Can Graph Reordering Speed Up Graph Neural Network Training? An Experimental Study

Nikolai Merkel
Technical University of Munich
nikolai.merkel@tum.de

Pierre Toussing
Technical University of Munich
pierre.toussing@tum.de

Ruben Mayer
University of Bayreuth
ruben.mayer@uni-bayreuth.de

Hans-Arno Jacobsen
University of Toronto
jacobsen@eecg.toronto.edu

ABSTRACT

Graph neural networks (GNNs) are a type of neural network capable of learning on graph-structured data. However, training GNNs on large-scale graphs is challenging due to iterative aggregations of high-dimensional features from neighboring vertices within sparse graph structures combined with neural network operations. The sparsity of graphs frequently results in suboptimal memory access patterns and longer training time. *Graph reordering* is an optimization strategy aiming to improve the graph data layout. It has shown to be effective to speed up graph analytics workloads, but its effect on the performance of GNN training has not been investigated yet. The generalization of reordering to GNN performance is nontrivial, as multiple aspects must be considered: GNN hyper-parameters such as the number of layers, the number of hidden dimensions, and the feature size used in the GNN model, neural network operations, large intermediate vertex states, and GPU acceleration.

In our work, we close this gap by performing an empirical evaluation of 12 reordering strategies in two state-of-the-art GNN systems, PyTorch Geometric and Deep Graph Library. Our results show that graph reordering is effective in reducing training time for CPU- and GPU-based training, respectively. Further, we find that GNN hyper-parameters influence the effectiveness of reordering, that reordering metrics play an important role in selecting a reordering strategy, that lightweight reordering performs better for GPU-based than for CPU-based training, and that invested reordering time can in many cases be amortized.

1 INTRODUCTION

Graph neural networks (GNNs) have recently emerged as a promising deep learning-based approach to learn on graph-structured data. GNNs show great success in different domains such as knowledge graphs [50], recommender systems [64], and natural language processing [61]. In GNN training, vertex representations are iteratively learned by aggregating features of the vertices' immediate neighbors followed by neural network transformations. Through this process, vertex representations are iteratively refined and capture both vertex and graph structure information which can be used for down-stream tasks.

Despite the great success of GNNs, the training process is resource-intensive. GNNs operate not only on the graph structure itself but also on high-dimensional feature vectors attached to vertices. Large intermediate representations are computed for the vertices in each layer leading to large memory overheads. Further, neural network operations are performed for each vertex; Therefore, GNNs are often trained on GPUs. Different GNN hyper-parameters such as the number of layers, the number of hidden dimensions, or the feature size influence the training time. Therefore, GNN training differs from traditional graph processing applications such as Breadth-first-search, Connected Components, Shortest Paths, and K-cores which perform lightweight computations, are typically short-running, only lead to small vertex states, are performed on the graph structure only, often exhibit dynamic computational patterns, and are commonly executed on CPUs.

Graph reordering is a technique to optimize the graph's data layout in a way that vertices that are frequently accessed together are stored close to each other in memory to improve locality. It has been shown that traditional graph processing can benefit from graph reordering [2]. Given that GNN training is often performed on scarce and expensive GPUs, reducing training time through graph reordering can lead to cost savings.

Recently, DGI [63], a framework for easy and efficient GNN *model evaluation* has been proposed which also benefits from graph reordering. However, it is not known if graph reordering is effective for GNN training. It is unclear how GNN-specific parameters and GPU-accelerated training influence the effectiveness of graph reordering. Further, it has not been investigated yet if graph reordering quality metrics can be used for graph reordering strategy selection, and if invested graph reordering time can be amortized. To close this research gap, we perform an extensive experimental study to investigate how effective graph reordering is in speeding

up GNN training on CPUs and GPUs. The main contributions of our work are:

(1) We perform extensive evaluations with 12 graph reordering strategies, two predominant GNN systems, different GNN models and GNN hyper-parameters, and 10 real-world graphs. In our experiments, we perform both GPU-based and CPU-based training. Based on our experiments, we find that graph reordering is an effective optimization to reduce GNN training time. A high-quality reordering strategy such as *Rabbit* results in speedups of up to 2.19x (average 1.25x) for CPU-based training and up to 2.43x (average 1.33x) for GPU-based training. We also investigate the effectiveness of graph reordering when sampling is applied. *Rabbit* achieves speedups of up to 3.68x (average 1.62x) for CPU-based training and up to 3.22x (average 1.57x) for GPU-based training.

(2) Graph reordering is a pre-processing task that is performed prior to GNN training. In our experiments, we investigate if invested graph reordering time can be amortized. We show that for CPU-based training, graph reordering time can be amortized by faster training. For GPU-based training, we find that costs can be saved due to reduced training time on monetary expensive GPUs, as the graph reordering can be performed on cheaper CPU machines.

(3) We study the relationship between graph reordering quality metrics and training speedup. We find that the metrics *average gap profile* and *average graph bandwidth* can be used in many cases to select a graph reordering that leads to a large speedup. However, we also find that graph reordering quality metrics are no perfect predictors for GNN speedup indicating that new metrics may be needed to better formalize a graph reordering goal.

Our paper is organized as follows. First, we introduce graph neural network training and graph reordering in Section 2. In Section 3, we describe the methodology of our experimental comparison. Then, we analyze our results for both systems in Section 4.1 and summarize our main findings in Section 5. We discuss related work in Section 6. Finally, we conclude our paper in Section 7.

2 BACKGROUND

Let $G = (V, E)$ be a directed graph consisting of a set of vertices V and a set of edges $E = V \times V$. The out-neighbors of vertex v are defined as $n^+(v) = \{u|(v, u) \in E\}$, the in-neighbors as $n^-(v) = \{u|(u, v) \in E\}$, and all neighbors of v as $n(v) = n^+(v) \cup n^-(v)$. The in- and out-degrees of vertex v are defined as $deg^+(v) = |n^+(v)|$ and $deg^-(v) = |n^-(v)|$, respectively. The mean degree $deg(G)$ of G is defined as $2 \cdot \frac{|E|}{|V|}$.

2.1 Graph Neural Network Training

Graph Neural Networks (GNNs) are a specialized category of neural networks tailored to graph-structured data, leveraging the connections inherent in such data. In GNN training, vertex representations are iteratively learned as follows: Initially, each vertex v is characterized by its feature vector $h_v^{(0)}$. Then, in each subsequent GNN layer, for each vertex v, the learned representations of its neighbors $n(v)$ are aggregated to $a_v^{(k)} = AGG^{(k)}(\{h_u^{(k-1)}|u \in N(v)\})$ and the vertex representation is updated to $h_v^{(k)} = UP^{(k)}(a_v^{(k)}, h_v^{(k-1)})$ by applying an update function UP based on the aggregation $a_v^{(k)}$ and its previous representation $h_v^{(k-1)}$ in layer $k-1$. This process
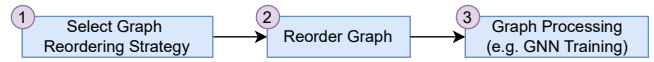

Figure 1: Graph Processing Pipeline.
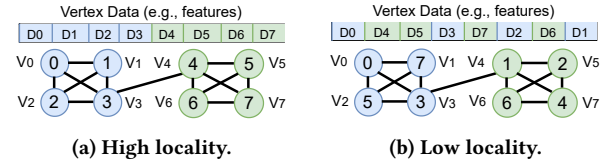

(a) High locality.　　(b) Low locality.

Figure 2: Two different orderings of the same graph.

is *permutation invariant*, meaning the outcome of the learning does not depend on the order of the nodes in the input graph.

2.2 Graph Reordering

Graph reordering is a pre-processing step in a graph processing pipeline (see. Figure 1). Graph reordering improves memory access locality in graph processing systems by locating vertices that are accessed frequently together close to each other in memory [2]. In graph data structures such as compressed sparse row or adjacency matrices, the ID of a vertex is used as a key, e.g., into an array. Therefore, the ID of a vertex influences its position in memory, and vertices with close IDs (small difference between the IDs) are also stored close to each other in memory.

The goal of graph reordering is to relabel the vertices of a graph in a way that vertices that are accessed together get close IDs and therefore are stored in consecutive memory. This optimization only changes the data layout but does not influence the graph structure or the processing result. For example, consider the graphs in Figure 2. The figure shows two different orderings of the *same* graph. The numbers in the cycles represent the ID of the vertices. For example, vertex v_1 has ID 1 in the left graph (see Figure 2a) and ID 7 in the right graph (see Figure 2b). The ID of vertex v_i defines the position of its data D_i in the vertex data array (represented above the graph) in which feature vectors or hidden representations of vertices are stored. The graph has two dense clusters c_1 (colored blue) and c_2 (colored green). Vertices in a cluster share many neighbors while there is only one edge between the clusters. In the left ordering of the graph (see Figure 2a), vertices of the same cluster have consecutive IDs, and their data is therefore consecutively stored in memory. In contrast, in the right ordering of the graph (see Figure 2b), the vertices of a cluster are not labeled consecutively, therefore, their data is spread across the vertex data array. In GNN training, for vertex v_0 the states D_1, D_2 and D_3 of its neighbors $n(v_0) = \{v_1, v_2, v_3\}$ need to be aggregated, which are in consecutive memory in the left ordering (see blue colored vertex data in Figure 2a) but spread across memory in the right ordering (see blue colored vertex data in Figure 2b). Hence, aggregating the neighbors' data in the first ordering has higher access locality than in the second ordering.

2.2.1 Graph Reordering Metrics. Different graph reordering metrics exist that describe reordering quality [5]. Let $\pi : V \to V$ be a bijection mapping vertices to their IDs. The *gap* between two vertices $u, v \in V$ is defined as $\xi_\pi(u, v) = |\pi(u) - \pi(v)|$. The larger the gap between two vertices, the more distant they are

from each other. The *vertex bandwidth* for a vertex $v \in V$ is defined as $\beta_v(G, \pi) = max\{\xi_\pi(v, u) | \forall u \in n(v)\}$, meaning it is defined by the ID of v and the ID of its most distant neighbor. In the following, three quality metrics (lower values are better) are defined: The *average gap profile* for graph G is defined as $\xi(G, \pi) = \frac{1}{|E|} \sum_{u,v \in E} \xi_\pi(u, v)$. The *graph bandwidth* of graph G is defined as: $\beta(G, \pi) = max\{\xi_\pi(v, u) | \forall(v, u) \in E\}$. The *average graph bandwidth* of graph G is defined as: $\widehat{\beta}(G, \pi) = \frac{1}{|V|} \sum_{v \in V} \beta_v(G, \pi)$.

Table 1: Graph reordering strategies used in our study.

Reordering Category	Reordering Strategies
GAP-based	MINLA
Degree- and Hub-based	Degree Sort, Hub Sort, Hub Cluster, SlashBurn
Window-based	Gorder
Partition-based	Rabbit, LDG, Metis
Fill-reducing-based	RCM, DFS, BFS

2.2.2 Graph Reordering Strategies. Graph reordering is a vibrant research field and many different graph reordering strategies exist [2, 5, 11, 29, 47, 58, 69]. According to Barik et al [5], graph reordering strategies can be categorized into the following categories: (1) Degree- and Hub-based graph reordering approaches mainly use the degree information of vertices for reordering. (2) Partition-based approaches divide the vertex set into partitions with the goal of minimizing the number of edges between partitions and balancing the number of vertices per partition. This category also includes community detection and graph clustering approaches, that detect densely connected clusters in graphs. Then, the reordering is performed based on the partitions or clusters, e.g., vertices of the same partition or cluster get consecutive IDs. (3) Window-based approaches slide a window over the vertices and maximize a score for the window. (4) GAP-based approaches minimize the *average gap profile*. (5) Fill-reducing-based approaches reorder a matrix in a way that the number of non-zero elements in the factorized matrix is minimized.

In the following, we introduce for each category graph reordering strategies. An overview can be found in Table 1.

Rabbit: Rabbit order is a lightweight community detection-based graph reordering strategy. Many real-world graphs have a community structure consisting of densely connected vertices that are accessed together in the processing. Rabbit order exploits this property of real-world graphs for reordering [2].

MINLA: The Minimum Linear Arrangement Problem (MINLA) [47] is a combinatorial optimization problem with the goal of minimizing the *average graph bandwidth* (as introduced in Section 2.2.1).

SlashBurn: SlashBurn iteratively identifies and removes hubs from the graph to create many small disconnected components. The goal is to bring the adjacency matrix to block-diagonal form [29].

Gorder: Gorder [58] is window-based graph reordering strategy. Let $S_s(u, v) = |n^-(u) \cap n^-(v)|$ be the number of common in-neighbors of vertices u and v, and let $S_n(u, v)$ be the number of times u and v are connected by an edge which can be either 0, 1 or 2, and let w be the window size. Gorder maximizes the "Gscore" which is defined as $\sum_{0 < \pi(v) - \pi(u) \leq w} (S_n(u, v) + S_s(u, v))$.

Degree Sort: Degree Sort is a lightweight approach where the vertices of the graph are sorted by degree and are consecutively labeled in that order [3].

Hub Sort: Similar to Degree Sort, in Hub Sort, vertices with a degree larger than a given threshold (Hub vertices) are sorted by degree and placed next to each other [69].

Hub Clustering: Similar to Hub Sort with the difference that hub vertices are not sorted but still placed next to each other in their original relative order [3].

RCM: Reverse-Cuthill-McKee (RCM) is a breadth-first-search (BFS) based approach. Different from BFS, vertices are sorted by degree before being added to the queue. The goal of RCM is to reduce the graph bandwidth [11].

DFS: The graph is traversed with depth-first search, and the vertices are labeled in the order in which they are visited.

BFS: The graph is traversed with breadth-first search, and the vertices are labeled in the order in which they are visited.

LDG: Linear Deterministic Greedy (LDG) [53] is a score-based streaming vertex partitioner. Vertices are streamed one after the other along with their neighbors and are assigned to partitions on the fly based on a score. A vertex v is assigned to partition i for which the highest score $s(i) = |P_i \cap n(v)| \cdot (1 - \frac{|P_i|}{C})$ is achieved. P_i is the set of vertices already assigned to partition i and C is a capacity limit per partition, e.g., $C = \frac{|V|}{k}$ with k being the number of partitions. In the first part of the equation, partitions are preferred to which neighbors of v are already assigned, while the second part ensures balancing so that partitions are of similar size.

Metis: Metis is an in-memory vertex partitioner. It uses a multi-level technique consisting of three phases [24]. First, the graph is coarsened, and thereby the size of the graph is reduced. Then, the coarsened graph is partitioned. Finally, the partitioning is projected back to the original graph and refined.

3 EXPERIMENTAL METHODOLOGY

We want to answer the following research questions:

Q1 Is graph reordering effective in reducing training time?

Q2 How do GNN parameters such as *number of layers*, *number of hidden dimensions*, and *feature size* influence the effectiveness of graph reordering for GNN training?

Q3 How does the effectiveness of graph reordering vary across different hardware configurations, including CPU-based training and GPU architectures?

Q4 Can invested graph reordering time be amortized by faster GNN training?

Q5 Are existing graph reordering quality metrics good indicators for GNN training speedup, and can they be used to select a graph reordering strategy?

Q6 Do inherent graph characteristics, such as community structure and density, influence the effectiveness of graph reordering?

Q7 Is graph reordering effective when sampling is applied?

In the following, we introduce our experimental setup to answer the research questions by describing which GNN systems, graph reordering strategies, GNN models, GNN parameters, infrastructure (GPU or CPU), and datasets we selected for our study, and which metrics we measured.

Systems: We perform experiments with the predominant GNN systems Deep Graph Library (DGL) [56] and PyTorch Geometric (PyG) [12]. Both systems are commonly used for GNN training and have large user communities[1], indicating that optimizing both systems with graph reordering is useful to a large user base. Further, both systems provide highly optimized CPU- and GPU-based training.

Reordering Strategies: We selected all 12 graph reordering strategies from five different categories which were introduced in Section 2.2 (see Table 1 for an overview). This selection covers at least one representative of each category. For graph reordering with *Metis*, we observed that the number of partitions can influence the effectiveness, therefore, we selected different numbers of partitions $k \in \{16, 128, 1024, 8192, 65536\}$, and treated each configuration as a different reordering *metis-16*, *metis-128*, *metis-1024*, *metis-8192*, and *metis-65536*. Therefore, we create in total 16 different reorderings per graph. Our common baseline is random graph reordering.

GNN architectures: We selected two representative GNN architectures, GAT [54] and GCN [25] which are commonly used [12, 56]. For both GNN models, the hyperparameters *number of hidden dimensions* and *number of layers* need to be configured. Our literature review indicates that the number of hidden dimensions typically ranges between 16 and 256. Regarding the number of layers, we find that most GNNs are designed with 2 layers as the default, followed by 3 layers, and only in rare instances, 4 layers [13, 17, 22, 30, 56, 60, 63, 72]. We also observe that the number of hidden dimensions is usually smaller than the feature size. Accordingly, we selected the number of hidden dimensions as 16, 32, 64, and 256, and the number of layers as 2, 3, and 4, to encompass the range of commonly used parameters.

Datasets: We selected graphs from different categories to investigate whether the graph reordering effectiveness depends on the graph type. Table 2 reports the graphs along with the number of vertices and edges, the mean degree, and the average local clustering coefficient[2]. In order to investigate the influence of *the number of features* on the graph reordering effectiveness, we set the feature size for all graphs to three different values 16, 64 and 512.

Metrics: We selected different graph reordering metrics (*average gap profile*, *graph bandwidth* and *average graph bandwidth*) introduced in Section 2 to compare graph reordering strategies against each other, to investigate their relationship with GNN training speedup, and to explore their utility for graph reordering selection. Further, we measure graph reordering time and GNN training time.

Infrastructure: We use different hardware setups to investigate the effectiveness of graph reordering for CPU- and GPU-based training. For GPU-based training, we use a machine with 256 GB main memory, 64 cores, and a Nvidia RTX 8000 GPU with 48 GB of VRAM. For CPU-based GNN training, we use a machine with 164 GB main memory and 8 cores. This machine is also used for graph reordering.

---

[1] *DGL* and *PyG* have 12.4k and 18.9k stars on GitHub, respectively. Further, *DGL* and *PyG* are forked 2.9k and 3.4k times on GitHub, respectively.

[2] Let $t(v)$ be the number of triangles connected to vertex v, and $tr(v) = 0.5 \cdot deg(v) \cdot (deg(v) - 1)$ be the number of triplets connected to v. According to [41], the local clustering coefficient of vertex v is defined as $c(v) = \frac{t(v)}{tr(v)}$ and the average local clustering of graph G as $C(G) = \frac{1}{|V|} \cdot \sum_{v \in V} c(v)$.

Table 2: Used graphs along with the number of vertices $|V|$, number of edges $|E|$, mean degree (MD) and average local clustering coefficient (LCC).

Graph	$|V|$	$|E|$	MD	LCC
web-BerkStan [27]	0.69M	6.65M	19.41	0.63
soc-pokec[27]	1.63M	22.30M	27.32	0.12
dimacs9-USA [26]	23.95M	28.85M	2.41	0.02
livejournal[27]	4.00M	34.68M	17.35	0.35
reddit [15]	0.23M	57.31M	491.99	0.58
products [7, 19]	2.40M	61.86M	51.54	0.44
wikipedia [26]	3.60M	77.58M	43.06	0.25
orkut [27]	3.07M	117.19M	76.28	0.17

(a) Speedup distribution. Mean represented as dashed line.

(b) Reduction of cache misses in percent (CPU).

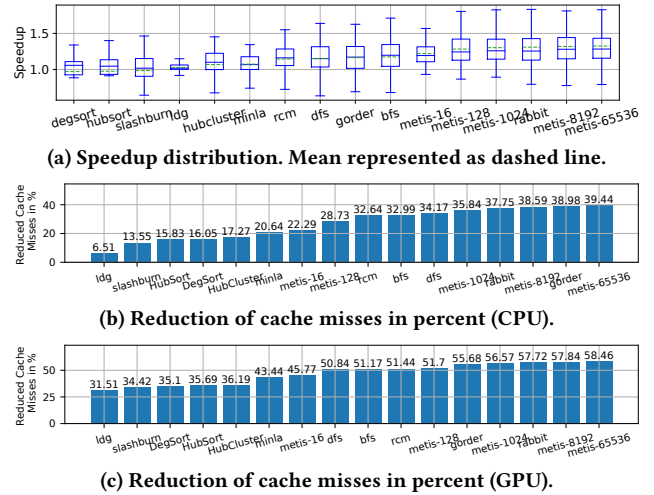**(c) Reduction of cache misses in percent (GPU).**

Figure 3: Overview of the graph reordering strategies in terms of speedup and cache misses. Larger values are better.

4 RESULTS

4.1 Overall Performance

In the following, we report the average speedups for a 2-layer GNN with a medium feature size of 64. The number of hidden dimensions is set to 16 as it is usually smaller than the feature size. Subsequently, we will vary the hyper-parameters to investigate the effectiveness of graph reordering under different configurations.

Figure 3a gives an overview of the average speedup for all graph reordering strategies to indicate which speedups can be expected on average from which strategy. We observe that the degree-based approaches *degsort*, *hubsort*, *slashburn*, lead to speedups of 0.97x, 0.98x, and 0.99x, meaning on average they slow down the training. The partitioning-based approaches *rabbit*, *metis-16*, *metis-128*, *metis-1024*, *metis-8192*, and *metis-65536* are effective in reducing GNN training time and lead to average speedups of 1.31x, 1.22x, 1.28x, 1.30x, 1.32x, and 1.33x, respectively, followed by *bfs*, *gorder*, *dfs*, *rcm*, *minla*, *hubcluster*, and *ldg* with speedups of 1.18x, 1.16x, 1.15x, 1.15x, 1.07x, and 1.03x, respectively. For *Metis*, we observe that the more partitions are used, the larger the speedup, indicating that the number of partitions is an important parameter that influences the effectiveness.

For two graphs, *products* and *web-BerkStan*, we measured cache misses both on the CPU and GPU to investigate if the graph reordering strategies that lead to larger speedups also lead to fewer cache misses. To count cache misses on the GPU, we identified the kernels that are most influenced by graph reordering and profiled the cache misses of these kernels. Both on the CPU and GPU, we observed that better performing graph reordering strategies lead to fewer cache misses, e.g., the best performing graph reordering strategy *metis-65536* reduced the number of cache misses by 39.44% and 58.46% on the CPU and GPU, respectively (see Figures 3b and 3c).

These results are plausible given the nature of GNN training. During GNN training, vertices iteratively aggregate features and hidden representations from neighboring vertices. Hence, vertices that are frequently accessed together, such as those within the same community, should have contiguous IDs to ensure locality in memory. Partitioning-based approaches like *Metis* and *rabbit* are specifically designed to identify communities, assigning similar IDs to vertices within the same community. This results in improved training speed and reduced cache misses. Other approaches, such as *bfs*, *dfs*, or *rcm*, also yield decent results as they preserve some locality by traversing the graph in a manner that inherently maintains structure. However, degree-based approaches are less effective. For instance, sorting vertices by degree and relabeling them in that order fails to respect the community structure, resulting in a data layout that does not reflect this structure. In Section 4.6, we compare the reordering approaches based on quality metrics that indicate data locality. Our findings show that partitioning-based approaches indeed achieve much better reordering quality metrics compared to degree-based approaches.

We conclude that **graph reordering is an effective optimization for reducing cache misses on both CPU and GPU, significantly accelerating GNN training. Particularly, the partitioning-based approaches *Metis* and *rabbit* demonstrate substantial potential**.

4.2 Influence of GNN Parameters

In the following, we investigate how the GNN parameters *hidden dimension*, *number of layers*, and *feature size* influence the effectiveness of graph reordering for GNN training.

4.2.1 Number of hidden dimensions. In the following experiments, we increase the number of hidden dimensions from 16 to 32, 64, and 256. The number of layers is set to 2, and the feature size is fixed at 256, considering that the number of hidden dimensions is typically smaller than the feature size.

CPU: For CPU-based training, we observe for both *PyG* and *DGL* that an increasing number of hidden dimensions reduces the effectiveness of graph reordering in most cases. For example, *rabbit* leads on average to speedups of 1.35x and 1.17x a for a hidden dimension of 16 and 256, respectively (see Figure 4a). These results seem plausible. The larger the number of hidden dimensions, the larger the intermediate vertex representations. In order to compute the representation of a vertex for the next layer, the representations of its neighbors need to be accessed. However, the cache size is limited and the larger the representations, the fewer fit into the cache and graph reordering becomes less effective. To validate this hypothesis, we measure the cache miss rate for the graph reordering

strategies and observe that the graph reordering strategies become indeed less effective in reducing cache misses if the number of hidden dimensions increases. For example, *rabbit* reduces the number of cache misses by 36% and 31% on *products* for a hidden dimension of 16 and 256, respectively (see Figure 7a).

GPU: For GPU-based training, we make similar observations. However, we observe one exception for GCN in *DGL*, where graph reordering is more effective in the face of a larger hidden dimension. Specifically, *rabbit* achieves an average speedup of 1.22x and 1.57x for a hidden dimension of 16 and 256, respectively (see Figure 4b). For GAT, the trends align with those observed in CPU-based training (see Figure 4c). This phenomenon can be attributed to GCN's limited computational demands and its memory-bound nature, where increasing the hidden dimensions necessitates fetching more data, exacerbating memory bandwidth limitations more than computational overhead. Therefore, reordering becomes more important with an increasing number of hidden dimensions in that scenario. We conclude that **in most cases graph reordering becomes less effective with an increasing number of hidden dimensions, but that in GPU-based training of lightweight GNN architectures, the opposite can be the case.**

4.2.2 Feature size. In the following experiments, we increase the feature size from 16 to 64 and 512. The number of hidden dimensions is set to 16, ensuring it is not larger than the feature size. The number of layers is fixed at 2.

CPU: For *DGL*, we observe in most cases, that the graph reordering strategies that lead to a speedup, become less effective in the phase of a larger feature size. For example, *rabbit* leads to an average speedup of 1.33x and 1.24x for a feature size of 16 and 512, respectively (see Figure 5a). For *PyG*, we also observe that a larger feature size decreases the effectiveness of graph reordering.

Regarding the effectiveness in reducing cache misses, we observe that the larger the feature size, the less effective are the graph reordering strategies to reduce the cache misses. These results are plausible, as the larger the feature size, the fewer features fit into the cache and graph reordering becomes less important. For example, *rabbit* reduces the number of cache misses by 60% and 32% for feature sizes of 16 and 512, respectively (see Figure 7b).

GPU: For *DGL*, we observe for GCN, that the effectiveness of graph reordering is decreasing with an increase in the feature size. For example, *rabbit*, leads to average speedups of 1.35x and 1.17x (see Figure 5b) for a feature size of 16 and 512, respectively. However, when training a GAT, we observe that the effectiveness is not much influenced by the feature size (see Figure 5c). Both observations are plausible. The GAT model is much more computation-heavy compared to GCN. Therefore, improving the locality for faster feature loading is not effective.

We conclude that **with an increasing feature size, the effectiveness of graph reordering can decrease.**

4.2.3 Number of layers. In the following experiments, we increase the number of layers from 2 to 3 and 4. The feature size is set to a medium value of 64, and the number of hidden dimensions is fixed at 16, as it is typically smaller than the feature size.

CPU: For CPU-based training, we observe for both *DGL* and *PyG* that the effectiveness of graph reordering is in most cases not much
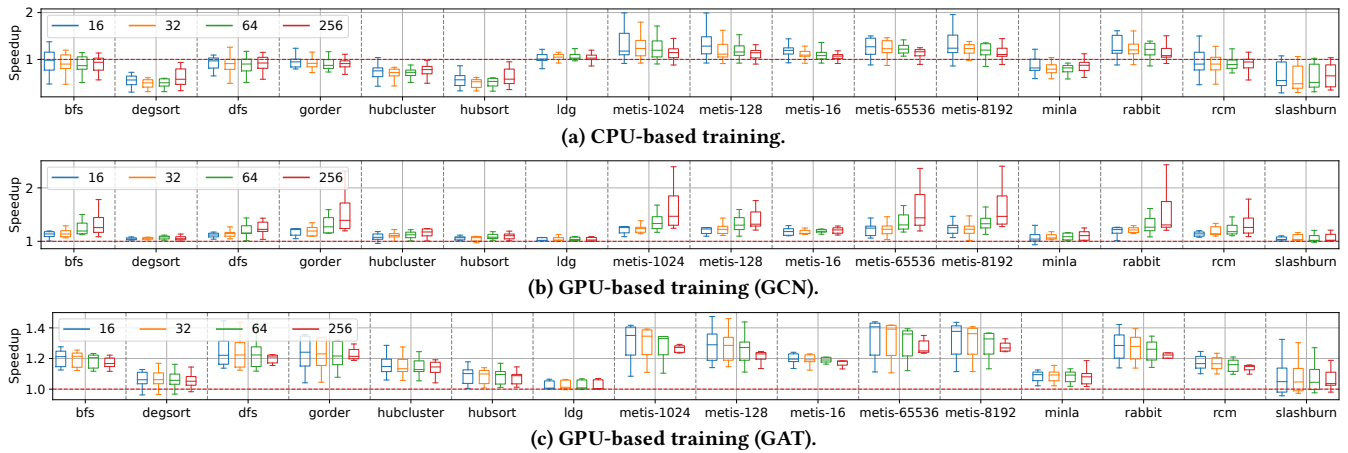
Figure 4: Increasing the number of hidden dimensions from 16 to 32, 64, and to 256 in *DGL*.
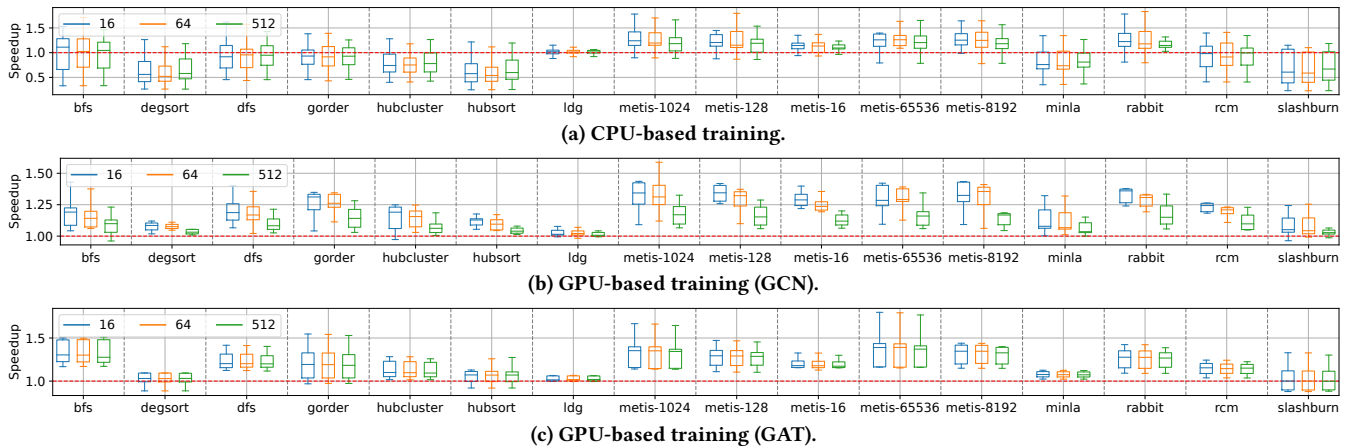

Figure 5: Increasing the feature size from 16 to 64 and to 512 in *DGL*. Larger values are better.
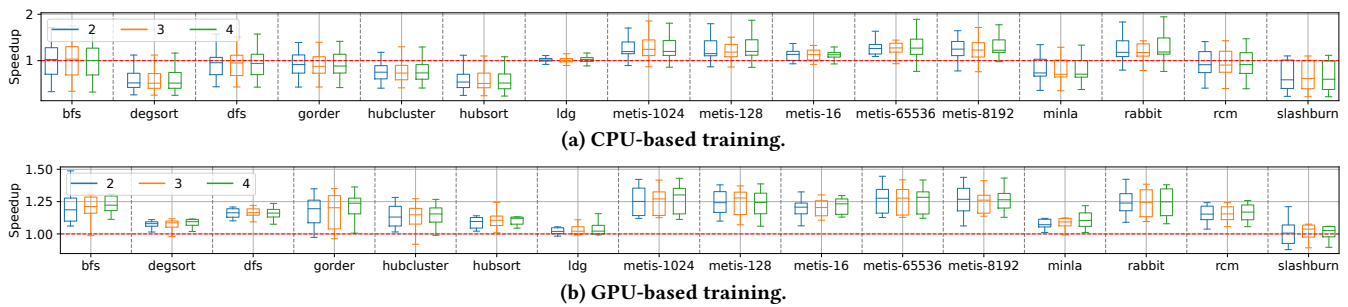

Figure 6: Increasing the number of layers from 2 to 3 and to 4 in *DGL*. Larger values are better.

influenced by the number of layers (see Figure 6a). The effectiveness can slightly increase or decrease, but there is no clear trend.

These results seem reasonable. In each layer, GNN computations are performed and the speedup should be similar per layer, therefore, stacking multiple layers on top of each other should not have much influence on the effectiveness of graph reordering. We also observe

that the effectiveness in reducing cache misses on *products* is not much dependent on the number of layers (see Figure 7c).

GPU: For GPU-based training, we make similar observations (see Figure 6b).

We conclude that **the number of layers has no clear influence on the effectiveness of graph reordering.**
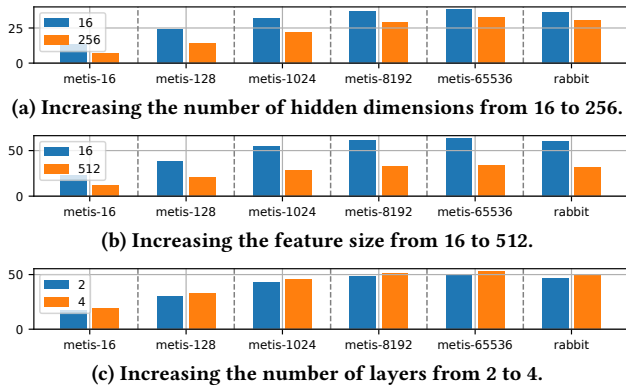
(a) Increasing the number of hidden dimensions from 16 to 256.

(b) Increasing the feature size from 16 to 512.

(c) Increasing the number of layers from 2 to 4.

Figure 7: Influence of increasing GNN parameters on cache miss reduction. Larger values are better.

(a) Speedups on CPU (DGL).

(b) Speedups on GPU (DGL).

(c) Speedups on CPU (PyG).

(d) Speedups on GPU (PyG).

Figure 8: Comparison of speedups achieved for GPU- and CPU-based training with DGL and PyG.

4.3 Influence of Graph Characteristics

In the following, we investigate how graph characteristics influence the effectiveness of graph reordering for both CPU- and GPU-based GNN training. To enable controlled experiments, we use different graph generators to create synthetic graphs with different varying properties: Specifically, we vary the mean degree and the local clustering coefficient, both of which are well-known and important metrics for describing graph datasets [41]. The mean degree influences the density of the graph, while the clustering coefficient indicates the presence of community structures.

Clustering Coefficient. We use the small-world model introduced by Watts and Strogatz [57] as it can be used to generate graphs with different clustering coefficients while keeping the number of edges and vertices constant. We create 6 graphs with clustering coefficients of 0.09, 0.16, 0.25, 0.38, 0.53, and 0.73 (higher numbers indicate a stronger community structure). Each graph consists of 3 M vertices and 60 M edges. We observe a strong correlation (Pearson correlation coefficients and Spearman rank correlation coefficients larger 0.91) between high clustering coefficients and high speedups for both *DGL* and *PyG* for CPU- and GPU-based training. This indicates that for graphs with a stronger community structure, graph reordering is more effective in speeding up GNN training. This result seems plausible because if a graph contains a community structure, this locality can be exploited.

Mean Degree. We use the R-MAT [8] model and Barabási–Albert preferential attachment [4] to generate graphs with different mean degrees. For both generators, we set the number of vertices to 3 M and create 6 graphs with mean degrees ranging from 10 to 120. We observe that the mean degree does not correlate with the speedup for both generators, indicating that the effectiveness of graph reordering is independent of the mean degree. One possible reason is that while the mean degree influences the overall graph density, it does not significantly affect the locality or community structure within the graph, which are more critical for the effectiveness of reordering techniques.

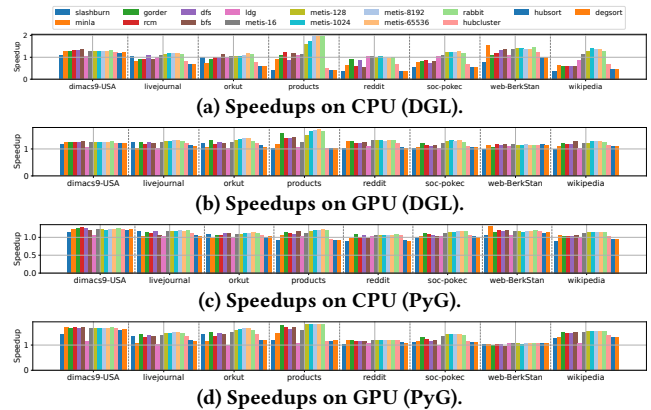We conclude that **graph reordering is more effective for graphs with stronger community structures**.

4.4 Influence of Infastructure

In the following, we investigate the effectiveness of graph reordering on different GPUs in *DGL*. In addition, to the NVIDIA RTX 8000 GPU (Turin architecture, 48 GB memory, 672 GB/s memory bandwidth, 16.3 TFLOPS peak), we use the NVIDIA Titan X GPU (Maxwell architecture, 12 GB memory, 337 GB/s memory bandwidth, 6.7 TFLOPS peak) representing an older architecture, and an NVIDIA Orin (Ampere architecture, 48 GB memory, 205 GB/s memory bandwidth, 5.3 TFLOPS peak) representing a newer architecture.

We observe that graph reordering is effective for all GPUs leading to average speedups of 1.13x, 1.17x, and 1.22x for Titan X, RTX 8000, and Orin, respectively. When employing an advanced reordering strategy such as *rabbit*, even higher average speedups of 1.19x, 1.27x, and 1.33x are achieved on the Titan X, RTX 8000, and Orin, respectively.

These results are plausible given that GNN training is memory-bottlenecked, making data layout improvements particularly beneficial. One possible explanation for the varying effectiveness of graph reordering across different GPUs could be the ratio of peak FLOPS to memory bandwidth: We observed that for all three GPUs, the higher the ratio of FLOPS per memory bandwidth, the more beneficial the graph reordering.

We conclude that **graph reordering is effective across different GPUs and is especially beneficial when the ratio of computational performance to memory bandwidth is high**.

4.5 GPU vs. CPU-based Training

In the following, we analyze how the effectiveness of graph reordering is influenced by the infrastructure (GPU and CPU). Due to limited GPU memory, it was not possible to run all experiments on the GPU (GPU memory is only 48 GB while the CPU server had 164 GB of memory). Therefore, we exclude experiments that did not run on both the CPU and GPU from the following discussion.

DGL: We report our results for CPU- and GPU-based training in Figures 8a and 8b, respectively. We observe for the graph *dimacs9-USA* that there is not much difference in terms of speedup for the different graph reordering strategies when comparing training on
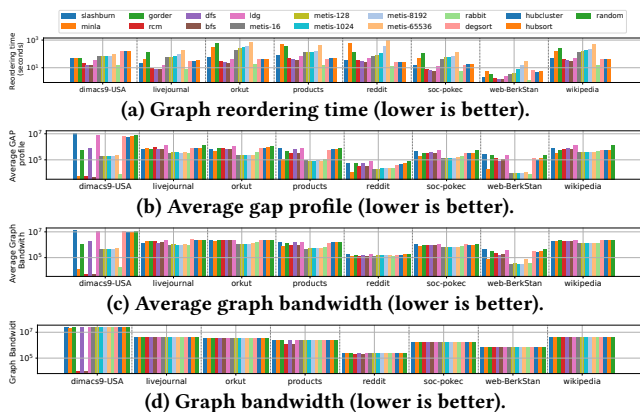
(a) Graph reordering time (lower is better).

(b) Average gap profile (lower is better).

(c) Average graph bandwidth (lower is better).

(d) Graph bandwidth (lower is better).

Figure 9: Reordering times and different reordering metrics.

the CPU versus training on the GPU. For *dimacs9-USA*, the largest difference is observed for *slashburn*, leading to a speedup of 1.08x on the CPU and a higher speedup of 1.15x on the GPU. On the remaining graphs, we observe that some strategies (mainly *slashburn*, *minla*, *hubcluster*, *hubsort*, *degsort*, *rcm*, and *dfs*) lead to slowdowns (speedups smaller 1x) for CPU-based training. However, on the GPU, these strategies perform much better and lead nearly in all cases to speedups. On average the speedup of graph reordering for GPU-based training is 1.28 times the speedup of CPU-based training. Interestingly, even some of the lightweight strategies can lead to speedups close to heavyweight graph reordering with *Metis* when training on the GPU. For example, *dfs* leads to speedups of 1.25x, 1.25x, 1.20x, 1.13x, and 1.18x on *dimacs9-USA*, *livejournal*, *reddit*, *web-BerkStan*, and *wikipedia* which is not much worse compared to *metis-65536* leading to 1.24x, 1.30x, 1.30x, 1.12x, and 1.27x.

PyG: Figures 8c and 8d show our results for CPU- and GPU-based training, respectively. Similar to DGL, we observe that graph reordering is more effective for GPU-based training than for CPU-based training. Also similar to DGL, for GPU-based training all graph reordering strategies lead to speedups compared to CPU-based training where some led to slowdowns. The average speedup on the GPU is 1.17 times the speedup of CPU-based training. It can also be seen that even lightweight strategies lead to good speedups on the GPU. For example, *dfs* leads to a speedup of 1.67x, 1.38x, 1.47x, 1.61x, 1.14x, 1.16x, 1.02x, and 1.44x on the graphs *dimacs9-USA*, *livejournal*, *orkut*, *products*, *reddit*, *soc-pokec*, *web-BerkStan*, and *wikipedia*, respectively. This is close to *metis-65536* with speedups of 1.67x, 1.48x, 1.66x, 1.83x, 1.17x, 1.43x, 1.06x, and 1.54x.

The results seem plausible. Due to the high computational performance of GPUs, they are more memory bottlenecked compared to CPUs, making data layout improvements more beneficial.

We conclude that **both in *DGL* and *PyG*, graph reordering is more effective for GPU training**. Further, we find that **for both systems lightweight graph reordering methods can lead to good results for GPU-based training, which was not the case for CPU-based training.**

4.6 Graph Reordering Quality Metrics

In the following, we analyze how different graph reordering metrics correlate with GNN training speedup. We compute the graph reordering metrics *average gap profile*, *graph bandwidth*, and *average graph bandwidth* introduced in Section 2 for each combination of graph and reordering strategy and normalize it with random reordering. Then, we compute the Pearson correlation coefficient and the Spearman rank correlation coefficient between the normalized reordering metrics and training speedup. Table 3 reports the results.

Table 3: Correlation between reordering metrics and training speedup measured with Pearson correlation coefficient (PC) and Spearman rank correlation coefficient (SC) and achieved speedup for selecting the best reordering strategy (Opt.), selecting the ordering strategy based on quality metrics (Qua.), and randomly selecting a reordering strategy (Ran.).

Metric	Sys.	PC	SC	Opt.	Qua.	Ran.
Avg. gap prof.	DGL	0.58	0.63	1.39	1.21	1.10
Avg. gap prof.	PyG	0.56	0.64	1.31	1.24	1.19
Avg. graph bdw.	DGL	0.61	0.64	1.39	1.21	1.10
Avg. graph bdw.	PyG	0.58	0.64	1.31	1.25	1.19
Graph bdw.	DGL	0.11	0.30	1.39	1.10	1.10
Graph bdw.	PyG	0.13	0.29	1.31	1.22	1.19

We observe for both *DGL* and *PyG*, that the *average graph bandwidth* leads to a Pearson correlation coefficient of 0.61 and 0.58, respectively, which is slightly higher than the *average gap profile* for which coefficients of 0.58 and 0.56 are achieved, respectively. In terms of Spearman rank correlation coefficient, for *DGL*, *average graph bandwidth* leads to a better coefficient of 0.64 compared to a coefficient of 0.63 for *average gap profile*. For *PyG*, for both *average gap profile* and *average graph bandwidth* a coefficient of 0.64 is achieved. This means both *average gap profile* and *average graph bandwidth* correlate with the speedup. In contrast, the quality metric *graph bandwidth* leads to a low Pearson correlation coefficient smaller 0.13 and a low Spearman rank correlation coefficient smaller 0.30. We also observe that the values for *graph bandwidth* are very similar across the various graph reordering strategies, however, as reported above, different orderings lead to different speedups. Therefore, *graph bandwidth* does not seem to be an expressive graph reordering metric.

In Figure 10, we plot the *average graph bandwidth* normalized with random reordering (lower is better) against speedup for the graphs *wikipedia* and *soc-pokec* for CPU and GPU-based training using both systems *PyG* and *DGL* on the GCN model. In many cases, a small *average graph bandwidth* value also leads to larger speedups, both for GPU and CPU training, indicating that minimizing the reordering metric is beneficial (see top left in Figures 10a-10d). However, there are also exceptions. For example, for the graph *wikipedia*, we observe that CPU training with *gorder* leads to high speedups on *DGL* and *PyG* comparable to *Metis*, although the *average graph bandwidth* of *gorder* is higher (worse). For the graph *soc-pokec*, *metis-16* leads to the best (lowest) *average graph bandwidth*, but in both *DGL* and *PyG* other graph reorderings lead to larger speedups on CPU and GPU. We make similar observations for GAT.

(a) Wiki graph (DGL).   **(b) Wiki graph (PyG).**   **(c) Soc-Poket graph (DGL).**   **(d) Soc-Poket graph (PyG).**
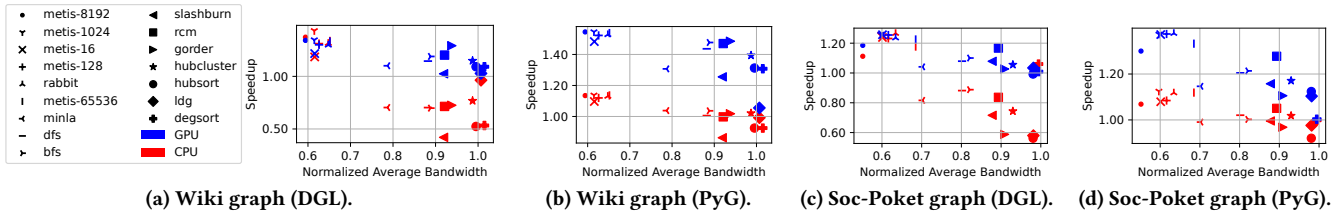
Figure 10: Average graph bandwidth normalized with random reordering versus speedup. The markers represent the graph reordering strategy, and the color of the markers indicates whether the training was performed on the GPU or CPU.

While it is plausible that in most cases a good reordering quality as measured by standard reordering metrics indicates high effectiveness, there are notable limitations. For instance, consider the metric *average gap profile*. Suppose we have two edges, $e_1 = (v_1, v_2)$ and $e_2 = (v_3, v_4)$, and the degree of all four vertices is 1. Now, let's examine two different reorderings: (1) v_1, v_2, v_3, and v_4 have IDs 1K, 100K, 900K, and 990K, respectively. (2) v_1, v_2, v_3, and v_4 have IDs 1K, 900K, 100K, and 990K, respectively. This means reordering (2) only differs from reordering (1) by switching the IDs of v_2 and v_3. The *gap* for both edges would be 180K and 1780K for the first and second reordering, respectively. Thus, the second reordering appears much worse compared to the first one. However, in the first reordering, the connected vertices are already far apart in memory, leading to cache misses. Consequently, both reorderings might be equally suboptimal, but the metric shows a much worse score for the second one. This suggests that once the *gap* between two vertices exceeds a certain threshold, the exact distance may no longer significantly impact performance, and the absolute value of the metric may no longer be representative. Similar problems occur in the other reordering metrics.

We conclude that **the graph reordering metrics *average graph bandwidth* and *average gap profile* correlate with GNN training speedup while *graph bandwidth* does not. However, there are exceptions indicating that the graph reordering metrics are not perfect, and more descriptive metrics may need to be developed.**

4.7 Graph Reordering Selection

In the following, we investigate if the graph reordering metrics *average gap profile*, *graph bandwidth*, and *average graph bandwidth* can be used to select a graph reordering strategy. For each graph, we select the graph reordering strategy that leads to the best reordering according to the quality metrics. This selection is independent of the GNN parameters and only depends on the graph. We compare this selection with the optimal selection, meaning we assume to already know for *each* scenario which graph reordering strategy performs best. The optimal selection is dependent on the GNN parameters, e.g., for a given graph not always the same graph reordering strategy is best. Random selection means that for each scenario a graph reordering strategy is selected randomly.

For both *DGL* and *PyG*, we find that using any one of the graph reordering metrics is at least as good as a random selection (see Table 3). For *DGL*, if the selection is based on *average gap profile*, *average graph bandwidth*, and *graph bandwidth*, we achieve average speedups of 1.21x, 1.21x, and 1.1x which is between a random selection (1.10x) and the optimal selection (1.39x). For *PyG*, if the

selection is based on *average gap profile*, *average graph bandwidth*, and *graph bandwidth*, we achieve average speedups of 1.24x, 1.25x, and 1.22x which are also between a random selection (1.19x) and the optimal selection (1.31x). For both *DGL* and *PyG*, *average graph bandwidth* leads to the best selections, consistent with the results from Section 4.6, which also highlights the limitations of the current reordering metrics. Therefore, we anticipate that more descriptive metrics will enhance reordering selection.

We conclude that **using the graph reordering metric *average gap profile* for graph reordering strategy selection outperforms a random selection, but there is still room for improvement to select the optimal strategy.**

4.8 Graph Reordering Time Amortization

Graph reordering is a pre-processing step that is invested to minimize graph processing time. In the following, we investigate after how many training epochs the graph reordering time can be amortized. We highlight, that training is often performed for hundreds of epochs, and often a hyper-parameter search is performed leading to even more training epochs. As mentioned in Section 4.5, some graph reordering strategies can lead to slowdowns and therefore amortization is not possible. In the following, we only consider scenarios in which graph reordering leads to speedups.

CPU: We observe that in many cases graph reordering is amortized in far less than 100 epochs for both GAT and GCN. For example in *DGL*, *metis-1024* amortizes in 4.0, 22.6, 4.5, 21.6, 3.7, and 11.1 epochs for the graphs *dimacs9-USA*, *livejournal*, *products*, *soc-pokec*, *web-BerkStan*, and *wikipedia* when training a GAT and in 21.0, 77.9, 29.2, 128.4, 416.2, and 61.0 epochs when training a GCN (see Figures 11a and 11b). *Rabbit* amortizes even faster in 0.8, 4.3, 0.4, 2.7, 0.6, and 1.6 epochs for the graphs *dimacs9-USA*, *livejournal*, *products*, *soc-pokec*, *web-BerkStan*, and *wikipedia* when training a GAT and in 3.5, 12.2, 2.9, 16.7, 10.3, and 6.6 epochs when training a GCN, respectively (see Figure 11b). In general training a GAT takes more time and graph reordering time can be amortized faster.

GPU: We also observe that graph reordering amortizes faster for GAT than for GCN. In general, GPU-based training is much faster than CPU-based training and graph reordering can take more time than training. For example in *DGL metis-1024* amortizes only after 188 and 4155 epochs when training a GAT and a GCN on *dimacs9-USA*, respectively (see Figures 11c and 11d). However, it is worth noting that *rabbit* amortizes in 39.8, 55.8, 81.1, 37.0, 159.7, 80.9, 49.4, and 75.7 epochs on the graphs *dimacs9-USA*, *livejournal*, *orkut*, *products*, *reddit*, *soc-pokec*, *web-BerkStan*, and *wikipedia*, respectively, when training a GAT (see Figure 11c). This means that even for GPU-based training graph reordering can be amortized. However, even

(a) GAT on DGL (CPU).

(b) GCN on DGL (CPU).

(c) GAT on DGL (GPU).
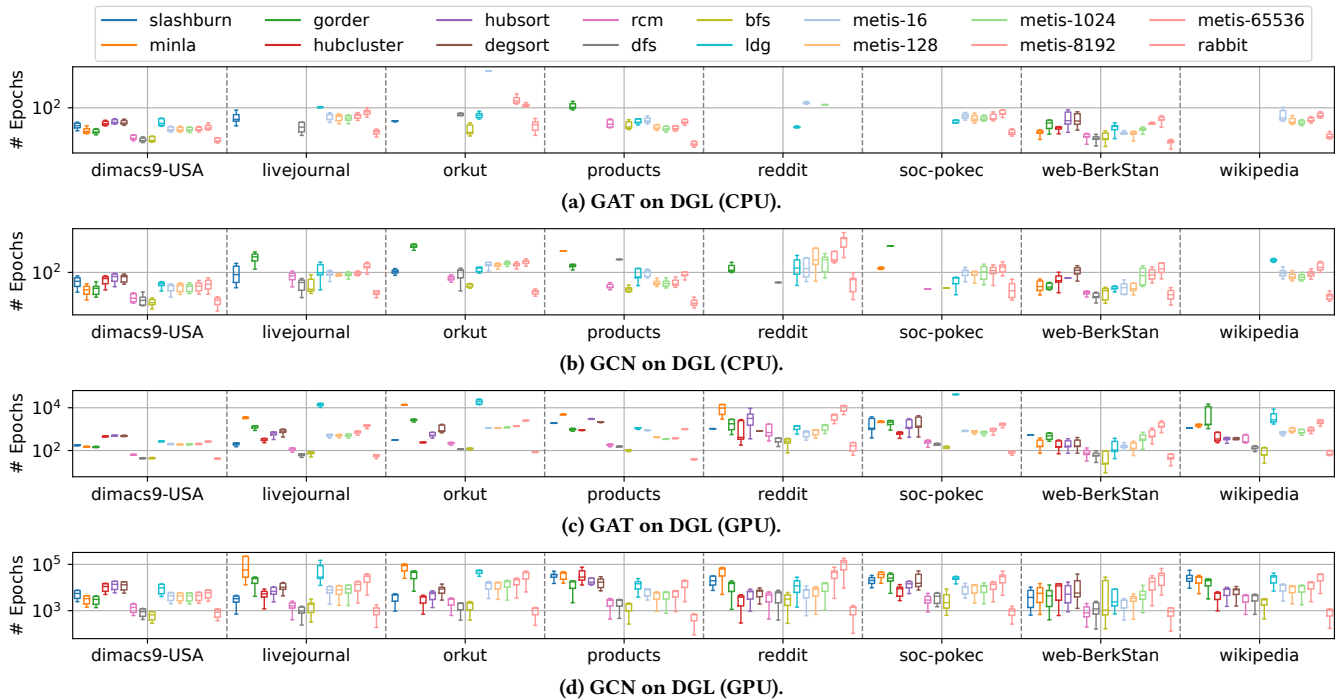

(d) GCN on DGL (GPU).

Figure 11: Number of training epochs until amortization. Lower values are better.

more important, graph reordering can be performed on a cheap CPU server and the training can be performed on a more expensive GPU afterwards. Therefore, we suggest pre-processing graphs on a CPU server to save costs when training on a GPU server.

We conclude that **for CPU-based training graph reordering time can be amortized in many cases. For GPU-based training, the training time can not always be amortized, however, pre-processing on a CPU can still save GPU costs in the training process.**

4.9 Sampling

We also investigate the effectiveness of graph reordering when neighborhood sampling is applied. In addition to the eight graphs in Table 2, we added two large-scale graphs, *twitter* [26] and *papers100* from the Open Graph Benchmark (OGB) [19], with 1.5 billion edges and 1.6 billion edges, respectively. For both graphs, full graph training is infeasible on a GPU due to memory limitations, necessitating the use of neighborhood sampling. Neighborhood sampling requires setting the fanout parameters, denoted as $(l_1, l_2, ..., l_n)$, where l_i represents the number of neighbors to sample in the ith layer. For our experiments, we use commonly employed fanout parameters: (25, 10) for 2-layer experiments [13, 16, 65], and a fanout of (15, 10, 5) for 3-layer experiments [23, 60, 65, 72]. The batch size is set to 1024. For both *DGL* and *PyG*, we use the GCN model for CPU- and GPU-based training on a server with 512 GB main memory and a NVIDIA A40 GPU with 48 GB device memory.

For *PyG*, we observe that graph reordering leads to significant speedups which are shown in Figures 12a and 12b for CPU- and GPU-based training, respectively. When training on a CPU, effective reordering strategies such as *rabbit* and *metis-65536* yield

(a) CPU-based training.

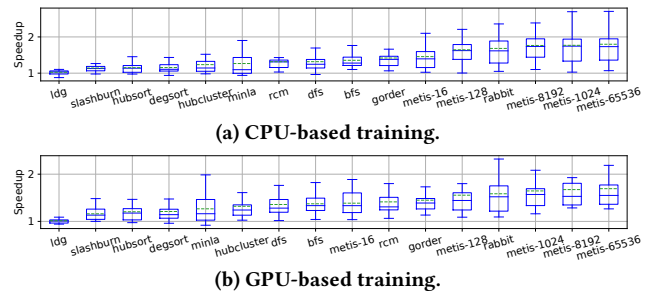**(b) GPU-based training.**

Figure 12: Speedup distribution for sampling-based training.

average speedups of 1.68x and 1.80x, and maximum speedups of 3.35x and 3.82x, respectively. For GPU-based training, *rabbit* and *metis-65536* achieve average speedups of 1.59x and 1.69x, and maximum speedups of 2.84x and 3.30x, respectively.

In the following, we investigate how the fanout influences the effectiveness of graph reordering. We experiment with three different fanout configurations: We use a *low* fanout of (5, 5) and (5, 5, 5), a *medium* fanout of (10, 10) and (10, 10, 10), and a *high* fanout of (15, 15) and (15, 15, 15), for 2-layer and 3-layer GNNs, respectively. Figure 13 shows the average speedup for all combinations of fanout and number of layers for CPU and GPU-based training, respectively. We observe that graph reordering is more effective in the face of higher fanouts: On the CPU, the speedup increases from 1.17x to 1.26x for a 2-layer GNN and from 1.38x to 1.62x for a 3-layer GNN as the fanout increases from low to high. On the GPU, the speedup increases from 1.24x to 1.26x for a 2-layer GNN and from 1.39x to 1.62x for a 3-layer GNN. Further, we observe that if the number
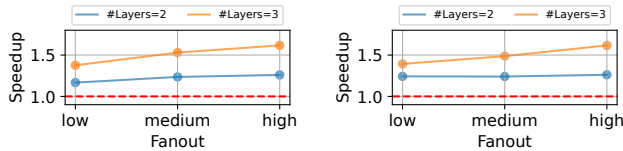
Figure 13: Speedup versus fanout. CPU (left) and GPU (right).

of layers increases, graph reordering becomes more effective: For instance, on the CPU, the speedup for a low fanout increases from 1.17x to 1.38x when increasing the number of layers from two to three, and for a high fanout, it increases from 1.27x to 1.66x. This is plausible as larger fanouts and more layers result in more sampling and feature fetching, making graph reordering more beneficial.

We further analyze which phase of the GNN training benefits from the speedup. GNN training with neighborhood sampling consists of two phases which are iteratively repeated: (1) sample a mini batch from the input graph and extract necessary features (data loading phase) and (2) perform the training on the sample (training phase). For CPU-based training, we find that for a low fanout, 41% of the speedup is achieved in the data loading phase and 59% in the training phase, while for a high fanout, 61% of the speedup is achieved in the data loading phase and 39% in the training phase. For GPU-based training, these figures are 88% and 12% for low fanout, and 99% and 1% for high fanout, respectively.

For *DGL*, we observe average speedups of 1.05x and 1.00x for CPU and GPU-based training, respectively, indicating that graph reordering is not effective to speed up GNN training in *DGL* when neighbor sampling is applied. We attribute this phenomenon to differences in the data loading implementation in *DGL*, which does not benefit from locality in graph ordering.

Additionally, we assess whether graph reordering time can be amortized in *PyG*. For CPU-based training, *rabbit* amortizes in 0.17, 0.20, 0.33, 0.35. 0.39. 0.57, 0.84, 1.23, 3.20, and 6.1 epochs on the graphs *products*, *livejournal*, *soc-pokec*, *wikipedia*, *orkut*, *web-BerkStan*, *dimacs9-USA*, *reddit*, *papers100*, and *twitter*, respectively. For GPU-based training, *rabbit* amortizes in 0.92, 0.95, 1.21, 1.75, 1.85, 2.20, 3.67, 4.36, 6.65, and 9.87 epochs on the graphs *products*, *dimacs9-USA*, *livejournal*, *web-BerkStan*, *wikipedia*, *orkut*, *soc-pokec*, *twitter*, *papers100*, and *reddit*, respectively. Therefore, graph reordering can be amortized for both CPU and GPU-based training, making it a practical optimization. Furthermore, graph reordering can be performed on a cost-effective CPU server, reducing the monetary cost of expensive GPU training time.

We conclude that **graph reordering is effective in *PyG* when sampling is applied, but not in *DGL*. Moreover, graph reordering is more effective with higher fanouts or an increased number of layers. Lastly, graph reordering time can be amortized for both CPU- and GPU-based training, highlighting its practicality as an optimization technique.**

5 LESSONS LEARNED

In the following, we summarize our main findings and relate them to the research questions introduced in Section 3.

Graph reordering can reduce GNN training time (Q1). Our experiments show that graph reordering is an effective optimization to reduce training time leading to speedups of up to 2.19x

(average 1.25x) and to 2.43x (average 1.33x) for CPU- and GPU-based training, respectively, when applying *Rabbit*. We observe large differences between the graph reordering strategies, however, in most cases the partition-based approaches *Metis* and *rabbit* lead to the largest speedups.

GNN-specific parameters can influence the effectiveness of graph reordering (Q2). For both CPU- and GPU-based training, we find that an increasing number of hidden dimensions and an increasing feature size can decrease the effectiveness of graph reordering while the number of layers has only little influence.

Graph reordering is effective across different hardware configurations (Q3). We find that graph reordering can speed up GNN training on a CPU and GPU. For both *DGL* and *PyG*, the achieved speedups on GPUs are larger than for CPU-based training. We observe that mostly the degree-based approaches can lead to a slowdown for CPU-based training, while they lead to speedups when training on a GPU. Further, we find that graph reordering is effective on different GPUs, and that especially if the GPU's ratio of computation performance to memory bandwidth is high, reordering is more beneficial.

Graph reordering time can be amortized (Q4). We find that graph reordering time can be amortized in many cases when training on a CPU. We observed that amortization is dependent on the GNN model architecture, e.g., GAT epoch times are higher than GCN epoch times, and therefore, amortization can be achieved faster. However, when training on a GPU, it depends on the model if graph reordering time can be amortized by faster training. Still, we suggest applying graph reordering when training on a GPU as the graph reordering can be performed on a cheap CPU machine, and monetary expensive training time on the GPU can be reduced.

Graph reordering quality metrics can be used for graph reordering selection (Q5). We find a correlation between graph reordering quality metrics and training speedup. Further, our experiments show that selecting a graph reordering strategy based on graph reordering quality metrics is much better than randomly selecting a strategy. However, there is room for improvement indicating that better graph reordering metrics need to be developed. As it is not straightforward to select the best reordering strategy for a given scenario, we see building an automatic, machine learning-based selector as a promising research direction. Such an approach has been successfully applied to graph partitioner selection for distributed graph processing [39]. For practitioners, we make the following recommendation: (1) For GPU-based training, we recommend performing the graph reordering on a cost-effective CPU machine to save monetary costs when training on the GPU. *Metis* should be selected if the pre-processing machine has sufficient memory, otherwise *rabbit* should be selected as it is much more light-weight in terms of memory and reordering time but still leads to good speedups. (2) For CPU-based training, we recommend applying *rabbit* for its quick amortization. However, if a hyper-parameter search is performed or a more expensive model such as GAT is trained, *Metis* should be applied. (3) For sampling-based training in *PyG* (CPU and GPU), we suggest applying *Metis* if enough memory is available, otherwise *rabbit* should be used.

Graph properties can influence the effectiveness of graph reordering (Q6). We find that graph reordering is more effective

for graphs with a stronger community structure, while the graph density has a lower impact.

Graph reordering can be effective if sampling is applied (Q7). In *PyG*, graph reordering significantly speeds up GNN training, particularly by accelerating the data loading phase (sampling and feature loading). The high-quality strategy *Rabbit* achieves speedups of up to 3.68x (average 1.62x) for CPU-based training and up to 3.22x (average 1.57x) for GPU-based training. Graph reordering is more effective with higher fanouts and more layers, and reordering time can be amortized faster compared to full graph training.

6 RELATED WORK

Graph reordering to optimize graph analytics. Graph reordering is a vibrant research area [2, 11, 29, 58] and different graph reordering strategies exist. It was shown that graph reordering can speed up graph analytics such as Breadth-first-search, Shortest Paths, or K-cores, but graph reordering time often can not be amortized by faster analytics.

In our work, we investigate graph reordering for GNN training which has unique characteristics such as high-dimensional intermediate states which are iteratively aggregated, high-dimensional feature vectors, neural network operations, and is executed for hundreds of epochs. Further, GNN training is often performed on GPUs to accelerate computationally-extensive neural network operations. Our extensive evaluation of 12 different reordering strategies shows the importance of graph reordering for both CPU- and GPU-based GNN training. Further, we investigate graph reordering metrics and their relationship with training speedup to understand if a maximization of the metrics is beneficial to achieve training speedups. In addition, we investigate if quality metrics can be used for the selection of graph reordering strategies.

Graph neural network systems. The recent success of GNNs led to many systems [12, 13, 30–33, 38, 46, 56, 59, 70, 72, 74] tailored to the specific needs of GNNs, however, it has not been investigated in which scenarios GNN systems can benefit from graph reordering.

In our work, we selected the two predominant systems *DGL* and *PyG* which are widely used and have large user communities. For both systems, we find that graph reordering is an effective optimization to speed up the training process on CPUs and GPUs.

We anticipate that graph reordering is also beneficial for other systems: Many systems such as DistDGL [72], GraphStorm [73], PaGraph [30], P3 [13], DistGNN [38], BGL [33], GNNLab [60], and DiskGNN [32] are built on top of *DGL* or Salient [23] which is built on top of *PyG*, and therefore, these systems may also benefit from graph reordering. Further, many research papers [20, 30, 59] mention that poor data locality is a significant challenge for efficient GNN training. Graph reordering directly addresses this issue and therefore may be effective also in other systems.

Optimizations to speed up GNNs. A recent study on GNN acceleration [35] identifies three graph-level optimizations: (1) graph partitioning, (2) graph sparsification, and (3) graph sampling. Below, we relate graph reordering to these optimizations.

(1) Graph partitioning Graph partitioning [18, 36, 37, 42, 45, 48, 68] is a pre-processing step for distributed graph processing systems with the goal to maximize locality on the machines of a compute cluster by assigning densely connected subgraphs (partitions) to machines. Thereby, communication between machines is minimized as vertices that are processed together are on the same machine. This optimization is similar to graph reordering where vertices that are frequently accessed together are stored close by in the memory. Different experimental studies [1, 14, 43, 55] investigated the effectiveness of graph partitioning for distributed graph analytics and showed that high-quality partitioning can speed up distributed graph processing. It was also shown that selecting a graph partitioning algorithm is challenging and that a machine learning-based selection works best [39]. In a recent experimental study [40], the effectiveness of graph partitioning was investigated for distributed GNN training and showed that GNN training time can be decreased by high-quality graph partitioning.

In our work, we investigate graph reordering as a related preprocessing step that optimizes the graph data layout to speed up GNN training on a *single* machine. Further, we find that the graph partitioning-based reordering approaches *Metis* and *rabbit* can also be used for graph reordering, and are effective in reducing GNN training time by improving the data locality. Similar to machine learning-based partitioner selection proposed in [39], we see applying such an approach to graph reordering selection as a promising research direction.

(2) Graph Sparsification. Graph sparsification reduces the graph size by dropping edges, thus decreasing memory footprint and speeding up graph processing [51]. It has been used to speed up graph processing [6, 21] and has also been applied to GNN training to reduce computation time and memory requirements [28, 52]. Graph sparsification can also improve the prediction performance of GNNs [49, 62, 71].

Graph sparsification and graph reordering are orthogonal optimizations. Reordering can still be applied to enhance data locality even after sparsification.

(3) Graph Sampling. Graph sampling reduces computational complexity and memory footprint by sampling a subset of nodes or edges for training, thus improving the efficiency and scalability of GNN training. According to a recent study [34], numerous graph sampling approaches exist that can be categorized into node-wise sampling [16, 44], layer-wise sampling [9, 75], subgraph-based sampling [10, 66], and heterogenous sampling [67].

Our study reveals that graph reordering can also accelerate sampling-based GNN training in *PyG*, primarily by reducing data loading time.

7 CONCLUSIONS

In our work, we investigate the effectiveness of graph reordering to reduce GNN training time. We find that graph reordering is an effective optimization to decrease training time for both CPU and GPU-based training. Our experiments show that the effectiveness is influenced by GNN parameters and that the graph reordering time can be amortized. Further, we find that graph reordering metrics correlate with training speedup, and can be used for graph reordering selection. However, graph reordering metrics are not perfect predictors for speedup indicating that new graph reordering metrics may need to be developed.

REFERENCES

[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming Graph Partitioning: An Experimental Study. *Proc. VLDB Endow.* 11, 11 (July 2018), 14. https://doi.org/10.14778/3236187.3236208

[2] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 22–31. https://doi.org/10.1109/IPDPS.2016.110

[3] Vignesh Balaji and Brandon Lucia. 2018. When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 203–214. https://doi.org/10.1109/IISWC.2018.8573478

[4] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–512. https://doi.org/10.1126/science.286.5439.509 arXiv:https://www.science.org/doi/pdf/10.1126/science.286.5439.509

[5] Reet Barik, Marco Minutoli, Mahantesh Halappanavar, Nathan R. Tallent, and Ananth Kalyanaraman. 2020. Vertex Reordering for Real-World Graphs and Applications: An Empirical Evaluation. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 240–251. https://doi.org/10.1109/IISWC50251.2020.00031

[6] Maciej Besta, Simon Weber, Lukas Gianinazzi, Robert Gerstenberger, Andrey Ivanov, Yishai Oltchik, and Torsten Hoefler. 2019. Slim graph: practical lossy graph compression for approximate graph processing, storage, and analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 35, 25 pages. https://doi.org/10.1145/3295500.3356182

[7] K. Bhatia, K. Dahiya, H. Jain, P. Kar, A. Mittal, Y. Prabhu, and M. Varma. 2016. The extreme classification repository: Multi-label datasets and code. http://manikvarma.org/downloads/XC/XMLRepository.html

[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[9] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rytstxWAW

[10] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 257–266. https://doi.org/10.1145/3292500.3330925

[11] E. Cuthill and J. McKee. 1969. Reducing the Bandwidth of Sparse Symmetric Matrices. In *Proceedings of the 1969 24th National Conference (ACM '69)*. Association for Computing Machinery, New York, NY, USA, 157–172. https://doi.org/10.1145/800195.805928

[12] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[13] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 551–568. https://www.usenix.org/conference/osdi21/presentation/gandhi

[14] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. 2018. A Study of Partitioning Policies for Graph Analytics on Large-Scale Distributed Platforms. *Proc. VLDB Endow.* 12, 4 (Dec. 2018), 14. https://doi.org/10.14778/3297753.3297754

[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[16] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[17] Loc Hoang, Xuhao Chen, Hochan Lee, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2021. Efficient Distribution for Deep Learning on Large Graphs. In *Workshop on Graph Neural Networks and Systems*. 1–9.

[18] Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2021. CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics. *SIGOPS Oper. Syst. Rev.* 55, 1 (jun 2021), 47–60. https://doi.org/10.1145/3469379.3469385

[19] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[20] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 119–132. https://doi.org/10.1145/3437801.3441585

[21] Anand Padmanabha Iyer, Aurojit Panda, Shivaram Venkataraman, Mosharaf Chowdhury, Aditya Akella, Scott Shenker, and Ion Stoica. 2018. Bridging the GAP: towards approximate graph analytics. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Houston, Texas) *(GRADES-NDA '18)*. Association for Computing Machinery, New York, NY, USA, Article 10, 5 pages. https://doi.org/10.1145/3210259.3210269

[22] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[23] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E. Leiserson, and Jie Chen. 2022. Accelerating Training and Inference of Graph Neural Networks with Fast Sampling and Pipelining. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 172–189. https://proceedings.mlsys.org/paper_files/paper/2022/file/afacc5db3e0e85b446e6c7727cd7dca5-Paper.pdf

[24] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[25] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations* (Palais des Congrès Neptune, Toulon, France) *(ICLR '17)*. https://openreview.net/forum?id=SJU4ayYgl

[26] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web* (Rio de Janeiro, Brazil) *(WWW '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1343–1350. https://doi.org/10.1145/2487788.2488173

[27] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[28] Dongyue Li, Tao Yang, Lun Du, Zhezhi He, and Li Jiang. 2021. AdaptiveGCN: Efficient GCN Through Adaptively Sparsifying Graphs. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management* (Virtual Event, Queensland, Australia) *(CIKM '21)*. Association for Computing Machinery, New York, NY, USA, 3206–3210. https://doi.org/10.1145/3459637.3482049

[29] Yongsub Lim, U Kang, and Christos Faloutsos. 2014. SlashBurn: Graph Compression and Mining beyond Caveman Communities. *IEEE Transactions on Knowledge and Data Engineering* 26, 12 (Dec 2014), 3077–3089. https://doi.org/10.1109/TKDE.2014.2320716

[30] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 401–415. https://doi.org/10.1145/3419111.3421281

[31] Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. 2020. G3: when <u>g</u>raph neural networks meet parallel <u>g</u>raph processing systems on <u>g</u>PUs. *Proc. VLDB Endow.* 13, 12 (aug 2020), 2813–2816. https://doi.org/10.14778/3415478.3415482

[32] Renjie Liu, Yichuan Wang, Xiao Yan, Zhenkun Cai, Minjie Wang, Haitian Jiang, Bo Tang, and Jinyang Li. 2024. DiskGNN: Bridging I/O Efficiency and Model Accuracy for Out-of-Core GNN Training. *arXiv preprint arXiv:2405.05231* (2024).

[33] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2023. BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 103–118. https://www.usenix.org/conference/nsdi23/presentation/liu-tianfeng

[34] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, and Dongrui Fan. 2022. Sampling Methods for Efficient Training of Graph Convolutional Networks: A Survey. *IEEE/CAA Journal of Automatica Sinica* 9, 2 (2022), 205–234. https://doi.org/10.1109/JAS.2021.1004311

[35] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, Dongrui Fan, Shirui Pan, and Yuan Xie. 2022. Survey on Graph Neural Network Acceleration: An Algorithmic Perspective. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, Lud De Raedt (Ed.). International Joint Conferences on Artificial Intelligence Organization, 5521–5529. https://doi.org/10.24963/ijcai.2022/772 Survey Track.

[36] Ruben Mayer and Hans-Arno Jacobsen. 2021. Hybrid Edge Partitioner: Partitioning Large Power-Law Graphs under Memory Constraints. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 14. https://doi.org/10.1145/3448016.3457300

[37] Ruben Mayer, Kamil Orujzade, and Hans-Arno Jacobsen. 2022. Out-of-Core Edge Partitioning at Linear Run-Time. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2629–2642. https://doi.org/10.1109/ICDE53745.2022.00242

[38] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha. 2021. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 76, 14 pages. https://doi.org/10.1145/3458817.3480856

[39] Nikolai Merkel, Ruben Mayer, Tawkir Ahmed Fakir, and Hans-Arno Jacobsen. 2023. Partitioner Selection with EASE to Optimize Distributed Graph Processing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2400–2414. https://doi.org/10.1109/ICDE55515.2023.00185

[40] Nikolai Merkel, Daniel Stoll, Ruben Mayer, and Hans-Arno Jacobsen. 2025. An Experimental Comparison of Partitioning Strategies for Distributed Graph Neural Network Training. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025*. OpenProceedings.org, 171–184. https://doi.org/10.48786/EDBT.2025.14

[41] Mark EJ Newman. 2003. The structure and function of complex networks. *SIAM review* 45, 2 (2003), 167–256.

[42] Joel Nishimura and Johan Ugander. 2013. Restreaming Graph Partitioning: Simple Versatile Algorithms for Advanced Balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Chicago, Illinois, USA) *(KDD '13)*. Association for Computing Machinery, New York, NY, USA, 9. https://doi.org/10.1145/2487575.2487696

[43] Anil Pacaci and M. Tamer Özsu. 2019. Experimental Analysis of Streaming Algorithms for Graph Partitioning. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 18. https://doi.org/10.1145/3299869.3300076

[44] Aditya Pal, Chantat Eksombatchai, Yitong Zhou, Bo Zhao, Charles Rosenberg, and Jure Leskovec. 2020. PinnerSage: Multi-Modal User Embedding Framework for Recommendations at Pinterest. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 2311–2320. https://doi.org/10.1145/3394486.3403280

[45] Md Anwarul Kaium Patwary, Saurabh Garg, and Byeong Kang. 2019. Window-Based Streaming Graph Partitioning Algorithm. In *Proceedings of the Australasian Computer Science Week Multiconference* (Sydney, NSW, Australia) *(ACSW 2019)*. Association for Computing Machinery, New York, NY, USA, Article 51, 10 pages. https://doi.org/10.1145/3290688.3290711

[46] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proc. VLDB Endow.* 15, 9 (may 2022), 1937–1950. https://doi.org/10.14778/3538598.3538614

[47] Jordi Petit. 2004. Experiments on the Minimum Linear Arrangement Problem. *ACM J. Exp. Algorithmics* 8, Article 2.3 (dec 2004), 33 pages. https://doi.org/10.1145/996546.996554

[48] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. HDRF: Stream-Based Partitioning for Power-Law Graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management* (Melbourne, Australia) *(CIKM '15)*. Association for Computing Machinery, New York, NY, USA, 10. https://doi.org/10.1145/2806416.2806424

[49] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. 2020. DropEdge: Towards Deep Graph Convolutional Networks on Node Classification. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Hkx1qkrKPr

[50] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*. Springer, 593–607.

[51] Daniel A. Spielman and Nikhil Srivastava. 2008. Graph sparsification by effective resistances. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing* (Victoria, British Columbia, Canada) *(STOC '08)*. Association for Computing Machinery, New York, NY, USA, 563–568. https://doi.org/10.1145/1374376.1374456

[52] Rakshith S Srinivasa, Cao Xiao, Lucas Glass, Justin Romberg, and Jimeng Sun. 2020. Fast graph attention networks using effective resistance based graph sparsification. *arXiv preprint arXiv:2006.08796* (2020).

[53] Isabelle Stanton and Gabriel Kliot. 2012. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Beijing, China) *(KDD '12)*. Association for Computing Machinery, New York, NY, USA, 1222–1230. https://doi.org/10.1145/2339530.2339722

[54] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. 2017. Graph attention networks. *stat* 1050, 20 (2017), 10–48550.

[55] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. 2017. An Experimental Comparison of Partitioning Strategies in Distributed Graph Processing. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 12. https://doi.org/10.14778/3055540.3055543

[56] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[57] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world' networks. *nature* 393, 6684 (1998), 440–442.

[58] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1813–1828. https://doi.org/10.1145/2882903.2915220

[59] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 359–375. https://doi.org/10.1145/3447786.3456247

[60] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 417–434.

[61] Liang Yao, Chengsheng Mao, and Yuan Luo. 2019. Graph convolutional networks for text classification. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 7370–7377.

[62] Yang Ye and Shihao Ji. 2023. Sparse Graph Attention Networks. *IEEE Transactions on Knowledge and Data Engineering* 35, 1 (2023), 905–916. https://doi.org/10.1109/TKDE.2021.3072345

[63] Peiqi Yin, Xiao Yan, Jinjing Zhou, Qiang Fu, Zhenkun Cai, James Cheng, Bo Tang, and Minjie Wang. 2023. DGI: An Easy and Efficient Framework for GNN Model Evaluation. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA,) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 5439–5450. https://doi.org/10.1145/3580305.3599805

[64] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 974–983.

[65] Hao Yuan, Yajiong Liu, Yanfeng Zhang, Xin Ai, Qiange Wang, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. Comprehensive Evaluation of GNN Training Systems: A Data Management Perspective. *Proc. VLDB Endow.* 17, 6 (may 2024), 1241–1254. https://doi.org/10.14778/3648160.3648167

[66] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=BJe8pkHFwS

[67] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V. Chawla. 2019. Heterogeneous Graph Neural Network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 793–803. https://doi.org/10.1145/3292500.3330961

[68] Chenzi Zhang, Fan Wei, Qin Liu, Zhihao Gavin Tang, and Zhenguo Li. 2017. Graph Edge Partitioning via Neighborhood Heuristic. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) *(KDD '17)*. Association for Computing Machinery, New York, NY, USA, 10. https://doi.org/10.1145/3097983.3098033

[69] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. 293–302. https://doi.org/10.1109/BigData.2017.8257937

[70] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1228–1242.

[71] Cheng Zheng, Bo Zong, Wei Cheng, Dongjin Song, Jingchao Ni, Wenchao Yu, Haifeng Chen, and Wei Wang. 2020. Robust Graph Representation Learning via Neural Sparsification. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Hal Daumé III and Aarti Singh (Eds.), Vol. 119. PMLR, 11458–11468. https://proceedings.mlr.press/v119/zheng20d.html

[72] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 36–44.

[73] Da Zheng, Xiang Song, Qi Zhu, Jian Zhang, Theodore Vasiloudis, Runjie Ma, Houyu Zhang, Zichen Wang, Soji Adeshina, Israt Nisa, et al. 2024. GraphStorm: all-in-one graph machine learning framework for industry applications. *arXiv*

preprint arXiv:2406.06022 (2024).

[74] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2094–2105. https://doi.org/10.

14778/3352063.3352127

[75] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. *Layer-dependent importance sampling for training deep and large graph convolutional networks.* Curran Associates Inc., Red Hook, NY, USA.