

Runtime Scheduling for Efficient Graph Neural Network Workloads on GPUs

Rowan C. Coyne

Department of Computer Science, University of North Texas, Denton, TX, USA.
rowan.work@unt.edu

Przemyslaw Maciuszek

Department of Computer Science, Binghamton University, Binghamton, NY, USA.
przemyslaw1979@binghamton.edu

Raffaella Pometta

School of Electrical Engineering and Computer Science, Oregon State University, Corvallis,
OR, USA.
raffaella.work@oregonstate.edu

Adina Kitzmann

Department of Computer Science, University of New Hampshire, Durham, NH, USA.
adina1991@unh.edu

Abstract

Graph neural networks have become a central computational method for learning over relational, networked, and structured data, yet their execution on graphics processing units remains persistently difficult to optimize. Unlike dense tensor workloads, graph neural network computation is shaped by irregular adjacency, sparse feature access, dynamic batching, and highly variable neighborhood expansion. These properties make runtime scheduling a first-order systems concern rather than a secondary implementation detail. This paper examines runtime scheduling for efficient graph neural network workloads on graphics processing units from a system-level perspective. It analyzes the structural tension between latency-sensitive inference and throughput-oriented training, between memory capacity and compute utilization, and between centralized cluster governance and edge deployment. The discussion considers kernel design, multi-stream execution, memory management, graph partitioning, and service-level orchestration as interdependent layers. It further examines infrastructure implications for multi-tenant platforms, observability, reproducibility, and policy compliance. Sustainability and robustness are treated as design constraints, not post hoc concerns, because scheduling choices directly influence energy consumption, carbon intensity, failure recovery, and fairness across users and applications. The paper argues that future graph neural network systems require adaptive, topology-aware, and governance-aware runtimes that can balance performance with accountability. Such runtimes must coordinate hardware, software, and institutional policy to support efficient, resilient, and equitable graph learning at scale. The conclusion identifies research directions for scheduling abstractions that remain portable across hardware generations and deployment environments.

Keywords

graph neural networks; runtime scheduling; GPU systems; systems infrastructure; sustainable computing; fairness; governance; robustness.

1. Introduction

Graph neural networks have emerged as a powerful family of machine learning models for data whose underlying structure is relational rather than grid-like. Early graph neural network formulations established the idea that node representations can be learned through repeated message passing over graph topology [4]. Subsequent work introduced spectral and spatial convolution approximations that made graph learning more practical on modern accelerators [1], [5]. Attention-based aggregation further improved representational flexibility by allowing nodes to weight neighbors differently [2], while inductive methods enabled generalization to unseen nodes and entirely new subgraphs [3]. These methodological advances have moved graph neural networks from small benchmark datasets toward industrial recommendation, fraud detection, drug discovery, traffic forecasting, and knowledge graph reasoning [6]. As a result, graph neural network workloads now sit at the intersection of machine learning, high-performance computing, and distributed systems.

Graphics processing units have been the primary execution platform for this progress because they offer high throughput for dense linear algebra. However, graph neural networks do not behave like standard dense neural networks. Their computation is driven by graph topology, which is often sparse, skewed, and dynamically changing. Feature aggregation may require irregular memory access, and neighborhood sampling can produce highly variable amounts of work per node or batch. These characteristics create runtime scheduling problems that cannot be solved solely by improving individual kernels. A runtime must decide how to partition graphs, how to batch nodes and edges, how to order memory transfers, how to overlap communication with computation, and how to allocate shared graphics processing unit resources among competing workloads. The scheduling policy therefore determines not only speed but also predictability, fairness, energy use, and fault tolerance.

System-level research on graph neural networks has produced frameworks such as Deep Graph Library and PyTorch Geometric, which abstract graph operations while relying on underlying tensor runtimes such as PyTorch and TensorFlow [13], [14], [15], [16]. These frameworks have made graph neural network development more accessible, but they also expose a gap between high-level graph semantics and low-level execution. A graph-centric operation may map to many possible kernel configurations, memory layouts, and communication patterns. Without runtime scheduling intelligence, performance can vary dramatically across graph sizes, degree distributions, and hardware topologies. This paper argues that efficient graph neural network workloads on graphics processing units require a runtime layer that treats graph structure, hardware state, service objectives, and governance constraints as joint inputs to scheduling.

The paper is organized as follows. Section 2 examines workload characteristics and the resulting runtime challenges. Section 3 discusses architectural foundations and system constraints. Section 4 analyzes scheduling strategies and structural trade-offs. Section 5 considers infrastructure, deployment, and governance. Section 6 explores sustainability, robustness, fairness, and policy implications. Section 7 concludes with forward-looking research directions. The central claim is that runtime scheduling is not merely an optimization technique but a socio-technical mechanism that mediates between computational efficiency and institutional responsibility.

2. Workload Characteristics and Runtime Scheduling Challenges

Graph neural network workloads differ from dense neural network workloads in ways that directly affect graphics processing unit scheduling. The first difference is irregularity. In a dense neural network, matrix multiplication produces predictable memory access and arithmetic intensity. In a graph neural network, each node aggregates information from a variable number of neighbors. High-degree nodes can dominate local computation, while low-degree nodes leave execution units underutilized. This imbalance is not incidental; it is a property of real-world graphs such as social networks, citation networks, and transaction graphs. Surveys of graph neural networks emphasize that scalability and efficiency remain persistent challenges because graph structure resists uniform parallelization [7], [8]. A runtime scheduler must therefore account for degree distribution, feature dimension, and edge sparsity when assigning work to graphics processing unit threads and blocks.

The second difference is memory behavior. Graph neural network training often requires storing node features, edge indices, intermediate embeddings, gradients, and optimizer states. For large graphs, the working set may exceed graphics processing unit memory, forcing out-of-core execution or distributed training. Even when the graph fits, irregular access patterns can reduce effective bandwidth. Memory management techniques developed for large language models, such as paged attention and memory pooling, illustrate how runtime systems can reduce fragmentation and improve utilization [17]. Graph neural networks can benefit from similar ideas, but the unit of management is not only sequence blocks; it is neighborhoods, subgraphs, and feature partitions. A scheduler must decide which subgraphs to keep resident, which to evict, and when to prefetch. These decisions interact with batching and kernel launch order.

The third difference is dynamism. Many graph neural network applications operate on graphs that change over time. New edges appear, old edges decay, and node features update continuously. Training may be incremental, and inference may need to reflect the latest graph state. Dynamic graphs make static scheduling insufficient because the optimal partition or batch size at one moment may be poor at the next. Runtime systems must monitor graph drift and adapt scheduling policies accordingly. This need for adaptation connects graph neural network scheduling to broader work on online serving systems, where latency, throughput, and freshness must be balanced under uncertainty [18]. In graph settings, freshness may refer to the recency of neighborhood information, while latency may refer to the time required to produce an embedding for a recommendation or fraud alert.

The fourth difference is multi-tenancy. In production environments, graphics processing units are often shared across multiple models, users, and experiments. A graph neural network training job may compete with inference services, data preprocessing, and other machine learning workloads. Scheduling policies must therefore address isolation, priority, and fairness. If one large graph training job monopolizes memory bandwidth, co-located inference jobs may suffer latency spikes. If inference requests receive absolute priority, training throughput may collapse. These trade-offs are not purely technical; they reflect organizational policies about which workloads matter and which users bear the cost of contention. Runtime scheduling thus becomes a site of governance, even when it is implemented in low-level system software.

3. Architectural Foundations and System-Level Constraints

Graphics processing units are organized around streaming multiprocessors, warp schedulers, shared memory, caches, and high-bandwidth memory. Efficient graph neural network execution requires mapping graph operations onto this architecture without assuming

regularity. Kernel designers have explored sparse-aware formats, warp-level cooperation, and memory coalescing to improve graph aggregation. Recent kernel-level research demonstrates that extremely fast graphics processing unit kernels for graph neural network training can be achieved by reorganizing sparse operations and controlling memory access patterns [12]. Such advances are important, but they do not eliminate the need for runtime scheduling. A kernel may be fast for a given partition and batch, yet the runtime must choose that partition and batch under changing conditions. Kernel efficiency and scheduling efficiency are complementary rather than substitutable.

Distributed training introduces additional constraints. Data parallelism replicates the model and partitions the graph, while model parallelism splits the model across devices. Pipeline parallelism overlaps stages of computation across devices to improve utilization [9]. Memory optimization systems such as ZeRO reduce redundant storage of optimizer states and gradients, enabling larger models and graphs [10]. More general scheduling frameworks show that the best parallelism strategy depends on the computation graph, communication costs, and device topology [11]. Graph neural networks add another layer because the data itself has structure. Partitioning a graph poorly can create excessive cross-device communication, while partitioning it too aggressively can imbalance load. A runtime scheduler must therefore reason about both model parallelism and graph partitioning, often simultaneously.

Memory capacity is a hard constraint that shapes scheduling. When a graph is too large for graphics processing unit memory, the runtime must stream features or subgraphs from host memory or remote storage. This creates a tension between prefetching and thrashing. Aggressive prefetching can hide latency but may evict useful data; conservative prefetching avoids waste but leaves devices idle. Similar tensions appear in large model serving, where memory management directly affects throughput and tail latency [17]. Graph neural network runtimes can borrow concepts such as block-level caching, eviction policies, and admission control, but they must adapt them to graph locality. Graph locality is often weak and nonuniform, so cache policies based on temporal reuse must be supplemented with structural heuristics, such as keeping high-degree hubs resident or prioritizing recently updated neighborhoods.

Heterogeneous hardware further complicates scheduling. Modern deployments may include multiple graphics processing unit generations, central processing units, network interfaces, and edge accelerators. A graph neural network inference service may run partially on an edge device and partially in a datacenter, depending on latency and privacy requirements. Scheduling across such a continuum requires abstractions for computation placement, data movement, and failure handling. Centralized cluster schedulers optimize global utilization but may not capture edge-specific constraints such as intermittent connectivity or energy limits. Conversely, edge-local schedulers may be responsive but lack global visibility. A hybrid runtime architecture is likely necessary, with hierarchical scheduling that delegates local decisions while retaining global policy control.

4. Runtime Scheduling Strategies and Structural Trade-offs

Runtime scheduling strategies for graph neural networks can be understood through several structural trade-offs. The first is latency versus throughput. Inference services often require low latency for individual queries, while training prioritizes aggregate throughput. A scheduler may batch many small inference requests to improve graphics processing unit utilization, but batching increases tail latency. For graph neural networks, batching is complicated because each request may involve a different subgraph. Combining subgraphs

into a single batch can improve kernel efficiency but may introduce padding, increase memory pressure, or violate freshness requirements. A topology-aware scheduler might group subgraphs with similar degree distributions or feature dimensions, reducing wasted work while preserving acceptable latency. This approach treats batching as a graph clustering problem rather than a simple queueing problem.

The second trade-off is fairness versus efficiency. Maximizing graphics processing unit utilization often means running long, resource-intensive jobs with minimal interruption. However, multi-tenant environments require fairness across users and services. A scheduler might use weighted fair queueing, priority classes, or resource caps to prevent starvation. In graph neural network workloads, fairness is complicated by the fact that jobs have different graph structures. Two jobs with the same number of nodes may have very different computational demands due to degree skew. A fair scheduler must therefore account for estimated work rather than simple job counts. This requires runtime profiling, online estimation, and possibly admission control. If the system cannot estimate workload cost accurately, fairness policies may inadvertently favor jobs with simpler graphs or smaller neighborhoods.

The third trade-off is adaptivity versus stability. Adaptive schedulers can respond to changing graph structure, contention, and hardware state, but frequent policy changes can cause overhead, oscillation, and unpredictable performance. Stable schedulers are easier to reason about and audit, but they may be inefficient under dynamic conditions. A practical runtime may use hysteresis, bounded adaptation, and periodic re-evaluation. For example, a scheduler might adjust subgraph batch sizes only when observed latency or memory pressure crosses a threshold. It might rebalance graph partitions only during training epochs rather than on every iteration. These design choices affect not only performance but also reproducibility. If scheduling decisions are too dynamic, experiments become difficult to replicate, which undermines scientific validity and regulatory auditability.

The fourth trade-off is centralization versus decentralization. A centralized scheduler has global visibility and can make coordinated decisions across devices, tenants, and services. It can enforce policy consistently and optimize cluster-wide objectives. However, it may become a bottleneck, a single point of failure, and a source of high communication overhead. A decentralized scheduler allows local autonomy and resilience, which is valuable in edge deployments and federated learning. However, local decisions may conflict, leading to resource fragmentation and unfairness. A hierarchical model can combine both approaches: a global policy layer defines priorities, quotas, and safety constraints, while local schedulers handle short-term placement, batching, and memory management. This mirrors broader trends in datacenter systems, where centralized control planes coexist with distributed data planes.

5. Infrastructure, Deployment, and Governance

Deploying graph neural network workloads on graphics processing units requires infrastructure that supports packaging, orchestration, monitoring, and lifecycle management. Frameworks such as Deep Graph Library and PyTorch Geometric provide graph abstractions, while PyTorch and TensorFlow provide tensor execution and autograd [13], [14], [15], [16]. In production, these libraries are usually deployed inside containers and orchestrated by cluster managers. The runtime scheduler must interact with container placement, device plugins, network policies, and storage systems. It must also expose metrics for latency, throughput, memory usage, errors, and queue depth. Observability is not optional; without it, scheduling policies cannot be tuned or audited. A graph neural network service that performs well on

average but fails unpredictably under skewed graphs is not robust enough for critical applications.

Governance enters through service-level objectives, access control, and compliance requirements. A financial institution may require that fraud detection models meet maximum latency bounds and provide explanations for adverse decisions. A healthcare system may require that patient graph data remain within a jurisdiction. A social platform may need to demonstrate that recommendation models do not systematically disadvantage certain groups. Runtime scheduling can support these goals by enforcing placement constraints, logging execution provenance, and allocating resources according to policy. However, scheduling alone cannot guarantee fairness or compliance. It must be coupled with model documentation, data governance, and human oversight. Model cards and similar reporting mechanisms provide structured descriptions of model purpose, performance, and limitations [21]. These artifacts can inform runtime policy by specifying which workloads require stricter latency, privacy, or fairness safeguards.

Infrastructure choices also affect reproducibility and technical debt. Machine learning systems often accumulate hidden dependencies, configuration drift, and undeclared feedback loops [22]. Graph neural network runtimes are particularly vulnerable because performance depends on graph preprocessing, sampling seeds, partitioning, and hardware configuration. If these factors are not captured in metadata, results may be difficult to reproduce. A governance-aware runtime should record scheduling decisions, resource allocations, and graph statistics alongside model checkpoints. This record enables debugging, auditing, and post hoc analysis. It also supports accountability when automated decisions have significant consequences. In this sense, the runtime is not merely an execution engine; it is part of the evidence infrastructure for responsible machine learning.

Deployment across cloud and edge environments adds further complexity. Edge devices may have limited memory, intermittent power, and variable connectivity. Cloud datacenters offer abundant resources but introduce communication latency and data transfer costs. A graph neural network application may partition computation so that sensitive local data remains on the edge while global aggregation occurs in the cloud. The scheduler must decide where to run each phase, how to compress intermediate representations, and how to recover from network failures. These decisions have policy implications because they determine data residency, energy consumption, and service equity. A rural edge deployment may have different constraints than an urban datacenter, and a fair system should not assume uniform infrastructure. Runtime scheduling must therefore be context-aware and capable of operating under diverse socio-technical conditions.

6. Sustainability, Robustness, Fairness, and Policy Implications

Sustainability has become a central concern for large-scale machine learning. Training and serving deep models consume significant energy, and the carbon intensity of that energy depends on location, time, and grid mix [19]. Green AI advocates for measuring efficiency not only in accuracy but also in computational cost and environmental impact [20]. Graph neural networks are not exempt from this critique. Their irregular memory access can reduce hardware utilization, extending execution time and increasing energy per useful operation. A runtime scheduler can improve sustainability by consolidating workloads, choosing energy-efficient device states, shifting flexible training jobs to low-carbon periods, and avoiding wasteful over-provisioning. However, these strategies must be balanced against latency and fairness. Shifting a job to a low-carbon period may delay results for users who need timely

updates. Consolidating workloads may improve efficiency but increase contention and tail latency.

Robustness is another critical dimension. Graph neural networks can be sensitive to adversarial perturbations, noisy edges, and distribution shift. A runtime scheduler cannot solve model-level robustness, but it can support resilience by isolating faulty jobs, retrying failed subgraphs, and degrading gracefully under resource shortages. For example, if a graphics processing unit fails during training, the scheduler can reassign partitions and restore from checkpoints. If inference latency exceeds a threshold, the scheduler can temporarily use a smaller neighborhood or a cached embedding. These degradation strategies must be policy-controlled because they affect prediction quality and fairness. A system that silently degrades service for some users but not others may create inequitable outcomes. Runtime policies should therefore specify when approximation is acceptable, how users are notified, and how quality is monitored.

Fairness in graph neural networks is closely tied to sampling and aggregation. If a scheduler prioritizes certain subgraphs or nodes for computational efficiency, it may inadvertently under-sample minority communities or peripheral regions of the graph. Over time, this can bias learned representations and downstream decisions. A fair runtime should monitor sampling coverage and ensure that scheduling decisions do not systematically disadvantage particular graph regions. This is challenging because fairness metrics for graphs are still an active research area. Nevertheless, runtime systems can provide transparency by logging which nodes, edges, and subgraphs were processed under which conditions. Such logs enable auditors to test for disparate impact and to trace performance disparities back to scheduling choices. Fairness is thus not only a model property but also a property of the computational pipeline.

Policy implications follow from these technical realities. Regulators may increasingly require documentation of training data, computational resources, and deployment environments. A runtime scheduler that records resource usage, placement decisions, and degradation events can help satisfy such requirements. Conversely, a scheduler that is opaque and adaptive without audit trails may make compliance difficult. Organizations should therefore treat scheduling policy as part of their governance framework. They should define who can set priorities, how quotas are enforced, how failures are handled, and how users can appeal automated decisions. These questions are not purely technical, but they cannot be answered without technical mechanisms. Runtime scheduling provides one such mechanism, bridging low-level resource management and high-level institutional accountability.

7. Conclusion

Runtime scheduling for efficient graph neural network workloads on graphics processing units is a multidimensional systems problem. It must address irregular computation, sparse memory access, dynamic graph structure, and multi-tenant contention while respecting latency, throughput, fairness, and sustainability goals. This paper has argued that kernel optimization alone is insufficient. Efficient graph neural network execution requires a runtime layer that coordinates graph partitioning, batching, memory management, device placement, and service-level policy. The structural trade-offs between latency and throughput, fairness and efficiency, adaptivity and stability, and centralization and decentralization cannot be resolved once and for all. They must be managed through adaptive, transparent, and governance-aware scheduling mechanisms.

Future research should develop scheduling abstractions that are portable across hardware generations and deployment environments. These abstractions should incorporate graph statistics, hardware telemetry, energy signals, and policy constraints. They should support hierarchical control, allowing global objectives to coexist with local autonomy. They should also produce audit trails that enable reproducibility, fairness analysis, and regulatory compliance. As graph neural networks move deeper into critical infrastructure, the runtime will become a site where technical efficiency and social responsibility meet. The challenge is not only to make graph neural networks faster on graphics processing units but also to make their execution accountable, resilient, and sustainable.

References

1. Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.
2. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2018). Graph attention 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* (Vol. 30, pp. 1024–1034).
4. 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.
5. Defferrard, M., Bresson, X., & Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems* (Vol. 29, pp. 3844–3852).
6. Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., & Leskovec, J. (2018). Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (pp. 974–983). ACM.
7. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2021). A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1), 4–24.
8. Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., & Sun, M. (2020). Graph neural networks: A review of methods and applications. *AI Open*, 1, 57–81.
9. Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., & Zaharia, M. (2019). PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (pp. 1–15). ACM.
10. Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2020). ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC20)* (pp. 1–16). IEEE.
11. Jia, Z., Zaharia, M., & Aiken, A. (2019). Beyond data and model parallelism for deep neural networks. In *Proceedings of Machine Learning and Systems* (Vol. 1, pp. 1–13).

12. 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).
13. Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, C., Zhou, J., Huang, Q., Ma, C., et al. (2019). Deep Graph Library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*.
14. Fey, M., & Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.
15. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems* (Vol. 32, pp. 8024–8035).
16. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (pp. 265–283). USENIX Association.
17. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., ... & Stoica, I. (2023). Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles* (pp. 611–626). ACM.
18. Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., & Stoica, I. (2017). Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (pp. 613–627). USENIX Association.
19. Strubell, E., Ganesh, A., & McCallum, A. (2019). Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics* (pp. 3645–3650). Association for Computational Linguistics.
20. Schwartz, R., Dodge, J., Smith, N. A., & Etzioni, O. (2020). Green AI. *Communications of the ACM*, 63(12), 54–63.
21. Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., Vasserman, L., Hutchinson, B., ... & Gebru, T. (2019). Model cards for model reporting. In *Proceedings of the Conference on Fairness, Accountability, and Transparency* (pp. 220–229). ACM.
22. Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency* (pp. 610–623). ACM.