

Memory-Efficient GNN Training via Adaptive Graph Data Layouts

Elsa Gerlach

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

Julianna Goj

School of Information Technology, University of Cincinnati, Cincinnati, OH, USA.
juliannawork@uc.edu

Prabowo Hariyah

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

Abstract

Graph neural networks have become a central computational method for learning over relational data, yet their training remains constrained by irregular memory access, adjacency storage expansion, and poor locality across accelerators and distributed systems. This paper examines memory-efficient graph neural network training through the lens of adaptive graph data layouts. Rather than treating layout as a static preprocessing decision, the paper positions adaptive layouts as a systems abstraction that couples graph structure, hardware topology, runtime scheduling, and governance concerns. It analyzes how vertex-centric, edge-centric, blocked, tiled, cache-aware, and hybrid representations interact with memory hierarchies, feature stores, sampling engines, and communication substrates. The discussion emphasizes structural trade-offs among preprocessing cost, layout mutation overhead, load balance, bandwidth utilization, fault tolerance, and reproducibility. It then extends the analysis to distributed and heterogeneous infrastructures, where partitioning, replication, caching, and compression determine whether training can scale without excessive synchronization or energy consumption. The paper also considers robustness, fairness, and policy implications, arguing that adaptive layouts are not merely performance artifacts but infrastructural choices that shape access, accountability, and sustainability. Case illustrations from graph sampling systems, kernel-level acceleration, and large-scale deployment environments show that memory efficiency depends on coordinated design across algorithms, compilers, storage, and operations. The conclusion identifies open challenges in portable adaptive layouts, benchmark standardization, carbon-aware scheduling, and equitable deployment. The central claim is that adaptive graph data layouts offer a viable path toward memory-efficient graph neural network training only when they are treated as governed socio-technical infrastructure rather than isolated optimization tricks.

Keywords

graph neural networks, adaptive data layouts, memory efficiency, distributed training, systems architecture, sustainability, governance, graph sampling.

1. Introduction

Graph neural networks have emerged as a dominant approach for learning from relational structures, including social networks, molecular graphs, transportation systems, recommendation graphs, and scientific knowledge bases [1]. Their core appeal lies in the ability to combine node features with neighborhood aggregation, allowing models to capture dependencies that are difficult to express in sequence or grid data. Early graph convolutional formulations established that localized spectral approximations could produce effective node representations while remaining computationally tractable [1]. Subsequent inductive methods extended this capability to unseen nodes and evolving graphs, making graph learning practical for dynamic industrial settings [2]. Attention-based graph models further improved flexibility by learning neighborhood importance weights rather than relying solely on fixed normalization [3]. Recurrent and message-passing graph models provided additional theoretical grounding for iterative state propagation over structured domains [4]. Collectively, these advances have made graph neural networks a general-purpose substrate for relational learning.

However, the success of graph neural networks has exposed a persistent systems bottleneck. Unlike dense tensors in convolutional or transformer models, graph data is irregular, sparse, and often highly skewed in degree distribution. Training requires repeated neighborhood sampling, feature gathering, aggregation, and gradient updates across large adjacency structures. These operations generate irregular memory access patterns that underutilize caches, increase translation overhead, and create load imbalance across parallel workers. The problem is not merely one of arithmetic intensity but of data movement. In many deployments, the cost of fetching and rearranging graph data dominates the cost of computation. Consequently, memory-efficient training has become a first-order research concern, with implications for hardware design, distributed scheduling, storage systems, and even environmental sustainability.

This paper argues that adaptive graph data layouts provide a unifying systems strategy for addressing these bottlenecks. A graph data layout is the physical arrangement of adjacency information, node features, edge attributes, and intermediate activations in memory. Traditional layouts such as compressed sparse row, compressed sparse column, and adjacency lists are often chosen once during preprocessing and then held static throughout training. Adaptive layouts, by contrast, change in response to graph structure, access frequency, hardware hierarchy, batch composition, and runtime feedback. They may reorder vertices, partition edges, tile subgraphs, compress low-traffic regions, or migrate hot data between memory tiers. The central question is not whether adaptation is always beneficial, but under what conditions its coordination costs are outweighed by gains in locality, bandwidth efficiency, and scalability. This framing shifts attention from isolated kernel optimizations to the architecture and governance of graph training infrastructures.

2. Background and Problem Formulation

The computational pattern of graph neural network training can be understood as a sequence of sparse operations interleaved with dense neural network computation. For each sampled batch, the system must identify target nodes, expand neighborhoods, retrieve features, aggregate messages, apply learned transformations, and update parameters. The irregularity arises because neighborhood sizes vary dramatically, and the set of accessed nodes changes across epochs. In static layouts, this variation produces uneven memory traffic and branch divergence. In distributed settings, it produces communication imbalance and straggler effects. Surveys of graph neural networks have documented the breadth of algorithmic variants,

including spectral, spatial, attention-based, recurrent, and generative models [16]. Reviews of methods and applications have similarly shown that graph learning is deployed across domains with very different latency, privacy, and scale requirements [17]. These differences imply that no single layout can be optimal for all graph neural network workloads.

A second source of difficulty is the memory hierarchy itself. Modern accelerators rely on high-bandwidth memory, multilevel caches, shared memory, and increasingly complex interconnect topologies. Efficient graph training requires that frequently accessed adjacency blocks and feature tiles remain close to compute units, while cold data stays in slower tiers. Yet graph structure is rarely aligned with hardware topology. Community structure, hub nodes, and long-tail degree distributions create nonuniform reuse. A layout that improves spatial locality for one subgraph may worsen it for another. As a result, static preprocessing can only approximate the access behavior of a dynamic training process. Sampling-based methods mitigate this problem by reducing the neighborhood size per batch, but they introduce variance and may require repeated random access to adjacency structures [7]. Variance-reduction techniques can improve convergence, but they also add memory overhead for control variates and cached history [7]. The design space is therefore multidimensional, involving trade-offs among accuracy, convergence speed, memory footprint, and system throughput.

Framework and compiler choices further shape what layouts are feasible. Libraries such as PyTorch Geometric and the Deep Graph Library provide abstractions for graph data, sampling, and message passing, but their performance depends heavily on underlying storage formats and kernel implementations [8], [9]. Distributed graph training systems such as DistDGL and PaGraph extend these abstractions to multiple machines by partitioning graphs, caching features, and scheduling communication [10], [11]. Surveys of graph neural network accelerators note that hardware and software co-design is essential because graph workloads stress memory systems differently from dense deep learning [12]. Studies of graph sampling and clustering algorithms show that partitioning and sampling strategies can reduce working-set size and improve locality [14], [15]. These efforts collectively demonstrate that layout is not a passive detail but an active component of the training architecture.

3. Adaptive Graph Data Layouts as a Systems Abstraction

Adaptive graph data layouts can be defined as representations that are selected, transformed, or reorganized during the training lifecycle in response to observed or predicted access patterns. This definition encompasses several mechanisms. Vertex reordering changes node identifiers to improve locality for frequently co-accessed nodes. Edge blocking groups edges into dense tiles that can be fetched efficiently by accelerators. Hybrid layouts combine compressed sparse formats for cold regions with blocked or dense formats for hot regions. Feature layouts may be column-major, row-major, paged, or quantized depending on reuse and bandwidth constraints. Sampling-aware layouts may store subgraph templates or precomputed neighborhoods to reduce random access. In each case, the layout is not merely a storage format but a control point that mediates between graph semantics and machine execution.

A recent kernel-level study of graph neural network training illustrates the potential of layout-aware acceleration. By focusing on max-k subgraph patterns and block-centric scheduling, it demonstrates that memory pressure can be reduced while preserving training throughput on graphics processing units [5]. The significance of this result is not limited to a single kernel. It shows that when adjacency structure is reorganized around computationally meaningful

blocks, the memory system can be used more predictably. Similar lessons appear in graph sampling systems, where clustering or partitioning nodes into dense batches reduces the number of cross-partition edges and improves cache behavior [14]. Graph sampling methods such as GraphSAINT likewise exploit sampled subgraphs to bound memory use while maintaining inductive learning capability [15]. These approaches collectively support the idea that adaptive layouts are a form of runtime resource management.

The systems abstraction also clarifies structural trade-offs. Adaptation consumes time, memory, and engineering complexity. Reordering vertices may invalidate cached node identifiers, requiring remapping across feature stores, optimizers, and checkpoints. Repartitioning edges may improve locality but disrupt training continuity and require synchronization. Compression may reduce memory footprint but introduce decoding overhead and potential accuracy loss. Quantization may accelerate feature gathering but complicate gradient computation and reproducibility. Finally, adaptive layouts can interact poorly with fault tolerance because recovery may require reconstructing not only model state but also the current layout state. A mature design must therefore specify adaptation triggers, consistency guarantees, rollback mechanisms, and observability. Without these, adaptive layouts risk becoming opaque optimizations that improve benchmark numbers while increasing operational fragility.

4. Memory Hierarchy and Training Pipeline Implications

The training pipeline for graph neural networks spans data ingestion, sampling, feature retrieval, message passing, aggregation, backpropagation, and parameter updates. Each stage has different memory access characteristics. Sampling is often dominated by pointer chasing and random reads. Feature retrieval is bandwidth-intensive and may benefit from tiling or compression. Message passing is irregular and sensitive to edge ordering. Aggregation may require atomic operations or segmented reductions. Backpropagation revisits intermediate activations, increasing peak memory pressure. Adaptive layouts can target one or more of these stages, but they must be evaluated against the entire pipeline. Optimizing sampling while worsening feature retrieval, for example, may produce no net gain.

Hardware-aware layout design must account for the distinction between capacity, bandwidth, and latency. High-bandwidth memory provides capacity and throughput but not necessarily low latency. Caches reward temporal and spatial locality but punish irregular access. Shared memory can act as a software-managed cache for adjacency tiles, but its limited size constrains how much of a neighborhood can be staged. Unified memory and page migration can simplify programming but may introduce unpredictable stalls. In distributed environments, remote memory access adds network latency and congestion. Systems such as DistDGL and PaGraph address these constraints through partitioning, caching, and communication scheduling [10], [11]. Their experiences show that layout decisions are inseparable from placement and replication decisions. A layout that is efficient on a single node may be inefficient when partitions must exchange boundary nodes.

General deep learning frameworks provide important infrastructure for these efforts. PyTorch’s tensor abstraction and autograd system enable flexible experimentation, but graph-specific layouts often require custom extensions or compiler support [18]. Early work on large-scale distributed deep networks established many of the synchronization and parameter-server patterns that still inform graph training systems [19]. More recent experience with large model training on graphics processing unit clusters highlights the importance of overlapping communication with computation and managing memory hierarchies at scale [20]. Although

graph neural networks differ from language models in structure, both domains confront the same fundamental issue: performance is limited by data movement, not only by arithmetic. Adaptive graph layouts are therefore part of a broader movement toward memory-centric system design.

5. Distributed and Heterogeneous Infrastructure

Distributed graph neural network training introduces additional layout dimensions. Graphs may be partitioned by vertex, by edge, or by subgraph. Each choice affects communication volume, load balance, and cache locality. Vertex partitioning often creates boundary nodes that must be replicated or exchanged. Edge partitioning can balance workload better for high-degree vertices but may duplicate features. Subgraph partitioning preserves locality within clusters but can create isolated partitions with poor global coverage. Systems such as DistDGL and PaGraph demonstrate that no single partitioning strategy dominates across all graph types and cluster configurations [10], [11]. Adaptive layouts in distributed settings must therefore respond not only to local memory pressure but also to network topology and tenant-level resource constraints.

Heterogeneous infrastructure adds further complexity. A training cluster may include graphics processing units with different memory capacities, central processing units with large but slower memory, and specialized accelerators optimized for sparse operations. Edge deployments may involve mobile devices, embedded systems, or intermittently connected nodes. In such environments, adaptive layouts can exploit heterogeneity by placing hot subgraphs on high-bandwidth devices and cold data on cheaper storage. However, heterogeneity also increases the risk of stragglers, version skew, and inconsistent performance. The layout must be portable enough to run across devices yet specialized enough to exploit local hardware features. Compiler-based approaches and runtime systems can help by abstracting layout transformations, but they require stable interfaces and cost models. Without these, operators may face a choice between portability and performance.

Governance and policy implications arise because distributed graph training often involves data that is sensitive, regulated, or unevenly distributed. Social graphs may contain personal relationships, financial graphs may encode transactions, and health graphs may link patients and providers. Adaptive layouts that cache, replicate, or compress data can create new privacy and compliance risks. If a layout moves hot features to a shared cache, it may bypass access controls designed for the original storage system. If a layout is learned from access patterns, it may leak information about which nodes or edges are frequently queried. If a layout is optimized for a particular population, it may systematically under-serve minority communities with different graph structures. These concerns connect layout design to broader questions of fairness, accountability, and infrastructure governance.

6. Robustness, Fairness, and Governance

Robustness in graph training is closely tied to memory layout because adversarial or unusual graph structures can trigger pathological access patterns. A malicious or naturally occurring hub node with extremely high degree can cause memory blowups, load imbalance, and denial-of-service effects in shared training infrastructure. Adaptive layouts can mitigate this by capping neighborhood expansion, isolating high-degree vertices, or dynamically partitioning hot regions. However, such adaptations can also introduce blind spots. If the system systematically samples around certain structures, it may fail to learn important long-range dependencies or may underrepresent peripheral communities. Robustness therefore

requires monitoring not only throughput and memory use but also the statistical coverage of sampled neighborhoods.

Fairness considerations are less obvious but equally important. Graph neural networks are often used in recommendation, credit scoring, fraud detection, and public service allocation. The layout determines which nodes and edges are cheap to access and which are expensive. If the layout favors dense, well-connected communities, the model may receive more accurate representations for those groups while producing noisier predictions for sparse or marginalized groups. If compression or quantization is applied unevenly, some subpopulations may experience higher error rates. If caching prioritizes frequently accessed nodes, minority viewpoints that are rare in the graph may be systematically deprioritized. These are not merely algorithmic fairness issues; they are infrastructural fairness issues. A fair graph training system must be able to audit how layout choices affect representation quality across groups.

Governance frameworks must therefore extend to the layout layer. Organizations need documentation of layout transformations, versioning of graph partitions, and audit trails showing which data was cached or replicated. Privacy-preserving techniques such as differential privacy, secure aggregation, and federated learning can be combined with adaptive layouts, but each combination creates new trade-offs. Federated graph learning, for example, may keep raw data local but still require layout coordination across clients. Differential privacy may add noise that interacts with sparse sampling and compression. Policy instruments such as data minimization, purpose limitation, and retention limits must be translated into technical controls at the storage and memory management layers. Adaptive layouts should not be exempt from these controls simply because they operate close to the hardware.

7. Sustainability and Deployment Economics

Sustainability is an increasingly important dimension of graph training infrastructure. Training large graph models consumes energy for computation, memory access, communication, and cooling. Adaptive layouts can reduce energy use by improving locality and reducing redundant data movement. They can also increase energy use if adaptation itself is expensive or if frequent reordering prevents hardware from entering low-power states. The net effect depends on workload characteristics, hardware efficiency, and scheduling policies. Carbon-aware scheduling could prioritize jobs that use adaptive layouts during periods of low-carbon electricity, but this requires visibility into both energy consumption and layout behavior. Without measurement and standardization, sustainability claims remain difficult to verify.

Deployment economics add another layer. Organizations may run graph training on cloud instances, on-premises clusters, or hybrid environments. Adaptive layouts can reduce memory requirements, allowing larger graphs to fit on existing hardware and postponing expensive upgrades. They can also improve multi-tenancy by limiting peak memory and reducing interference between jobs. However, the engineering cost of building and maintaining adaptive layout systems is substantial. Teams must develop expertise in graph algorithms, compilers, distributed systems, and operations. They must also manage the risk of vendor lock-in if layout optimizations depend on proprietary hardware or software. Open standards and portable abstractions can mitigate this risk, but they may lag behind the fastest proprietary implementations.

Case illustrations show both promise and caution. A social media company training recommendation models might use adaptive layouts to cache features for highly active users while compressing cold profiles. A scientific computing team might use subgraph tiling to fit molecular graph batches into accelerator memory. A public health agency might use federated graph learning with layout coordination across hospitals. In each case, the layout is not merely a performance choice; it determines who can participate, what data is retained, and how quickly insights can be produced. The deployment context therefore shapes the acceptable trade-offs among speed, cost, privacy, and equity.

8. Evaluation Considerations and Case Illustrations

Evaluating adaptive graph layouts requires more than reporting epoch time or peak memory. Researchers should measure end-to-end pipeline performance, including preprocessing, adaptation, sampling, feature retrieval, communication, and checkpointing. They should report variance across graph types, degree distributions, batch sizes, and hardware configurations. They should test robustness to adversarial structures, such as high-degree hubs and disconnected components. They should assess fairness by comparing model quality across subpopulations defined by graph position, degree, or community membership. They should document energy consumption and carbon intensity where possible. Without such multidimensional evaluation, layout claims may be overfitted to narrow benchmarks.

Benchmarks and testbeds are also needed. Existing graph learning libraries and sampling systems provide valuable baselines [6], [8], [9], [14], [15]. Kernel-level studies show that significant gains are possible when memory access is reorganized around graph structure [5]. Distributed systems demonstrate that partitioning and caching can scale training to very large graphs [10], [11]. Surveys and reviews synthesize these lessons and identify recurring trade-offs [12], [16], [17]. What remains underdeveloped is a common evaluation framework that treats layout as a first-class variable. Such a framework would need to specify graph datasets, hardware profiles, adaptation policies, privacy constraints, and fairness metrics. It would also need to support reproducibility, including seed management and layout versioning.

From a systems architecture perspective, the most promising direction is composable adaptation. Rather than building monolithic systems that hard-code a single layout, future platforms could expose layout as a policy that can be selected, combined, and audited. A compiler might generate multiple layout candidates and use runtime feedback to switch among them. A scheduler might place subgraphs according to both memory pressure and fairness constraints. A governance layer might approve or reject layout transformations based on privacy and retention rules. This vision aligns with broader trends in machine learning systems toward adaptive, hardware-aware, and policy-aware execution. It also acknowledges that graph training is a socio-technical activity, not merely a numerical computation.

9. Open Challenges and Research Agenda

Several open challenges remain. First, adaptation overhead must be bounded. Frequent reordering or repartitioning can consume more resources than it saves. Research is needed on lightweight adaptation triggers, incremental layout updates, and cost models that predict when adaptation will pay off. Second, portability must be improved. Layouts optimized for one accelerator may not transfer to another. Portable intermediate representations and compiler passes could help, but they must preserve performance across diverse backends. Third, robustness must be tested against adversarial graphs and noisy data. Adaptive systems can be gamed if attackers understand how layouts are selected. Fourth, fairness and privacy must be

integrated into layout design rather than added afterward. This requires new metrics, audits, and governance mechanisms.

Fifth, sustainability requires measurement and accountability. Energy and carbon reporting should become standard in graph training research. Sixth, reproducibility must be strengthened. Layout transformations change execution order and floating-point behavior, which can affect results. Systems should record layout versions, randomization seeds, and adaptation histories. Seventh, human factors matter. Operators need tools to understand why a layout was chosen, what data was moved, and what risks were introduced. Explainability at the systems level is as important as explainability at the model level. Finally, interdisciplinary collaboration is essential. Computer scientists, engineers, social scientists, policymakers, and domain experts must work together to ensure that memory-efficient graph training advances both performance and public interest.

10. Conclusion

Memory-efficient graph neural network training via adaptive graph data layouts is a promising but demanding systems research direction. Graph neural networks are powerful because they respect relational structure, but that same structure creates irregular memory access, load imbalance, and scaling challenges. Adaptive layouts offer a way to align graph data with hardware hierarchies, runtime conditions, and distributed infrastructure. They can improve locality, reduce bandwidth pressure, and enable larger graphs to be trained on existing hardware. Yet adaptation is not free. It introduces preprocessing cost, consistency risks, portability challenges, and governance questions. The most effective systems will be those that treat layout as a policy-infused infrastructure layer, with clear triggers, auditable transformations, and multidimensional evaluation. The central conclusion is that memory efficiency in graph neural network training cannot be achieved through a single kernel or format. It requires coordinated design across algorithms, compilers, memory systems, distributed schedulers, and policy frameworks. If pursued with robustness, fairness, and sustainability in mind, adaptive graph data layouts can help make graph learning more scalable, equitable, and environmentally responsible.

References

1. Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In International Conference on Learning Representations.
2. 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).
3. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2018). Graph attention networks. In International Conference on Learning Representations.
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. 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).

6. Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2019). How powerful are graph neural networks? In *International Conference on Learning Representations*.
7. Chen, J., Zhu, J., & Song, L. (2018). Stochastic training of graph convolutional networks with variance reduction. In *Proceedings of the 35th International Conference on Machine Learning* (pp. 941–949).
8. Fey, M., & Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.
9. Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., ... Zhang, Z. (2019). Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*.
10. Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., ... Karypis, G. (2020). DistDGL: Distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)* (pp. 36–44).
11. Liu, Z., Zhou, K., Yang, F., Li, X., & Chen, Z. (2021). PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *Proceedings of the ACM Symposium on Cloud Computing* (pp. 401–415).
12. Abadal, S., Jain, A., Guirado, R., López-Alonso, J., & Alarcón, E. (2021). Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys*, 54(9), 1–38.
13. Jia, Z., Lin, S., Gao, M., Zaharia, M., & Aiken, A. (2019). Improving the accuracy, scalability, and performance of graph neural networks with Roc. In *Proceedings of Machine Learning and Systems* (Vol. 1, pp. 187–198).
14. Chiang, W. L., Liu, X., Si, S., Li, Y., Bengio, S., & Hjelm, R. D. (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 and Data Mining* (pp. 257–266).
15. Zeng, H., Zhou, H., Srivastava, A., Kannan, R., & Prasanna, V. (2020). GraphSAINT: Graph sampling based inductive learning method. In *International Conference on Learning Representations*.
16. 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.
17. Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., ... Sun, M. (2020). Graph neural networks: A review of methods and applications. *AI Open*, 1, 57–81.
18. 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).
19. Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., ... Ng, A. Y. (2012). Large scale distributed deep networks. In *Advances in Neural Information Processing Systems* (Vol. 25, pp. 1223–1231).

20. Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., ... Zaharia, M. (2021). Efficient large-scale language model training on GPU clusters using Megatron-LM. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1–15).