

Graph Sampling and GPU Kernel Co-Design for Scalable GNN Training

Cloe Balaguer

Department of Computer Science, Colorado State University, Fort Collins, CO, USA.
cloe.work@colostate.edu

Faas van Liendert

Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS, USA.
faas1992@ku.edu

Abstract

Graph neural networks have become a central computational approach for learning over relational data, but their training pipelines remain difficult to scale because they combine irregular memory access, rapidly expanding neighborhoods, and tightly coupled sampling and execution stages. This paper examines graph sampling and GPU kernel co-design as an integrated systems strategy for scalable graph neural network training. Rather than treating sampling as a preprocessing utility or GPU kernels as isolated optimization targets, the analysis argues that scalability emerges from the joint design of graph traversal policies, memory hierarchies, workload balancing, and deployment governance. The paper first reviews foundational graph neural network architectures and identifies the structural bottlenecks that arise when neighborhood expansion is mapped onto parallel hardware. It then evaluates major sampling families, including layer-wise importance sampling, clustering, and subgraph sampling, with attention to their statistical consequences and infrastructure costs. The discussion next turns to GPU kernel co-design, considering sparse aggregation, feature gathering, fusion opportunities, and memory-centric scheduling. It also explores how distributed deployment, multi-tenant serving, and sustainability constraints reshape the design space. Robustness and fairness are treated as system-level properties rather than post hoc corrections, because sampling policies and kernel scheduling can amplify or suppress particular regions of a graph. The paper concludes that scalable graph neural network training requires an interdisciplinary architecture in which algorithmic sampling, hardware-aware execution, and institutional governance co-evolve. Such co-design can improve throughput and energy efficiency while also making training behavior more predictable, auditable, and equitable.

Keywords

graph neural networks, graph sampling, GPU kernels, scalable training, systems co-design, memory hierarchy, distributed infrastructure, fairness, sustainability.

1. Introduction

Graph neural networks have moved from a specialized model family to a general infrastructure for relational learning in recommendation, fraud detection, chemistry, transportation, and knowledge representation. Foundational work established that message passing over graph structure can learn representations sensitive to topology and node attributes [1]. Subsequent architectures such as graph convolutional networks simplified

spectral-inspired propagation into neighborhood aggregation [2], while inductive methods enabled generalization to unseen nodes through learned aggregators [3]. Attention-based variants introduced adaptive weighting across neighbors [4], and theoretical work clarified the expressive limits and structural sensitivities of message-passing models [5]. Together, these developments created a powerful but computationally demanding class of models whose training behavior is shaped as much by data movement and scheduling as by model mathematics.

The central difficulty is that graph neural network training is not a conventional dense tensor workload. Neighborhood expansion creates irregular access patterns, variable fan-out, and repeated gathers across high-dimensional features. A single training step may involve sampling, indexing, copying, aggregation, normalization, gradient computation, and synchronization across devices. As graphs grow to billions of edges, the dominant cost shifts from arithmetic to data movement, coordination, and memory pressure. This shift makes isolated improvements insufficient. A faster matrix multiplication kernel may deliver little benefit if sampling produces poorly balanced batches or if feature gathering saturates memory bandwidth. Conversely, an elegant sampling algorithm may fail in production if it cannot be mapped efficiently onto GPUs or distributed workers.

This paper argues that graph sampling and GPU kernel co-design should be understood as a unified systems problem. Sampling determines which subgraphs, nodes, and edges enter the training path. GPU kernels determine how those inputs are transformed, aggregated, and returned as gradients. When sampling is designed without regard to hardware behavior, it can create skewed workloads, scattered memory accesses, and synchronization stalls. When kernels are designed without regard to sampling semantics, they may optimize a local operation while degrading statistical quality or increasing variance. Co-design treats sampling as part of the memory hierarchy and treats kernel scheduling as part of the learning policy.

The discussion is organized around several system-level concerns. It examines foundational bottlenecks, sampling families, GPU kernel strategies, deployment architecture, robustness, fairness, sustainability, and policy implications. The goal is not to present a single optimal pipeline but to map the structural trade-offs that arise when graph learning is deployed at scale. These trade-offs include throughput versus statistical fidelity, locality versus coverage, centralized control versus federated autonomy, and energy efficiency versus model quality. The paper concludes that scalable graph neural network training is a socio-technical infrastructure challenge, not merely an algorithmic or hardware problem.

2. Foundations and Scalability Bottlenecks in Graph Neural Network Training

Early graph neural network formulations established iterative message passing as a general mechanism for structured data [1]. Graph convolutional networks then demonstrated that localized aggregation could be trained efficiently on semi-supervised tasks [2]. Inductive approaches replaced full-graph training with neighborhood sampling and learned aggregation functions, making it possible to generalize to unseen nodes [3]. Attention mechanisms allowed models to weight neighbors according to learned relevance [4], while expressiveness studies showed that standard message passing has limitations in distinguishing certain graph structures [5]. These foundations matter for systems design because each architectural choice changes memory access, branching, and aggregation patterns.

The first scalability bottleneck is neighborhood explosion. In a multi-layer graph neural network, the receptive field of a single node expands exponentially with depth when full

neighborhoods are used. This expansion increases memory consumption and makes mini-batch construction expensive. Even moderate depth can require gathering features from a large fraction of the graph. The problem is not only computational cost but also irregularity. Nodes have different degrees, so some batches contain dense hubs while others contain sparse leaf regions. GPU threads assigned to different nodes may therefore finish at very different times, reducing utilization and increasing tail latency.

The second bottleneck is feature movement. Graph neural network layers often combine sparse adjacency operations with dense feature transformations. Sparse aggregation gathers neighbor features, while dense transformations apply learned weights. On GPUs, the sparse phase may underutilize arithmetic units because it is dominated by indirect memory accesses. The dense phase may be efficient but depends on the feature layout produced by sampling. If sampled nodes are scattered across memory, feature gathering becomes bandwidth-bound. If they are clustered, dense operations become more efficient but may reduce coverage or introduce bias. This tension between locality and representativeness is central to co-design.

The third bottleneck is synchronization and orchestration. Training pipelines must coordinate sampling workers, feature stores, GPU kernels, gradient reducers, and parameter servers or collective communication libraries. Each stage may have different latency characteristics. A sampling stage optimized for throughput may produce batches too slowly for the GPU, while a GPU kernel optimized for large batches may increase memory peaks and force smaller samples. Distributed training adds network communication and straggler effects. The result is a pipeline whose end-to-end performance cannot be predicted from component benchmarks alone. Frameworks such as PyTorch Geometric and Deep Graph Library provide abstractions and optimized primitives [10], [11], while platforms such as AliGraph illustrate the complexity of production graph learning infrastructure [12]. These systems show that scalability is achieved through coordinated design rather than isolated kernel improvements.

3. Graph Sampling as a Structural and Statistical Interface

Graph sampling is often described as a way to reduce computation, but it is more accurately a structural and statistical interface between the graph and the training system. Sampling decides which nodes and edges become visible to the model, how gradients are estimated, and what memory access patterns the GPU will encounter. FastGCN introduced importance sampling to reduce variance in layer-wise graph convolution [6]. Layer-dependent importance sampling further refined this idea by adapting sampling distributions across depth [7]. Cluster-GCN partitioned graphs into dense subgraphs to improve locality and reduce neighborhood expansion [8]. GraphSAINT used subgraph sampling with normalization to maintain unbiasedness and improve training efficiency [9]. These approaches differ not only in statistical properties but also in the hardware behavior they induce.

From a systems perspective, sampling policies determine the shape of the workload. Node-wise sampling produces many small gathers and high indexing overhead. Layer-wise sampling can control fan-out but may create dependencies between layers that complicate pipelining. Clustering creates dense blocks that map well to GPU caches and matrix operations but can disconnect the sampled subgraph from global structure. Subgraph sampling balances coverage and locality by selecting connected regions, yet it may require expensive preprocessing and partitioning. Each choice affects batch construction latency, memory footprint, and the amount of inter-worker communication in distributed settings.

Sampling also determines fairness and robustness properties. If a sampler consistently over-represents high-degree nodes, the model may become biased toward hubs and neglect peripheral communities. If it under-samples rare classes or minority groups, predictive performance may degrade for those populations. If sampling is randomized without stratification, variance may be acceptable on average but unstable for small communities. These concerns are not separate from performance. In practice, fairness-aware sampling may require more memory, more communication, or more complex indexing. Conversely, aggressive locality-oriented sampling may improve throughput while quietly reducing coverage. A co-designed system must therefore expose sampling decisions as tunable governance parameters rather than hidden implementation details.

The infrastructure implications are substantial. Sampling may be performed on CPU workers, GPU-resident structures, or a hybrid pipeline. CPU sampling is flexible and can use mature graph libraries, but it introduces host-to-device transfer overhead. GPU sampling reduces transfer costs but requires dynamic memory allocation, compaction, and load balancing. Distributed sampling introduces partitioning, replication, and consistency concerns. In large deployments, the sampler becomes a service with its own latency, failure modes, and observability requirements. Treating it as a mere preprocessing step obscures these operational realities. A mature architecture must log sampling distributions, measure batch skew, and allow operators to trace how sampling choices affect model quality and resource consumption.

4. GPU Kernel Co-Design and Memory-Centric Execution

GPU kernel co-design begins with the recognition that graph neural network training is memory-centric. Sparse aggregation, feature gathering, and gradient scattering all stress the memory hierarchy. Early accelerator work such as HyGCN explored hybrid architectures for graph convolution [15], while AWB-GCN addressed runtime workload imbalance through rebalancing [16]. GNNAdvisor demonstrated adaptive runtime support for GNN acceleration on GPUs [17]. Later kernel-level work has pushed further by redesigning the relationship between graph structure and GPU execution, including extreme fast kernel designs for training [18]. These efforts show that improving graph neural network performance requires more than porting dense neural network kernels to graph data.

A key co-design principle is fusion. Rather than materializing sampled subgraphs, copying features, and launching separate kernels for aggregation and transformation, a fused kernel can load neighbor features, apply transformations, aggregate, and update node representations within a tighter memory scope. Fusion reduces intermediate memory traffic and kernel launch overhead. However, it also increases register pressure, complicates dynamic branching, and may reduce occupancy. The optimal fusion depth depends on graph degree distribution, feature dimension, and GPU architecture. A sampler that produces uniform block sizes may enable deeper fusion, while a sampler that produces variable fan-out may require adaptive kernel variants.

Another principle is workload balancing. Graph neighborhoods are irregular, and GPU threads or warps assigned to different nodes may have very different amounts of work. Techniques such as degree-aware scheduling, warp-level aggregation, and dynamic work stealing can improve utilization. But balancing interacts with sampling. If the sampler emits highly skewed batches, the kernel must compensate with more complex scheduling. If the sampler emits highly uniform batches, kernel design becomes simpler but may sacrifice structural diversity. Co-design therefore involves negotiating between sampling regularity and

execution regularity. The goal is not perfect uniformity but predictable bounded variance that the kernel can handle efficiently.

Memory layout is equally important. Compressed sparse row formats are common, but they may not be optimal for all aggregation patterns. Blocked formats can improve cache locality for clustered subgraphs, while edge-centric layouts may benefit certain attention or message-passing schemes. Feature layout also matters. Row-major features favor dense transformations, while column-major or tiled layouts may improve gather efficiency for certain dimensions. These choices must be evaluated together with sampling. A clustered sample may favor blocked sparse formats, while a random sample may favor simpler compressed formats with high parallelism. The broader lesson from optimized attention kernels and transformer data movement studies is that hardware-aware layout and fusion can dominate nominal arithmetic counts [19], [20]. Graph neural networks exhibit even more irregularity, making this lesson especially relevant.

GPU kernel co-design also has deployment consequences. Specialized kernels may be tuned for a particular GPU generation, graph distribution, or feature dimension. In heterogeneous clusters, the same training job may run across different devices. A co-designed system must either compile multiple kernel variants and dispatch dynamically or accept reduced performance on unsupported hardware. It must also manage memory fragmentation caused by dynamic sampling. Furthermore, kernel-level optimizations can reduce energy consumption by avoiding unnecessary data movement, which is increasingly important for sustainable infrastructure. Yet these optimizations may be opaque to operators. If a fused kernel changes numerical behavior or reduces determinism, it can complicate debugging, auditing, and reproducibility. Therefore, kernel co-design must include observability and configuration control, not only raw speed.

5. System Architecture, Deployment, and Infrastructure Governance

At production scale, graph neural network training is a distributed system. Graphs may be partitioned across machines, features stored in external databases, and sampling performed by dedicated workers. Frameworks such as PyTorch Geometric and Deep Graph Library provide programming abstractions and optimized kernels [10], [11], while platforms such as AliGraph illustrate the need for comprehensive graph management, storage, and computation services [12]. The architecture must decide where graph structure lives, where features are cached, and where gradients are aggregated. These decisions affect latency, fault tolerance, cost, and privacy.

A common pattern is partitioned training with parameter servers or collective communication. Each worker samples a local partition, computes gradients, and exchanges updates. If partitions are poorly balanced, some workers become stragglers. If high-degree nodes span partitions, communication increases. Graph partitioning must therefore consider both cut size and feature locality. Sampling can mitigate these problems by selecting subgraphs that align with partitions, but it may also reduce global coverage. Governance mechanisms are needed to monitor partition quality and adjust sampling boundaries over time. In multi-tenant environments, resources are shared across teams and applications. A training job that aggressively samples and caches features may starve other jobs or inflate memory pressure. Infrastructure governance must define quotas, priorities, and fairness policies across tenants.

Data governance is equally important. Graph data often contains sensitive relationships. Sampling may expose edges or attributes to workers that would not otherwise see them.

Differential privacy, access control, and secure aggregation become part of the systems design. If sampling is randomized for privacy, it may increase variance and require more training steps, creating a trade-off between privacy and efficiency. If sampling is deterministic for reproducibility, it may leak information through repeated access patterns. Policy makers and platform operators must therefore treat sampling configuration as a governed artifact. Logs should record sampling distributions, access patterns, and model performance across subpopulations without exposing raw sensitive data.

Deployment also raises the question of centralized versus federated architecture. Centralized graph training simplifies global sampling and partitioning, but it concentrates data and computational power. Federated or edge-assisted training can preserve locality and reduce data movement, but it complicates neighborhood expansion because edges may cross administrative boundaries. In such settings, sampling must respect network topology and policy constraints. A co-designed system may use hierarchical sampling, where local subgraphs are sampled at the edge and global summaries are exchanged periodically. This approach can improve privacy and bandwidth efficiency, but it may introduce staleness and bias. The architecture must therefore balance statistical fidelity against governance requirements.

Sustainability is another governance dimension. Large-scale graph training consumes substantial energy through GPU computation, memory traffic, and network transfer. Kernel fusion, sampling efficiency, and partition locality can reduce energy per training step, but they may also increase model development complexity. Organizations need metrics that connect system efficiency to useful model outcomes. A model that trains faster but requires more hyperparameter searches may not be sustainable overall. Carbon-aware scheduling can shift flexible training jobs to times or regions with cleaner energy, but this requires infrastructure that can checkpoint, migrate, and resume graph training pipelines. Because graph sampling state and kernel caches are complex, checkpointing is not trivial. Sustainability must therefore be designed into the training architecture from the beginning.

6. Robustness, Fairness, and Sustainability Under Scale

Robustness in graph neural network training is not only about handling hardware failures. It is about maintaining stable learning behavior when data distributions shift, partitions fail, or sampling introduces variance. Graph data is often non-stationary in production. New users, transactions, or relationships change degree distributions and community structure. A sampler tuned for yesterday's graph may become inefficient or biased tomorrow. Robust systems must monitor graph drift, adapt sampling rates, and detect degradation in model quality. They must also handle stragglers and partial failures. The tail at scale problem is directly relevant because large distributed systems experience rare but costly latency spikes that dominate end-to-end performance [22]. Graph training pipelines are especially vulnerable because a single slow partition can delay gradient synchronization across all workers.

Fairness is intertwined with sampling and kernel scheduling. If sampling over-represents dense regions, learned representations may favor well-connected nodes and communities. If kernel scheduling prioritizes large batches, minority nodes may be grouped into small, delayed batches. If feature caching favors frequently accessed nodes, peripheral communities may experience higher latency and lower freshness. These effects can compound over training. Fairness-aware graph learning therefore requires system-level transparency. Operators should be able to measure representation quality across demographic or structural subgroups, trace how sampling distributions affect those measures, and adjust policies accordingly. The

machine learning fairness literature provides concepts for measuring bias and mitigating disparate impact [21], but applying them to graph systems requires new observability and control mechanisms.

There is also a robustness-fairness trade-off. Aggressive rebalancing may improve throughput by making batches uniform, but it may also amplify majority structures if uniformity is achieved by dropping rare edges. Privacy-preserving sampling may protect sensitive relationships but increase variance for small groups. Energy-aware scheduling may shift jobs to low-carbon regions but increase network latency and reduce freshness for latency-sensitive applications. These trade-offs cannot be resolved by a single technical metric. They require governance processes that include domain experts, affected communities, and infrastructure operators. In this sense, scalable graph neural network training is a socio-technical system in which technical defaults encode policy choices.

Sustainability must be considered alongside robustness and fairness. A system that is fast but fragile may consume more energy through retries and debugging. A system that is fair but inefficient may be abandoned in favor of less equitable alternatives. The most sustainable architecture is therefore one that is predictable, observable, and adaptable. Kernel co-design can reduce energy per operation, sampling design can reduce redundant computation, and governance can prevent wasteful over-provisioning. However, these benefits depend on measuring the right quantities. Throughput alone is insufficient. Operators need metrics for energy per useful gradient, variance per sampled batch, coverage per community, and tail latency per tenant. These metrics connect system performance to social and environmental outcomes.

7. Forward-Looking Perspectives and Policy Implications

The next generation of graph neural network systems will likely move toward adaptive co-design. Instead of fixing a sampler and a kernel ahead of time, the system may observe graph structure, hardware state, and model quality during training, then reconfigure sampling rates, kernel variants, and partition layouts dynamically. This vision resembles autotuning and runtime adaptation in high-performance computing, but it adds statistical and governance constraints. Adaptation must not destabilize learning or violate privacy policies. It must be explainable to operators and auditable by regulators. The technical challenge is to build control loops that are fast enough to improve efficiency but conservative enough to preserve correctness and fairness.

Policy implications follow from this interdependence. First, infrastructure transparency matters. Organizations deploying graph neural networks should document sampling policies, kernel assumptions, and hardware dependencies because these choices affect model behavior. Second, fairness auditing should include system-level factors such as sampling coverage, latency distribution, and feature freshness. Third, sustainability reporting should consider not only training energy but also the energy cost of data movement, retries, and inefficient sampling. Fourth, privacy regulation should recognize that graph structure itself can reveal sensitive information, so sampling and caching policies must be governed accordingly. Fifth, reproducibility requires that kernel optimizations and sampling randomness be recorded in sufficient detail for independent evaluation.

These policy concerns are not external to engineering. They shape the design space. For example, a requirement to audit sampling distributions may favor samplers that are easy to instrument, even if they are slightly slower. A requirement to serve multiple tenants fairly

may favor shared infrastructure with strong isolation and quota enforcement. A requirement to reduce carbon emissions may favor checkpointable pipelines that can pause and resume. In each case, the optimal technical design depends on governance objectives. Therefore, researchers and practitioners should treat governance as a first-class design input rather than a compliance burden added after deployment.

The most promising path forward is interdisciplinary co-design. Graph theorists can contribute sampling methods that preserve structural properties. Systems researchers can design GPU kernels and memory hierarchies that match those methods. Machine learning scientists can evaluate statistical fidelity and robustness. Policy scholars and ethicists can help define fairness and accountability criteria. Infrastructure operators can provide feedback on deployment constraints. This collaboration is difficult because each field uses different metrics and timescales. Yet the complexity of graph neural network training at scale makes single-discipline optimization insufficient. The systems that succeed will be those that align algorithmic sampling, hardware execution, and institutional governance around shared objectives.

8. Conclusion

Scalable graph neural network training cannot be achieved by optimizing sampling or GPU kernels in isolation. Sampling determines the statistical and structural properties of each training batch, while GPU kernels determine how efficiently those batches are executed. Their interaction shapes throughput, memory pressure, energy consumption, robustness, and fairness. This paper has argued for a co-design perspective that treats sampling as part of the memory hierarchy and kernel scheduling as part of the learning policy. It reviewed foundational architectures and bottlenecks, examined sampling families and their systems implications, analyzed GPU kernel strategies, and discussed deployment, governance, robustness, fairness, and sustainability.

The central conclusion is that scalability is an architectural property of the entire pipeline. Frameworks and platforms provide essential abstractions [10], [11], [12], and specialized kernels demonstrate the value of hardware-aware execution [15], [16], [17], [18]. Attention and transformer optimization studies reinforce the importance of data movement and fusion [19], [20]. Fairness and robustness research reminds us that system design has social consequences [21], [22]. Future progress will require adaptive, observable, and governable systems that can reconfigure sampling and execution as graphs, hardware, and policies change. Such systems will not eliminate trade-offs, but they can make those trade-offs explicit, measurable, and subject to deliberate choice. That is the foundation for scalable, sustainable, and equitable graph learning infrastructure.

References

1. Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1), 61-80.
2. Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.
3. Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems* (pp. 1024-1034).
4. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2018). Graph attention networks. In *International Conference on Learning Representations*.

5. Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2019). How powerful are graph neural networks? In International Conference on Learning Representations.
6. Chen, J., Ma, T., & Xiao, C. (2018). FastGCN: Fast learning with graph convolutional networks via importance sampling. In International Conference on Learning Representations.
7. Zou, D., Hu, Z., Wang, Y., Jiang, S., Sun, Y., & Gu, Q. (2019). Layer-dependent importance sampling for training deep and large graph convolutional networks. In Advances in Neural Information Processing Systems (pp. 11249-11259).
8. Chiang, W. L., Liu, X., Si, S., Li, Y., Bengio, S., & Hsieh, C. J. (2019). Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (pp. 257-266).
9. Zeng, H., Zhou, H., Srivastava, A., Kannan, R., & Prasanna, V. (2020). GraphSAINT: Graph sampling based inductive learning method. In International Conference on Learning Representations.
10. Fey, M., & Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. In ICLR Workshop on Representation Learning on Graphs and Manifolds.
11. Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, A., Zhou, J., Huang, Q., Ma, Y., Liu, T., & Li, J. (2019). Deep Graph Library: Towards efficient and scalable deep learning on graphs. In ICLR Workshop on Representation Learning on Graphs and Manifolds.
12. Zhu, R., Zhao, K., Yang, H., Lin, W., Zhou, C., Ai, B., Li, Y., & Li, J. (2019). AliGraph: A comprehensive graph neural network platform. Proceedings of the VLDB Endowment, 12(12), 2094-2105.
13. Jia, Z., Lin, S., Gao, R., Zaharia, M., & Aiken, A. (2020). Improving the accuracy, scalability, and performance of graph neural networks with ROC. Proceedings of Machine Learning and Systems, 2, 187-198.
14. Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., & Dai, Y. (2019). NeuGraph: Parallel deep neural network computation on large graphs. In 2019 USENIX Annual Technical Conference (pp. 443-458).
15. Yan, M., Deng, L., Hu, X., Liang, L., Feng, Y., Ma, X., Ye, H., & Xie, Y. (2020). HyGCN: A GCN accelerator with hybrid architecture. In 2020 IEEE International Symposium on High Performance Computer Architecture (pp. 15-29).
16. Geng, T., Li, A., Shi, R., Wu, C., Wang, T., Li, Y., Haghi, P., Tumeo, A., Ding, C., & Herbordt, M. (2020). AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (pp. 922-936).
17. Wang, Y., Feng, B., Li, G., Li, S., Xie, Y., & Ding, Y. (2021). GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In 15th USENIX Symposium on Operating Systems Design and Implementation (pp. 515-531).
18. Peng, H., Xie, X., Shivdikar, K., Hasan, M. A., Zhao, J., Huang, S., ... & Ding, C. (2024, April). Maxk-gnn: Extremely fast gpu kernel design for accelerating graph neural

networks training. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (pp. 683-698).

19. Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In Advances in Neural Information Processing Systems (pp. 16344-16359).
20. Ivanov, A., Dryden, N., Ben-Nun, T., Li, S., & Hoefler, T. (2021). Data movement is all you need: A case study on optimizing transformers. Proceedings of Machine Learning and Systems, 3, 711-732.
21. Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K., & Galstyan, A. (2021). A survey on bias and fairness in machine learning. ACM Computing Surveys, 54(6), 1-35.
22. Dean, J., & Barroso, L. A. (2013). The tail at scale. Communications of the ACM, 56(2), 74-80.