

Cache-Aware Graph Processing for High-Throughput GNN Training

Loek Lommert

Department of Computer Science, University of Alabama at Birmingham, Birmingham, AL, USA.

loek.work@uab.edu

Marcelina Flisek

Department of Computer Science, University of Houston, Houston, TX, USA.

marcelina.work@uh.edu

Banjeet Dalal

Department of Computer Science, Colorado State University, Fort Collins, CO, USA.

banjeetwork@colostate.edu

Abilio Canellas

School of Computing, Clemson University, Clemson, SC, USA.

abiliocanellas@clemson.edu

Abstract

Graph neural network training has become a central workload in contemporary machine learning systems, yet its performance remains constrained by irregular memory access, poor cache locality, and high communication overhead. This paper examines cache-aware graph processing as a system-level strategy for high-throughput graph neural network training. It argues that cache behavior cannot be treated as a secondary optimization concern but must be elevated to a first-class design principle spanning graph representation, partitioning, scheduling, hardware acceleration, distributed execution, and operational governance. The analysis synthesizes prior work on graph-parallel systems, graph neural network algorithms, deep learning frameworks, and large-scale training infrastructure to identify structural trade-offs among throughput, accuracy, robustness, fairness, and sustainability. Particular attention is given to how cache-aware data layouts, vertex reordering, edge tiling, feature caching, mini-batch sampling, and heterogeneous acceleration shape end-to-end training performance. The paper also considers governance and policy implications, including energy accountability, equitable access to accelerated infrastructure, privacy-preserving graph learning, and the risk that cache-aware optimizations may disproportionately benefit dense or high-degree regions of a graph. Deployment pathways across cloud, edge, and high-performance computing environments are discussed, along with the need for observability into cache hit behavior, memory bandwidth, and communication bottlenecks. The conclusion emphasizes that high-throughput graph neural network training requires not only faster kernels but also coherent architectural, institutional, and policy frameworks that treat memory locality as a socio-technical resource.

Keywords

graph neural networks; cache-aware processing; high-throughput training; memory hierarchy; distributed systems; socio-technical infrastructure; governance; sustainability.

1. Introduction

Graph neural networks have emerged as a powerful computational paradigm for learning over relational data, with applications ranging from recommendation and fraud detection to drug discovery and transportation forecasting. Their training process combines the irregular access patterns of graph analytics with the dense numerical demands of deep learning, creating a workload that stresses memory hierarchies in ways that conventional tensor-centric systems were not designed to handle. Early large-scale data processing systems such as MapReduce established the value of distributed abstraction for massive datasets [1], while graph-specific systems such as Pregel and PowerGraph introduced vertex-centric and edge-centric programming models that exposed the tension between locality and parallelism [2], [3]. These foundations remain relevant because graph neural network training inherits many of the same structural problems, especially when neighborhood expansion produces scattered reads and unpredictable cache misses.

The introduction of graph convolutional networks and related message-passing architectures demonstrated that relational learning could be integrated into differentiable end-to-end pipelines [4]. Subsequent models such as GraphSAGE and graph attention networks expanded the design space by incorporating sampling, inductive learning, and attention mechanisms [5], [6]. Yet these algorithmic advances also intensified system-level pressure. Each training step may require gathering features from multiple hops of neighbors, aggregating them, and updating representations in a manner that resists simple blocking and reuse. As a result, high-throughput graph neural network training is not merely a matter of adding accelerators; it requires cache-aware graph processing that aligns data movement with the memory hierarchy at every layer of the stack.

This paper argues that cache awareness should be treated as a cross-cutting architectural principle rather than a local kernel optimization. It examines how graph representation, partitioning, sampling, scheduling, distribution, and governance interact with cache behavior. The discussion is deliberately system-oriented and socio-technical, because choices about caching and locality have consequences for cost, energy, robustness, fairness, and policy. The central claim is that high-throughput graph neural network training depends on coordinated design decisions that make memory locality explicit, measurable, and accountable across hardware, software, and institutional boundaries.

2. Foundations of Graph Neural Network Training and Memory Behavior

Graph neural network models derive their expressive power from iterative message passing over graph structure. The foundational graph neural network model formalized this idea as a state transition process over nodes and edges [7], while later work on graph isomorphism networks clarified the representational limits and design conditions of message-passing architectures [8]. In practice, message passing translates into a sequence of gather, transform, and scatter operations. These operations are computationally simple in isolation, but their memory behavior is highly irregular. Neighbor lists are often sparse, feature vectors may be large, and the number of active nodes can vary dramatically across batches. Consequently, cache misses, memory latency, and bandwidth contention frequently dominate runtime.

Sampling and decoupling strategies have been introduced to reduce neighborhood explosion. GraphSAGE samples fixed-size neighborhoods to make training more scalable and inductive [5], while GraphSAINT constructs subgraphs that preserve structural properties during training [10]. Decoupled approaches such as SIGN precompute multi-hop features to reduce

repeated message passing [9]. These techniques improve throughput, but they also shift the burden to data layout and cache management. If sampled subgraphs are stored without regard for locality, the system may still suffer from scattered memory access. If precomputed features are cached poorly, they can increase memory pressure without delivering proportional gains.

Frameworks such as PyTorch Geometric and the Deep Graph Library provide abstractions for building graph neural networks and executing them on heterogeneous hardware [11], [12]. However, abstraction alone does not solve the memory hierarchy problem. Performance depends on how adjacency structures, features, gradients, and intermediate activations are placed in cache, shared memory, and high-bandwidth memory. The foundation of high-throughput training is therefore an understanding of graph neural network workloads as memory-bound and communication-sensitive, not merely compute-bound. This understanding motivates cache-aware representations and execution strategies that treat locality as a primary design objective.

3. Cache-Aware Graph Representation and Data Layout

The representation of a graph determines the baseline locality of every subsequent operation. Compressed sparse row formats and related adjacency structures are common because they reduce storage overhead, but their access patterns can be unfriendly when neighborhoods span distant memory regions. Vertex reordering, clustering, and partitioning can improve spatial and temporal locality by placing frequently co-accessed vertices or edges near one another. Similarly, edge tiling and block-based layouts can transform unpredictable gathers into more regular accesses that fit within cache lines. These transformations are not semantically neutral; they can affect load balance, communication volume, and the cost of incremental updates in dynamic graphs.

Feature caching is another critical dimension. Node features are often much larger than adjacency metadata, and repeated access to popular nodes can dominate bandwidth. Cache-aware designs may pin high-degree or frequently sampled features in faster memory, compress low-precision embeddings, or stage features in shared memory for reuse across neighboring computations. A recent GPU kernel design for accelerating graph neural network training demonstrates that cache-aware tiling and shared-memory reuse can substantially improve throughput [14]. This result reinforces a broader lesson: the best-performing systems do not merely optimize arithmetic; they orchestrate data movement so that expensive memory traffic is minimized.

Framework integration matters because cache-aware layouts must be expressible within deep learning ecosystems. TensorFlow and PyTorch provide flexible execution graphs and operator libraries [15], [16], but graph-specific data structures often sit outside their default optimization paths. A practical cache-aware system therefore needs conversion layers, memory planners, and profiling hooks that allow graph representations to be transformed without breaking autograd, distributed synchronization, or checkpointing. The design challenge is to make locality-enhancing transformations transparent enough for users and explicit enough for compilers and runtime schedulers.

4. Architectural Design for Cache-Aware GNN Training

At the architectural level, cache-aware graph neural network training spans CPUs, GPUs, accelerators, interconnects, and storage. GPUs offer massive parallelism and high memory bandwidth, but their cache hierarchies are limited relative to the irregular working sets of

graph workloads. CPUs provide larger caches and more flexible memory management, but they may lack the throughput required for dense feature transformations. Heterogeneous frameworks such as MXNet illustrate the difficulty of coordinating computation across devices with different memory characteristics [18]. Effective designs often partition work so that regular dense operations run on accelerators while irregular gathers, sampling, and metadata traversal are handled by cache-friendly CPU paths or specialized kernels.

Pipelining and memory optimization further shape achievable throughput. Pipeline parallelism can overlap communication and computation across mini-batches or layers [19], while memory optimization techniques such as ZeRO reduce redundant model-state storage and improve the feasibility of large models [20]. For graph neural networks, however, model-state memory is only part of the problem. Intermediate activations, sampled subgraphs, and feature caches can consume substantial memory and compete for cache capacity. Architectural decisions must therefore balance model parallelism, data parallelism, graph partitioning, and cache residency. A system that maximizes tensor-core utilization without controlling graph-induced cache thrashing may still deliver disappointing end-to-end performance.

The architecture must also incorporate observability. Cache hit rates, memory bandwidth utilization, interconnect congestion, and queue depths are essential signals for adaptive scheduling. Without such telemetry, cache-aware policies become static guesses that may fail under changing graph structure, batch composition, or multi-tenant load. A robust architecture should expose locality metrics to both runtime schedulers and human operators, enabling dynamic repartitioning, feature migration, and priority adjustment. In this sense, cache awareness is not only a kernel property but a systemic capability for sensing and responding to memory behavior.

5. Scheduling, Partitioning, and Distributed Execution

Distributed graph neural network training introduces additional trade-offs between locality and scalability. Graph partitioning aims to minimize cross-partition edges, but cache-aware partitioning must also consider feature locality, degree distribution, and communication patterns. Systems such as DistDGL demonstrate the importance of partitioning for billion-scale graph training [13]. Yet even a well-partitioned graph can suffer if mini-batch sampling creates highly skewed workloads. High-degree vertices may become hotspots, while low-degree regions leave resources underutilized. Scheduling policies must therefore combine graph-aware load balancing with cache-aware placement of features and adjacency data.

Communication is a central constraint. When partitions exchange node embeddings or gradients, interconnect bandwidth and latency can dominate runtime. Remote direct memory access, collective communication libraries, and hierarchical aggregation can reduce overhead, but they also require careful coordination with cache policies. For example, if frequently accessed boundary vertices are cached locally, the system can reduce communication at the cost of memory pressure and potential staleness. Conversely, aggressive synchronization improves consistency but may destroy the locality gains that motivated caching in the first place. Distributed execution thus becomes a negotiation among consistency, freshness, locality, and throughput.

Scheduling also intersects with data preprocessing and pipeline design. Systems such as Spark are often used to prepare graph data, normalize features, and construct adjacency structures [17]. If preprocessing produces partitions that are hostile to cache reuse, downstream training

pays a persistent penalty. A cache-aware pipeline should therefore treat preprocessing as part of the training architecture, not as a separate extract-transform-load stage. Partition formats, feature encodings, and sampling indices should be generated with the target memory hierarchy in mind. This end-to-end perspective is essential for high-throughput training at scale.

6. Throughput Trade-Offs, Robustness, and Fairness

Maximizing throughput often requires approximations that can affect model quality, robustness, and fairness. Neighborhood sampling reduces memory traffic and improves cache behavior, but it may underrepresent rare or peripheral nodes. Attention mechanisms can focus computation on salient edges, yet they may also amplify existing structural biases [6]. Cache-aware prioritization of high-degree or frequently accessed nodes can improve performance while systematically neglecting nodes that are structurally disadvantaged. If such choices are made only for speed, they can entrench inequities in downstream predictions, especially in applications involving credit, employment, housing, or public services.

Robustness is similarly entangled with cache design. Adversarial edges, noisy features, and distribution shifts can alter which parts of the graph are hot in cache and which are cold. A system optimized for a stable access distribution may degrade sharply when the graph changes or when malicious actors manipulate connectivity. Cache-aware training should therefore include stress testing, anomaly detection, and adaptive eviction policies that prevent any subgraph from monopolizing memory indefinitely. Robustness is not only a model property; it is also a property of the data movement strategy that determines which information remains available to the learner.

Fairness-aware system design requires metrics that connect cache behavior to representational outcomes. For example, operators could monitor whether sampled batches systematically exclude certain communities, whether feature caches disproportionately retain majority-group attributes, or whether partitioning creates uneven latency across tenants. These metrics do not automatically produce fairness, but they make structural trade-offs visible. Throughput and fairness need not be mutually exclusive, yet they will remain in tension unless cache-aware architectures include explicit governance hooks for auditing, adjusting, and contesting allocation decisions.

7. Governance, Policy, and Sustainability Implications

Cache-aware graph processing is embedded in socio-technical infrastructure, which means its governance cannot be reduced to technical benchmarking. Organizations that operate large graph training systems make consequential decisions about data retention, access control, model reuse, and energy consumption. Caching features and embeddings can improve efficiency, but it can also create persistent copies of sensitive information outside original data boundaries. Privacy regulations and institutional policies must therefore address cache lifetime, eviction, encryption, and auditability. A cache that is optimized solely for hit rate may become a compliance liability if it retains personal or proprietary data longer than necessary.

Sustainability is another policy-relevant dimension. Graph neural network training consumes substantial energy, and memory movement is a major contributor. Cache-aware designs can reduce energy by avoiding redundant data transfers, but they can also encourage larger models, larger graphs, and more aggressive overprovisioning. The net environmental effect depends on deployment context, hardware efficiency, utilization, and the broader energy mix.

Policymakers and infrastructure providers should therefore demand transparency about energy use and encourage shared, multi-tenant accelerators where appropriate, rather than treating cache optimization as an unconditional good.

Governance also concerns equitable access to accelerated infrastructure. Universities, smaller enterprises, and public-interest organizations may lack the capital to build or rent high-throughput GPU clusters. If cache-aware graph training becomes dependent on specialized hardware and proprietary runtime optimizations, the result may be a concentration of capability among a few well-resourced actors. Public policy can mitigate this by supporting shared research infrastructure, open benchmarks, and reproducible systems research. The technical community can contribute by publishing cache-aware methods that are portable, measurable, and usable across diverse deployment settings.

8. Deployment Pathways and Operational Considerations

Deployment of cache-aware graph neural network training varies across cloud, edge, and high-performance computing environments. Cloud platforms offer elastic accelerators and managed storage, but network and multi-tenancy can introduce unpredictable cache interference. Edge deployments may prioritize latency, privacy, and bandwidth conservation, making local feature caching attractive but constrained by device memory. High-performance computing environments provide fast interconnects and large memory hierarchies, yet job schedulers and batch policies may limit adaptive cache control. A single cache-aware strategy is unlikely to fit all contexts; instead, systems should expose tunable policies for partitioning, prefetching, eviction, and synchronization.

Operational monitoring must extend beyond loss curves and hardware utilization. Teams need visibility into cache hit rates by data structure, communication volume by partition, straggler effects, and energy per training step. Frameworks such as PyTorch Geometric and the Deep Graph Library can serve as integration points [11], [12], while distributed graph engines such as DistDGL offer lessons for partitioning and sampling [13]. Pipeline parallelism and memory optimization methods provide additional design patterns [19], [20]. The practical challenge is to compose these elements into a deployment architecture that remains stable under changing graphs, workloads, and organizational requirements.

Sustainability and robustness in deployment also depend on lifecycle management. Graph data changes, models are retrained, and hardware is replaced. Cache formats and partitions must be versioned, validated, and migrated without silently degrading performance or fairness. Operational runbooks should include rollback procedures for cache policies, audits of retained embeddings, and periodic revalidation of partitioning assumptions. High-throughput training is therefore not a one-time configuration but an ongoing operational discipline that connects machine learning engineering with infrastructure governance.

9. Future Research Directions

Future research should pursue cache-aware graph processing as a co-design problem across algorithms, compilers, runtimes, and hardware. One promising direction is adaptive graph representation, in which the system changes layout, compression, and caching policy in response to observed access patterns. Another is fairness-aware cache management, where eviction and prefetching decisions are evaluated not only by hit rate but also by their impact on underrepresented subgraphs. A third direction is sustainable acceleration, including energy-aware scheduling and memory hierarchies that reward data reuse without encouraging unnecessary model growth.

There is also a need for shared benchmarks and reproducible evaluation. Many cache-aware claims are difficult to compare because they use different graphs, batch sizes, hardware, and accuracy targets. Community benchmarks should report throughput, latency, memory bandwidth, cache behavior, energy, and fairness metrics together. They should include dynamic graphs, adversarial perturbations, and multi-tenant scenarios. Such benchmarks would help researchers and practitioners understand when a cache-aware optimization is genuinely beneficial and when it merely shifts cost to another part of the system.

Finally, research should examine the governance of cache-aware infrastructure. Technical mechanisms for auditing cached data, limiting retention, and explaining allocation decisions are still immature. Interdisciplinary work involving systems researchers, legal scholars, policymakers, and domain experts is needed to translate cache behavior into accountable institutional practice. The next generation of graph neural network training systems should therefore be evaluated not only by speed but by their ability to support robust, fair, and sustainable learning over relational data.

10. Conclusion

Cache-aware graph processing is a necessary foundation for high-throughput graph neural network training. The irregular structure of graphs makes memory locality a first-order determinant of performance, and the complexity of modern training pipelines makes cache behavior a cross-cutting concern that spans representation, partitioning, scheduling, acceleration, distribution, and operations. This paper has argued that cache awareness should be designed into the system from the beginning, rather than added as a late-stage kernel optimization. It has also emphasized that cache decisions have socio-technical consequences, including effects on energy use, privacy, fairness, robustness, and equitable access to computational resources.

High-throughput training will increasingly depend on architectures that can sense memory behavior, adapt to changing graph structure, and account for the distributional consequences of locality optimization. The technical community should develop portable and reproducible cache-aware methods, while institutions should create governance frameworks that make data retention, energy consumption, and allocation decisions transparent. By treating memory locality as both an engineering resource and a policy-relevant infrastructure concern, future graph neural network systems can achieve higher throughput without sacrificing responsibility, resilience, or public trust.

References

1. Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107–113.
2. Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (pp. 135–146). ACM.
3. Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., & Guestrin, C. (2012). PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (pp. 17–30). USENIX Association.
4. Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.

5. Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems*, 30.
6. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2018). Graph attention networks. In *International Conference on Learning Representations*.
7. 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.
8. Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2019). How powerful are graph neural networks? In *International Conference on Learning Representations*.
9. Frasca, F., Rossi, E., Eynard, D., Chamberlain, B., Bronstein, M., & Monti, F. (2020). SIGN: Scalable inception graph neural networks. *arXiv preprint arXiv:2004.11198*.
10. Zeng, H., Zhou, H., Srivastava, A., Kannan, R., & Prasanna, V. (2020). GraphSAINT: Graph sampling based inductive learning method. In *International Conference on Learning Representations*.
11. Fey, M., & Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428*.
12. Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., Zhang, Z., & Li, A. (2019). Deep Graph Library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*.
13. Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., Gan, Q., Zhang, Z., & Karypis, G. (2020). DistDGL: Distributed graph neural network training for billion-scale graphs. In *IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms* (pp. 36–46). IEEE.
14. 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).
15. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., & Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (pp. 265–283). USENIX Association.
16. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32.
17. Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*. USENIX Association.

18. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., & Zhang, Z. (2015). MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274.
19. 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.
20. 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 (pp. 1–16). IEEE.