS with AoT sampling significantly

decreases working set sizes of the first-stage sampling compared to LBS.

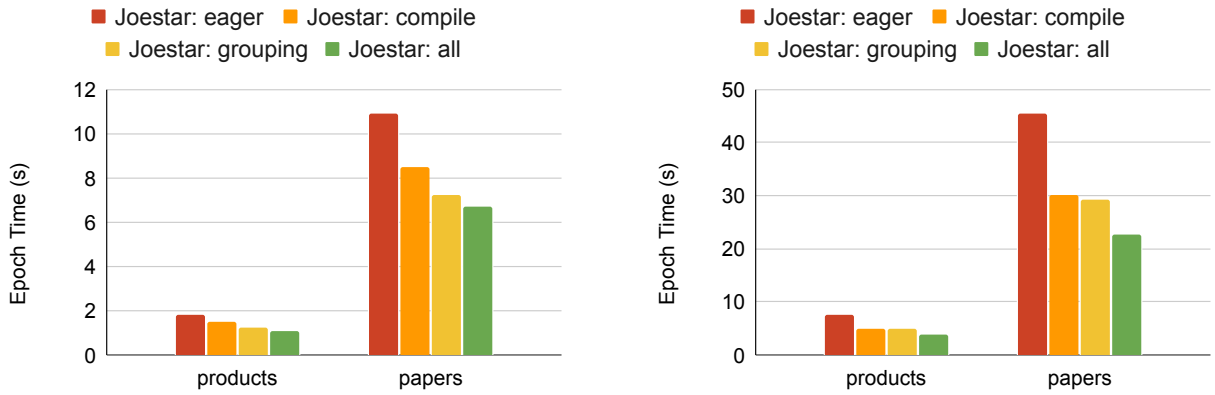
We also measure the impacts of multistage sampling on the final model quality, since an extra stage of sampling may introduce more noise and bias to the training process (Section 3.3). Table 5.7 compares the model accuracy of JOESTAR-I and JOESTAR-II with single-step sampling methods (DGL). The "target accuracy" row represents the results reported by DGL, while we list the differences in final accuracy of JOESTAR beneath it. We observe that JOESTAR-I produces excellent model quality across all datasets and models with less than 0.15% accuracy drops. The minor accuracy gap, if statistically significant at all, is likely due to the loss of randomness from multistage sampling: the second-stage sampling reuses the same subset of neighbors sampled in the first stage before the macro-batch is refreshed. For JOESTAR-II, due to another level of approximation from historical embeddings, the accuracy degration is slightly larger but still under 0.6%. Given a limited training budget, the higher throughput of JOESTAR-II allows more training experiments with different hyper-parameters to be conducted, which can potentially help to find a better model and compensate the impacts on accuracy.

Table 5.7: Impacts of multistage sampling methods in JOESTAR-I and JOESTAR-II are minor compared to single-step neighbor sampling method.

Methods	`products`		`papers`		`mag240m-c`	
	GraphSAGE	GAT	GraphSAGE	GAT	GraphSAGE	GAT
Target Accuracy	78.9	79.5	64.9	64.6	65.9	65.7
JOESTAR-I	-0.1	+0.21	-0.13	+0.04	-0.08	-0.03
JOESTAR-II	+0.21	-0.42	-0.25	-0.56	-0.29	-0.48

5.4.4 Effects of Joint Optimizations

This section provides a detailed ablation study of various optimizations implemented in JOESTAR's compilation and profiling passes. Starting from the eager version of JOESTAR, we incrementally enable optimizations such as operator fusion with PyTorch-2 compilation (JOESTAR: compile), kernel invocation grouping (JOESTAR: grouping) and profile-guided optimizations (JOESTAR: all). The eager JOESTAR is equipped with optimized kernels for graph structrual transformations, thus already achieving speedups over vanilla DGL. We run different variants of JOESTAR with GraphSAGE and GAT, and demonstrate the effects of optimizations in Fig. 5.14. The improvements contributed by each optimization components direclty depend on the model architecture while less sensitive to the dataset of choice. When GNN models are relatively light in computation, such as GraphSAGE, all three optimization techniques contribute to final performance gains together. Model



(a) GraphSAGE

(b) GAT

Figure 5.14: Effects of joint optimizations in JOESTAR on the end-to-end training time.

compilation provides the most sensible speedup by optimizing away the feature gathering and fusing some dense tensor operations while CudaGraphs reduces 15%–20% of the overall overhead due to kernel invocations. The benefit of kernel invocation grouping is much less significant for GAT, which is more GPU-bound by longer-running kernels. Instead, the model compilation and profile-guided optimizations have more pronounced impacts on its performance. The former removes graph structural operations (e.g., adding self-loops and graph reversal) from model computation, enabling more aggressive fusion of intensive kernel operations. During profile-guided optimizations, JOESTAR discover the default behavior of updating features before aggregation in GAT layers to be computationally sub-optimal. It delays the matrix multiplication in feature updates to a later stage after the attention mechanism and feature aggregation, which saves substantial amounts of computation due to the shrinking structures of subgraphs sampled by neighbor sampling. This optimization relies on data statistics specific to the sampling algorithms, which is inherently difficult to capture with purely compilation-based approaches.

Fig. 5.15 performs a detailed runtime breakdown of sampling, feature gathering and model computation time for one training iteration of GraphSAGE. This experiment uses `products` for its relatively small size, allowing us to fit the entire dataset in the GPU memory. Therefore, we are able to run DGL entirely on the GPU to demonstrate its performance bottlenecks and compare it with variants of JOESTAR. Simply by substituting graph structural operations and message passing layers in DGL with JOESTAR's customized implementations, we achieve a substantial reduction in both sampling and model computation time. Operator fusion and kernel invocation grouping further remove feature gathering overheads and bring a 2.2× speedup in model computation. Lastly, our profile-guided optimizations capitalize on remaining opportunities for operator reordering and tensor format choices, though the

performance enhancement in this particular scenario is more modest.

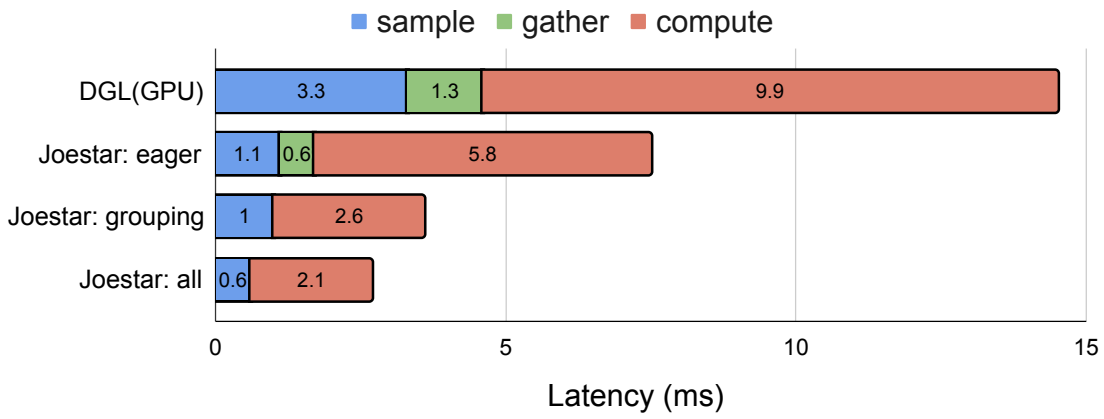


Figure 5.15: Breakdown of end-to-end training time for different optimization components in JOESTAR.

5.5 Conclusion

In this chapter, we presented JOESTAR, a GPU-centric framework designed for training GNNs on large-scale graphs exceeding the GPU memory capacity. JOESTAR employs multistage sampling to minimize PCIe data transfers while harnessing GPU hardware for sampling operations. A key innovation of JOESTAR is the co-location of sampling and model computation on the GPU, enabling novel optimization techniques that holistically address graph sampling, feature gathering, and model computation. Through comprehensive system- and kernel-level optimizations, JOESTAR establishes a new performance standard for GNN training on a single GPU, processing billion-edge graph datasets in under 10 seconds per epoch with negligible impact on model accuracy (less than 0.5%).

Chapter 6

Conclusion and Future Work

Scaling GNN training to large-scale, real-world datasets remains a fundamental challenge limiting their broader adoption and application. Achieving this scalability with both cost-effectiveness and computational efficiency would democratize access for academic researchers with constrained resources while simultaneously offering industry practitioners better training throughput per computational dollar invested. Current GNN training systems typically operate under the strict requirement of maintaining the entire dataset in either CPU or GPU memory to ensure acceptable performance. Extending GNN training to larger and more economical memory tiers presents significant challenges due to the inherently irregular data access patterns associated with graph sampling and the bandwidth limitations of lower memory hierarchies.

This thesis presents a novel approach to addressing this challenge through the introduction of *multistage sampling*. Multistage sampling encourages I/O traffic reduction and enhances data reuse by decomposing fine-grained sampling operations into multiple stages with adaptive degrees of granularity. Based on this methodological framework, this thesis investigates two complementary systems, HANOI and JOESTAR, engineered to address the constraints of limited host memory and GPU memory, respectively.

1. HANOI (Chapter 4) explores the design space of out-of-core GNN training through a comprehensive algorithm-system co-design approach. Prior solutions either compromise model quality for efficiency or depend heavily on host memory caching for acceptable performance. HANOI establishes an improved design frontier by striking a better balance between computational throughputs and model accuracy through its accuracy-aware multistage sampling design. It delivers training throughput and accuracy comparable to in-memory systems while utilizing only a fraction of the host memory, enabling single-node training to scale to datasets substantially larger than those currently explored.

2. JOESTAR (Chapter 5) streamlines single-node in-memory GNN training by eliminating CPUs from the system critical path and optimizing GPU hardware utilization. By leveraging multistage sampling principles, it offloads the majority of sampling operations to the GPU, which excels at high-throughput fine-grained sampling. Through compilation and profiling-based techniques, JOESTAR jointly optimizes GPU-based sampling and model computation. This approach reveals optimization opportunities overlooked in existing systems, including cross-stage operator fusion and profiling-guided end-to-end training optimization. JOESTAR achieves state-of-the-art performance on billion-scale datasets with a single GPU.

Looking ahead, we envision several promising avenues for future research. First, the methodologies taken by the proposed systems differ from the existing line of GNN training works, which mostly focus on various forms of caching and kernel-level optimizations. We believe the techniques proposed in the thesis are largely orthogonal to those works and can be combined with them to further improve the performance. Particularly, the integration of caching and multistage sampling can be a promising direction in out-of-core GNN training, where the storage I/O remains one of the major bottlenecks. It would also be interesting to explore automatic operator fusion of the diverse sparse tensor operators in GNN training. Currently, most of the fused kernels are implemented manually, which hinders the extensibility of the system and optimization space a compiler can explore. Second, the proposed systems are mainly designed for single-node training. While the extension to distributed data parallel training through data replication is straightforward, it remains an open question how to best combine data partitioning and multistage sampling. Other parallelization strategies, such as model parallelism and pipeline parallelism, are also promising directions for future work. Finally, we believe the multistage sampling framework can be extended to other domains beyond GNNs, such as non-graph based recommendation systems and even more general machine learning workloads where data preparation represents a significant portion of the training time. As machine learning models continue to grow in the demand of training data, multistage sampling can be an effective approach in enabling more cost-effictive storage hardware without sacrificing data ingestion performance and model accuracy.

References

[1] D. Easley, J. Kleinberg, et al. *Networks, crowds, and markets: Reasoning about a highly connected world*. Vol. 1. Cambridge university press Cambridge, 2010.

[2] D. Chakrabarti and C. Faloutsos. "Graph mining: Laws, generators, and algorithms". In: *ACM computing surveys (CSUR)* 38.1 (2006), 2–es.

[3] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive data sets*. Cambridge university press, 2020.

[4] L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Stanford InfoLab, Nov. 1999. URL: http://ilpubs.stanford.edu:8090/422/.

[5] P. Ribeiro, P. Paredes, M. E. Silva, D. Aparicio, and F. Silva. "A survey on subgraph counting: concepts, algorithms, and applications to network motifs and graphlets". In: *ACM computing surveys (csur)* 54.2 (2021), pp. 1–36.

[6] W. L. Hamilton. "Graph Representation Learning". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14.3, pp. 1–159.

[7] W. L. Hamilton, R. Ying, and J. Leskovec. "Representation learning on graphs: Methods and applications". In: *arXiv preprint arXiv:1709.05584* (2017).

[8] T. Kipf and M. Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *International Conference on Learning Representations*. ICLR '16. 2016.

[9] W. L. Hamilton, R. Ying, and J. Leskovec. "Inductive Representation Learning on Large Graphs". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS '17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 1025–1035. ISBN: 9781510860964.

[10] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. "Graph Attention Networks". In: *International Conference on Learning Representations*. ICLR '18. 2018. URL: https://openreview.net/forum?id=rJXMpikCZ.

[11] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. "How Powerful are Graph Neural Networks?" In: *International Conference on Learning Representations*. ICLR '19. 2019. URL: https://openreview.net/forum?id=ryGs6iA5Km.

[12] M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li. "Simple and Deep Graph Convolutional Networks". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by H. D. III and A. Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 1725–1735.

[13] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka. "Representation Learning on Graphs with Jumping Knowledge Networks". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by J. Dy and A. Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 5453–5462. URL: https://proceedings.mlr.press/v80/xu18c.html.

[14] T. N. Kipf and M. Welling. *Variational Graph Auto-Encoders*. 2016. arXiv: 1611.07308 [stat.ML]. URL: https://arxiv.org/abs/1611.07308.

[15] M. Zhang and Y. Chen. "Link prediction based on graph neural networks". In: *Advances in neural information processing systems* 31 (2018).

[16] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling. "Modeling relational data with graph convolutional networks". In: *The semantic web: 15th international conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, proceedings 15*. Springer. 2018, pp. 593–607.

[17] Z. Hu, Y. Dong, K. Wang, and Y. Sun. "Heterogeneous graph transformer". In: *Proceedings of the web conference 2020*. 2020, pp. 2704–2710.

[18] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. "Neural message passing for quantum chemistry". In: *International conference on machine learning*. PMLR. 2017, pp. 1263–1272.

[19] X. Zhu, Z. Ghahramani, and J. Lafferty. "Semi-Supervised Learning Using Gaussian Fields and Harmonic Functions". In: *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*. ICML'03. Washington, DC, USA: AAAI Press, 2003, pp. 912–919. ISBN: 1577351894.

[20] A. Grover and J. Leskovec. "Node2vec: Scalable Feature Learning for Networks". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 855–864. ISBN: 9781450342322. DOI: 10.1145/2939672.2939754. URL: https://doi.org/10.1145/2939672.2939754.

[21] B. Perozzi, R. Al-Rfou, and S. Skiena. "DeepWalk: Online Learning of Social Representations". In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '14. New York, New York, USA: Association for Computing Machinery, 2014, pp. 701–710. ISBN: 9781450329569. DOI: 10.1145/2623330.2623732. URL: https://doi.org/10.1145/2623330.2623732.

[22] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. "LINE: Large-Scale Information Network Embedding". In: *Proceedings of the 24th International Conference on World Wide Web*. WWW '15. Florence, Italy: International World Wide Web Conferences Steering Committee, 2015, pp. 1067–1077. ISBN: 9781450334693. DOI: 10.1145/2736277.2741093. URL: https://doi.org/10.1145/2736277.2741093.

[23] C. Gao et al. "A Survey of Graph Neural Networks for Recommender Systems: Challenges, Methods, and Directions". In: *ACM Trans. Recomm. Syst.* 1.1 (Mar. 2023). DOI: 10.1145/3568022. URL: https://doi.org/10.1145/3568022.

[24] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. 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. London, United Kingdom: Association for Computing Machinery, 2018, pp. 974–983. ISBN: 9781450355520. DOI: 10.1145/3219819.3219890. URL: https://doi.org/10.1145/3219819.3219890.

[25] A. Derrow-Pinion, J. She, D. Wong, O. Lange, T. Hester, L. Perez, M. Nunkesser, S. Lee, X. Guo, B. Wiltshire, et al. "Eta prediction with graph neural networks in google maps". In: *Proceedings of the 30th ACM international conference on information & knowledge management*. 2021, pp. 3767–3776.

[26] F. Borisyuk, S. He, Y. Ouyang, M. Ramezani, P. Du, X. Hou, C. Jiang, N. Pasumarthy, P. Bannur, B. Tiwana, et al. "Lignn: Graph neural networks at linkedin". In: *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2024, pp. 4793–4803.

[27] H. Yang. "AliGraph: A Comprehensive Graph Neural Network Platform". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 3165–3166. ISBN: 9781450362016. DOI: 10.1145/3292500.3340404. URL: https://doi.org/10.1145/3292500.3340404.

[28] R. Lam, A. Sanchez-Gonzalez, M. Willson, P. Wirnsberger, M. Fortunato, F. Alet, S. Ravuri, T. Ewalds, Z. Eaton-Rosen, W. Hu, et al. "Learning skillful medium-range global weather forecasting". In: *Science* 382.6677 (2023), pp. 1416–1421.

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[30] M. Wang et al. "Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs". In: *International Conference on Learning Representations*. ICLR '19. 2019.

[31] M. Fey and J. E. Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.

[32] N. Corporation. *cuSPARSE*. https://docs.nvidia.com/cuda/cusparse/. 2007-2025.

[33] W. Hu, M. Fey, H. Ren, M. Nakata, Y. Dong, and J. Leskovec. "Ogb-lsc: A large-scale challenge for machine learning on graphs". In: *arXiv preprint arXiv:2103.09430* (2021).

[34] A. Khatua, V. S. Mailthody, B. Taleka, T. Ma, X. Song, and W.-m. Hwu. "IGB: Addressing The Gaps In Labeling, Features, Heterogeneity, and Size of Public Graph Datasets for Deep Learning Research". In: *In Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23)*. KDD '23. 2023. DOI: 10.48550/ARXIV.2302.13522. URL: https://arxiv.org/abs/2302.13522.

[35] Y. Lee, J. Chung, and M. 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. New York, New York: Association for Computing Machinery, 2022, pp. 932–945. ISBN: 9781450386104. DOI: 10.1145/3470496.3527391. URL: https://doi.org/10.1145/3470496.3527391.

[36] Y. Park, S. Min, and J. W. Lee. "Ginex: SSD-enabled Billion-scale Graph Neural Network Training on a Single Machine via Provably Optimal In-memory Caching". In: *Proceedings of the VLDB Endowment*. Vol. 15. 11. 2022.

[37] R. Waleffe, J. Mohoney, T. Rekatsinas, and S. Venkataraman. "MariusGNN: Resource-Efficient Out-of-Core Training of Graph Neural Networks". In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23. Rome, Italy: Association for Computing Machinery, 2023, pp. 144–161. ISBN: 9781450394871. DOI: 10.1145/3552326.3567501. URL: https://doi.org/10.1145/3552326.3567501.

[38] Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang. "FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[39] Z. Xie, M. Wang, Z. Ye, Z. Zhang, and R. Fan. "Graphiler: Optimizing graph neural networks with message passing data flow graph". In: *Proceedings of Machine Learning and Systems* 4 (2022), pp. 515–528.

[40] K. Wu, M. Hidayetoğlu, X. Song, S. Huang, D. Zheng, I. Nisa, and W.-m. Hwu. "Hector: An efficient programming and compilation framework for implementing relational graph neural networks in GPU architectures". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2024, pp. 528–544.

[41] P. Gong, R. Liu, Z. Mao, Z. Cai, X. Yan, C. Li, M. Wang, and Z. Li. "gSampler: General and Efficient GPU-based Graph Sampling for Graph Learning". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, pp. 562–578. ISBN: 9798400702297. DOI: 10.1145/3600006.3613168. URL: https://doi.org/10.1145/3600006.3613168.

[42] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. "Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc". In: *Proceedings of the 3rd Conference on Machine Learning and Systems (MLSys)*. Mar. 2020.

[43] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding. "GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs". In: *15th USENIX symposium on operating systems design and implementation (OSDI 21)*. 2021.

[44] L. Bottou, F. E. Curtis, and J. Nocedal. "Optimization Methods for Large-Scale Machine Learning". In: *SIAM Review* 60.2 (2018), pp. 223–311.

[45] J. Chen, J. Zhu, and L. Song. "Stochastic Training of Graph Convolutional Networks with Variance Reduction". In: *International Conference on Machine Learning*. 2018, pp. 941–949.

[46] J. Chen, T. Ma, and C. Xiao. "FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=rytstxWAW.

[47] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu. "Layer-Dependent Importance Sampling for Training Deep and Large Graph Convolutional Networks". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/91ba4a4478a66bee9812b0804b6f9d1b-Paper.pdf.

[48] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh. "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*. KDD '19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 257–266. ISBN: 9781450362016. DOI: 10.1145/3292500.3330925. URL: https://doi.org/10.1145/3292500.3330925.

[49] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna. "GraphSAINT: Graph Sampling Based Inductive Learning Method". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=BJe8pkHFwS.

[50] M. Fey, J. E. Lenssen, F. Weichert, and J. Leskovec. "GNNAutoScale: Scalable and expressive graph neural networks via historical embeddings". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 3294–3304.

[51] A. Tripathy, K. Yelick, and A. Buluç. "Reducing Communication in Graph Neural Network Training". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[52] J. Peng, Z. Chen, Y. Shao, Y. Shen, L. Chen, and J. Cao. "Sancus: Staleness-Aware Communication-Avoiding Full-Graph Decentralized Training in Large-Scale Graph Neural Networks". In: *Proc. VLDB Endow.* 15.9 (May 2022), pp. 1937–1950. ISSN: 2150-8097. URL: https://doi.org/10.14778/3538598.3538614.

[53] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen. "Understanding and Bridging the Gaps in Current GNN Performance Optimizations". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 119–132. ISBN: 9781450382946. DOI: 10.1145/3437801.3441585. URL: https://doi.org/10.1145/3437801.3441585.

[54] Y. Wang, B. Feng, Z. Wang, T. Geng, K. Barker, A. Li, and Y. Ding. "{MGG}: Accelerating graph neural networks with {Fine-Grained}{Intra-Kernel}{Communication-Computation} pipelining on {Multi-GPU} platforms". In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 2023, pp. 779–795.

[55] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha. "DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3480856. URL: https://doi.org/10.1145/3458817.3480856.

[56] Z. Gong, H. Ji, Y. Yao, C. W. Fletcher, C. J. Hughes, and J. Torrellas. "Graphite: optimizing graph neural networks on CPUs through cooperative software-hardware techniques". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022, pp. 916–931. ISBN: 9781450386104. DOI: 10.1145/3470496.3527403. URL: https://doi.org/10.1145/3470496.3527403.

[57] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. "Hygcn: A gcn accelerator with hybrid architecture". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 15–29.

[58] H. Zeng and V. Prasanna. "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms". In: *proceedings of the 2020 ACM/SIGDA international symposium on field-programmable gate arrays*. 2020, pp. 255–265.

[59] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, et al. "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 922–936.

[60] J. Li, A. Louri, A. Karanth, and R. Bunescu. "GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 775–788.

[61] H. You, T. Geng, Y. Zhang, A. Li, and Y. Lin. "Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design". In: *2022 IEEE In-*

ternational Symposium on High-Performance Computer Architecture (HPCA). IEEE. 2022, pp. 460–474.

[62] C. Zheng, H. Chen, Y. Cheng, Z. Song, Y. Wu, C. Li, J. Cheng, H. Yang, and S. Zhang. "ByteGNN: Efficient Graph Neural Network Training at Large Scale". In: *Proc. VLDB Endow.* 15.6 (Feb. 2022), pp. 1228–1242. ISSN: 2150-8097. DOI: 10.14778/3514061.3514069. URL: https://doi.org/10.14778/3514061.3514069.

[63] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[64] K. Huang, H. Jiang, M. Wang, G. Xiao, D. Wipf, X. Song, Q. Gan, Z. Huang, J. Zhai, and Z. Zhang. "FreshGNN: Reducing Memory Access via Stable Historical Embeddings for Graph Neural Network Training". In: *Proceedings of the VLDB Endowment* 17.6 (2024), pp. 1473–1486.

[65] X. Zhang, Y. Shen, Y. Shao, and L. Chen. "DUCATI: A dual-cache training system for graph neural networks on giant graphs with the GPU". In: *Proceedings of the ACM on Management of Data* 1.2 (2023), pp. 1–24.

[66] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu. "Pagraph: Scaling gnn training on large graphs via computation-aware caching". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 401–415.

[67] J. Dong, D. Zheng, L. F. Yang, and G. Karypis. "Global Neighbor Sampling for Mixed CPU-GPU Training on Giant Graphs". In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. KDD '21. Virtual Event, Singapore: Association for Computing Machinery, 2021, pp. 289–299. ISBN: 9781450383325. DOI: 10.1145/3447548.3467437. URL: https://doi.org/10.1145/3447548.3467437.

[68] M. Ramezani, W. Cong, M. Mahdavi, A. Sivasubramaniam, and M. Kandemir. "GCN meets GPU: Decoupling "When to Sample" from "How to Sample"". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 18482–18492. URL: https://proceedings.neurips.cc/paper/2020/file/d714d2c5a796d5814c565d78dd16188d-Paper.pdf.

[69] L. Xu et al. "In-Database Machine Learning with CorgiPile: Stochastic Gradient Descent without Full Data Shuffle". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 1286–1300. ISBN: 9781450392495. DOI: 10.1145/3514221.3526150. URL: https://doi.org/10.1145/3514221.3526150.

[70] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis. "Distdgl: distributed graph neural network training for billion-scale graphs". In: *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE. 2020, pp. 36–44.

[71] D. Zheng, X. Song, C. Yang, D. LaSalle, and G. 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*. KDD '22. Washington DC, USA: Association for Computing Machinery, 2022, pp. 4582–4591. ISBN: 9781450393850. DOI: 10.1145/3534678.3539177. URL: https://doi.org/10.1145/3534678.3539177.

[72] S. Gandhi and A. P. Iyer. "P3: Distributed Deep Graph Learning at Scale". In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021, pp. 551–568. ISBN: 978-1-939133-22-9. URL: https://www.usenix.org/conference/osdi21/presentation/gandhi.

[73] D. Zhang, X. Huang, Z. Liu, J. Zhou, Z. Hu, X. Song, Z. Ge, L. Wang, Z. Zhang, and Y. Qi. "AGL: A Scalable System for Industrial-Purpose Graph Machine Learning". In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pp. 3125–3137. ISSN: 2150-8097. DOI: 10.14778/3415478.3415539. URL: https://doi.org/10.14778/3415478.3415539.

[74] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo. "BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 103–118. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/liu-tianfeng.

[75] T. Kaler, A.-S. Iliopoulos, P. Murzynowski, T. B. Schardl, C. E. Leiserson, and J. Chen. "Communication-Efficient Graph Neural Networks with Probabilistic Neighborhood Expansion Analysis and Caching". In: *Proceedings of Machine Learning and Systems* (2023).

[76] M. Ramezani, W. Cong, M. Mahdavi, M. T. Kandemir, and A. Sivasubramaniam. "Learn Locally, Correct Globally: A Distributed Algorithm for Training Graph Neural Networks". In: *The International Conference on Learning Representations*. 2022.

[77] J. B. Park, V. S. Mailthody, Z. Qureshi, and W.-m. Hwu. "Accelerating sampling and aggregation operations in gnn frameworks with gpu initiated direct storage accesses". In: *arXiv preprint arXiv:2306.16384* (2023).

[78] J. Mohoney, R. Waleffe, H. Xu, T. Rekatsinas, and S. Venkataraman. "Marius: Learning Massive Graph Embeddings on a Single Machine". In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021, pp. 533–549. ISBN: 978-1-939133-22-9. URL: https://www.usenix.org/conference/osdi21/presentation/mohoney.

[79] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. "Chaos: Scale-out graph processing from secondary storage". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015, pp. 410–424.

[80] X. Zhu, W. Han, and W. Chen. "{GridGraph}:{Large-Scale} graph processing on a single machine using 2-level hierarchical partitioning". In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 2015, pp. 375–386.

[81] K. Vora, G. Xu, and R. Gupta. "Load the Edges You Need: A Generic I/O Optimization for Disk-Based Graph Processing". In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '16. Denver, CO, USA: USENIX Association, 2016, pp. 507–522. ISBN: 9781931971300.

[82] Y. Lu, S. Y. Meng, and C. De Sa. "A General Analysis of Example-Selection for Stochastic Gradient Descent". In: *ICLR*. 2022.

[83] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. "Open Graph Benchmark: Datasets for Machine Learning on Graphs". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 22118–22133. URL: https://proceedings.neurips.cc/paper/2020/file/fb60d411a5c5b72b2e7d3527cfc84fd0-Paper.pdf.

[84] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann. *Pitfalls of Graph Neural Network Evaluation*. 2019. arXiv: 1811.05868 [cs.LG].

[85] M. McPherson, L. Smith-Lovin, and J. M. Cook. "Birds of a Feather: Homophily in Social Networks". In: *Annual Review of Sociology* 27.1 (2001), pp. 415–444. DOI: 10.1146/annurev.soc.27.1.415. eprint: https://doi.org/10.1146/annurev.soc.27.1.415. URL: https://doi.org/10.1146/annurev.soc.27.1.415.

[86] G. Karypis and V. Kumar. "A fast and high quality multilevel scheme for partitioning irregular graphs". In: *SIAM Journal on scientific Computing* 20.1 (1998), pp. 359–392.

[87] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. "FENNEL: Streaming Graph Partitioning for Massive Scale Graphs". In: *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. WSDM '14. New York, New York, USA: Association for Computing Machinery, 2014, pp. 333–342. ISBN: 9781450323512. DOI: 10.1145/2556195.2556213. URL: https://doi.org/10.1145/2556195.2556213.

[88] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov. "Streaming Graph Partitioning: An Experimental Study". In: *Proc. VLDB Endow.* 11.11 (July 2018), pp. 1590–1603. ISSN: 2150-8097. DOI: 10.14778/3236187.3236208. URL: https://doi.org/10.14778/3236187.3236208.

[89] M. F. Faraj and C. Schulz. "Buffered Streaming Graph Partitioning". In: *ACM J. Exp. Algorithmics* 27 (Oct. 2022). ISSN: 1084-6654. DOI: 10.1145/3546911. URL: https://doi.org/10.1145/3546911.

[90] T. Kaler, N. Stathas, A. Ouyang, A.-S. Iliopoulos, T. Schardl, C. E. Leiserson, and J. Chen. "Accelerating Training and Inference of Graph Neural Networks with Fast Sampling and Pipelining". In: *Proceedings of Machine Learning and Systems*. Ed. by D. Marculescu, Y. Chi, and C. Wu. Vol. 4. 2022, pp. 172–189. URL: https://proceedings.mlsys.org/paper/2022/file/35f4a8d465e6e1edc05f3d8ab658c551-Paper.pdf.

[91] M. F. Balin and Ü. Çatalyürek. "Layer-Neighbor Sampling—Defusing Neighborhood Explosion in GNNs". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 25819–25836.

[92] L. Hoang, X. Chen, H. Lee, R. Dathathri, G. Gill, and K. Pingali. "Efficient Distribution for Deep Learning on Large Graphs". In: *Proceedings of the Workshop on Graph Neural Networks and Systems*. 2021. URL: https://chenxuhao.github.io/docs/gnnsys-2021.pdf.

[93] A. Kyrola, G. Blelloch, and C. Guestrin. "Graphchi: Large-scale graph computation on just a {PC}". In: *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 2012, pp. 31–46.

[94] F. McSherry, M. Isard, and D. G. Murray. "Scalability! But at What Cost?" In: *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*. HOTOS'15. Switzerland: USENIX Association, 2015, p. 14.

[95] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. "Mosaic: Processing a Trillion-Edge Graph on a Single Machine". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: Association for Computing Machinery, 2017, pp. 527–543. ISBN: 9781450349383. DOI: 10.1145/3064176.3064191. URL: https://doi.org/10.1145/3064176.3064191.

[96] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. "Squeezing out All the Value of Loaded Data: An out-of-Core Graph Processing System with Reduced Disk I/O". In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '17. Santa Clara, CA, USA: USENIX Association, 2017, pp. 125–137. ISBN: 9781931971386.

[97] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. "PyTorch-BigGraph: A large scale graph embedding system". In: *Proceedings of Machine Learning and Systems* 1 (2019), pp. 120–131.

[98] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou. "GNNLab: a factored system for sample-based GNN training over GPUs". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 417–434.

[99] Y. Wang, B. Feng, and Y. Ding. "QGTC: accelerating quantized graph neural networks via gpu tensor core". In: *Proceedings of the 27th ACM SIGPLAN symposium on principles and practice of parallel programming*. 2022, pp. 107–119.

[100] S. W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W.-m. Hwu. "Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture". In: *Proc. VLDB Endow.* 14.11 (July 2021), pp. 2087–2100. ISSN: 2150-8097. DOI: 10.14778/3476249.3476264. URL: https://doi.org/10.14778/3476249.3476264.

[101] L. Dhulipala, G. E. Blelloch, and J. Shun. "Theoretically efficient parallel graph algorithms can be fast and scalable". In: *ACM Transactions on Parallel Computing (TOPC)* 8.1 (2021), pp. 1–70.

[102] N. Corporation. *Unified Addressing*. Mar. 2025. URL: https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__UNIFIED.html (visited on 03/31/2025).

[103] Z. Zhu, P. Wang, Q. Hu, G. Li, X. Liang, and J. Cheng. "FastGL: A GPU-Efficient Framework for Accelerating Sampling-Based GNN Training at Large Scale". In: *arXiv preprint arXiv:2409.14939* (2024).

[104] A. Jangda, S. Polisetty, A. Guha, and M. Serafini. "Accelerating graph sampling for graph machine learning using GPUs". In: *Proceedings of the sixteenth European conference on computer systems*. 2021, pp. 311–326.

[105] H. Zhang, Z. Yu, G. Dai, G. Huang, Y. Ding, Y. Xie, and Y. Wang. "Understanding gnn computational graph: A coordinated computation, io, and memory perspective". In: *Proceedings of Machine Learning and Systems* 4 (2022), pp. 467–484.

[106] RAPIDS. *cuGraph*. 2025. URL: https://docs.rapids.ai/api/cugraph/stable (visited on 03/31/2025).

[107] M. F. Balin, D. LaSalle, and Ü. V. Çatalyürek. *Cooperative Minibatching in Graph Neural Networks*. 2024. arXiv: 2310.12403 [cs.LG]. URL: https://arxiv.org/abs/2310.12403.

[108] Z. Sheng, W. Zhang, Y. Tao, and B. Cui. "OUTRE: An OUT-of-Core De-REdundancy GNN Training Framework for Massive Graphs within A Single Machine". In: *Proc. VLDB Endow.* 17.11 (July 2024), pp. 2960–2973. ISSN: 2150-8097. DOI: 10.14778/3681954.3681976. URL: https://doi.org/10.14778/3681954.3681976.

[109] *ParlayLib: A Toolkit for Programming Parallel Algorithms on Shared-Memory Multicore Machines*. May 2025. URL: https://cmuparlay.github.io/parlaylib (visited on 05/16/2025).

[110] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, et al. "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2024, pp. 929–947.

[111] N. Corporation. *NVIDIA TensorRT*. https://docs.nvidia.com/tensorrt/index.html. 2021-2025.

[112] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. "{TVM}: An automated {End-to-End} optimizing compiler for deep learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594.

[113] *CUDA C++ Programming Guide*. May 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-graphs (visited on 05/16/2025).