Algorithms and Systems for Scalable Machine Learning over Graphs

by

Roger Waleffe

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2024

Date of final oral examination: 12/04/2024

The dissertation is approved by the following members of the Final Oral Committee:
Theodoros Rekatsinas, Research Scientist, Apple Inc.
Shivaram Venkataraman, Assistant Professor, Computer Sciences
Steve Wright, Professor, Computer Sciences
Dimitris Papailiopoulos, Associate Professor, Electrical and Computer Engineering

ACKNOWLEDGMENTS

There have been many people who have helped me and collaborated with me throughout my PhD and my life; I am very grateful for them.

I would like to especially thank my advisor Theodoros Rekatsinas, who has supported and guided me for the entirety of my PhD. Working with Theo, I have grown immensely as a researcher, presenter, and writer. I will always be grateful for these teachings and experience. Though we were often in different timezones, Theo always met with me early in the morning or late in the evening and went above and beyond with his dedication to my PhD.

I would also like to sincerely thank the other members of my dissertation committee: Shivaram Venkataraman, Stephen J Wright, and Dimitris Papailiopoulos, whose questions and expertise have helped to enhance my research, this dissertation, and my PhD. In particular, Shivaram has been an invaluable collaborator during the research and development of this thesis and I have truly enjoyed all of the time working with him. I would additionally like to thank Shivaram and Steve for also providing advisory and procedural support during my PhD at UW-Madison.

I owe a great deal of thanks to my coauthors, professors, and fellow graduate students at UW-Madison and beyond who have been a part of my PhD. Jason Mohoney and Patrick Okanovic were particularly key collaborators who helped substantially with one or more of my PhD papers. I would also like to thank Emmanouil-Vasileios Vlatakis-Gkaragkounis, Xiangyao Yu, Wenqi Jiang, Vasilis Mageirakos, Nezihe Merve Gürel, Devesh Sarda, Zifan Liu, and all of the staff and members of the UW-Madison CS department.

To those at NVIDIA, including Mohammad Shoeybi, Bryan Catanzaro, Brandon Norick, Duncan Riach, Wonmin Byeon, Deepak Narayanan, and many others, I am very grateful for the opportunities you have provided and I am looking forward to continuing to work together in the future.

This thesis is the culmination of an academic journey that started well before my PhD itself. I would like to thank all of my teachers in the Middleton-Cross Plains Area School District who prepared me for college and beyond. I also owe many thanks to my professors during my time as an undergraduate at UW-Madison; I hold my undergraduate education at UW-Madison in the highest regard. The Math, Physics, and Computer Sciences departments are truly exceptional places to learn and grow.

A special thank you goes to Cary B Forest, who provided me with the opportunity to start my research career at the Wisconsin Plasma Physics Laboratory (WiPPL) during my undergraduate studies. Cary's encouragement and excitement for his work were inspiring, and the joy I experienced working with Cary led me to continue with a career in research. Many others contributed to this enjoyable learning experience, including Ethan E Peterson, Douglass A Endrizzi, Jason Milhone, Joseph Olson, Michael Clark, John Wallace, Jan Egedal, and Vladimir Mirnov.

Finally, I would like to thank my family and those close to me who have supported me during my PhD. The path to my PhD has been easier with you as a part of this journey.

To my parents, I will be forever grateful for the life and opportunities you have provided for me and my brother. This thesis is no doubt a reflection and result of your unconditional love and support over the last 27 years. To my brother, thank you for all of your love, support, and encouragement.

CONTENTS

ABSTRACT

Many forms of data are best represented as graphs with entities (nodes) and relationships (edges) between them. By combining this structure with neural network computation, Graph Neural Networks (GNNs) have emerged as the state-of-the-art approach for machine learning on graph data and have enabled advancements in navigation, structural biology, and weather forecasting. Training GNNs over large graphs, however, is highlighted as a major challenge in the literature: Real world graphs contain billions of nodes and edges, each of which can be associated with high-dimensional (possibly learned) feature vectors that form the input to GNNs. Moreover, the node representations at internal GNN layers depend on feature vectors in the nodes' multi-hop neighborhood. These workload characteristics necessitate storing feature vectors off device (e.g., in CPU memory) and leveraging mini-batch training coupled with multi-hop neighborhood sampling algorithms for learning GNNs over large-scale graphs; yet, we find that these requirements can lead to GPU underutilization and sublinear scaling in existing systems, leading to increased monetary costs and runtime over massive graphs.

Motivated by the above, this dissertation aims to develop cost-effective, scalable GNN training over large graphs. We move towards this goal by addressing a sequence of technical challenges, starting with mixed CPU-GPU training on a single machine and advancing to disk-based, distributed training across multiple machines and GPUs.

First, we focus on maximizing GPU utilization during mixed CPU-GPU mini-batch training on a single machine with a single GPU and with the full graph in CPU memory. We present a pipelined architecture for asynchronous training in this setting to overlap data preparation and movement with GNN computation. Asynchronous machine learning, however, introduces the possibility of concurrent mini batches accessing stale parameters or overwriting each other's work, thus slowing convergence or limiting accuracy

compared to synchronous training. To address this issue, we introduce a new policy for mixed CPU-GPU training that ensures asynchronous parallel preparation and transfer of mini batches is equivalent to a serial one by one execution. Finally, we introduce a new data structure and algorithm for neighborhood sampling that minimizes redundant computation and data access when constructing multi-hop neighborhoods. We implement the above techniques in MariusGNN, a new system for GNN training, and show that in-CPU sampling in MariusGNN can be up to 14× faster compared to state-of-the-art systems. Moreover, we show that end-to-end training in MariusGNN can be up to 4× faster than these systems, even as they use four GPUs and MariusGNN uses only one.

Next, we focus on GNN training over large graphs which exceed the CPU memory capacity of a single machine. Efficient, high-accuracy training in this setting relies on min-edge-cut graph partitioning algorithms, which maximize the number of neighbors each node has within the same partition, thereby maximizing the number of neighbors readily available for training (e.g., accessing neighbors in different partitions may require cross-machine communication in distributed settings). Yet, min-edge-cut partitioning over large graphs remains a challenge: Existing offline methods (e.g., METIS) are effective, but they require orders of magnitude more memory and runtime than GNN training itself, while computationally efficient algorithms (e.g., streaming greedy approaches) suffer from increased edge cuts. Thus, in this dissertation we introduce GREM, a novel min-edge-cut partitioning algorithm that can efficiently scale to large graphs. GREM builds on streaming greedy approaches but continuously refines prior vertex assignments during streaming, rather than freezing them after an initial greedy selection. Our theoretical analysis and experimental results show that this refinement is critical to minimizing edge cuts and enables GREM to reach partition quality comparable to METIS but with 8-65× less memory and 8-46× faster.

Given a partitioned graph, we then focus on disk-based GNN training.

In this case, graph partitions are stored on disk, with subsets loaded into memory as needed for mixed CPU-GPU training on the induced subgraph. We introduce a series of partition replacement policies that ensure 1) the entire graph appears in memory for training each epoch with a near-minimal number of partition swaps (and thus IO) and 2) that models learned with disk-based training exhibit accuracy similar to those trained with the full graph in memory. We combine the above policies in MariusGNN with a partition buffer that supports prefetching and writing partitions to disk asynchronously, thus introducing the first system that utilizes the entire memory hierarchy—including disk—for GNN training. We evaluate disk-based training in MariusGNN against state-of-the-art systems for learning GNN models and find that it achieves the same level of accuracy up to 8× faster than these systems. Moreover, disk-based training enables MariusGNN to train over large graphs that do not fit in CPU memory using just a single, cheap machine rather than an expensive, large-memory machine or multi-machine deployment, leading to monetary cost reductions of up to 64×.

Finally, for the case when parallelization is desired to accelerate training, we study distributed training of GNNs on billion-scale graphs that are partitioned across machines. We introduce Armada, a new end-to-end system for distributed multi-GPU and multi-machine training. Armada leverages a disaggregated architecture to improve efficiency; we find that on common cloud machines, GNN neighborhood sampling and feature loading bottleneck training in existing multi-GPU deployments. Disaggregation allows Armada to independently allocate resources for these operations and ensure that expensive GPUs remain saturated with computation. We evaluate Armada against state-of-the-art systems for distributed GNN training and find that, in the same setting for which existing systems achieve only 2.3× and 1.7× speedup with eight instead of one GPU, Armada's disaggregated architecture leads to a 7.5× speedup. This linear scaling leads to runtime improvements of up to 4.8× and cost reductions of up to 3.4× compared to baselines.

1 INTRODUCTION

1.1 Motivation

Graphs are ubiquitous data structures, valued for their ability to represent entities (*nodes*) and the relationships (*edges*) between them. This versatility makes them widely applicable across domains, where they are used to model various types of data, including social networks, transportation systems, biological networks, and knowledge graphs (Zafarani et al., 2014; Fairchild et al., 1988; Brohee and Van Helden, 2006; Derrow-Pinion et al., 2021). Motivated by their prevalence and the recent success of machine learning (ML) across many disciplines, including computer vision (He et al., 2016; Huang et al., 2017; Dosovitskiy et al., 2020) and natural language processing (Brown et al., 2020; Radford et al., 2019; OpenAI, 2023), there is growing interest in applying ML over graphs to enable richer analysis and prediction on this form of data (Chami et al., 2021; Kipf and Welling, 2016).

The desire to apply machine learning to graphs has led to the development of specialized models known as *Graph Neural Networks (GNNs)*, which have emerged as the defacto approach for ML over graph-structured inputs (Chami et al., 2021). Behind this success is the ability of GNNs to combine graph connectivity directly with neural network computation (Kipf and Welling, 2016; Hamilton et al., 2017; Veličković et al., 2018). This property allows GNNs to achieve state-of-the-art accuracy on diverse tasks. In fact, GNN-based models are currently used to predict the travel time for route options in navigation apps (Derrow-Pinion et al., 2021), to accurately predict protein structures (Jumper et al., 2021), and to create the most accurate weather forecasts (GraphCast (Lam et al., 2022)). We describe the latter as a motivating application in more detail in Chapter 2.

While impressive, state-of-the-art results require training GNNs over massive amounts of graph data. For example, GraphCast was trained on

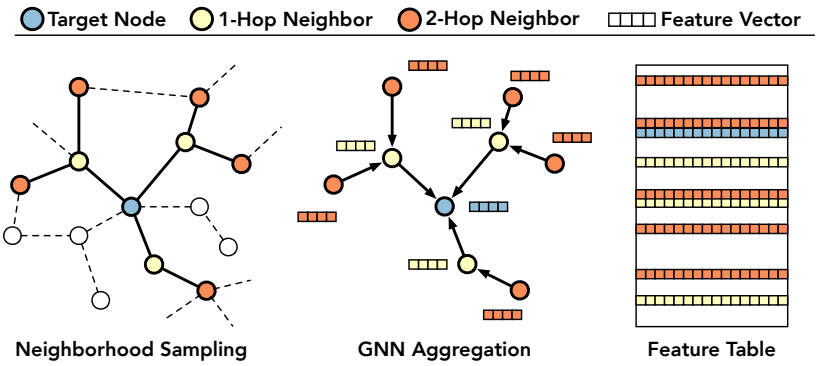


Figure 1.1: GNNs compute representations for graph nodes that can be used for downstream tasks by learning to aggregate local information with information from nodes' multi-hop neighborhoods. The local information for each node is encoded in a feature vector and the feature vectors for all nodes in the graph are stored together in a lookup table.

53TB over four weeks using 32 Google Cloud TPU v4 nodes (totaling an estimated cost of $70,000 as of 10/2024), limiting the development of such a model to those with sufficient resources. As such, there is a compelling need for cost-effective, scalable systems for GNN training to democratize further expansion of GNN applications.

As highlighted in the literature, however, cost-effective, scalable GNN training over large-scale graphs is challenging (Chami et al., 2021; Zheng et al., 2022; Thorpe et al., 2021; Gandhi and Iyer, 2021). These challenges arise from the unique properties of the GNN workload itself: First, graphs used in production settings contain billions of nodes and edges, each of which can be associated with high-dimensional feature vectors that describe the node (or edge) and form the inputs to GNNs. Thus, the storage overhead for these vectors can require hundreds of GBs to TBs of memory (Ilyas et al., 2022; Maass et al., 2017) (Table 1.1), easily exceeding the capacity of the GPUs needed to accelerate GNN computation (e.g., matrix-vector multiplications) over these features (e.g., an NVIDIA V100 GPU has 16GB of memory). Moreover, for certain GNN models, the feature vectors are

Table 1.1: Graph sizes and storage overheads for common, representative graphs that GNNs are applied to. Large graphs exceed the memory capacity of GPU accelerators, but can often fit in main memory or the disk of a single machine (e.g., AWS P3 GPU instances range from 61-488GB of CPU memory and contain up to 16TB of disk storage).

Graph	Nodes	Edges	Feat. Dim	Memory (GB)		
				Edges	Feat.	Tot.
Papers100M (Hu et al., 2020)	111M	1.62B	128	13	57	70
Mag240M (Hu et al., 2021)	122M	1.30B	768	10	375	385
Freebase86M (Google, 2018)	86M	338M	100	4	69	73
WikiKG90Mv2 (Hu et al., 2021)	91M	601M	100	7	73	80
Hyperlink-2012 (Meusel et al., 2014)	3.5B	128B	50	2k	1.4k	3.4k
Facebook15 (Ching et al., 2015)	1.4B	1T	100	8k	560	8.5k

actually learned during training, rather than predefined, further complicating the challenge of working with this data. Second, the node representations computed by multi-layer GNNs and used for downstream tasks depend on the feature vectors of the nodes' multi-hop neighborhood. This dependency causes the GNN computation to scale exponentially with the number of GNN layers and results in irregular accesses patterns to storage (Figure 1.1).

Given these two workload characteristics—graph storage overheads and GNN neighborhood dependencies—state-of-the-art systems, such as Deep Graph Library (DGL) (Wang et al., 2019; Zheng et al., 2020a, 2022), PyTorch Geometric (PyG) (Fey and Lenssen, 2019), and Salient++ (Kaler et al., 2022, 2023), address the challenges of large-scale GNN training by leveraging multi-hop neighbor sampling algorithms (Hamilton et al., 2017) coupled with mixed CPU-GPU mini-batch training over one or multiple machines with attached GPU(s). In this setting, CPU memory is used for graph storage. Training then proceeds as follows: *Mini batches*, consisting of a random sample of graph nodes (or edges), a sample of each nodes' multi-hop neighborhood, and the corresponding feature vectors for all nodes and neighbors, are prepared on the CPU before being transferred to the GPU(s)

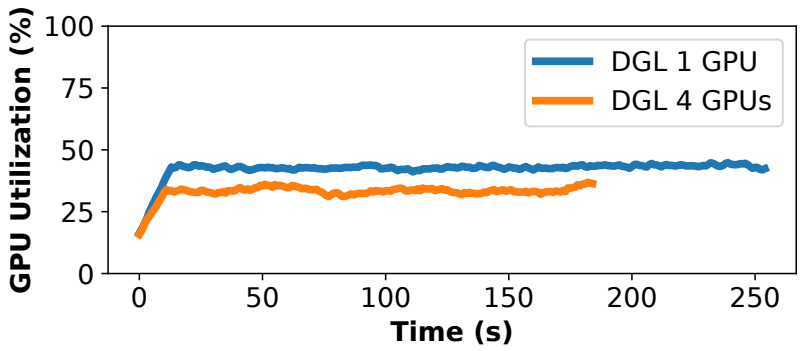


Figure 1.2: Average per-GPU utilization of DGL during one epoch of GraphSage (Hamilton et al., 2017) GNN training on OGBN-Papers100M (Hu et al., 2020). Experimental details are provided in Section 3.4.

for GNN computation; once on the GPU, one training iteration consists of a forward and backward pass through the GNN to calculate gradients and update the learnable GNN parameters (weights). If applicable, any updates to learned feature vectors resulting from the training iteration are also calculated and then transferred and written back to CPU memory. One round of training (called an epoch) completes when all nodes (or edges) have been used to create mini batches.

Despite these techniques, however, two primary inefficiencies remain for large-scale GNN training. First, in many cases we find that existing systems underutilize their available hardware, particularly as compute resources (i.e., GPUs) are scaled. For example, the average GPU utilization of DGL on a common GNN benchmark is about 45% and drops to just 35% when using 4 GPUs (Figure 1.2). In fact, we find that two state-of-the-art systems (Kaler et al., 2023; Zheng et al., 2020a) yield only 2.3× and 1.7× speedup respectively when using eight instead of one GPU (instead of 8×). Second, existing systems require graph data and feature vectors to be stored on the CPU of GPU machines used for training. This requirement necessitates that expensive compute resources are allocated in proportion to the graph size. As a result, current systems incur increased monetary

costs and runtime over massive graphs due to resource-hungry deployments coupled with underutilization of the allocated hardware.

1.2 Dissertation Goal

Motivated by the above, this dissertation aims to enable cost-effective, scalable GNN training over large graphs by optimizing resource utilization and leveraging the entire memory hierarchy—including disk—for training. To achieve this goal, we tackle the following sequence of technical problems that build on each other and progress from mixed CPU-GPU training on a single machine to disk-based, distributed training over multiple machines. A more detailed discussion of these challenges is presented in Chapter 2.

Problem 1: Efficient In-Memory Mixed CPU-GPU Training First, we aim to maximize utilization during mixed CPU-GPU mini-batch training on a single machine with a single GPU and when graphs are assumed to fit in CPU memory. GPU underutilization in this setting can be attributed to two primary challenges: 1) data movement overheads resulting from the need to transfer feature vectors between CPU and GPU memory and 2) CPU bottlenecks arising from multi-hop neighborhood sampling.

More specifically, mixed CPU-GPU mini-batch training, which typically proceeds *synchronously*—one mini batch is prepared, transferred, and then used for computation at a time—leads to low throughput (and utilization, e.g., Figure 1.2) as the the GPU sits idle during mini batch preparation and transfer. While synchronous training is the default in many frameworks (Zheng et al., 2020b), *asynchronous* training has gained popularity for increasing GPU utilization. In this case, multiple mini batches are prepared and transferred in parallel to ensure that the GPU is always busy with GNN computation. While asynchronous training can improve utilization, it can result in suboptimal convergence—requiring more mini batches to be processed to reach the accuracy of synchronous training, or failing to reach that accuracy altogether. The problem occurs when multiple concurrent

mini batches contain the same nodes, and thus feature vectors, and these feature vectors are being learned during training; since mini batches are prepared and transferred in parallel, any feature vectors present in concurrent mini batches will be missing updates that are generated by other concurrent mini batches containing the same features (we say these feature vectors are *stale*). *Therefore, new algorithms for mixed CPU-GPU training that leverage asynchrony while maintaining synchronous convergence are needed to maximize GPU utilization without accuracy loss.*

Even with asynchronous training, however, it can be difficult to maximize GPU utilization during mixed CPU-GPU training due to the overheads associated with multi-hop neighborhood sampling. Despite recent efforts to address this issue (Chen et al., 2018; Zou et al., 2019; Ramezani et al., 2020; Zeng et al., 2020; Chiang et al., 2019), we find that existing sampling algorithms in state-of-the-art systems bottleneck training and lead GPUs to sit idle. For example, when training a three-layer GNN on a graph with 100M nodes, we find that PyTorch Geometric's CPU-based neighborhood sampler takes 1200ms to sample the requisite three-hop neighborhoods for each mini batch; yet, the GPU-based operations for training take only 170ms. *Therefore, new techniques and implementations for multi-hop neighborhood sampling are needed to minimize the overhead of this operation and unblock mixed CPU-GPU GNN mini-batch training.*

Problem 2: Scalable Min-Edge-Cut Graph Partitioning Next, we aim to support training over large-scale graphs for which the graph structure and feature vectors may exceed the CPU memory capacity of a single machine. Training in this case requires that the graph nodes and features are split into *partitions*. These partitions can then used to enable training in one of two ways: 1) they can be stored on disk, with subsets loaded into CPU memory for training (called *disk-based* or *out-of-core* training) or 2) they can be split and loaded into memory across multiple machines to facilitate distributed training.

In either case, the partitioning has a direct impact on the subsequent training process, as GNNs require access to the multi-hop neighborhoods of graph nodes. In a disk-based setup, partitioning limits the number of neighbors available for training, as neighborhood sampling can only be performed over the nodes present in CPU memory (multi-hop sampling results in random access patterns to memory which are too expensive to run over block storage). Consequently, disk-based training can result in reduced model accuracy if the in-memory partitions lack neighbors for certain nodes. By contrast, in the distributed setting, neighborhood sampling can be performed across the whole graph, but doing so requires machines to communicating with each other as needed (Shao et al., 2024). While beneficial for model accuracy, cross-machine neighborhood sampling can lead to a communication bottleneck that limits the scalability and throughput of distributed GNN training.

The above issues can by mitigated by partitioning the graph using *min-edge-cut partitioning* algorithms that minimize the number of edges with endpoints in different partitions (called *cut edges*). Such a partitioning ensures that the subset of graph data in each partition is dense (i.e., for a given node most of its neighbors are also in the same partition). For disk-based training, this implies most of a node's neighbors will be available in memory when the node itself is in memory. For distributed training, min-edge-cut partition reduces cross-machine communication and has been shown to lead to an order of magnitude faster training compared to random partitioning (Merkel et al., 2023; Zheng et al., 2022). Thus, min-edge-cut partitioning is widely used in GNN systems.

Yet, min-edge-cut partitioning over large graphs remains a challenge: State-of-the-art offline methods (e.g., METIS (Karypis and Kumar, 1997)) are effective due to their ability to iteratively refine partitions across the whole graph, but they require orders of magnitude more memory and runtime than GNN training itself, while computationally efficient algorithms (e.g.,

streaming greedy approaches (Abbas et al., 2018)) suffer from increased edge cuts due to fixed greedy partition assignments. *Thus, there is a critical need for new min-edge-cut partitioning algorithms that can efficiently scale to large graphs on common hardware.*

Problem 3: Efficient, High-Accuracy Out-of-Core Training Given a partitioned graph, we aim to support training using the entire memory hierarchy—including the cheap and high capacity disk—to enable resource-efficient, economical GNN training deployments over large graphs (that do not fit in CPU memory on a single machine), rather than paying for additional resources to scale training.

As described above, disk-based training involves storing the partitioned graph on disk and then loading subsets of partitions into memory for mixed CPU-GPU training on the induced in-memory subgraph. In this setting, traversing the whole graph (i.e., training on the whole graph) is achieved by swapping partitions between the disk and CPU, according to a *partition replacement policy*, until all graph nodes (or edges) have appeared in memory. Although partitions can be accessed sequentially, disk-to-CPU partition swapping can lead to IO-bound training and result in GPU underutilization. Buffer management techniques combined with specialized data orderings (i.e., traversing the nodes or edges in the graph in a specific order) can be used to partially address this issue by overlapping and reducing IO to disk; yet, we find that existing locality-aware data orderings (e.g., space-filling curves (McSherry et al., 2015)) still result in IO-bound training due to the frequent amount of partition swaps they require. Moreover, we find that partition replacement policies which focus only on minimizing IO harm accuracy when training GNNs (due to the data orderings they introduce for training which conflict with the random orderings preferred for stochastic gradient descent). For example, we find that the accuracy of disk-based GNN models can drop by up to 16% on common benchmarks when compared to models trained with the entire graph in CPU memory. *Therefore, new*

partition replacement policies for disk-based training are needed that mitigate the overheads of partition swapping while simultaneously allowing for disk-based training to achieve high accuracy.

Problem 4: Scalable Distributed Training Finally, in the case where the storage overhead of a graph exceeds the CPU memory capacity of a single machine (and distributed training is preferred over disk-based training), or when parallelization is desired to accelerate training, we aim to support cost-effective, scalable distributed GNN training over large graphs using common cloud offerings.

To do so, we focus on optimizing the utilization of expensive GPU resources during training as compute resources are scaled. As the number of GPUs used for training increases, however, so too does the overhead of mini batch preparation on the CPU (neighborhood sampling and feature loading)– distributed data parallel training with weak scaling, commonly used for GNN training, requires preparing one mini batch per GPU for each training iteration. For example, even with optimized sampling implementations and zero cross-machine communication, we find that on common cloud machines mini batch preparation on the CPU can be up to an order of magnitude slower than mini batch computation on the GPU when using eight GPUs for training. Thus, existing systems which rely only on the fixed set of CPU resources attached to the GPU machines used for training to prepare batches are unable to parallelize mini batch preparation sufficiently to saturate multiple accelerators and suffer from sublinear speedups as compute resources are scaled (as highlighted above). Sublinear speedups lead to higher-than-necessary total training cost and runtime over massive graphs, as expensive compute resources sit idle. *The above observations motivate a new architecture for large-scale GNN training that supports scaling each part of the workload independently.*

1.3 Contributions

To address the problems stated in Section 1.2 above, this dissertation makes the following technical contributions.

Asynchronous Training with Synchronous Convergence We present a *pipelined architecture for asynchronous mixed CPU-GPU training* that overlaps data access, transfer, and computation to achieve high GPU utilization. Specifically, we assign a dedicated set of workers to each stage in the training process. Workers then read and write the input and output of their respective stage to a set of queues. This architecture allows each stage to run in parallel, and allows more workers to be assigned to slower stages in the pipeline to maximize throughput. Using this architecture, we can learn feature vectors for the 42M node, 1.5B edge Twitter graph (Kwak et al., 2010) an order of magnitude faster than state-of-the-art systems which use synchronous training (Zheng et al., 2020b): Using a single GPU, pipelined training requires 3.5 hours whereas synchronous training requires 35 hours on this graph. Our pipelined architecture is described in detail in Mohoney et al. (2021) and in Section 3.1.

We combine our pipelined architecture with a new technique for asynchronous training, which we term *Optimistic Asynchrony Control (OAC)*, to ensure that parallel processing of batches in mixed CPU-GPU settings results in the same accuracy as synchronous, one-by-one execution. Motivated by optimistic methods for concurrency control in database systems (termed OCC (Kung and Robinson, 1981; Tu et al., 2013)), OAC allows mini batches to be prepared and transferred in parallel, but ensures that batches which access the same parameters are updated before computation to have the correct feature vectors. Specifically, we highlight that even during asynchronous training, the order in which batches pass through the GPU computation step defines a one-by-one order over batches. OAC ensures that asynchronous training results in the same updates to model parameters as synchronous training would have produced according to that

order of batches. To do so, we leverage timestamps for each feature vector to track the most recent versions in the presence of multiple options and utilize an on-GPU feature cache to track the updates from batches prepared and transferred in parallel. This cache allows us to validate that a mini batch has the correct parameter values just before it enters the GNN computation step. We show that OAC results in identical convergence to synchronous training and identical throughput to asynchronous training; this allows OAC to achieve the best of both worlds, resulting in the fastest time-to-accuracy (how long it takes to reach a given accuracy) for GNN training. OAC is described in detail in Waleffe and Mohoney (2024) and in Section 3.2.

Multi-Hop Neighborhood Sampling without Redundancy We introduce *a new data structure to minimize the overhead of multi-hop neighborhood sampling* by minimizing redundant computation and data access. We term the new data structure *DENSE*, as it uses a Delta Encoding of Neighborhood SamplEs for vertices in multi-hop neighborhoods. More specifically, we identify that current approaches for sampling multi-hop neighborhoods resample one-hop neighbors for graph nodes multiple times when constructing a single multi-hop neighborhood—due to the graph structure, the same nodes can appear repeatedly across different branches and depths of the neighborhood. This resampling leads to redundant computation and data access and can limit pipeline throughput when training multi-layer GNNs. DENSE allows us to cache and reuse previously-sampled one-hop neighbors to construct multi-hop neighborhoods and enables in-CPU sampling that is up to 14× faster than state-of-the-art systems. DENSE also enables GNN forward pass computations up to 8× faster than competing systems by utilizing optimized dense GPU kernels rather than custom kernels for sparse matrix representations. DENSE is described in detail in Waleffe et al. (2023) and in Section 3.3.

Efficient In-Memory Mixed CPU-GPU Training We combine the above contributions (asynchronous pipelined training, OAC, and DENSE)

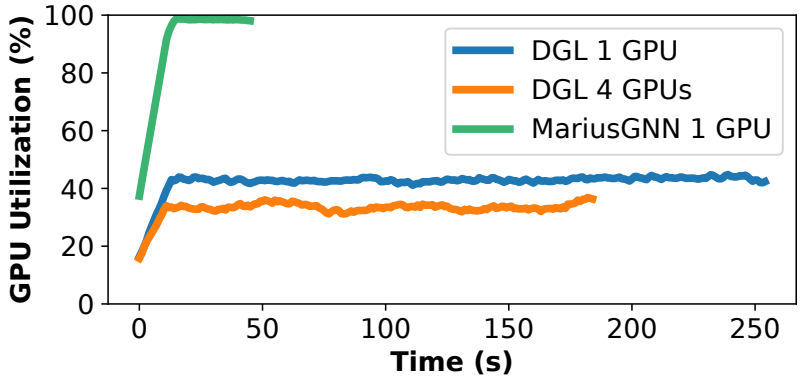


Figure 1.3: Average per-GPU utilization of our MariusGNN system compared to the state-of-the-art system DGL during one epoch of GraphSage GNN training on OGBN-Papers100M. Details are provided in Section 3.4.

in the MariusGNN framework (Mohoney et al., 2021; Waleffe and Mohoney, 2024; Waleffe et al., 2023) to maximize GPU utilization and minimize the cost and runtime of mixed CPU-GPU GNN training. In the same setting for which the state-of-the-art system DGL achieves only 45% GPU utilization, MariusGNN achieves 100% utilization (Figure 1.3) and can train to the same model accuracy 4× faster, even when DGL uses multiple GPUs. In fact, we find that MariusGNN in-memory training can be up to 7× faster than the best existing systems. Additional details are discussed in Section 3.4.

Scalable Min-Edge-Cut Graph Partitioning We present a *novel memory-efficient min-edge-cut partitioning algorithm called GREM* (Greedy plus Refinement for Edge-cut Minimization) that aims to address the bottleneck of partitioning in existing out-of-core or distributed GNN pipelines. GREM can efficiently scale to massive graphs on common hardware by processing streaming chunks of graph edges and returns partitions with edge cuts comparable to state-of-the-art offline methods (e.g., METIS). For example, in the same setting in which METIS requires 8000s and 630GB, GREM can partition the graph with similar edge cuts in 175s using 9.3GB.

GREM's partitioning algorithm builds on existing streaming greedy

approaches (Abbas et al., 2018). Specifically, GREM iterates over the graph edges in chunks and greedily assigns the vertices in each chunk to partitions. The key idea behind GREM, however, is that it allows prior vertex assignments to be modified throughout the process, rather than freezing them after an initial greedy selection (as in existing algorithms). This approach, inspired by offline algorithms, refines the partitioning by leveraging lightweight statistics accumulated during streaming (these statistics provide estimates of the number of neighbors per node in each partition).

We analyze theoretically GREM's expected number of edge cuts versus chunk size, providing insight into its expected behavior. This analysis, confirmed by experiments, shows that refinement is critical for minimizing edge cuts when using small chunk sizes (e.g., $\leq$10% of the edges) and thus for minimizing GREM's computational requirements (which are proportional to chunk size): We show that GREM with a chunk size of 10% and METIS cut a similar number of edges, but GREM does so with 8$\times$ less memory and runtime. GREM even achieves comparable results with a chunk size of 1%, leading to further reductions and enabling GREM to partition the largest public graphs (e.g., Hyperlink-2012 (Meusel et al., 2014), which contains 3.5B nodes and 128B edges) with only 500GB of memory. GREM is described in detail in Waleffe et al. (2024) and in Chapter 4.

Min-IO, High-Accuracy Disk-Based Training We *develop partition replacement polices for disk-based GNN training* that allow us to maintain high GPU utilization and accuracy when scaling to large graphs that do not fit in CPU memory (Chapter 5). Specifically, we introduce the *Buffer-aware Edge Traversal Algorithm (BETA)*, a partition replacement policy that ensures all graph edges appear in memory for training each epoch with near minimal partition swaps (and thus IO), outperforming locality-based algorithms such as Hilbert orderings (Hilbert, 1891). We combine the BETA ordering with an in-memory partition buffer that supports prefetching and writing partitions to disk asynchronously to hide the remaining IO overheads.

BETA is described in detail in Mohoney et al. (2021) and in Section 5.2.

We study the effect BETA has on GNN model accuracy and find that during training it leads to successive mini batches with correlated training examples (i.e., mini batches processed close together during training contain many of the same nodes and edges), a property that conflicts with the independently distributed assumption of ML training data, leading to lower GNN accuracy compared to training with the full graph in memory. To close this accuracy gap, while still minimizing disk IO, we introduce a partition replacement policy termed *COMET (COrrelation Minimizing Edge Traversal)*. COMET builds on BETA, but separates the granularity of data storage and access from data transfer by utilizing two levels of partitioning (physical and logical partitions). COMET also decouples mini batch generation from partition replacement, allowing for the inclusion of graph nodes (or edges) in mini batches to be deferred until later in an epoch rather than always including them the first time they appear in memory. This technique helps to further shuffle the order in which graph nodes (or edges) are used for training. Additionally, to maximize throughput and accuracy, we provide automated rules for setting COMET's hyperparameters (e.g., the number of physical and logical partitions) (Section 5.4). Using COMET, we are able to reduce the gap between the accuracy of disk-based training and that of training with the full graph in memory by up to 80% compared to BETA without any increase in runtime. COMET is described in detail in Waleffe et al. (2023) and in Section 5.3.

We couple the above disk-based partition replacement policies with the efficient in-memory training included in MariusGNN (discussed above) to introduce the first system for GNN training that utilizes the full memory hierarchy (disk, CPU, and GPU). Our experiments show that MariusGNN's disk-based, single-GPU training can be 8× faster than eight-GPU deployments of existing systems. This improvement yields monetary cost reductions of an order of magnitude. We find that for graphs where state-of-the-art

systems can take six days and $1720 dollars to train, MariusGNN needs only eight hours and $36 dollars for training, a 48× reduction in monetary cost. Moreover, we show that single-machine, disk-based training can be sufficient for large-scale graphs: *We use MariusGNN to train a GNN over the entire hyperlink graph from the Common Crawl 2012 web corpus, a graph with 3.5B nodes (web pages) and 128B edges (hyperlinks between pages) (Table 1.1).* MariusGNN can learn feature vectors for all *3.5B nodes using only a single machine with one GPU, 60GB of RAM, and a large SSD, leading to a cost of just $564/epoch.* Experimental details and results for disk-based training are provided in Section 5.5.

Cost-Effective, Scalable Distributed Training Finally, building on all of the above contributions, we introduce Armada, *a new distributed architecture for large-scale GNN training, that disaggregates graph storage, the CPU resources used for neighborhood sampling, and the GPU resources used for model computation*, in order to achieve memory-efficient, cost-effective, and scalable GNN training on common hardware. Concretely, Armada consists of: 1) A partitioning layer that implements GREM. 2) A storage layer to store the partitioned graph, implemented over cheap disk-based storage. 3) A distributed mini batch preparation layer consisting of a set of workers running on cheap CPU-only machines; workers read graph partitions from storage and prepare batches (i.e., perform neighborhood sampling) for training. 4) A distributed model computation layer that utilizes a set of GPU machines to perform training over the prepared batches.

We chose a disaggregated architecture to optimize resource utilization. By independently scaling the batch preparation layer, we can ensure that GPUs in the computation layer remain saturated with mini batches during training. As a result, in the same setting for which existing systems achieve only 2.3× and 1.7× speedup with eight instead of one GPU, Armada achieves a 7.5× speedup. Additionally, for massive graphs, Armada can leverage the storage layer for primary graph storage, rather than the CPU memory

of the batch preparation layer (or compute layer, as in existing systems), providing the option to train with fewer machines, leading to lower cost, memory-efficient training deployments (Armada with only one machine is similar to MariusGNN).

Despite the flexibility of disaggregation, challenges arise due to the communication overhead between various components. Thus, we carefully design Armada with a focus on minimizing communication between and within layers. In particular, Armada includes two optimizations to reduce the data sent between mini batch preparation and compute workers: 1) mini batch preparation workers group mini batches destined for different GPUs on the same compute worker and transfer them together, rather than independently, in order to enable greater compression (mini batch grouping), and 2) compute workers in Armada maintain a cache of frequently accessed data in their local CPU memory (feature caching). Together, these optimizations enable Armada to scale each layer in the architecture independently without communication bottlenecks.

We evaluate Armada's disaggregated architecture for GNN training and compare against existing state-of-the-art systems. Using popular GNN architectures, we show that while existing systems scale sublinearly, Armada does not, leading to runtime improvements of up to 4.8× and monetary cost reductions up to 3.4× compared to the latest distributed systems. Armada is discussed in detail in Waleffe et al. (2024) and Chapter 6

1.4 Organization

The rest of this dissertation is organized as follows. In Chapter 2 we provide background on GNNs and GNN training and expand on the challenges for large-scale training discussed in Section 1.2. In Chapter 3, we focus on maximizing GPU utilization during in-memory training on a single machine and discuss pipelined training, OAC, and the DENSE data structure for neighborhood sampling. In Chapter 4, we present GREM, a novel algorithm

for efficient min-edge-cut partition over massive graphs. In Chapter 5, we focus on partition replacement policies for disk-based training to scale to graphs which do not fit in CPU memory while still using only a single machine. In Chapter 6, we present the Armada system for disaggregated, distributed GNN training. Finally, Chapter 5 discusses potential areas for future work and then concludes this dissertation.

2 BACKGROUND AND CHALLENGES

In this chapter, we discuss necessary background on GNNs and GNN mini-batch training, and highlight the challenges for GNN training over large-scale graphs that this dissertation aims to address.

2.1 Background on GNNs

This dissertation focuses on training GNNs for *node classification* and *link prediction*. These tasks are defined as follows: Given a graph $G = (V, E)$ with nodes V and edges E, the task of node classification is to assign the correct label to a given node $v \in V$ from a set of possible labels. The task of link prediction entails predicting whether a pair of nodes $(v_1, v_2) \in V \times V$ should be connected by an edge or not.

GNNs achieve state-of-the-art accuracy on the above tasks by learning to combine local information about graph nodes (e.g., features of each specific node) with information from their neighborhood in G. Local information for each node is encoded in a *feature vector*: For a node $v \in V$, we denote its feature vector as h_v^0. These feature vectors can be either fixed (e.g., the conference of a paper in a citation graph) or learned parameters part of the GNN model itself (e.g., a learned representation of the type of an entity node in a knowledge graph) (feature vectors that are learned are commonly called *embeddings*). In fact, learnable feature vectors are commonly used for link prediction. All feature vectors for the graph are stored together in a lookup table indexed by node ID.

Given the feature vectors for graph nodes, GNN models learn to combine this local information with information from neighboring nodes: For a given node v, the k-th layer of a multi-layer GNN model computes a new vector representation for v, called h_v^k, that is defined recursively as $h_v^k = AGG(h_v^{(k-1)}, \{h_u^{(k-1)} \colon u \in N_v\}; \Theta_k)$. Here, N_v is the set of one-hop neighbors for node v and AGG denotes the aggregation function, parameterized by Θ_k,

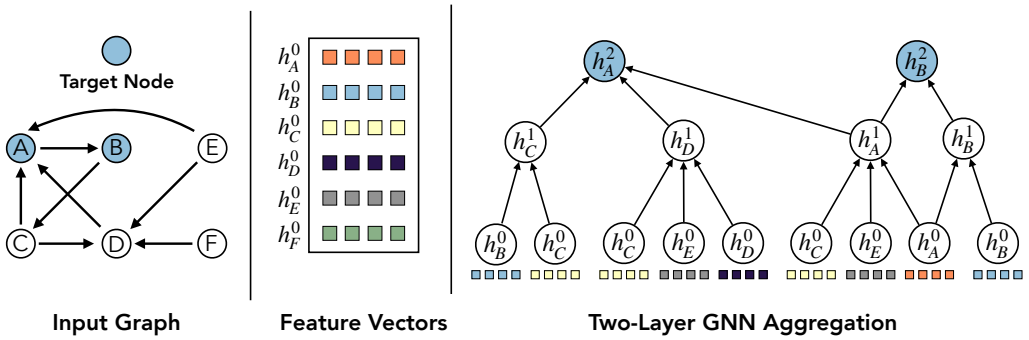


Figure 2.1: Example two-layer GNN aggregation for nodes {A, B} using a sample of their two-hop incoming neighborhood (two neighbors per node).

used to combine the local representation of a node with the representations of its neighbors. For example, a simple GNN layer may compute $h_v^k = W_k * (h_v^{(k-1)} + \sum_{u \in N_v} h_u^{(k-1)})$ with a learned weight matrix W_k. As a result of the recursive definition of h_v^k, observe that the representation of a node v after k layers depends on the k-hop neighborhood of node v. An example two-layer GNN aggregation is shown in Figure 2.1.

GNN model computation completes by feeding h_v^k to a task specific classifier or score function. To perform node classification, h_v^k can be fed into a fully-connected layer followed by a softmax, while for link prediction, $h_{v_1}^k$ and $h_{v_2}^k$ are given as input to a score function (typically referred to as a *decoder*, e.g., DistMult (Yang et al., 2014)) which decides whether the two nodes should be connected by an edge or not. For example, a common score function is the simple vector dot product $f(h_{v_1}^k, h_{v_2}^k) = h_{v_1}^k \cdot h_{v_2}^k$ with the requirement that the two vectors are such that $f(h_{v_1}^k, h_{v_2}^k) \approx 1.0$ if nodes v_1 and v_2 are connected via an edge and $f(h_{v_1}^k, h_{v_2}^k) \approx 0.0$ otherwise.

2.2 GNN Mini-Batch Training

GNN training is similar to standard supervised neural network training; it is performed using labeled training examples and gradient descent. In the case of node classification, a single training example consists of a *target node*

$v \in V$ and its associated class label. For link prediction, a single training example consists of a pair of target nodes $(v_1, v_2) \in V \times V$ and a binary class label indicating whether these two nodes are connected by an edge or not in G. Given a training example, a k-layer GNN computes a prediction for the training example by computing h_v^k for node classification or $h_{v_1}^k$ and $h_{v_2}^k$ for link prediction (and then uses a classifier or score function as described above). The prediction can then be compared to the expected label in order to compute gradients and update the learned GNN parameters.

GNN training, however, contains unique challenges not present for conventional neural networks (Chami et al., 2021; Thorpe et al., 2021; Gandhi and Iyer, 2021; Kaler et al., 2022). First, for large graphs, the storage overhead for the feature vectors can require hundreds of GBs to TBs of memory (as shown in Table 1.1). For example, storing the feature vectors requires 385GB for a common paper citation graph and 1.4TB for the 2012 web hyperlink graph with 3.5B nodes. Thus, for medium to large-scale graphs, the storage overheads required for GNN training exceed the memory capacity of GPU accelerators, and the feature vectors must be stored in CPU memory. Moreover, computing h_v^k for a node v requires the feature vectors for all nodes in the k-hop neighborhood of v; the number of nodes in this neighborhood grows exponentially in size as k increases. As a result, the storage overhead of the feature vectors for all nodes in the multi-hop neighborhood can also exceed the GPU memory capacity. To mitigate this issue, it's necessary to employ multi-hop neighborhood sampling for large-scale GNN training (Hamilton et al., 2017; Kaler et al., 2022) (in which a subset of the full k-hop neighborhood is randomly select and used for computing the GNN aggregation).

Together, the above challenges necessitate the use of mixed CPU-GPU mini-batch training for learning GNN models over large-scale graphs. Mini-batch training consists of two phases. The first phase, referred to as *mini batch preparation*, begins by randomly sampling (generally without replace-

ment) a set of training examples from the input graph. The next step is to sample the necessary multi-hop neighborhoods for all target nodes in the batch. A *mini batch* (or just batch) is prepared once the feature vectors for all target nodes and their neighbors have been loaded. Given the storage overhead of the feature vectors requires that they are stored in CPU memory (as highlighted above), mini batch preparation typically occurs on the CPU. After a batch is prepared, it can be transferred to the GPU to perform the GNN forward pass (e.g., to compute h_v^k) and to compute the loss and gradients needed for updating the model parameters (e.g., the per-layer GNN weights Θ used to parameterize the aggregation functions); these model parameters are generally stored in GPU memory. We refer to this second phase of GNN training on the GPU as *mini batch computation*. If applicable, updates for learnable feature vectors are transferred and written back to CPU memory so they can be read by future batches. One round of training (i.e., one *epoch* over the whole graph) completes when all nodes or edges have been sampled and used as training examples.

2.2.1 Challenge: Data Movement and Staleness

A consequence of mixed CPU-GPU training is that it introduces data movement overheads due to feature vectors being stored off device (i.e., on the CPU instead of the GPU). Depending on whether training proceeds *synchronously* or *asynchronously*, these overheads can lead to low GPU utilization or reduced model accuracy. We review the difference between these two training paradigms and discuss these challenges in detail next.

Synchronous Training Conventional machine learning algorithms proceed in a synchronous manner. In this setting, only one mini batch is processed at a time. That is, one mini batch is prepared on the CPU, then transferred to the GPU for computation, and then used to update the model parameters (and, if applicable, any learned feature vectors) before preparation on the subsequent batch begins. For example, in Figure 2.3a, we show

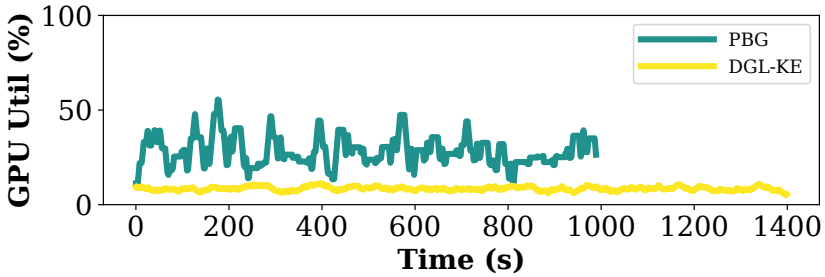


Figure 2.2: The GPU utilization of DGL-KE and PBG for one training epoch on the Freebase86m (Google, 2018) knowledge graph. These systems exhibit low GPU utilization due to data movement overheads.

a mini batch which reads three feature vectors $\{a, c, f\}$ and updates them to $\{a', c', f'\}$. A second mini batch will wait to start until the first batch is completed. This second batch may read, for example, vectors $\{a', b, h\}$ and update them to $\{a'', b', h'\}$. *Notice that the second batch is guaranteed to see the update from a to a' from the first batch.* This is the strength of synchronous training—all updates from one batch are seen by all subsequent batches—and leads to fast model convergence (i.e., the model requires fewer mini batches to learn).

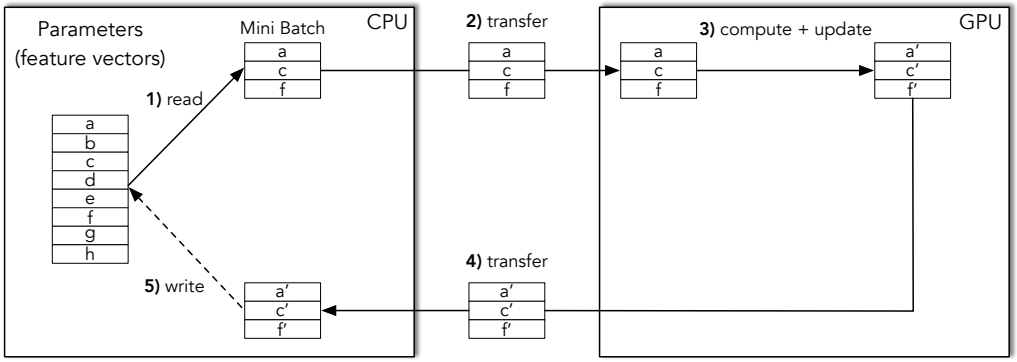
Synchronous ML is the default in many training frameworks but it suffers from low throughput (fewer batches are processed per unit time) because the GPU is idle during mini batch preparation and data transfer (steps 1, 2, 4, and 5 in Figure 2.3a) (i.e., the GPU utilization is low). In fact, current state-of-the-art systems, including DGL-KE (Zheng et al., 2020b), and Pytorch BigGraph (PBG) (Lerer et al., 2019), exhibit poor GPU utilization due to synchronous data movement overheads: Figure 2.2 shows the GPU utilization during one training epoch when using a single GPU for DGL-KE and PBG. As shown, DGL-KE only utilizes 10% of the GPU, and the average utilization for PBG is less than 30%.

Asynchronous Training To improve utilization, asynchronous (or pipelined) training is a common approach. In this case, multiple worker threads continuously prepare and transfer batches to the GPU in parallel.

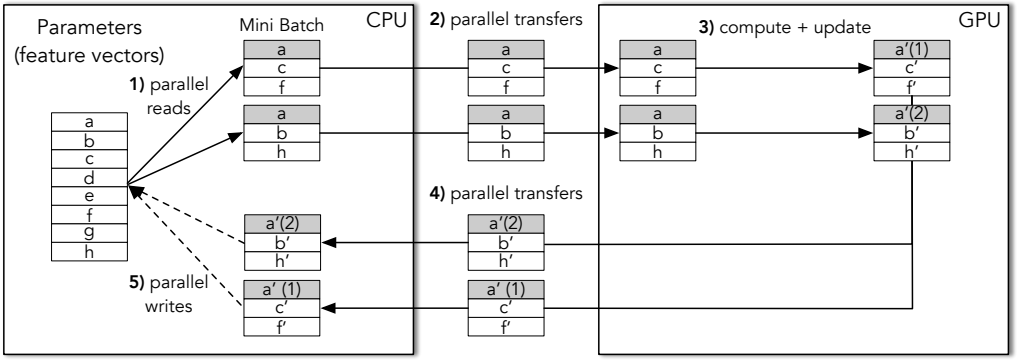
Once on the GPU, mini batches are pushed onto a queue, which is constantly polled by a GPU worker thread performing the GNN computation. The goal is to keep the GPU busy: as soon as it finishes computation on one batch, ideally another batch is waiting in GPU memory and ready for processing. By preparing and transferring batches in parallel, asynchronous training overlaps data preparation and data movement of future batches with computation on the current batch, leading to improved throughput and GPU utilization compared to synchronous training.

While asynchronous training can increase throughput, it can negatively affect model convergence and accuracy in the presence of learnable feature vectors. This is because a key property of synchronous training no longer holds—it is no longer the case that all feature updates from one batch are seen by all subsequent batches. In fact, any mini batches prepared and transferred concurrently that share the same nodes (and thus feature vectors) will miss any updates to these features generated by the other concurrent mini batches—in other words, these feature vectors are *stale*.

Consider the example in Figure 2.3b. Two batches are prepared in parallel. One reads $\{a, c, f\}$ and another reads $\{a, b, h\}$. Notice that these batches *overlap*, i.e. they both read the same feature vector, a in this case. The updates from the first batch (e.g., $\{a, c, f\}$) have not made it back to CPU memory before the second batch (e.g., $\{a, b, h\}$) is started. This is in contrast with the synchronous setting. When both batches reach the GPU, one will be pulled of the queue and processed first, followed by the second batch. In this example, both batches will update a to their own version of a' (we differentiate each version of a' by $a'(1)$ or $a'(2)$ when needed). When these batches subsequently write their updates back to CPU memory, only the last write will persist, for example $a'(1)$. The processing of these two batches in the asynchronous setting has resulted in a different state of features $\{a'(1), b', c', d, e, f', g, h'\}$ than the result of processing these batches one at a time in the synchronous setting $\{a'', b', c', d, e, f', g, h'\}$.



(a) Synchronous Training



(b) Asynchronous Training

Figure 2.3: Example of synchronous versus asynchronous mixed CPU-GPU mini-batch training. Synchronous training proceeds one batch at a time, whereas asynchronous training processes multiple batches in parallel. These parallel batches can overlap (access the same parameters) leading to staleness.

As such, asynchronous training can require more mini batches to be processed compared to synchronous training to reach the same accuracy, or may even fail to reach that accuracy all together. The exact accuracy difference depends on many factors including: the number of feature vectors per batch, the total number of feature vectors, the feature vector access pattern across batches, the number of parallel threads, etc. It is possible to tune some of these parameters to reduce parallel batches with overlapping features and improve the convergence of asynchronous training, but changing

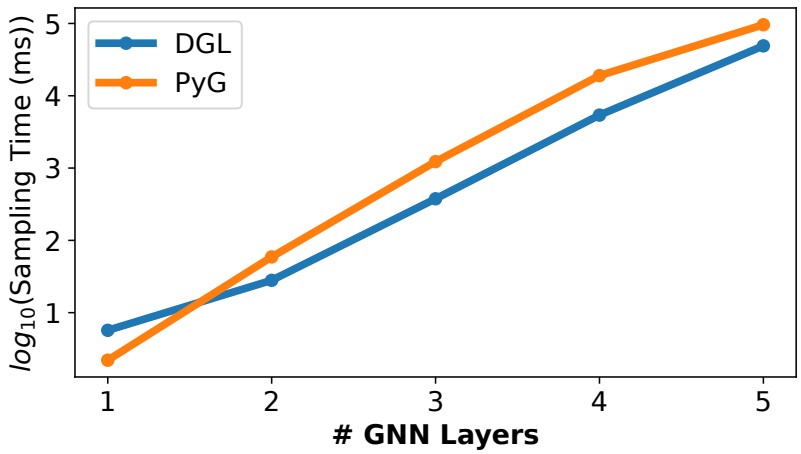


Figure 2.4: The time for multi-hop sampling in DGL and PyG for GNNs of varying depth on the OGBN-Papers100M citation graph. Multi-hop sampling overheads grow exponentially with the number of GNN layers.

these parameters can also affect model accuracy and throughput. In general, asynchronous training can reach similar accuracy to synchronous training when the overlap between concurrent batches is low (Niu et al., 2011), but in other cases, it can result in degraded convergence, potentially rendering it unusable. We remark that *all GNN weights (e.g., the* Θ*s used to parameterize aggregation functions) are updated for every mini batch* (i.e., the overlap of these parameters is always 100% across batches). Thus, these parameters always need to be update synchronously, regardless of whether mini batches are prepared and transferred in a synchronous or asynchronous manner.

In Section 3.1 and 3.2, we present a pipelined architecture for GNN training that allows for asynchronous mini batch preparation and transfer together with synchronous updates to GNN model parameters. We then introduce *Optimistic Asynchrony Control (OAC)* a method to ensure that our pipelined training is guaranteed to reach the same model accuracy as synchronous training even in the presence of learned feature vectors stored in CPU memory and transferred in parallel to the GPU.

2.2.2 Challenge: Multi-Hop Neighborhood Sampling

In addition to data movement, multi-hop neighborhood sampling overheads on the CPU can also bottleneck mixed CPU-GPU mini-batch training and lead the GPU to sit idle. As described above, these overheads grow exponentially with the number of GNN layers. For example, in Figure 2.4 we show the time for multi-hop sampling in two state-of-the-art systems as the number of GNN layers increases. With just three layers, both systems require roughly one second for the requisite neighborhood sampling; with five layers, sampling takes almost 100 seconds. These timings can be an order of magnitude slower than GNN computation. For example, the GPU operations for training take only 170ms when using the three-layer GNN. Even if multiple mini batches are prepared in parallel (e.g., asynchronous training), the overhead of multi-hop neighborhood sampling limits the overall throughput of GNN training. We present the *DENSE* data structure to mitigate these overheads in Section 3.3.

2.3 Scaling Training Beyond CPU Memory

When the storage overhead of a graph exceeds the CPU memory capacity of a single machine, or when parallelization across multiple machines is desired to accelerate training, prior works rely on graph partitioning (Lerer et al., 2019; Shao et al., 2024). In this case, graph nodes (and their features) are split into p disjoint *partitions*. According to the node partitions, the graph edges are grouped into p^2 *edge buckets*, where all edges in edge bucket (i, j) have their source node in partition i and their destination node in partition j (see example in Figure 2.5). To perform training, these partitions and edge buckets can either be 1) stored on disk and periodically brought into CPU memory for training (called *disk-based training*) or 2) loaded into CPU memory on separate machines for *distributed training*.

Both disk-based and distributed training build on single-machine, mixed CPU-GPU mini-batch training with the full graph in memory, but with a few

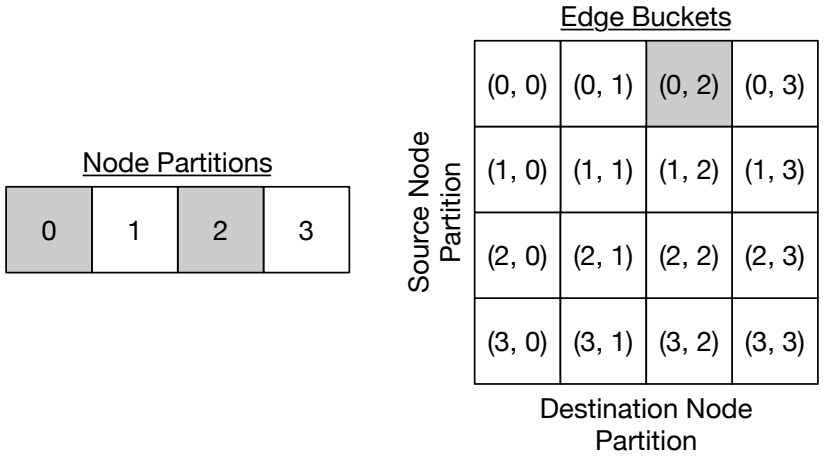


Figure 2.5: Partitions and edge-buckets with $p = 4$. All edges in edge-bucket (0, 2) have a source node in node-partition 0 and a destination node in node-partition 2.

notable differences: For disk-based training, mixed CPU-GPU mini-batch training, particularly mini batch preparation (e.g., multi-hop neighborhood sampling), is performed only over the partitions (and the edge buckets between them) that are loaded in CPU memory. Thus, a single training epoch over all nodes (or edges) in the graph requires swapping partitions into and out of CPU memory as needed. Disk-based training in this manner has the following advantages: random access to graph features (e.g., as a result of neighborhood sampling) occurs only over graph data in memory, but not over graph data stored on disk (where random access is prohibitively expensive); instead, access to graph data stored on disk consists of only large reads and writes to graph partitions and their edge buckets (which are also stored sequentially).

Similarly, for distributed training, each machine is responsible for running mixed CPU-GPU training in parallel. Unlike in disk-based training, however, mini batch preparation can still be performed over the whole graph; that is, mini batch preparation is also distributed—Each machine is responsible for preparing batches in parallel and sampling the required

multi-hop neighborhoods across the whole graph, by communicating with other machines as needed.

2.3.1 Challenge: Scalable Min-Edge-Cut Partitioning

Cross-machine neighborhood sampling and feature loading can lead to a communication bottleneck that fundamentally limits the scalability and throughput of distributed GNN training across a set of machines (Kaler et al., 2023). Yet at the same time, neighborhood sampling across just a subset of graph partitions in CPU memory (e.g., as in disk-based training), can lead to lower model accuracy as a result of reduced neighborhood information available for nodes during GNN training.

To mitigate these issues, existing systems rely on partitioning algorithms that minimize the number of edges with endpoints in different partitions (and thus machines) (Zheng et al., 2022). Such edges are known as *cut edges*. Min-edge-cut partitioning ensures that graph nodes have as many neighbors together with them in the same partition as possible, and thus that these neighbors are available in memory during disk-based training or available on the same machine (without communication across machines) during distributed training. Thus, min-edge-cut partitioning is widely used in GNN systems; In fact, for distributed training it can lead to an order of magnitude faster training compared to random partitioning (Merkel et al., 2023; Zheng et al., 2022).

Min-edge-cut partitioning, however, becomes increasingly expensive with graph size. For instance, many systems utilize the offline algorithm METIS (Karypis and Kumar, 1997) due to its ability to effectively minimize edge cuts by iteratively refining partitions across the whole graph and its comparatively efficient implementation (Merkel et al., 2023; Shao et al., 2024; Lin et al., 2023); yet, METIS takes 8000s and requires a special machine with 630GB of memory to partition a common benchmark graph (the 1.6B edge OGBN-Papers100M), whereas GNN training takes only 549s (10 epochs, one GPU) and can run on cloud machines with 244GB of memory (Waleffe

et al., 2023) (details in Section 6.3). Although the partitioning overhead can be amortized across models, it still presents a bottleneck to GNN training. To address this issue, streaming algorithms iterate over the graph and assign vertices to partitions greedily (Abbas et al., 2018). While these algorithms offer improved scalability (e.g., FENNEL (Tsourakakis et al., 2014)), they tend to result in more edge cuts (Zhang et al., 2018); e.g., we find a streaming greedy approach cuts up to 4× more edges than METIS.

In Chapter 4, we present *GREM*, a novel min-edge-cut partitioning algorithm to address the bottlenecks of graph partitioning in existing disk-based or distributed GNN training pipelines.

2.3.2 Challenge: Disk-Based Training

Given a partitioned graph, a key remaining challenge for disk-based training is that partition swaps can lead to IO-bound training; recall that in order to complete one epoch, all training examples (e.g., graph nodes or edges) need to be brought into memory for training, which requires swapping partitions and their edge buckets to and from disk as needed, according to a partition replacement policy. We find that these partition swaps can be expensive and lead GPUs to sit idle in existing systems which utilize disk-based training (Lerer et al., 2019). Thus, to efficiently scale training beyond CPU memory using disk, it is necessary to mitigate the overheads that arise from swapping partitions.

To address this issue, prior work in IO-bound settings leverage buffer management techniques (e.g., prefetching, asynchronous reads/writes to disk) to hide and reduce IO (Ramakrishnan et al., 2003; Hellerstein et al., 2007). In particular, these works highlight the importance of the order in which data is accessed on overall throughput (for GNN training, this order corresponds to the order in which training examples are used to generate mini batches during one epoch) (McSherry et al., 2015). When the data order exhibits good locality (i.e., many successive mini batches access nodes from the same partitions), overall epoch IO is typically reduced (due to

less swaps), yielding improved throughput and utilization. Additionally, if the data order is known ahead of time, the CPU can prefetch partitions needed in the near future and use Belady's optimal replacement algorithm to decide which partitions to evict (Belady, 1966), helping to further hide and minimize IO overheads.

While existing locality-aware data orderings over graphs (e.g., Hilbert space filling curves) have been shown to improve locality of accesses and performance of common graph algorithms such as PageRank (McSherry et al., 2015; Maass et al., 2017), we find that these orderings still result in IO-bound GNN training due to a non-optimal amount of swaps (Section 5.2). Moreover, when data orderings which only focus on minimizing IO are applied to GNN training, they lead to reduced model accuracy (these orderings heavily conflict with the random orderings preferred by stochastic gradient descent). In fact, we find that accuracy can drop by up to 16% on common GNN benchmarks with using disk-based training instead of training with the entire graph in CPU memory. To address these challenges, in Chapter 5 we propose *BETA*, a buffer-aware partition replacement policy and data ordering which results in a near-optimal number of swaps, and *COMET*, a policy which build on BETA but increases the amount of randomness present in the order in which examples are used for training, leading to increased accuracy for GNN training.

2.3.3 Challenge: Scalable Distributed Training

As highlighted above, multi-hop neighborhood sampling overheads on the CPU are a major challenge for mixed CPU-GPU mini-batch training; this challenge becomes even more apparent for distributed GNN training over multiple GPUs. In fact, we find that even when there is zero communication involved, and we use our optimized sampling implementation introduced in Chapter 3, mini batch preparation can bottleneck distributed GNN training, leading to GPU underutilization and unnecessarily expensive training. This problem is exacerbated on common cloud machines with fast GPUs and

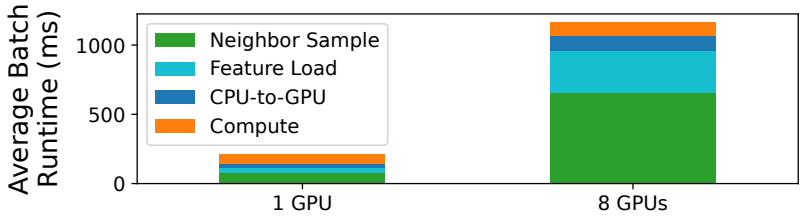


Figure 2.6: Breakdown of the average runtime per training iteration in our state-of-the-art system MariusGNN (GraphSage-Large on OGBN-Papers100M; details in Section 6.3). Despite optimized implementations, neighborhood sampling plus feature loading on the CPU dominates GNN runtime as compute resources are scaled.

fixed CPU resources.

For example, in Figure 2.6 we show the average time for mini batch preparation and computation across training iterations on a common GNN benchmark. Figure 2.6 shows that multi-hop sampling (even when optimized as described in Waleffe et al. (2023) and Chapter 3) and feature loading—which together encompass mini batch preparation—dominate overall training time as the number of GPUs increases. This occurs because distributed data parallel training with weak scaling, commonly used to distribute GNN training, requires preparing one mini batch per GPU for each training iteration. For example, eight GPU training requires preparing eight times more data, leading to an increase in neighborhood sampling and feature loading times, but not an increase in GNN computation time because each mini batch is processed in parallel across the eight GPUs.

Given the runtime discrepancy, it's necessary to increasingly parallelize mini batch preparation across CPUs as the number of GPUs increases in order to keep them busy with computation. Existing systems, however, rely only on the fixed set of CPU resources attached to GPU machines for this parallelization, fundamentally hindering their ability to prepare batches (Kaler et al., 2022; Zheng et al., 2022). To highlight this issue, in Figure 2.7, we show the CPU utilization of the state-of-the-art system

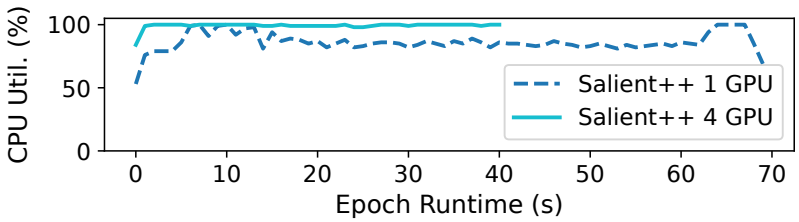


Figure 2.7: CPU utilization in the state-of-the-art system Salient++ when training a GraphSage-Small GNN on OGBN-Papers100M (details in Section 6.3). Nearly all CPU resources are used to parallelize mini batch preparation and minimize training time with one GPU; the CPU resources are insufficient for multi-GPU training, leading to sublinear speedups.

Salient++ (Kaler et al., 2023) during GNN training. Salient++ requires more than 80 percent of the CPU to prepare batches in parallel and fully utilize one GPU. When training with four GPUs, the CPU is fully saturated, limiting the throughput of mini batch preparation, leading to sublinear scaling (1.6× instead of the possible 4×), and resulting in expensive GPUs sitting partially idle.

The above observations motivate a disaggregated system for large-scale GNN training that supports scaling each part of the workload independently. While disaggregation has improved resource efficiency in traditional ML settings (Graur et al., 2022; Jin et al., 2024), prior work on disaggregated GNN training (Dorylus (Thorpe et al., 2021)) has focused only on utilizing serverless functions and full multi-hop neighborhoods (i.e., no sampling). Full neighborhoods, however, lead to expensive communication for multi-layer GNNs and a serverless architecture limits the type of models that can be trained efficiently without GPUs. On a common GNN benchmark, we find that Dorylus plateaus at 89.6s/epoch ($3.75/epoch); GPU-based systems can be 12× faster and 53× cheaper (details in Table 6.1 left).

In Chapter 6, we present Armada, a new system for GNN training which disaggregates CPU-based mini batch preparation from GPU-based GNN computation, allowing the resources dedicated to parallelizing each part of

the workload to be scaled independently, and enabling scalable distributed GNN training over multiple GPUs.

2.4 Weather Prediction: A Motivating Application

As a motivating application for large-scale GNN training, in this Section we consider the task of global weather forecasting, which has significant implications ranging from predicting the surface temperature at various locations around the globe to predicting the path of a hurricane. Below, we describe how this task can be implemented using GNNs as well as the challenges and benefits in doing so.

How to Use GNNs for Weather Prediction To map the task of weather forecasting to GNN training and inference, the first step is to define the input graph and feature vectors. In this case, a natural choice is to use graph nodes to represent latitude and longitude grid points and to use edges to connect nearby grid points together (potentially with edge weights describing distance and location). Node feature vectors can then be used to describe physical data variables (e.g., temperature) at each latitude and longitude grid point and at some specific time. This implies that the total number of graph features is $num\_time * num\_lat * num\_long * feature\_dim$ (if the graph contains data about num_time timestamps, num_lat latitude grid points, num_long longitude grid points, and $feature\_dim$ data variables). These feature vectors can be downloaded from historical records (e.g., ERA5 reanalysis (Hersbach et al., 2020)).

Given the input graph and feature vectors, a GNN can be trained to forecast the weather in similar fashion to the task of node classification as described above. In this setting, a single GNN input example consists of a node and its feature vector at some timestamp t (and the necessary feature vectors of neighboring nodes at timestamp t) and the "label" consists of the node features at some later timestamp $t + \alpha$. In this way, the GNN is trained

to predict the weather for each node at some later timestamp $t + \alpha$ given the weather at that node and its neighbors at some earlier timestamp t. Once the GNN is trained, it can be fed the weather (features) for a timestamp t and used to forecast the weather (features) for timestamp $t + \alpha$.

We remark that the above is meant to provide only a high level overview. In practice, more sophisticated modeling is used (e.g., different grids better suited for the globe, multi-resolution grids to capture local and global weather patterns, etc.). More details can be found in Lam et al. (2022).

Workload Challenges GNN training for the application of weather prediction is primarily limited by the ability to efficiently scale to the large corpus of existing weather data, a task which is necessary to perform both research and achieve state-of-the-art results. This limitation prevents only a select few from working on GNN-based weather forecasting models. For example, just a single year's worth of basic training data (one degree latitude and longitude grid, 84 data variables) is over 200GB, but realistic training scenarios require many TBs (e.g., 53TB as described in the introduction).

This Dissertation: Towards Democratizing GNN Training Motivated by the need to support GNN training in the presence of TBs of node feature vectors, as described in Chapter 1, this dissertation aims to enable cost-effective, scalable GNN training over massive graphs, even with limited resources. As a demonstration of the algorithms and systems described in this thesis, we apply them to the task of weather forecasting.

We used six data variables (temperature, geopotential, humidity, eastward wind, northward wind, vertical wind) at 13 pressure levels in the atmosphere and a one degree latitude and longitude grid (plus six additional features: 1) the day of the year, 2) the time of day, 3) sin(longitude), 4) cos(longitude), 5) sin(latitude), 6) cos(latitude)). We use an α of six hours. In this setting, the MariusGNN system introduced in this dissertation (Waleffe et al., 2023), can perform training over an entire years worth of data (with feature vectors at every hour in the year) in just four minutes using

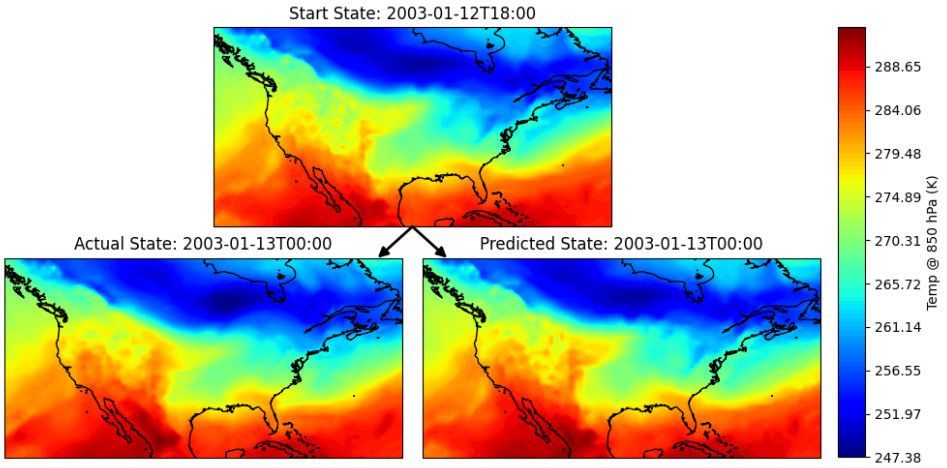


Figure 2.8: Example six hour GNN-based temperature forecast for the United States given an initial state in January 2023. The GNN model used to make this prediction was trained using the MariusGNN system described in this thesis on a single machine with a single GPU.

just a single machine (and disk) and a single NVIDIA V100 GPU.

In Figure 2.8, we visualize a sample weather forecast using our trained GNN model. Starting from an initial state at timestamp t (top), we show the temperature forecast six hours in the future (bottom right) and compare to the actual temperature that truly occurred at that time (bottom left).

Implications GNN-based weather forecasts have potential to improve accuracy due to their ability to capture patterns in the spatial data which may not be represented in the explicit physical equations used by current numerical weather prediction algorithms. It also offers the potential for greater efficiency; for example, GraphCast needs just one minute to generate a forecast compared with six hours for existing state-of-the-art numerical weather prediction algorithms Lam et al. (2022). These efficiency improvements allow for the use of larger ensemble models (the combination of many predictions using slightly different initial states) which helps to better predict rare weather events and provide early warnings when they do occur.

We hope that the contributions in this thesis help to enable researchers to more quickly, easily, and cheaply continue to develop GNN-based weather prediction models. Yet, while we have used this application as an example in this section, GNNs are being applied to many applications beyond global weather forecasting which can have different requirements and challenges (e.g., learnable feature vectors). Thus, we focus primarily in this thesis on general, abstract GNN training over arbitrary graphs.

2.5 Related Work

We highlight works related to this dissertation which seek to address similar challenges to those described above in Section 2.2 and 2.3.

Systems for ML over Graph Data Many systems support GPU training of GNNs (Gandhi and Iyer, 2021; Jia et al., 2020; Kaler et al., 2022; Zhu et al., 2019; Lin et al., 2020; Dong et al., 2021; Wu et al., 2021). Two such popular systems are DGL (Wang et al., 2019) and PyTorch Geometric (Fey and Lenssen, 2019). Complementary to these systems, many works focus on scaling different dimensions of GNN training: To reduce the overhead of mixed CPU-GPU training, some works highlight the importance of GPU-oriented data communication or caching (Min et al., 2021a,b; Kaler et al., 2022; Lin et al., 2020; Dong et al., 2021). Additional works focus on optimized GPU kernels (Wang et al., 2021b). In general, these works focus on orthogonal challenges of GNN training than those discussed here and these ideas can be incorporated into our systems (MariusGNN and Armada). Finally, there are works that focus on scaling the training of non-GNN based graph models (Zheng et al., 2020b; Lerer et al., 2019; Akyildiz et al., 2020).

Large-Scale Training To scale GNN training to graphs that exceed the CPU memory capacity of a single machine, many works opt for a distributed multi-machine approach (Zheng et al., 2022; Gandhi and Iyer, 2021; Jia et al., 2020; Zhu et al., 2019; Wang et al., 2021a). Some of these works share design choices with the systems discussed in this dissertation. For example, recent

work introduces DistDGLv2 as a distributed version of DGL (Zheng et al., 2022) and utilizes METIS partitioning, colocation of data with mini batch computation, and asynchronous mini batch preparation, but they do not use GREM or disaggregation to scale training. Several of these works aim to reduce cross-machine communication during multi-hop sampling, either by using min-edge-cut partitioning (Zheng et al., 2022) or by employing feature replication on each machine (Liu et al., 2023; Kaler et al., 2023). Other works distribute training in a serverless manner (Thorpe et al., 2021). Still other works focus on scaling training using disk storage; these works primarily focus on training for link prediction using non-GNN models (Sun et al., 2021; Lerer et al., 2019). In this dissertation, we focus on disk-based GNN support for both node classification and link prediction as well as scalable distributed GNN training for these tasks.

Neighborhood Sampling Many works focus on reducing the overhead of neighborhood sampling. Initial approaches sample a fixed number of neighbors per node (Hamilton et al., 2017), while follow-up works sample a fixed number of neighbors per layer (Chen et al., 2018; Zou et al., 2019). Other works decouple the sampling frequency from the mini batch frequency (Ramezani et al., 2020). In this dissertation, we focus on sampling a fixed number of neighbors per node with minimal redundancy. Still other works focus on making mini-batch training more efficient by increasing the density of edges between nodes in a mini batch (Zeng et al., 2020; Chiang et al., 2019). These contributions can be incorporated in our work and are orthogonal to our study. Finally, recent works utilize GPUs to speed up sampling (Dong et al., 2021; Jangda et al., 2021). The systems introduced in this dissertation support GPU-based sampling but use CPU-based sampling to scale to large graphs.

3 EFFICIENT MIXED CPU-GPU MINI-BATCH TRAINING

In this Chapter, we focus on techniques to maximize GPU utilization, and thereby minimizing cost and runtime, during mixed CPU-GPU mini-batch training. In Section 3.1, we present a pipelined architecture for asynchronous training that overlaps data preparation and movement with computation to achieve high throughput. Then, in Section 3.2 we introduce Optimistic Asynchrony Control (OAC) to ensure that asynchronous training results in models with accuracy equivalent to those obtained through synchronous training. In Section 3.3, we introduce the DENSE data structure to minimize the redundant computation present in multi-hop neighborhood sampling, and thus the overhead of this mini batch preparation step.

We implement the above techniques in a new system which we call MariusGNN (Waleffe et al., 2023); in Section 3.4, we present end-to-end results for mixed CPU-GPU GNN training in MariusGNN and compare to existing state-of-the-art systems. We show that MariusGNN with one GPU can train to the same accuracy 4× faster than existing systems, even when these systems use multiple GPUs. We also show that DENSE enables in-CPU multi-hop sampling that is up to 14× faster than the corresponding implementations in state-of-the-art baselines.

3.1 Asynchronous Pipelined Training for High Throughput

MariusGNN uses a pipelined architecture for mixed CPU-GPU GNN training. We discuss the overall design and then the details of each pipeline stage.

Pipeline Design Our architecture divides mixed CPU-GPU mini-batch training into a five-stage pipeline with queues separating each stage (Figure 3.1). Four stages are responsible for data preparation and movement

operations, and one stage is responsible for GNN model computation and on-GPU parameter updates. The four data movement stages have a configurable number of worker threads, while the model computation stage uses only a single worker to ensure that GNN parameters stored on the GPU are always updated synchronously (see Section 2.2). We next describe the different stages of the pipeline and responsibilities of each during training:

Stage 1: Prepare The first pipeline stage is responsible for mini batch preparation and occurs on the CPU. As described in Section 2.2, this step consists of sampling and loading a set of training examples (e.g., nodes or edges) from the input graph, sampling the necessary multi-hop neighborhoods for these training examples, and loading the corresponding feature vectors that form the input to GNN models. Once a batch is prepared by a worker thread, it is pushed onto a queue, which we call the *GPU transfer queue*, and the worker can begin preparing another mini batch.

Stage 2: Transfer The input to this stage consists of prepared mini batches from the previous stage. Worker threads in this stage continuously read mini batchs from the GPU transfer queue and asynchronously transfer them from CPU memory to GPU memory using CUDA Steams (Nickolls et al., 2008). Once on the GPU, the are added to the *GNN input queue*.

Stage 3: Compute The compute stage is the only stage that takes place entirely on the GPU. The compute worker thread reads prepared mini batches off of the GNN input queue and computes the GNN output for the target nodes, using the latest GNN model weights that are stored in GPU memory. The GNN output is then used to compute gradients and update the GNN model parameters. Any updates to learnable feature vectors (i.e., gradients that need to be added to the previous version of these vectors) are added to the *GNN output queue* to be transferred from GPU memory back to CPU memory. After completing the GNN computation for one mini batch, the GPU proceeds immediately to computation on the next batch available in the GNN input queue in GPU memory.

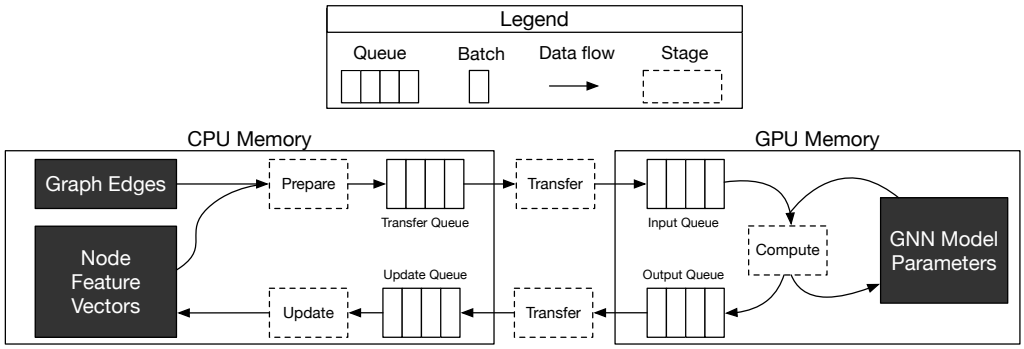


Figure 3.1: Overview of pipelined GNN training in MariusGNN.

Stage 4: Transfer The updates to feature vectors generated in Stage 3 and placed in the GNN output queue are read by worker threads and transferred from GPU memory back to CPU memory. We use similar mechanisms as in Stage 2. Once on the CPU, the feature vector updates are added to the *update queue* in CPU memory.

Stage 5: Update The final stage in our pipeline consists of worker threads which read updates to learnable feature vectors from the update queue and apply them to the feature vector lookup table in CPU memory so that they can be read by future batches.

This pipelined architecture allows MariusGNN to asynchronously prepare and transfer mini batches from the CPU to the GPU (and any updates back if needed), ensuring that the GPU does not sit idle while waiting for data preparation or data movement, helping to optimize GPU utilization and throughput during GNN training.

3.2 OAC: Optimistic Asynchrony Control for High Accuracy

As described in Section 2.2, the main challenge when using a pipelined architecture is that, for learnable feature vectors, asynchronous processing of mini batches introduces *staleness* that can lead to lower model accuracy (Kyrola

et al., 2012). We now introduce Optimistic Asynchrony Control (OAC), a protocol for mixed CPU-GPU training that allows for parallel preparation and transfer of batches but eliminates staleness and guarantees that asynchronous training reaches the same model accuracy as synchronous training. OAC can be applied to any mixed CPU-GPU training pipeline in which learned *parameters* are stored in CPU memory but transferred and updated on the GPU during training (i.e., not just GNN training); thus, here we use more general language and refer to feature vectors and GNN computation simply as *parameters* and *model computation* respectively.

3.2.1 Overview and Key Ideas

OAC is motivated by database concurrency control which solves an analogous challenge. To increase throughput, databases allow multiple transactions to run in parallel, but they ensure that the final result is equivalent to a serial execution of each transaction. Initial approaches relied on locking of data objects to prevent two simultaneous transactions from running in parallel and modifying the same records (known as pessimistic approaches). Optimistic methods for concurrency control (termed OCC (Kung and Robinson, 1981; Tu et al., 2013)), however, always allow transactions to be processed in parallel, but validate that no concurrent transactions conflict before writing updates to the database. If validation fails, two transactions accessed the same data concurrently and one transaction must abort to prevent updating the database to an inconsistent state.

For OAC, rather than adopting a pessimistic approach and preventing batches which overlap (i.e., access the same parameters in CPU memory) from running in parallel, we take an optimistic approach and allow all batches to run concurrently but validate parallel batches against each other for overlap before model computation. Unlike in conventional OCC where overlapping transactions require aborts, in OAC we show how batches which access the same parameters concurrently can be updated just before the model computation step to have the correct parameters. Here, the correct

parameters refer to the values each batch would have had if they had been processed one at a time (i.e., if synchronous training had been used).

Key Ideas Specifically, the key contributions of OAC are as follows. First, we highlight that in pipelined mixed CPU-GPU training, the order in which batches pass through the GPU computation step defines a one-by-one, serial order over batches. Given this order, the goal of OAC is to ensure that asynchronous training results in the same updates to model parameters as synchronous training would have produced according to this order. To achieve this goal, we introduce timestamps for each parameter to track different versions and allow us to easily decide which value to accept for a parameter in the presence of multiple options. Finally, we add an on GPU parameter cache to track the parameter sets of concurrent batches. This cache allows us to validate that a batch has the correct parameter values just before it enters computation to produce model updates.

3.2.2 OAC: Optimistic Asynchrony Control

We now discuss the OAC protocol in detail. To aid in the description, we continue with the running example of two batches which access parameters labeled $\{a, c, f\}$ and $\{a, b, h\}$, the synchronous and asynchronous processing of which has previously been described in Section 2.2 (Figure 2.3).

3.2.2.1 The Serial Order

The goal of OAC is to ensure that asynchronous training produces updates to model parameters that are equivalent to synchronous training according to some order of batches. The first question is then: what order of batches? Recall from Section 3.1 that in our pipelined architecture, batches are read and transferred in parallel from the CPU to an on GPU *input queue*. The GPU then reads batches one by one from this queue for computation. We refer to this one by one order of mini batches as the *computation order*.

This existence of the computation order is necessary for OAC. Given the computation order, the goal is to ensure that asynchronous training

produces the same parameters in CPU memory as those that would have been produced had mini batches been prepared and transferred synchronously according the computation order. We remark that even though batches are processed by the GPU sequentially in asynchronous training, by default asynchronous training does not result in the same parameters in CPU memory as synchronous training (see Section 2.2).

3.2.2.2 Equivalence to The Serial Order

Given the computation order defined in Section 3.2.2.1, we now seek to ensure that asynchronous training is equivalent to synchronous training according to this sequence of batches.

Parameter Timestamps To achieve this goal, OAC introduces timestamps for each CPU parameter (Figure 3.2a). Parameter timestamps are used to determine the correct parameter value in the presence of multiple versions. The OAC protocol requires that parameter timestamps are read together with parameters themselves. Critically, to write a parameter, OAC requires that the writer has a larger timestamp for the parameter of interest than the timestamp for which it is trying to overwrite. Smaller timestamps are not allowed to overwrite larger timestamps. The importance of this point will be highlighted below. Timestamps are initialized to minus one.

Parameter Locks OAC also introduces per-parameter locks which must be acquired before reading or writing a parameter and its corresponding timestamp (Figure 3.2a). These locks ensure that each read and write of a parameter value plus its timestamp is atomic, and are needed in the presence of concurrent reader/writer threads; a parameter may be more than just a single float (e.g., a feature vector for GNN training), thus reading or writing it may require more than just one atomic harware operation.

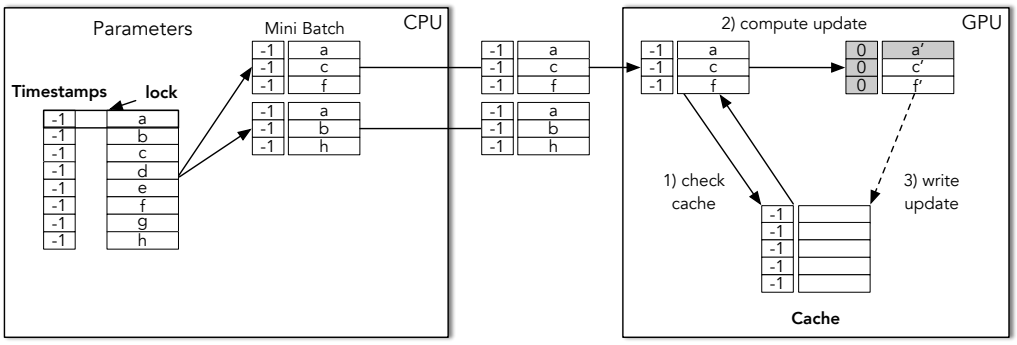
Given parameter timestamps and locks, mini batch preparation and transfer in OAC proceeds as before in asynchronous pipelined training—batches of parameters can be read and transferred to and from the GPU

in parallel (Figure 3.2a). The key difference between OAC and standard pipelined training comes once batches reach the GPU.

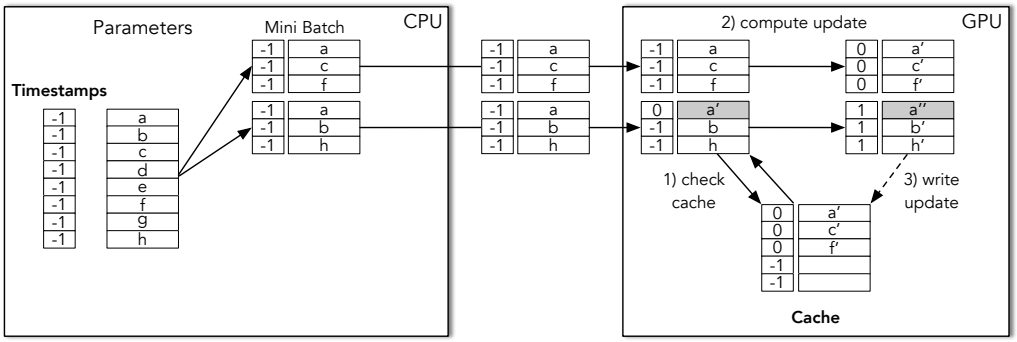
On-GPU Parameter Validation When a batch is removed from the GPU input queue for computation, OAC requires that it first check an on-GPU cache of parameter values; this cache serves to track the parameters and updates of concurrent batches. If any parameters in the batch are present in the cache, and the values in the cache have a larger timestamp, then the parameters in the batch must be replaced by the values in the cache. Then, after the model computation and the parameters in the batch have been updated, they are assigned a new timestamp according to a GPU timestamp counter, which is then atomically incremented. Assigning timestamps to parameters immediately after they are update by the GPU means timestamps capture the computation order. Finally, before being placed on the GPU output queue to be transferred back to the CPU, updated parameter values and their timestamps are written to the GPU cache. Batches which were prepared and transferred in parallel can then subsequently validate against the cache and ensure that their parameters have the correct values according to synchronous training with the computation order.

Example Returning to our running example, we assume batch $\{a, c, f\}$ is first to be removed from the GPU input queue (i.e., first in the computation order). As shown in Figure 3.2a, based on the OAC protocol, this batch checks the cache before proceeding. In this case, there is nothing to do, as it is the first batch to be processed. The batch can then proceed with the model computation and parameter updates (e.g., $\{a, c, f\} \rightarrow \{a', c', f'\}$). After the parameters have been updated, the GPU assigns these updates a new timestamp (e.g., $\{a', c', f'\}$ are assigned timestamp zero in Figure 3.2a).

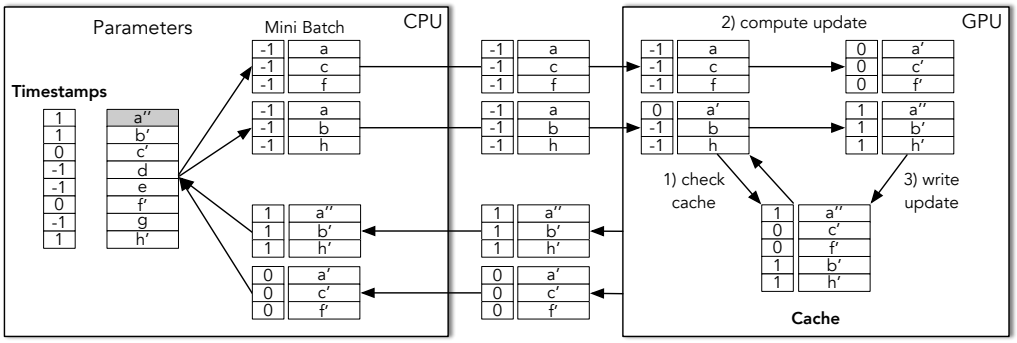
The other batch in our running example, $\{a, b, h\}$, is next to be processed by the GPU. Thus, it is the second batch in the computation order and the OAC protocol must ensure that the computation for this batch is equivalent to the computation that would have occurred had training waited for the



(a) The first of two parallel batches is processed by the GPU. It is then assigned the timestamp zero and its parameter updates are written to the on-GPU cache.



(b) The second of two parallel batches is processed by the GPU. It notices a newer version of parameter a in the cache, and thus updates its value $a \rightarrow a'$ before model computation. In doing so, this batch ensures it has the same parameter values as it would have had according to synchronous training after the first batch.



(c) The two parallel batches write their updates to CPU memory. Only the value of a with the larger timestamp will persist, ensuring that the final parameter states are equivalent to those obtained through synchronous training.

Figure 3.2: Example of OAC parallel batch processing while ensuring equivalence to a synchronous execution of batches.

updates $\{a, c, f\} \rightarrow \{a', c', f'\}$ to make it back to the CPU before preparing and transferring $\{a, b, h\}$. Processing of this batch is depicted in Figure 3.2b. Based on the OAC protocol, the batch must validate its parameters against concurrent batches by checking the GPU cache. In this case, there is a conflict for parameter a versus a'. In other words, a concurrent batch has modified $a \rightarrow a'$. The timestamp of a' in the cache is zero while the batch has timestamp minus one for its version of a. Thus, the value of a' is the more recent version for this parameter according to the computation order and the batch must replace a with a' before it proceeds with computation. As such, the batch $\{a, b, h\}$ is first updated to $\{a', b, h\}$, and then updated by the model computation to $\{a'', b', h'\}$. As before, after the computation step the parameters are assigned timestamp one, and the parameter updates are added to the cache. The value for a' with timestamp zero is replaced by a'', the newer timestamp version of this parameter. The resulting cache state is shown in Figure 3.2c.

To complete our running example, the two batches can be transferred back to the CPU in parallel. They can also write their updates to CPU memory in parallel, but must grab per-parameter locks and obey the timestamp rules. If the batch with timestamp one acquires the lock to parameter a first, it will update $a \rightarrow a''$ and update the timestamp to one. When the batch with timestamp zero subsequently acquires the lock to parameter a (now a''), it will notice that the CPU timestamp is larger than its timestamp. Thus, it must not overwrite this value. It is required to do nothing and release the lock. This ensures that older updates according to the computation order do not overwrite newer updates. Notice that the final state of our CPU parameters is $\{a'', b', c', d, e, f', g, h'\}$—this is equivalent to the final state achieved with synchronous training using the computation order of batches (as described initially in Section 2.2).

3.2.2.3 Implementation

The main OAC protocol has been presented in the previous section. We now discuss several considerations required to implement OAC in practice.

Cache Eviction While we have discussed adding parameter values and their timestamps to the on-GPU cache, we have not discussed when to evict values. Eviction is required to prevent the cache size from growing indefinitely, as it is assumed for mixed CPU-GPU training that the full set of parameters is too large to fit in GPU memory. Intuitively, a parameter can be evicted when we can ensure that all subsequent batches to be processed by the GPU will have already had the chance to see this parameter value when they were prepared on the CPU. In other words, a parameter with timestamp x on the GPU can be evicted when we can guarantee that future batches seen by the GPU which contain this parameter will have read a timestamp greater than or equal to x for this value from CPU memory.

In OAC, we utilize additional metadata to help implement cache eviction. First, the CPU tracks the *maximum finished consecutive timestamp (MFCT)* received from the GPU. This value is updated atomically after each batch finishes writing updates to CPU memory. It corresponds to the largest continuous batch timestamp which has finished. For example, if batches with timestamp $\{0, 1, 2, 5, 6\}$ have completed, regardless of the order in which they finished, the MFCT is 2. If batches 3 and 4 later arrive on the CPU, the MFCT will then be 6. Just before the parameters in a batch are read, this value is read atomically and added to the batch metadata. *In this way, each batch knows the maximum timestamp for which it can guarantee that all updates equal or prior to this value in the computation order were present in CPU memory when it began reading parameter values.* Thus, a batch with MFCT equal to y will not need to read parameter values with timestamp less than or equal to y from the GPU validation cache.

The final metadata required for cache eviction is a dictionary of outstanding batches (batches that are somewhere in the pipeline) and their MFCT

values. Upon creation, each batch is assigned an ID (not necessarily the same as its computation order timestamp); the pair $\{BatchID : BatchMFCT\}$ is then atomically added to an *outstanding batches* dictionary. When batches finish writing updates back to CPU memory, their key-value pair is atomically removed from this data structure. For cache eviction, when a batch reads its MFCT value and adds its key-value pair to the outstanding batches dictionary, it also atomically calculates *the minimum MFCT value across all batches in the dictionary*. This value is stored in the batch's *safe to evict (STE)* metadata field. When batches are read by the GPU, all parameter values with timestamp less than or equal to the STE metadata field can be evicted. In OAC, the GPU performs this eviction for each batch it processes.

When a batch calculates its safe to evict timestamp, even if this batch somehow beats all other outstanding batches to the GPU queue, it is still okay to perform eviction as described above. This is because all subsequent outstanding batches have an MFCT greater or equal to the STE value. That means that they were prepared on the CPU after the entries we wish to evict had already been persisted in CPU memory. Any future batches that will be prepared on the CPU will also have an MFCT greater than or equal to this value. Thus no batch will ever again reach the GPU and find a timestamp for a parameter less than the STE value but greater than than the timestamp it read from CPU memory. Therefore, no batch will ever again need to read timestamps less than or equal to the STE value from the cache and they can be safely evicted.

Deadlock Prevention When implementing OAC, one also needs to consider whether the per-parameter locks introduced for ensuring atomic reads and writes to parameter values and their timestamps can introduce deadlocks. In principle the answer is yes, however we have not observed this phenomena in practice. One possible reason is that these locks are extremely lightweight. They are held only for the time it takes to read or write a few bytes from CPU memory. For this reason, currently we do

nothing to prevent deadlocks in OAC. If deadlocks were prevalent, a simple solution would be to require that all batches read and write the parameters in a global order. For example, this could be achieved by assigning the parameter values IDs and then reading and writing in sorted ID order.

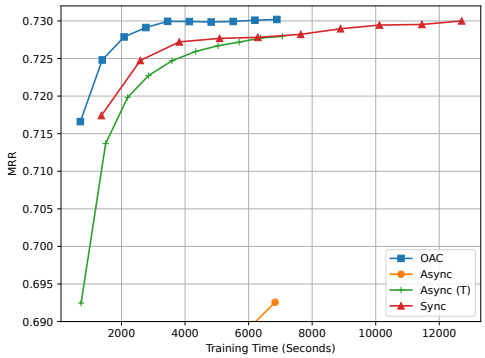
3.2.3 Empirical Evaluation of OAC

In this section, we evaluate OAC by comparing its throughput and convergence with standard synchronous and asynchronous training. We show that OAC achieves synchronous convergence while maintaining asynchronous throughput, offering the best of both methods for mixed CPU-GPU training.

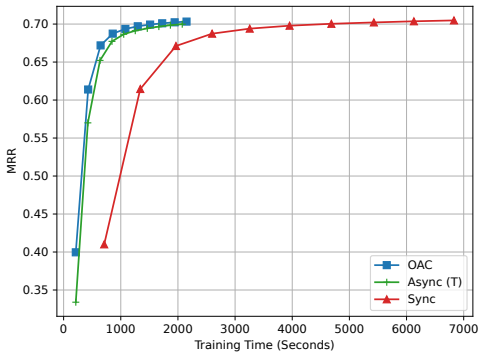
3.2.3.1 Experimental Setup

We evaluate OAC on the task of link prediction and use learned feature vectors for all nodes in the input graph (see Section 2.1). We run experiments on the Freebase86m knowledge graph (Google, 2018). This graph has roughly 86 million nodes, and we use a feature vector of size 50 for each node (called the embedding dimension). This leads to 17GB of storage overhead for these features (each parameter is four bytes). We store features in CPU memory. We measure link prediction accuracy using the metric Mean Reciprocal Rank (MRR) (Mohoney et al., 2021). This quantity tries to capture how well learned feature vectors can be used to recover the edges of the graph and takes values between zero and one—the higher the better. We run experiments on two machines. One with 80 CPU cores and one with 20 CPU cores. Both machines have one NVIDIA Tesla V100 GPU.

We compare OAC with synchronous and asynchronous training in terms of *convergence*, i.e., the accuracy with respect to the number of training iterations (batches that have passed through the GPU computation step), and *time-to-accuracy*, i.e., how long (wall clock training time) does it take for a model to reach a given accuracy. Time-to-accuracy is the primary practical metric of interest. For example, if asynchronous training increases throughput by a factor of two, it can process twice as many batches as



(a) Setup 1: We train a one-layer GraphSage (Hamilton et al., 2017) plus DistMult (Yang et al., 2014) GNN with embedding dimension 50 using 20 neighbors per node and a batch size of 50k examples. Training was performed on a machine with 80 CPU cores and one NVIDIA Tesla V100. Each batch accesses roughly 2 million feature vectors.



(b) Setup 2: We train a zero-layer DistMult GNN (feature vectors are passed directly into the DistMult score function) with embedding dimension 50 using a batch size of 500k examples. Training was performed on a machine with 20 CPU cores and one NVIDIA Tesla V100. Each batch access roughly 1 million feature vectors.

Figure 3.3: Time-to-accuracy for OAC versus asynchronous and synchronous approaches in two different settings. In both cases, OAC provides the best time-to-accuracy for mixed CPU-GPU GNN mini-batch training. For these experiments, accuracy is measured using MRR.

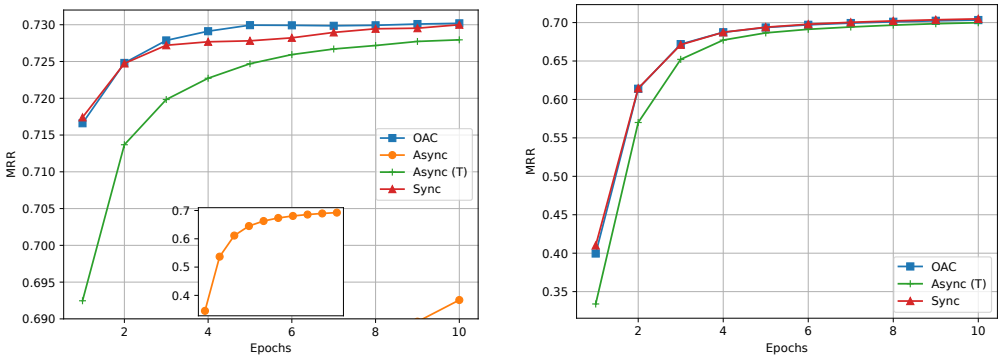
synchronous training per unit time. However, if it also requires twice as many batches to converge to the desired accuracy, then the overall training time of the two methods will be the same.

3.2.3.2 Time-To-Accuracy

We plot model accuracy versus wall clock training time for OAC, asynchronous, and synchronous training using two different setups in Figure 3.3. Using each method, we trained for ten epochs and measure the MRR (accuracy) after each one. Recall that one epoch refers to one full pass over the training examples and results in a fixed number of batches. Thus all methods see the same total number of batches pass through the GPU.

In Figure 3.3a, asynchronous training is shown by the orange line (labeled Async) and is barely visible in the bottom middle of the plot. The reason for this is that for asynchronous training we maximize the number of parallel threads and queue sizes to maximize pipeline throughput (see Section 3.1). Blindly running asynchronous training, however, causes a severe degradation in model accuracy. Thus, we also include a tuned version of asynchronous training (Async (T)), where we manually tune the number of threads and queue sizes for each pipeline stage in order to find the best configuration. This tuning allows us to regain much of the lost accuracy of asynchronous training with very little throughput loss. Both Async and Async (T) finish the ten epochs roughly twice as fast as synchronous training (roughly 6500s versus 12500s), but synchronous training ends with a higher MRR (accuracy). OAC outperforms all other methods and does not require the manual tuning of Async (T). For OAC, we simply maximize the number of threads and queue sizes in the pipeline and use the method described in Section 3.2.2. Figure 3.3a shows that OAC achieves the fastest time-to-accuracy. This is because it maintains the throughput of asynchronous training, also finishing the ten epochs in 6500s, while matching the convergence of synchronous training (which we highlight below).

Figure 3.3b shows a second experiment using a different model and hardware configuration. In this case, tuned asynchronous training manages to achieve similar accuracy to synchronous training while learning more than three times faster. We do not show the default asynchronous training as it is roughly equivalent to the tuned setup. The reason asynchronous training is able to perform well in this case is twofold: First, batches access roughly half the number of features of the setup in Figure 3.3a. Second, the machine has one quarter of the CPU resources, limiting the number of batches it can process concurrently. Both of these differences result in fewer concurrent batches which overlap—recall that the fewer conflicts there are, the more likely asynchronous training is going to perform well. That said, however,



(a) Setup 1 (Figure 3.3a) (b) Setup 2 (Figure 3.3b)

Figure 3.4: Convergence of OAC versus asynchronous and synchronous approaches for the experimental settings described in Figure 3.3. In both cases, OAC converges at the same rate as synchronous training; this is not true for asynchronous training, particularly in Setup 1.

in Figure 3.3b, OAC also trains over three times faster than synchronous training and is guaranteed to reach the same model quality.

3.2.3.3 Convergence

To highlight that OAC converges at the same rate as synchronous training, we plot the accuracy from the experiments in Section 3.2.3.2/Figure 3.3 again in Figure 3.4, but this time versus epoch number rather than versus training time. Recall that for each epoch, all methods see the same number of batches pass through the GPU, thus Figure 3.4 shows the accuracy with respect to the number of processed batches.

Figure 3.4 shows that OAC matches the convergence of synchronous training. In contrast, in Figure 3.4a, asynchronous training converges so poorly, that it requires its own scaling of the vertical axis. Tuned asynchronous training converges faster, reaching 0.6925, 0.725, and 0.7275 MRR after 1, 5, and 10 epochs respectively. Synchronous training and OAC both achieve 0.7175 MRR after one epoch and 0.73 MRR after 10 epochs. In this case, OAC converges slightly faster than synchronous training during

the middle epochs. This is because OAC is guaranteed to be equivalent to *some* synchronous order, not necessarily the exact synchronous order that was used for training by the synchronous experiment in Figure 3.4a.

The convergence of each method for the training setup of Figure 3.3b is shown in Figure 3.4b. As described above, in this case asynchronous training actually performs quite well—it converges at nearly the same rate as synchronous training. Again, however, OAC is guaranteed to converge at the rate of some synchronous order; in Figure 3.4b, the OAC and synchronous lines nearly perfectly overlap.

3.3 DENSE: Efficient Multi-hop Neighborhood Sampling

Given a pipelined architecture plus OAC for mixed CPU-GPU GNN training, we now focus on minimizing the overhead of multi-hop neighborhood sampling to improve the throughput of mini batch preparation, and thus training overall, in this architecture. To do so, we introduce the Delta Encoding of Neighborhood SamplEs data structure or DENSE for short. DENSE's key idea is to minimize single-hop sampling redundancy while constructing multi-hop neighborhoods by caching and reusing previous samples. Dense also allows for efficient GNN forward passes using dense kernels for linear algebra operations. We discuss each in turn.

3.3.1 Neighborhood Sampling With DENSE

Recall from Section 2.1 that to compute the representation of a set of target nodes after k GNN layers, we need to sample their k-hop neighborhood. For example, in Figure 2.1, we showed how to compute the representation of target node A after two layers. To compute h_A^2 we used h_C^1 and h_D^1 by sampling C and D from A's one-hop neighborhood. Computing h_A^2, however, uses h_A^1 which requires sampling the one-hop neighborhood of node A again. Performing these two one-hop sampling operations independently

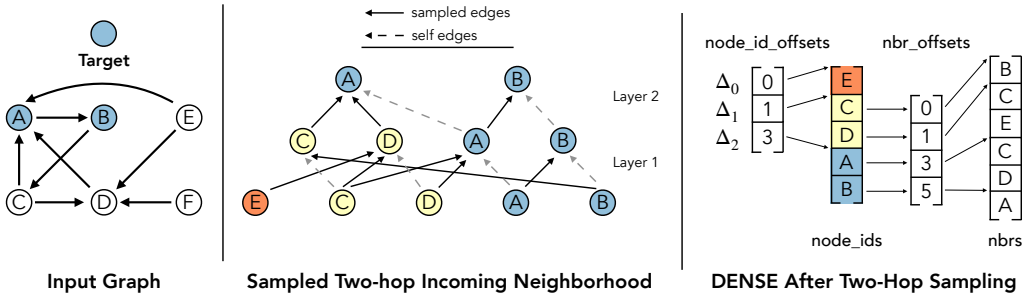


Figure 3.5: Example two-hop neighborhood sample for target nodes $\{A, B\}$ and the corresponding DENSE data structure. For one-hop neighbors, we sample at most two nodes from incoming edges.

requires traversing the one-hop neighborhood of node A multiple times, introducing redundant computation. We identify that such redundant sampling computations contribute to the overheads of multi-hop sampling in existing systems (see Section 2.2).

Using DENSE, we sample one-hop neighbors for each node in the k-hop neighborhood *only once*. We take advantage of the fact that k-hop sampling is recursive: we can construct a sample of the $(i+1)$-hop neighborhood for a set of target nodes by sampling the one-hop neighbors of all nodes N in the i-hop neighborhood. However, we may have previously sampled one-hop neighbors for some nodes $P \subseteq N$. We use this property to construct the $(i+1)$-hop neighborhood by sampling one-hop neighbors only for the nodes $N \setminus P$. We define these nodes to be Δ_{k-i}. The rest of the $(i+1)$-hop neighborhood is completed by reusing the previous one-hop samples for nodes in P. To track the nodes for which one-hop samples are required at each iteration, we uses the DENSE data structure. DENSE is constructed by stacking the $k+1$ Δ's and the corresponding one-hop neighbors. We show an example for the two-hop neighborhood of the target nodes $\{A, B\}$ in Figure 3.5. DENSE consists of three Δ's: $\Delta_2 = \{A, B\}$, $\Delta_1 = \{C, D\}$, $\Delta_0 = \{E\}$. Notice that unlike in Figure 2.1, here, the sampled one-hop neighbors of A, $\{C, D\}$ are used in both GNN layers.

Algorithm 1 DENSE Multi-hop Neighborhood Sampling

Require: target_nodes: unique node IDs for k-hop sampling;
fanouts: max # of neighbors to sample per hop
1: node_id_offsets = [0]; node_ids = target_nodes
2: nbr_offsets = []; nbrs = []; Δ_k = target_nodes
3: **for** $i \in [k \dots 1]$ **do**
4: Δ_i_nbrs, Δ_i_offsets = `oneHopSample`(Δ_i, fanouts[i])
5: nbr_offsets = `cat`(Δ_i_offsets, nbr_offsets + `len`(Δ_i_nbrs))
6: nbrs = `cat`(Δ_i_nbrs, nbrs)
7: Δ_{i-1} = `computeNextDelta`(Δ_i_nbrs, node_ids)
8: node_id_offsets = `cat`([0], node_id_offsets + `len`(Δ_{i-1}))
9: node_ids = `cat`(Δ_{i-1}, node_ids)
10: **return** DENSE(node_id_offsets, node_ids, nbr_offsets, nbrs)

DENSE is built using four arrays: 1) `node_ids` contains all graph node IDs involved in the sample, 2) `nbrs` contains the sampled one-hop neighbors for nodes in `node_ids`, 3) `nbr_offsets` identifies where the neighbors for each node ID start in `nbrs`, and 4) `node_id_offsets` identifies groups of node IDs in `node_ids` corresponding to each Δ. Given a set of target node IDs and a k-layer GNN, we sample the k-hop neighborhood for each target node and creates the four arrays in DENSE according to Algorithm 1. We define the target nodes to be Δ_k and initialize each array in DENSE as shown (Line 1-2). Sampling then proceeds for k rounds (Line 3). Each iteration $i \in [k \dots 1]$ starts by sampling the one-hop neighbors for the nodes in Δ_i (Line 4). Given the set of nodes Δ_i and a maximum number of neighbors to sample per node f (the layer *fanout*), one-hop sampling returns up to f neighbors for each node $j \in \Delta_i$ as a list Δ_i`_nbrs` with the neighbors for each node j sequential starting at the offset given by Δ_i`_offsets`. When a node has more than f neighbors, only f will be sampled, but if a node has less than f neighbors, all neighbors will be returned.

We perform one-hop sampling using CPU multi-threading. We store two sorted versions of the in-memory edge list: 1) sorted in ascending order of source node ID, and 2) sorted in ascending order of destination node ID. We create an array that, for each node ID in memory, stores the offsets

corresponding to its outgoing and incoming edges in each of the two edge lists. Given these structures, we can sample incoming and outgoing edges for any set of nodes in parallel using all available CPU threads.

Given the one-hop neighbors for Δ_i, the next step in Algorithm 1 is to stack these one-hop samples on the existing arrays in DENSE (Line 5-6). At this point, we computes Δ_{i-1} as the unique nodes in Δ_i`_nbrs` that do not appear in the DENSE `node_ids` array (Line 7). As with one-hop sampling, we compute Δ_{i-1} using multi-threading on the CPU. The nodes in Δ_{i-1} are then added to DENSE (Line 8, 9). Multi-hop sampling completes after adding Δ_0 to DENSE (neighbors are not needed for Δ_0).

While one-hop sample reuse in DENSE allows us to minimize redundant computation in multi-hop sampling, it also leads to k-hop neighborhoods with two noteworthy differences compared to existing sampling algorithms. First, sample reuse implies that the resulting neighborhood fanouts (max number of neighbors per node per hop) are not guaranteed to match the requested fanouts (input to Algorithm 1). Instead, the fanout for a node j at each hop is equal to the fanout requested for the first hop which required neighbors of j. As such, in the common case where GNN fanouts are requested according to a decreasing sequence away from the target nodes, DENSE provides at least as many neighbors as requested for a node j at each layer. Second, sample reuse in DENSE reduces randomness in multi-hop neighborhoods compared to existing algorithms by preventing different subsets of the one-hop neighbors for a given node from existing in the same multi-hop neighborhood. We study the implication of this reduced randomness on the accuracy of GNN training in Section 3.4 but find that training with DENSE can reach comparable accuracy to existing systems.

3.3.2 Forward Pass Computation With DENSE

After sampling is completed, we transfer DENSE to the GPU so it can be used to compute the GNN forward pass. We also transfer an array H^0 containing the feature vectors for each node ID in the DENSE `node_ids`

array. On the GPU we create and add a fifth array to DENSE called `repr_map` that stores the index in H^0 containing the feature vector for each node ID in the DENSE `nbrs` array.

To complete a forward pass over a k-layer GNN, we iterate over each layer $i \in [1 \dots k]$ and perform the next two steps:

(Step 1) We compute the output H^i of layer i. Given DENSE and the representations in H^{i-1}, the output of layer i is the representation for all nodes in the DENSE `node_ids` array which occur after `node_id_offsets`[1]. For example, in Figure 3.5, the output of the first GNN layer is h^1 for the nodes $\{C, D, A, B\}$. The output representations are computed according to the i^{th} GNN layer's aggregation function.

(Step 2) We update DENSE on the GPU as shown in Algorithm 2: We remove nodes and their one-hop neighbors that are no longer needed for subsequent layers. This step ensures that the output nodes from this iteration, which will be used as input to compute the representations of the nodes in Step 1 for the next iteration, correspond to all nodes in the DENSE `node_ids` array. This property allows us to use the same implementation across GNN layers. In Figure 3.5, node E and the neighbors of $\{C, D\}$ are not needed after layer one and can be removed from DENSE.

DENSE allows use to use optimized dense GPU kernels for the linear algebra operations in each GNN layer. For example, in Algorithm 3, we show how to use DENSE and H^{k-1} to compute the output of the k^{th} GNN layer H^k for the GNN aggregation: $h_i^{(l+1)} = h_i^{(l)} + \sum_{j \in Nbrs_i} h_j^{(l)}$. We first use the `repr_map` array in DENSE to select the node representations for all neighbors in the `nbrs` array (Line 1). Neighborhood aggregation can then be performed using a dense segment sum that is well suited for parallelization on GPU hardware (Line 2): Recall that the one-hop neighbors for the nodes in DENSE are stored sequentially and separated by the `nbr_offsets` array. After selecting the neighbor representations in Line 1, the representations for the one-hop neighbors of each node will also be sequential in GPU

Algorithm 2 On GPU DENSE Update After Layer i

Require: node_id_offsets, node_ids, nbr_offsets, nbrs, repr_map
1: Δ_{i-1} = node_ids[: node_id_offsets[1]]
2: Δ_i = node_ids[node_id_offsets[1] : node_id_offsets[2]]
3: Δ_i_nbrs = nbrs[: nbr_offsets[`len`(Δ_i)]]
4: nbrs = nbrs[`len`(Δ_i_nbrs):]
5: repr_map = repr_map[`len`(Δ_i_nbrs):] - `len`(Δ_{i-1})
6: nbr_offsets = nbr_offsets[`len`(Δ_i):] - `len`(Δ_i_nbrs)
7: node_ids = node_ids[node_id_offsets[1]:]
8: node_id_offsets = node_id_offsets[1:] - `len`(Δ_{i-1})
9: **return** node_id_offsets, node_ids, nbr_offsets, nbrs, repr_map

Algorithm 3 k^{th} GNN Layer Additive Aggregation

Require: DENSE; H^{k-1}: layer input vector representations
1: nbr_repr = H^{k-1}.`index_select`(DENSE.repr_map)
2: nbr_aggr = `segment_sum`(nbr_repr, DENSE.nbr_offsets)
3: self_repr = H^{k-1}[DENSE.node_id_offsets[1] :]
4: H^k = nbr_aggr + self_repr
5: **return** H^k

memory. Thus, neighborhood aggregation for each node consists of adding a set of sequential vectors and all nodes can aggregate in parallel. The last step to compute the layer output is to combine the aggregated neighbor representations for each node with their own representations (Lines 3-4).

3.4 Results: Mixed CPU-GPU Training in MariusGNN

We implement and evaluate pipelined training and DENSE in MariusGNN. We evaluate MariusGNN on four large-scale graphs (see Table 1.1 for dataset statistics), including two from the OGB large-scale challenge (Hu et al., 2021), and compare against the popular state-of-the-art GNN systems DGL and PyG. Our experiments show that:

1. For mixed CPU-GPU mini-batch training, MariusGNN reaches the same level of accuracy 2-7× faster and cheaper than DGL and PyG

Table 3.1: AWS cloud GPU instances used for experiments.

AWS Machine	($/hr)	GPUs	CPUs	CPU Mem (GB)
P3.2xLarge	3.06	1	8	61
P3.8xLarge	12.24	4	32	244
P3.16xLarge	24.48	8	64	488

across all datasets for both node classification and link prediction.

2. DENSE allows MariusGNN to reduce mini batch multi-hop neighborhood sampling and compute times by up to 14× and 8× respectively compared to DGL and PyG.

3.4.1 Experimental Setup

We discuss the setup used throughout the experiments.

Baselines We compare end-to-end GNN training over large-scale graphs in MariusGNN against DGL 0.7 and PyG 2.0.3 (late 2021 releases). In addition to end-to-end performance, we evaluate the effect of DENSE on training by measuring the time for multi-hop sampling and GNN forward and backward pass computation in these systems and MariusGNN (Section 3.4.3). Furthermore, we compare the multi-hop sampling time in MariusGNN to the state-of-the-art sampling implementation in NextDoor (Jangda et al., 2021) (Section 3.4.3). We do not use NextDoor for end-to-end GNN training as their open-source release supports only limited GNN computation over small graphs which fit in GPU memory.

Hardware Setup We evaluate all systems using AWS P3 instances (Table 3.1). We use the cheapest P3 instance which has enough CPU memory for training with the full graph in memory (for the graphs in these experiments, that is either a P3.8xLarge or P3.16xLarge). We allow baseline systems to use the maximum number of GPUs they support and available in the instance. *MariusGNN uses only one GPU for all experiments.*

Node Classification: Datasets, Models, and Metrics We use the two largest OGB node classification graphs: Mag240M and Papers100M (Papers) (Hu et al., 2020, 2021). For Mag240M we use only the paper nodes and citation edges, denoted as Mag240M-Cites (Mag). Based on the graph memory overheads, Papers100M and Mag240M-Cites require a P3.8xLarge and P3.16xLarge respectively for in-memory training. We train a three-layer GraphSage (GS) (Hamilton et al., 2017) GNN on both datasets, a common choice for these graphs (Kaler et al., 2022; Zheng et al., 2022). We use 30, 20, and 10 neighbors per layer (ordered away from the target nodes) and sample from both incoming and outgoing edges. We report multi-class classification accuracy averaged over three runs and train for ten epochs. We find that PyG multi-GPU training runs out of CPU memory for Mag240M-Cites, hence, for PyG on this dataset we switch to single-GPU training.

Link Prediction: Datasets, Models, and Metrics For link prediction, we use the largest OGB link prediction graph—WikiKG90Mv2 (Wiki) (Hu et al., 2021). As a second graph for large-scale link prediction, we use Freebase86M (FB) (Zheng et al., 2020b). Both datasets fit in CPU memory on an AWS P3.8xLarge machine. We train a GraphSage GNN on both datasets, and the more computationally expensive GAT (Veličković et al., 2018) on Freebase86M (the smaller dataset). Both GNNs use a single layer. We use 20 neighbors sampled from incoming and outgoing edges for GraphSage and 10 incoming neighbors for GAT. We evaluate the accuracy of link prediction models using the commonly reported MRR metric (Mohoney et al., 2021; Zheng et al., 2020b; Lerer et al., 2019) using the DistMult (Yang et al., 2014) score function and train all systems for a fixed number of epochs: We use five epochs on Freebase86M and ten on WikiKG90Mv2. We report the MRR for a single run due to cost considerations, but report runtime averaged across all training epochs.

Both DGL and PyG provide limited support for training link prediction at scale: PyG does not provide a negative sampler. We implemented negative

sampling in PyG based on the negative sampling used in MariusGNN. DGL provides a negative sampler but the implementation limits the amount of negative samples that can be used to train in a reasonable amount of time. As such, for DGL we use five times fewer negative samples per training edge compared to MariusGNN to prevent GPU out-of-memory issues. We find that neither baseline supports multi-GPU training for this task: the data loader implementation for link prediction in PyG supports only a single GPU and DGL's multi-GPU training ran out of CPU memory on the AWS P3.8xLarge machines we used for these experiments.

Hyperparameters We use the same values for hyperparameters which define the GNN model and training process across systems. We choose these values to be those used by OGB or prior works (Hu et al., 2020, 2021; Hamilton et al., 2017) to achieve high accuracy on each dataset. However, to prevent GPU out-of-memory, for PyG on Mag240M-Cites, we use a smaller batch size (half) than DGL and MariusGNN. While we make sure to request the same number of neighbors per layer for each system, differences in mini batches are expected due to the use of different sampling algorithms. For throughput parameters specific to each system and independent of the computation (e.g., the number of data loader threads), we tune each system and use the best configuration.

3.4.2 End-to-End System Comparisons

We discuss end-to-end training results for MariusGNN, DGL, and PyG on node classification and link prediction tasks.

Results are reported in Tables 3.2-3.4. For each experiment we train all systems for the same fixed number of epochs and measure 1) the per-epoch runtime, 2) model accuracy or MRR, and 3) the monetary cost per epoch based on AWS pricing. Next, we highlight key takeaways among all end-to-end results before focusing on each setting (Table) in more detail.

Key Takeaway MariusGNN provides the fastest *and* cheapest training

option to comparable accuracy for all dataset and model combinations on both learning tasks. Differences in training time and cost can be orders of magnitude: Baseline systems can take six days and $1720 dollars for training (see training on Wiki in Table 3.3), yet MariusGNN needs only eight hours or $94 dollars for the same dataset.

Node Classification We focus on end-to-end results for node classification in more detail (Table 3.2). With graph data stored in main memory, MariusGNN with one GPU trains 4× and 3× faster than DGL (the fastest baseline) using four and eight GPUs on Papers100M and Mag240M-Cites respectively. All three systems reach similar accuracy on both datasets. There are two reasons for the reduced runtime of MariusGNN in this setting. First, the DENSE data structure allows for faster CPU-based mini-batch sampling and GPU-based GNN computation in MariusGNN compared to baseline systems (evaluated in Section 3.4.3). Second, while both DGL and PyG support multi-GPU training, they both underutilize the additional compute resources: DGL and PyG four-GPU training on Papers100M are only 1.4× and 1.1× faster than their single-GPU performance respectively, and DGL eight-GPU training on Mag240M-Cites is only 2.2× faster than with one-GPU (single-GPU baselines not reported in Table 3.2).

While all systems reach comparable accuracy for training with the full graph in memory (within 1%), MariusGNN accuracy is 0.55% and 0.3% lower than the closest baseline (PyG) on Papers100M and Mag240M-Cites respectively. We hypothesize that this is because DENSE reuses previously sampled one-hop neighbors across layers when constructing multi-layer GNN dataflow graphs. Sample reuse leads to fewer opportunities for one-hop neighborhood randomness resulting in fewer unique nodes in the sampled multi-hop neighborhood for each mini batch (quantified in Table 3.5). Although this leads to an accuracy reduction for multi-layer GNNs in MariusGNN, sample reuse is isolated to a single mini batch. Over the course of training, the one-hop neighbors of each node are still randomized

Table 3.2: MariusGNN, DGL, and PyG for node classification on large-scale graphs using a GraphSage GNN. Using a single GPU, MariusGNN can reach the same level of accuracy as multi-GPU baselines 3-4× faster and cheaper.

	Epoch (min.)		Accuracy		Cost ($/epoch)	
Dataset	Papers	Mag	Papers	Mag	Papers	Mag
MariusGNN	**0.77**	**2.57**	66.38	63.17	**0.16**	**1.05**
DGL	3.07	7.83	66.98	63.73	0.63	3.19
PyG	8.01	19	66.93	63.47	1.63	7.75

Table 3.3: MariusGNN, DGL, and PyG for link prediction on large-scale graphs. All systems use a GraphSage GNN and one GPU. MariusGNN reaches comparable accuracy to baselines 6-7× faster and cheaper. (OOT: out of time)

	Epoch (min.)		MRR		Cost ($/epoch)	
Dataset	FB	Wiki	FB	Wiki	FB	Wiki
MariusGNN	**17.5**	**46.6**	.7285	.4655	**3.57**	**9.38**
DGL	152	844	.7091	OOT	31.0	172
PyG	108	312	.7267	.4683	22.0	63.6

Table 3.4: Comparison of GraphSage (GS) and GAT GNN training in MariusGNN, DGL, and PyG for link prediction on Freebase86M. Baselines bottlenecked by CPU-based mini batch preparation result in similar training time and cost on GraphSage and the more computationally expensive GAT.

	Epoch (min.)		MRR		Cost ($/epoch)	
Model	GS	GAT	GS	GAT	GS	GAT
MariusGNN	**17.5**	**52.6**	.7285	.7331	**3.57**	**10.7**
DGL	152	151	.7091	.6516	31.0	30.8
PyG	108	107	.7267	.7252	22.0	21.8

across batches, allowing MariusGNN to achieve comparable accuracy to baselines while training with DENSE.

Link Prediction We now focus on the task of link prediction. End-to-end results for all systems on Freebase86M and WikiKG90Mv2 are reported in Table 3.3. MariusGNN in-memory training is 6× and 7× faster than the best

Table 3.5: Comparison of the time required for mini-batch neighborhood sampling, GPU-based GNN computation, and the number of nodes/edges sampled per mini batch in MariusGNN (which uses DENSE), DGL, and PyG for GraphSage GNNs of varying depth on OGBN-Papers100M (Hu et al., 2020). Using DENSE, in MariusGNN sampling is 14× and GPU computation is 8× faster for a four-layer GNN. These speedups occur in part because DENSE allows MariusGNN to sample fewer nodes/edges to construct mini batches. (OOM: out of memory)

	CPU Sampling Time (ms)					GPU Computation Time (ms)				
#Layers	1	2	3	4	5	1	2	3	4	5
MariusGNN	**1.4**	**18**	**103**	**401**	**1.8k**	4	**6.1**	**21**	**153**	OOM
DGL	5.7	28	376	5.4k	49k	4.7	29	215	1231	OOM
PyG	2.2	59	1227	19k	96k	**3.2**	13	168	OOM	OOM

	Number of Nodes/Edges Sampled Per Mini Batch				
	1	2	3	4	5
MariusGNN	12k/13k	136k/181k	1M/2M	6M/17M	23M/91M
DGL	13k/20k	182k/278k	2M/4M	9M/37M	33M/222M
PyG	13k/20k	178k/258k	2M/4M	9M/32M	31M/174M

baseline on the two datasets respectively. While PyG and MariusGNN reach comparable model quality, DGL is lower due to its use of fewer negative samples. On WikiKG90Mv2, DGL does not complete the ten training epochs within two days. To compare system performance for different models, we report results for GraphSage and GAT GNNs on Freebase86M in Table 3.4. Interestingly, DGL and PyG exhibit similar runtimes for GraphSage and the more computationally expensive GAT. This result supports the fact that baseline systems are bottlenecked by CPU-based sampling operations rather than GPU-based model computation.

3.4.3 Effect of DENSE on Training

We have shown that end-to-end training in MariusGNN is faster than existing systems. A key reason for this result is the efficient mini-batch

sampling and forward pass computation using the DENSE data structure. In this section, we report the effect of DENSE on training. Recall that training consists of two phases: 1) CPU-based mini batch construction via neighborhood sampling and 2) GPU-based GNN forward and backward pass computation. While DENSE is co-designed for both efficient sampling and GNN computation, we seek to understand the effect of DENSE on each individual training phase. As a result, we measure the average time per mini batch for 1) CPU neighborhood sampling and 2) GPU training in MariusGNN (using DENSE) and compare against the corresponding methods used in DGL and PyG. For these experiments, we use a GraphSage GNN on the OGBN-Papers100M dataset and vary the number of GNN layers from one to five. For each layer, we request a max of 10 incoming and 10 outgoing neighbors per node from each system. We use the same hyperparameters for MariusGNN, DGL, and PyG and train all systems with the graph in main memory using one GPU.

We report the average CPU-based neighborhood sampling time for each system in Table 3.5. DENSE allows MariusGNN to sample multi-hop neighborhoods faster than baseline systems for all configurations. For three, four, and five layers, MariusGNN is 3.7×, 14×, and 26× faster than the best baseline. GNN training on the GPU in MariusGNN is also faster than DGL and PyG (Table 3.5). DENSE leads to 8× faster computation compared to the best baseline for three- and four-layer GNNs. We find that for five-layer GNNs, mini batches become too large and cause all three systems to run out of memory on the AWS NVIDIA V100 GPUs with 16GB of memory (but could be used on new GPUs with 80GB).

We investigate to what extent the sampling and computation improvements in MariusGNN can be attributed to the reuse of neighborhood samples in DENSE compared to implementation co-design choices, i.e., parallel sampling on the CPU and dense kernels on the GPU. In Table 3.5, we report the average number of unique nodes and edges sampled per mini batch for

Table 3.6: Comparison of the time required for GPU-based mini-batch neighborhood sampling in MariusGNN (using DENSE) and NextDoor for GraphSage GNNs of varying depth on LiveJournal. Sample reuse in DENSE leads to better scaling with respect to the number of GNN layers, allowing MariusGNN to outperform optimized sampling implementations.

	GPU Sampling Time (ms)				
#Layers	1	2	3	4	5
M-GNN	1	2.5	9.6	25	32
NextDoor	0.1	0.5	6.5	135	OOM

each system. DENSE allows for mini batch construction using fewer samples than baselines. For example, constructing a three-hop neighborhood in DENSE requires sampling half as many nodes and edges compared to DGL and PyG (for the same number of target nodes). While sample reuse in DENSE is evident, mini batch sizes in MariusGNN, DGL, and PyG are all the same order of magnitude, yet DENSE sampling and computation improvements are more significant (e.g., 14× and 8×). This result validates the co-design of DENSE: parallel CPU sampling algorithms and the use of dense GPU kernels, together with one-hop sample reuse, lead to the improved throughput in MariusGNN.

Comparison Against Accelerated Sampling Kernels To further evaluate the benefit of DENSE on GNN training, we compare multi-hop sampling in MariusGNN to the state-of-the-art accelerated sampling implementation of NextDoor (Jangda et al., 2021). NextDoor uses GPUs to reduce sampling times and employs optimized GPU kernels for parallelization, load balancing, and caching. These kernels allow NextDoor to outperform multi-hop sampling implementations in existing systems, but their open-source release requires that graphs fit in GPU memory. While in MariusGNN we focus primarily on CPU-based multi-hop sampling for mixed CPU-GPU training to scale to large graphs (as discussed throughout this section), MariusGNN also includes support for GPU-based multi-hop

sampling with DENSE for end-to-end training on smaller graphs without CPU involvement. Unlike our CPU-based sampling implementation which uses optimized parallel algorithms to construct DENSE, our GPU-based sampling implementation builds DENSE using only default PyTorch functions. We compare GPU-based sampling using DENSE in MariusGNN to the optimized sampling kernels in NextDoor by measuring the average multi-hop sampling time per mini batch for a GraphSage GNN of varying depth on the LiveJournal dataset (which fits in GPU memory with 4.8M nodes and 69M edges) (Leskovec and Krevl, 2014). For each layer, we sample 20 outgoing neighbors per system.

GPU-based sampling times for MariusGNN and NextDoor are shown in Table 3.6. The optimized sampling kernels in NextDoor have lower overhead and better parallelization for one-hop sampling compared to the default PyTorch functions used by MariusGNN. These kernels lead to faster sampling for one- and two-layer GNNs. For deeper GNNs however, MariusGNN is comparable to or faster than NextDoor. This is because as the number GNN layers increases DENSE has more opportunities to minimize redundant one-hop sampling compared to NextDoor by reusing previous samples across layers. Table 3.6 shows that redundant sampling can bottleneck even the most optimized sampling implementations. DENSE avoids this bottleneck and can scale to five-layer GNNs with little sampling overhead.

3.5 Summary

In this Chapter, we introduced a pipelined architecture for asynchronous mixed CPU-GPU training. We then presented OAC, a protocol to ensure that asynchronous training reaches the same model accuracy as synchronous training in the presence of learnable featured vectors stored in CPU memory. We also introduced the DENSE data structure to minimize redundant computations and data access during multi-hop neighborhood sampling. Finally, we showed experimentally that MariusGNN, our system for GNN

training which implements the above contributions, exhibits the fastest and cheapest mixed CPU-GPU mini-batch training when compared to popular state-of-the-art systems for both node classification and link prediction tasks.

4 SCALABLE MIN-EDGE-CUT GRAPH PARTITIONING

In this Chapter, we introduce GREM (Greedy plus Refinement for Edge-cut Minimization), a novel algorithm that enables efficient min-edge-cut graph partitioning over massive graphs on a commodity machine. We describe the optimization objective and algorithm, then analyze it theoretically. Finally, we present experimental results comparing GREM to existing state-of-the-art min-edge-cut partitioning algorithms.

4.1 GREM: Greedy plus Refinement for Edge-Cut Minimization

Optimization Objective Given a graph $G = (V, E)$, we assume as input an edge list (E) stored on disk in a random order. Our goal is to partition the nodes V into a set of p partitions, each of size $\lceil V/p \rceil$ (i.e., a balanced partitioning), according to an algorithm that 1) minimizes the number of cross-partition edges and 2) can scale to massive graphs given a fixed amount of CPU memory (but unlimited disk space) (the edge list for large graphs may not fit in memory on a single, or even multiple machines (e.g., Hyperlink-2012's 128B edges require 2TB (Meusel et al., 2014)).

Existing Solutions The above problem (balanced min-edge-cut partitioning) is NP-Hard, even for the case of just two partitions. Furthermore, for $p > 2$, no finite approximation algorithm exists unless P=NP (Andreev and Räcke, 2004). As such, existing algorithms for min-edge-cut partitioning rely on heuristics: As highlighted in the introduction and Section 2.3, traditional offline partitioning algorithms (e.g., METIS) operate over whole graph and minimize edge cuts through iterative partition refinement. These methods have been shown to approach the optimal partitioning in practice, but are unable to scale efficiently to large graphs due to memory and runtime require-

ments. To reduce the computational requirements of partitioning, streaming greedy algorithms iterate over the graph and assign vertices greedily to partitions based on prior partition assignments and a subset of graph edges currently buffered in the streaming process (Stanton, 2014; Alistarh et al., 2015; Abbas et al., 2018; Patwary et al., 2019; Stanton and Kliot, 2012; Faraj and Schulz, 2022; Petroni et al., 2015; Jain et al., 1998; Tsourakakis et al., 2014). While these algorithms have better scalability, they often lead to partitionings with lower quality (more edge cuts) than offline methods due to their use of fixed greedy decisions (e.g., 4× more edge cuts than METIS; experimental results provided in Section 4.3).

Key Idea To combine the advantages of offline and streaming methods, GREM employs a streaming greedy approach, but with one key addition: Rather than freezing the partition assignment for a node after an initial greedy selection, GREM leverages running statistics accumulated during streaming to continuously reevaluate prior assignments and refine the result. Inspired by offline algorithms, this refinement is critical to minimizing edge cuts in the resulting partitioning.

Detailed Algorithm GREM partitions an input graph into $p = 2$ partitions as described in Algorithm 4; we focus on $p = 2$ because GREM returns a partitioning for $p > 2$ by first partitioning the graph into two parts, and then recursively re-partitioning each part into two new parts as needed. We identify the two partitions by index *zero* and *one*. Each node starts unassigned (index *minus one*) and each partition starts with size zero (Line 1-2). We also initialize two numerical values for each node (`nbr_counts`, Line 3); the purpose of these values is to provide a running estimate of the number of neighbors each node has in each partition.

GREM then proceeds by iterating over the edge list in chunks (Line 4). For each chunk, the edges are loaded into memory (`c_edges`) and the set of unique nodes (`c_nodes`) contained in those edges is computed (Lines 5-6). For the first chunk (Line 7), as there are no existing partition assignments

Algorithm 4 GREM Bipartite Graph Partitioning

Require: num_nodes: number of nodes in the graph; edges: graph edges; c_size: chunk size for GREM; `seed_algo`: seed partition algorithm for GREM; P: max partition size (in nodes)

```
 1: parts = minus_ones(num_nodes)
 2: part_sizes = zeros(2)
 3: nbr_counts = zeros(num_nodes, 2)
 4: for i = 0 to ceil(len(edges)/c_size) − 1 do
 5:    c_edges = read(edges[i ∗ c_size : (i + 1) ∗ c_size])
 6:    c_nodes = unique_nodes_in_edges(c_edges)
 7:    if i == 0 then
 8:       parts[c_nodes] = seed_algo(c_edges, num=2)
 9:       part_sizes = [num_zeros(parts), num_ones(parts)]
10:       nbrs_0, nbrs_1 = cnt_nbrs(c_nodes, c_edges, parts)
11:       nbr_counts[c_nodes] = [nbrs_0, nbrs_1]
12:    else
13:       for n ∈ c_nodes do
14:          old_part = parts[n]
15:          nbrs_0, nbrs_1 = cnt_nbrs(n, c_edges, parts)
16:          if old_part ≠ −1 then
17:             nbrs_0 = (nbr_counts[n, 0] + nbrs_0) / 2
18:             nbrs_1 = (nbr_counts[n, 1] + nbrs_1) / 2
19:          parts[n] = assign(nbrs_0, nbrs_1, part_sizes, P)
20:          part_sizes = fix_sizes(parts[n], old_part, part_sizes)
21:          nbr_counts[n] = [nbrs_0, nbrs_1]
```

that can be used to make greedy decisions, we use a *seed* partitioning algorithm on the in-memory edges (e.g., METIS) to assign all nodes in memory to one of the two partitions (Line 8). The partition sizes and the estimated number of neighbors per node in each partition (calculated based on the in-memory edges and existing partitioning; Algorithm 5 - `cnt_nbrs`) are then updated (Lines 9-11).

For the remaining chunks (Line 12), GREM assigns nodes to partitions greedily. For each node n (Line 13), we start by estimating the number of neighbors in each partition using the current chunk's edges and most recent partition assignments (Line 15). If the node is unassigned (i.e., this is the

Algorithm 5 GREM Helper Functions

```
 1: cnt_nbrs(nodes, edges, parts):
 2:     local_nbr_counts = zeros(len(nodes), 2)
 3:     for n ∈ nodes do
 4:         for (src, dst) ∈ edges do
 5:             if src == n or dst == n then
 6:                 nbr = src if dst == n else dst
 7:                 if parts[nbr] ≠ -1 then
 8:                     local_nbr_counts[n][parts[nbr]] += 1
 9:     return  local_nbr_counts
10:
11: assign(nbrs_0, nbrs_1, part_sizes, P):
12:     if nbrs_0 < nbrs_1 and part_sizes[1] < P then return 1
13:     if nbrs_1 < nbrs_0 and part_sizes[0] < P then return 0
14:     return  arg_min(part_sizes)
15:
16: fix_sizes(new_part, old_part, part_sizes):
17:     part_sizes[new_part] += 1
18:     if old_part ≠ −1 then part_sizes[old_part] −= 1
```

first chunk containing the node), these neighbor estimates are used directly: To minimize edge cuts, we assign the node to the partition containing most of its neighbors, unless the partition is full (Line 19; Algorithm 5 - `assign`). The partition sizes are then updated (Line 20; Algorithm 5 - `fix_sizes`) and the neighbor estimates for the node are saved (Line 21). *The algorithm, as described so far, represents a streaming greedy approach with fixed assignments.*

Instead of fixing an initial greedy decision for each node, GREM reevaluates a node's partition assignment each time it reappears in memory. Specifically, for a previously assigned node n (Line 16), we refresh our estimate of the number of neighbors in each partition using an average of the estimate from the current chunk and the estimates accumulated from prior chunks (Lines 17-18). Node n is then assigned to a partition greedily using these updated estimates (Line 19), which are then saved for future

use (Line 21). We highlight that, by repeatedly averaging the accumulated neighbor estimates with the most recent ones, we are computing a weighted average of the estimates across all prior chunks containing the node, with the weight of each preceding chunk decreasing by a factor of two.

Updating prior greedy assignments based on the weighted average of neighbor estimates has the following advantages: First, nodes (which reappear) are not greedily assigned based on the estimates from only one chunk (as in existing algorithms)—these estimates can be noisy, particularly for small chunks when nodes have only a few neighbors in memory. Second, by weighting the average, more value is placed on recent estimates which are likely to be more accurate (as partition assignments may have changed since prior estimates were computed). The end result is a continuous refinement of greedy decisions throughout the algorithm.

4.2 Theoretical Analysis of GREM

We now analyze the number of edge cuts returned by GREM versus chunk size. We focus on chunk size as it directly affects the computational overhead of the algorithm. As chunk size decreases, so does GREM's memory requirement and runtime; only the active chunk of edges needs to be in memory and the time for the initial seed partitioning algorithm on the first chunk dominates the time for the simple greedy processing of subsequent chunks. We compare the expected number of edge cuts when using fixed greedy assignments (as in existing algorithms) to that of the refined greedy assignments employed by GREM.

Fixed Greedy Assignments We focus on the assignment of a specific node n and assume all other nodes are assigned to partitions. Among all edges, let node n have k neighbors, with k_0 in partition zero, and k_1 in partition one. Without loss of generality, we assume $k_0 \geq k_1$. Observe that, with a chunk size of $|E|$ (i.e., all edges), our greedy algorithm will assign node n to partition zero to minimize edge cuts.

To analyze the effect of chunk size, we ask, what is the probability node n will be assigned to partition zero if only $|E| * x$ edges (sampled uniformly) are used to make the decision (i.e., if we use a chunk size of $|E| * x$)? Let k'_0 and k'_1 be the number of neighbors of node n in partition zero and one that are present in the sampled $|E| * x$ edges. Then we seek to calculate $Pr(k'_0 \geq k'_1 | k_0 \geq k_1, x)$. We assume that $k'_0 + k'_1 = x * k$ (i.e., sampling $|E| * x$ edges leads to sampling $k * x$ neighbors). Then k'_0 (or k'_1) is a random variable sampled from a Hypergeometric distribution describing the probability of sampling (without replacement) a specific number of neighbors in partition zero (one) from a finite population of size k, containing k_0 (k_1) total neighbors in partition zero (one), using $k * x$ draws. We also have that $k'_1 = k * x - k'_0$ and $Pr(k'_0 \geq k'_1) = Pr(k'_0 \geq k * x - k'_0) = Pr(k'_0 \geq 0.5 * k * x) = 1 - Pr(k'_0 < 0.5 * k * x)$. The latter can be calculated using the cumulative distribution function (CDF) of the Hypergeometric distribution and describes the probability of correctly assigning node n given a chunk size of $|E| * x$ (correct here means making the same greedy decision as the one made if all edges are available).

Given the probability of correctly assigning node n, we can calculate the expected number of correctly assigned nodes T in the whole graph. Assuming nodes are independent, we have: $E[T] = \sum_{i=1}^{|V|}(1 - Pr(k_0^{i'} < 0.5 * k^i * x))$ with $k_0^{i'} \sim$ Hypergeometric(k^i, k_0^i, $k * x$), k^i the number of neighbors (among all edges) of node i, and k_0^i the number of these neighbors in the partition containing more of node i's neighbors. Finally, the expected number of edge cuts C is:

$$E[C] = \sum_{i=1}^{|V|}(k^i - k_0^i) * (1 - Pr(k_0^{i'} < 0.5 * k^i * x)) + k_0^i * Pr(k_0^{i'} < 0.5 * k^i * x) \quad (4.1)$$

since $k^i - k_0^i$ edges are cut for node i if it is correctly assigned and k_0^i edges are cut otherwise. Equation 4.1 can be calculated given k^i and k_0^i for each

node i (k_0^i can be estimated given an existing graph partitioning or by making assumptions about a graph's connectivity).

The Benefit of Refinement We now ask how the expected number of edge cuts $E[C]$ changes if greedy decisions are updated (refined) based on a weighted average of neighbor estimates across chunks (as in GREM). We focus on the simplest case: We assume two chunks (α and β), each of size $|E| * x$ are used to assign a given node n to a partition. Let $k'_{0,\alpha}$ and $k'_{0,\beta}$ be the number of neighbors of node n in partition zero among the sampled edges in chunk α and β respectively (and likewise for $k'_{1,\alpha}$, $k'_{1,\beta}$ and partition one). In the two chunk case, the weighted average simplifies to a regular average (which can be simplified to a sum): We seek to calculate $Pr(k'_{0,\alpha} + k'_{0,\beta} \geq k'_{1,\alpha} + k'_{1,\beta} | k_0 \geq k_1, x)$.

Observe that $k'_{0,\alpha}+k'_{0,\beta}$ is the number neighbors of node n in the $2*(|E|*x)$ edges formed by the union of chunk α and β (each chunk is disjoint). Given this, the expected number of cut edges $E[C]$, when averaging over two chunks each of size $|E| * x$, can be calculated using Equation 4.1 with x replaced by $2x$. In other words, refinement across chunks increases the *effective chunk size* (but not actual chunk size) of the algorithm, leading to better neighbor estimates. Similar intuition applies when generalizing the analysis beyond two chunks, which we omit for brevity.

In Figure 4.3, based on the analysis in this section, we plot the expected number of edge cuts $E[C]$ versus chunk size with and without refinement. Figure 4.3 highlights that refining greedy assignments based on neighbor estimates averaged across multiple chunks leads to fewer edge cuts, particularly for small chunk sizes; in fact, with this refinement, GREM can partition the graph with near minimal edge cuts even with chunk sizes $\leq$10%. See Section 4.3 for more details.

Table 4.1: Statistics for graphs used in the experiments.

Graph	Nodes	Edges
FB15K-237 (Toutanova et al., 2015)	14.5k	272k
OGBN-Products (Hu et al., 2020)	2.5M	62M
OGB-WikiKG90Mv2 (Hu et al., 2021)	91M	601M
OGBN-Papers100M (Hu et al., 2020)	111M	1.62B

4.3 Empirical Analysis of GREM

We now evaluate GREM on common large-scale graphs and compare to METIS, the SoTA min-edge-cut algorithm used by existing GNN systems. Our experiments show that GREM can efficiently scale min-edge-cut partitioning to large graphs, leading to up to 45× and 68× reduction in runtime and memory overheads compared to METIS.

Experimental Setup We start by discussing the setup used in our experiments. We report results using Open Graph Benchmark (OGB) datasets (Hu et al., 2020, 2021); we use OGBN-Papers100M (111M nodes, 1.62B edges) and OGB-WikiKG90Mv2 (91M nodes, 601M edges) for large-scale studies, and OGBN-Products (2.5M nodes, 62M edges) plus FB15K-237 (Toutanova et al., 2015) (14.5K nodes, 272K edges) for microbenchmarks. These graphs are summarized in Table 4.1. For all experiments, we measure the resulting number of edge cuts, runtime, and peak memory usage and average over three runs. For METIS, we use recursive partitioning and the implementation provided by PyMetis (Kloeckner et al., 2022).

Partitioning Quality: Number of Edge Cuts In Figure 4.1, we show the number of edge cuts that result from running GREM and METIS on three common graphs. With a chunk size of 10%, GREM partitions the graph with similar quality to METIS. For example, in the most challenging case ($p = 128$ partitions), GREM cuts just 0.5% and 1% more of the graph than METIS on OGBN-Products and OGB-WikiKG90Mv2 respectfully. GREM even achieves comparable results with a chunk size of 1%. Overall,

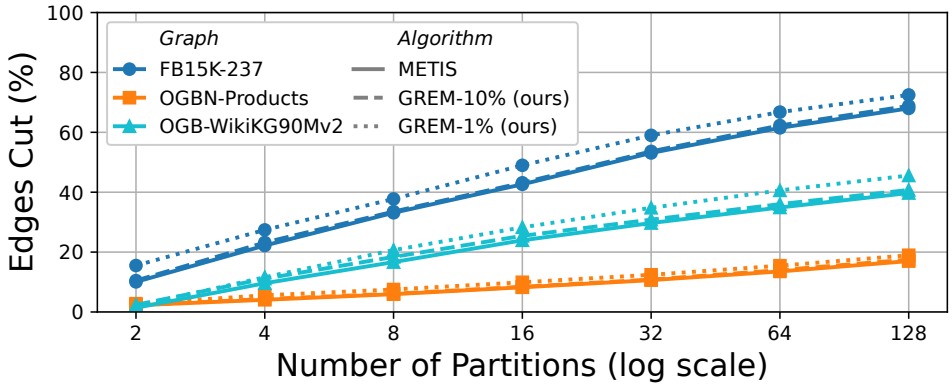


Figure 4.1: Percentage of edges cut when using GREM versus METIS on three common graphs. GREM achieves comparable edge cuts to METIS, even with a chunk size of just 10% or 1%.

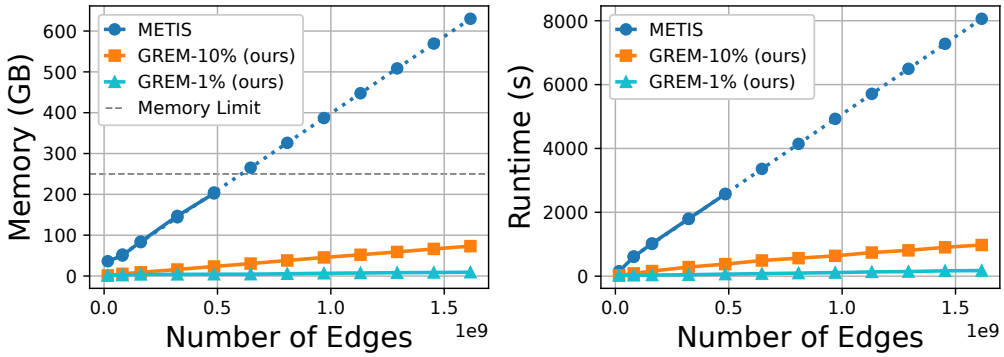


Figure 4.2: Memory usage and runtime of GREM and METIS when partitioning subgraphs of OGBN-Papers100M of various size. GREM reduces the computational requirements of partitioning.

Figure 4.1 shows that with small chunk sizes (e.g., $\leq 10\%$), GREM can partition graphs with comparable edge cuts to METIS.

Partitioning Overhead: Runtime and Memory Next, we evaluate the peak memory usage and runtime of GREM versus METIS. To do so, we use both algorithms to partition subgraphs of varying size (number of edges), taken from OGBN-Papers100M (1.6B edges total), into two parts ($p = 2$). Results are shown in Figure 4.2. We plot only $p = 2$ for simplicity;

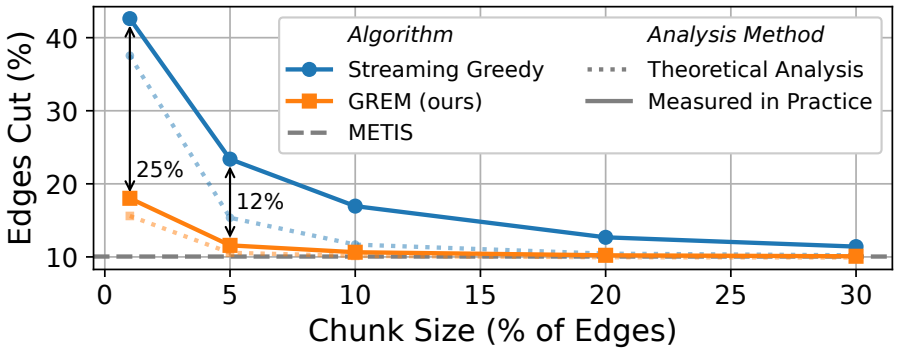


Figure 4.3: Percentage of edges cut versus chunk size when using GREM on FB15K-237 (the hardest graph to partition in Figure 4.1) compared to standard streaming greedy approaches. We focus on chunk sizes $\leq 30\%$ where the computational benefit of these methods compared to METIS (which is shown for reference, but partitions using the full graph, rather than in chunks) is maximal.

as p increases, peak memory remains constant and the runtime of each algorithm increases by the same factor (both GREM and METIS partition recursively for $p > 2$). Figure 4.2 (left) shows that METIS is able to partition 600M edges on the machine used for these experiments (250GB of memory). Based on the scaling of memory and runtime, we estimate that METIS needs 8000s and requires a machine with 630GB of memory to partition the entire OGBN-Papers100M graph; we confirmed this estimate on a special machine with 750GB. GREM, however, can partition the entire graph in just 976s with 73GB (8.2 and 8.3× reduction) or 175s with 9.3GB (46 and 65× reduction) when using a chunk size of 10% or 1% respectively.

The Benefit of Refinement Finally, we study the benefit of the refined greedy assignments used by GREM compared to the fixed greedy assignments of conventional streaming algorithms. For both approaches, we show in Figure 4.3 the number of edge cuts versus chunk size when partitioning FB15K-237 (the hardest graph to partition in Figure 4.1) into $p = 2$ partitions (given the recursive nature of GREM, similar results hold for

$p > 2$). We include both the expected number of edge cuts from the theoretical analysis in Section 4.2, and the number of edge cuts measured when running the algorithms in practice.

Figure 4.3 shows that for small chunk sizes (e.g., $\leq$10%), refinement is critical to minimizing edge cuts; we observe a reduction of up to 25% of the graph (at a chunk size of 1%). These improvements allow GREM to use smaller chunk sizes (e.g., 1-10%) without suffering a significant increase in edge cuts compared to METIS. For example, with a chunk size of 5%, GREM and METIS differ in edge cuts by <1% of the graph; this difference would be 13% with fixed greedy assignments. The consequence of these additional edge cuts is slower and more expensive GNN training—We observe that training in Armada (Chapter 6) is up to 2.4× slower when using the streaming greedy algorithm rather than GREM (for a chunk size of 1%). This confirms recent results which highlight that high quality partitioning algorithms (e.g., METIS), can lead to faster GNN training compared to streaming greedy approaches (e.g., LDG) (Merkel et al., 2023).

Summary GREM can partition large-scale graphs with comparable quality to METIS but with orders of magnitude less computational resources, helping to address the bottleneck of min-edge-cut partitioning for disk-based or distributed GNN training.

5 MIN-IO AND HIGH-ACCURACY DISK-BASED GNN TRAINING

In this Chapter, we focus on disk-based GNN training to scale to large graphs which do not fit in CPU memory on a single machine. Disk-based training has one key benefit: it leverages all available resources on a given machine, including the cheap and high capacity disk for primary graph storage, eliminating the need to pay for additional machines to scale training.

We implement disk-based training in MariusGNN (Waleffe et al., 2023). As described in Section 2.3, we store graph partitions on disk and load a subset of them into a buffer in main memory. In-buffer partitions are then used to construct mini batches according to mixed CPU-GPU training (Chapter 3). We provide an overview of MariusGNN's architecture for disk-based training in Section 5.1. We then develop partition replacement polices for swapping partitions between the buffer and disk that allow MariusGNN to iterate over all available training examples in the graph. In Section 5.2, we introduce the Buffer-aware Edge Traversal Algorithm (BETA) to minimize IO while ensuring that all graph edges appear in memory for training each epoch. Then, in Section 5.3 we highlight that BETA can lead to low GNN model accuracy due to the non-random data ordering it introduces (i.e., the order in which edges are used as training examples; see Section 2.3). We introduce the COrrelation Minimizing Edge Traversal (COMET) partition replacement policy to build on BETA, but help shuffle the order in which graph edges are used for training each epoch. In Section 5.4, we provide automated rules for tuning COMET's hyperparameters.

As highlighted in the introduction, in Section 5.5 we show with experiments over four datasets using popular GNN architectures that MariusGNN's disk-based, single-GPU training can be 8× faster than eight-GPU deployments of state-of-the-art systems. This improvement yields monetary cost reductions of an order of magnitude. We find that for graphs where existing

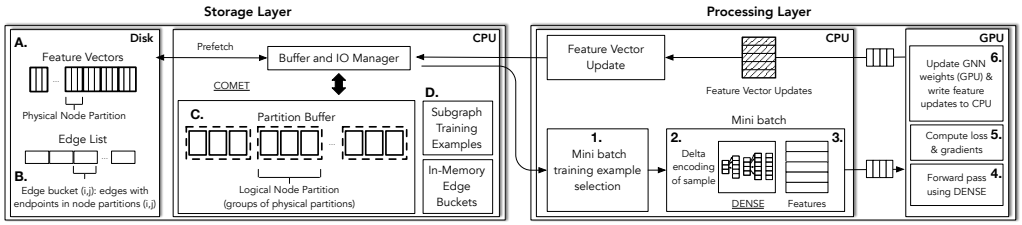


Figure 5.1: MariusGNN disk-based training. The lifecycle of a mini batch consists of Steps 1-6. According to a partition replacement policy (e.g., COMET), the storage layer periodically updates graph partitions in memory during training (Steps A-D).

systems can take six days and $1720 dollars to train a GNN, MariusGNN needs only eight hours and $36 dollars for training, a 48× reduction in monetary cost (WikiKG90Mv2, Table 5.2). Moreover, we show that single-GPU, disk-based training can be sufficient for large-scale graphs: *We use MariusGNN to train a GNN over the entire hyperlink graph from the Common Crawl 2012 web corpus, a graph with 3.5B nodes (web pages) and 128B edges (hyperlinks between pages) (Table 1.1).* MariusGNN can learn vector representations for all *3.5B nodes using only a single machine with one GPU, 60GB of RAM, and a large SSD, leading to a cost of just $564/epoch.*

5.1 Overview: Disk-Based Training in MariusGNN

MariusGNN implements out-of-core pipelined training using two modules: 1) a processing layer and 2) a storage layer. Figure 5.1 shows a diagram of disk-based GNN training in MariusGNN.

MariusGNN represents a graph as an edge list. In addition, feature vectors for nodes are stored sequentially in a lookup table split into p *physical partitions* on disk. The edge list is organized according to *edge buckets*: Given a pair of partitions (i, j), we define edge bucket (i, j) to be the collection of all edges in the graph with a source node in partition i and

a destination node in partition j (see Figure 2.5). Edges in each edge bucket are stored sequentially on disk.

Disk-based training in MariusGNN proceeds in epochs. We start each epoch with all partitions on disk. We consider an epoch completed only when all training examples in the graph have been processed once. At the beginning of each epoch, MariusGNN groups the physical partitions into a collection of $l \leq p$ *logical partitions*. The grouping is randomized and each physical partition is assigned to one logical partition. Grouping occurs without data movement: only an in-memory dictionary between logical and physical partitions is maintained. This two-level partitioning scheme is key to achieving high-accuracy GNNs (Section 5.3). Given the logical partitions, let S_i be a set of logical partitions such that all corresponding physical partitions fit in memory. According to *a partition replacement policy*, MariusGNN constructs a sequence $S = \{S_1, S_2, \dots\}$ (to be consecutively loaded into memory during training) such that each training example appears in at least one S_i. We then use S to obtain a sequence $X = \{X_1, X_2, \dots\}$. X_i is a subset of training examples in S_i such that when S_i is in memory, all (and only) training examples from X_i are used to generate mini batches. We describe the techniques used by MariusGNN to select X_i in Section 5.3.

Sequences S and X are generated in a task-aware manner. In the case of node classification, training examples are graph nodes and in the case of link prediction, they are pairs of nodes (see Section 2.1). Thus, for link prediction, pairs of node partitions need to be accessed at the same time, while for node classification node partitions can be considered individually. For link prediction, MariusGNN uses the COMET partition replacement policy to generate S and X. COMET ensures that all edges in the graph, which correspond to training examples, will appear in at least one $S_i \in S$. COMET also allows MariusGNN to minimize disk-to-CPU IO while maximizing accuracy for disk-based training. We describe COMET in Section 5.3. For node classification, MariusGNN defaults to a simple

policy that caches all labeled nodes used as training examples in memory to achieve both high accuracy and throughput, although other policies can be used (Section 5.3.2). Finally, the above design also allows MariusGNN to support training with the full graph in memory: S_1 contains the whole graph and X_1 contains all training examples.

To complete one epoch (for either task), the storage layer in MariusGNN uses the sequence S to determine which node partitions and edge buckets to bring into the *CPU partition buffer* and in what order. MariusGNN uses a buffer with a capacity of c physical partitions. When the set of c physical partitions in S_i are placed in the buffer, all c^2 pairwise edge buckets are also loaded into memory. After loading S_i, X_i is passed to the processing layer.

The processing layer generates mini batches from X_i in a random order. At this point, training proceeds according to mixed CPU-GPU mini-batch training as described in Chapter 3: MariusGNN performs multi-hop neighborhood sampling using the DENSE data structure to construct a mini batch (Section 3.3). To improve throughput, DENSE allows MariusGNN to minimize redundant computation during sampling by reusing neighborhood samples required as input to different GNN layers. Neighborhood sampling is performed only over graph nodes and edges in main memory. DENSE and the corresponding feature vectors are then transferred to the GPU to complete processing of the mini batch. DENSE is co-designed such that the forward pass for the GNN is computed using kernels that are optimized for dense linear algebra operations. After the forward pass, MariusGNN computes the loss and gradients. We update GNN parameters on the GPU and if applicable, updates to learnable feature vectors are transferred back to CPU memory and used to update the node features in the partition buffer. As described in Section 3.1, we perform all data preparation and transfers in a pipelined manner (also shown in Figure 5.1).

After training completes on the mini batches generated from X_i, the storage layer updates the partitions in the buffer from S_i to S_{i+1} by swapping

the necessary logical partitions between disk and CPU memory (one or more physical partitions) together with the corresponding edge buckets. This process repeats until the epoch is completed.

5.2 BETA: A Partition Replacement Policy with Minimal IO

In this section, we describe how to minimize IO overheads during disk-based training. In particular, we introduce BETA, a partition replacement policy that ensures all edges appear in memory each epoch with a near-minimal number of partition swaps (and thus IO) from disk to CPU memory.

5.2.1 Buffer-aware Edge Traversal Algorithm (BETA)

We develop a partition replacement policy that minimizes the number of swaps for edge-based training examples. As described in Section 5.1, we assume as input a graph that is partitioned into p^2 edge buckets corresponding to p node partitions (in this section, we assume that logical partitions and physical partitions are the same). Then, one training epoch requires iterating over all edges in the p^2 edge buckets.

We start by deriving a lower bound on the number of swaps necessary to complete one training epoch for a buffer of size c and p ($p >= c$) partitions. To derive the lower bound, we view the partition replacement policy as a sequence of partition buffers S over the epoch, where each item in the sequence S_i describes what node partitions are in the buffer at that point in training. Each successive buffer differs by one swapped partition. Given such a sequence, all edges in edge bucket (i, j) can be used for training when partitions i and j appear together in the buffer (edges in self edge buckets (i.e. (i, i)) can be processed when jus i appears in the buffer). Note that i and j must appear together at least once in some S_i otherwise not all edge buckets (and thus edges) in the graph can be processed. Viewed in this light, *we seek the shortest (minimum swaps) buffer sequence where all node*

partition pairs appear together in the buffer at least once.

Lower Bound We assume that initializing the first full buffer does not count as part of the total number of swaps as all partition replacement policies must incur this cost. Thus, there are $\frac{p(p-1)}{2}$ (the total number of partition pairs) minus $\frac{c(c-1)}{2}$ (the number of partition pairs we get in the first buffer) remaining partition pairs that must appear together in the partition buffer. On any given swap, the most new pairs we can cover is if the partition entering the buffer has not been paired with anything already in the buffer (everything in the buffer has already been paired with everything else in the buffer). Thus, for each swap, the best we can hope for is to get $c-1$ pairs we have not already seen. With this in mind a lower bound on the minimum number of swaps required is:

$$\left\lceil \frac{\frac{p(p-1)}{2} - \frac{c(c-1)}{2}}{c-1} \right\rceil \tag{5.1}$$

We use this lower bound to evaluate the performance of different policies in the next section. We experimentally show that the new policy we propose (BETA) is nearly optimal with respect to this bound.

BETA We describe the *Buffer-aware Edge Traversal Algorithm (BETA)*, an algorithm to compute the sequence of partition buffers S that ensures all partition pairs (i, j) appear together in at least one buffer with a close to optimal number of partition swaps.

Algorithm 6 describes how BETA generates the sequence of partition buffers. Consider a partition buffer that was initialized with the first c node partitions in the graph (Line 2). The remaining $p-c$ node partitions start on disk (Line 3). To generate the partition buffer sequence we then proceed as follows: First we fix the leading $c-1$ node partitions in the buffer and swap each of the outstanding partitions into the final buffer spot, one at a time (Line 6-8). Each swap creates a new partition buffer in the sequence. Once this is complete, the fixed $c-1$ partitions have been paired in the

Algorithm 6 BETA Partition Replacement Policy

Require: p partitions stored on disk index by $[0 \dots p]$

```
 1: S = {}
 2: in_memory = [0 ... c − 1]
 3: on_disk = [c ... p − 1]
 4: S.append(in_memory)
 5: while on_disk.size() > 0 do
 6:    for i in range(on_disk.size()) do
 7:       swap(in_memory[−1], on_disk[i])
 8:       S.append(in_memory)
 9:    n = 0
10:    for i in range(c − 1) do
11:       if i ≥ on_disk.size() then
12:          break
13:       n = n + 1
14:       in_memory[i] = on_disk[i]
15:       S.append(in_memory)
16:    on_disk = on_disk[n : end]
17: return S
```

buffer with all other node partitions in the graph and are therefore no longer needed this epoch. We refresh our buffer by replacing the finished $c - 1$ partitions with new node partitions from the unfinished set on disk (Line 10-15). The incoming partitions can then be deleted from the on-disk set (Line 16) since they are now in the buffer. As before, each swap results in a partition buffer added to the sequence. We repeat this process until there are no remaining unfinished node partitions (Line 5 and 11-12).

Note that the final partition buffer sequence S can be easily converted into an iteration (ordering) over edge buckets that can be used for training. For BETA, we add edge bucket (i, j) to the order the first time partitions i and j appear together in a buffer. This ordering includes all edge buckets and thus all graph edges. Stated in the language of Section 5.1, we obtain the sequence $X = \{X_1, X_2, \dots\}$, where X_i is the set training examples used to generate mini batches when S_i is in memory, by adding all edges in edge bucket (a, b) to the first X_i for which S_i contains both a an b. We show an

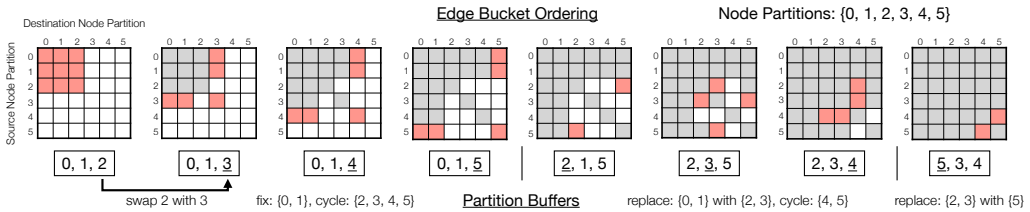


Figure 5.2: Example of BETA for $p = 6$ and $c = 3$. The sequence of partition buffers corresponds to first fixing $\{0, 1\}$, then replacing $\{0, 1\}$ with $\{2, 3\}$, fixing $\{2, 3\}$, and finally replacing $\{2, 3\}$ with $\{5\}$. Each successive buffer differs by one swap. The corresponding edge buckets that are used for training are shown above the buffers: For each partition buffer in the sequence, all previously unused edge buckets which have their source and destination node partitions in the buffer are used for training.

example of the BETA policy in Figure 5.2.

We observe that BETA has a number of useful properties that make it advantageous to implement in practice. Since all partitions are symmetrically processed we do not need to track any extra state or use any priority mechanisms. Further, for every disk IO (swap) with a fixed set of $c - 1$ partitions (Line 7 in Algorithm 6), the incoming node partition has yet to be paired with any other partition in the buffer. This means there are $c - 1$ new edge buckets available for training before we need to perform another swap—the most possible (excluding self edge buckets)—allowing us to overlap IO operations behind longer compute times. The only bottleneck arises when the fixed $c - 1$ partitions are replaced, but this only happens at most $\left\lfloor \frac{p-c}{c-1} \right\rfloor + 1$ times in one epoch. Additionally, BETA can be randomized to create different graph traversals by shuffling which partitions start in the buffer, by permuting the buffer and/or on disk set before Line 6 in Algorithm 6, or by permuting the on disk set before Line 10 in Algorithm 6.

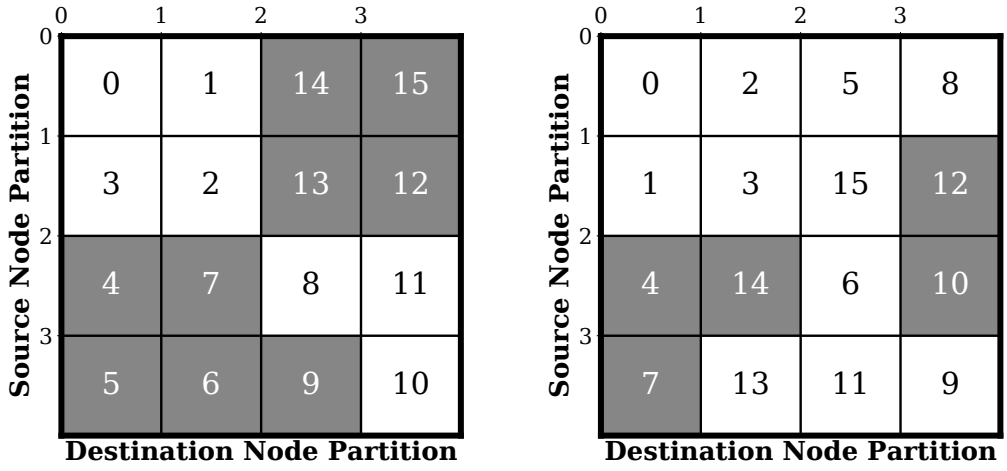
Finally, we analyze the number of swaps generated by BETA: given p

partitions and a buffer of size c the number of swaps is

$$\begin{gathered}(p-c)+(x+1)\left[(p-c)-\frac{1}{2}x(c-1)\right] \\ \text{where } x=\left\lfloor\frac{p-c}{c-1}\right\rfloor.\end{gathered} \tag{5.2}$$

Comparison with Hilbert and Lower Bound We compare the number of swaps incurred by BETA with the analytical lower bound and with the number of swaps incurred by iterating over graph edge buckets (and thus edges) according to space-filling curves: Space filling curves like Hilbert (Hilbert, 1891) attempt to define a graph traversal where edges buckets located close together in the $n \times n$ matrix of all edge buckets are also close together in the ordering (e.g., Figure 5.3a). During traversal, whenever an edge bucket is encountered for which both partitions are not in the buffer, a swap is required (since the ordering is known ahead of time, the optimal eviction policy can be used which evicts the partition used again farthest in the future (Belady, 1966)). We also compare to a second version of Hilbert, termed Hilbert Symmetric, which modifies the former by including edge buckets (i, j) and (j, i) successively in the order. A key advantage of BETA when compared to these methods is that it is buffer-aware, i.e., the algorithm knows the buffer size and specifically aims to minimize partition swaps. In contrast, space-filling curve based orderings are unaware of this information, aiming instead to process edge buckets with locality in the $n \times n$ matrix close together during training.

We illustrate how BETA compares to a Hilbert space-filling curve on a small $p = 4, c = 2$ case in Figure 5.3. We see that while Hilbert requires nine swaps, BETA only requires five. We also performed simulations to compare each method. Figure 5.4 shows the number of swaps when varying p and using a buffer with size $\frac{p}{4}$ for BETA, Hilbert, and Hilbert Symmetric, together with the lower bound. BETA yields nearly optimal performance across configurations and requires significantly less IO than the other methods. For



(a) Hilbert (b) BETA

Figure 5.3: Comparison of the order in which edge buckets are used for training (labeled by the number in each edge bucket) for Hilbert and BETA with $p = 4$ and $c = 2$. Gray cells indicate a partition swap was required before that edge bucket could be used for training.

a detailed empirical study of BETA versus Hilbert orderings during training, we refer the reader to Mohoney et al. (2021).

We leave an investigation of a provably-optimal policy for future work. Our initial studies have shown that there exist cases of p and c where no policy can match the lower bound as well as cases where a policy which requires slightly fewer swaps than BETA exists. Thus, the optimal algorithm requires swaps somewhere between the lower bound and BETA in Figure 5.4.

Finally, we note that once a partition replacement policy (e.g., S) has been selected, we can further mitigate IO overhead by using a prefetching thread to load node partitions in the background as they are needed in the near future. Correspondingly when a partition needs to be evicted from memory, we perform asynchronous writes using a background writer thread.

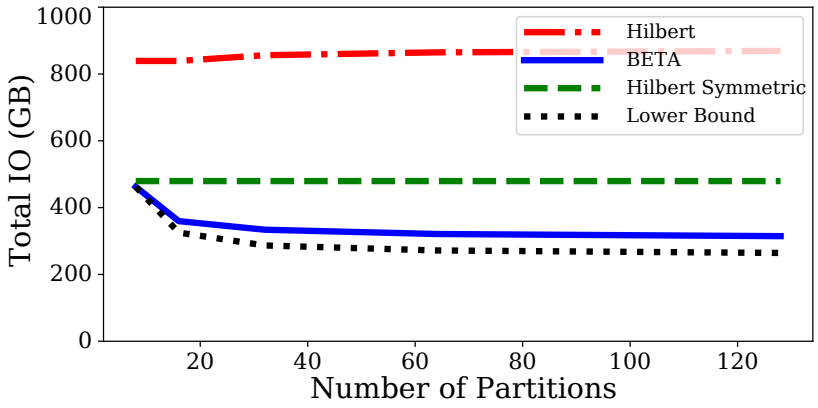


Figure 5.4: Simulated total IO performed during on epoch of training on Freebase86m (Google, 2018) with a feature vector size of 100. BETA outperforms existing algorithms and achieves near-optimal IO.

5.3 COMET and High-Accuracy Partition Replacement Policies

We now discuss the partition replacement policies used by MariusGNN for disk-based training. As described in Section 5.1, MariusGNN uses different policies for link prediction and node classification. We discuss each in turn.

5.3.1 Policies for Link Prediction

MariusGNN uses the COMET policy to maximize throughput while maintaining high accuracy when training link prediction models (where training examples correspond to graph *edges*). Before we introduce COMET, we discuss why BETA (and more generally any policy that only optimizes for throughput without considering the order of examples for training) leads to biased training and hence harms the accuracy of the learned GNN models.

Greedy policies, e.g., BETA, that focus on minimizing IO for high throughput produce correlated training examples that bias learning and lead to low model accuracy. Recall that we define $S = \{S_1, S_2, \dots\}$ to be the sequence of partition sets which will be loaded into memory during one epoch

Figure 5.5: Greedy sequence of partitions in memory S and training examples X that are correlated (e.g., those generated by BETA). For instance, the examples in X_2 all come from edge buckets containing partition four.

and $X = \{X_1, X_2, \dots\}$ to be the sequence of training examples used to generate mini batches for each $S_i \in S$ (Section 5.1). To minimize IO, greedy policies swap partitions between S_i and S_{i+1} such that the new partitions brought into memory maximize the number of new training examples that can be generated from the in-memory graph. For example, the BETA policy minimizes IO by bringing one new physical partition p^* in memory to obtain S_{i+1} and uses the edges (training examples) that correspond to node pairs formed by combining p^* with all other partitions in memory to construct X_{i+1}. This process makes all training examples in X_{i+1} be correlated: they all have one endpoint in the new partition p^*. We show an example of this problem in Figure 5.5. As highlighted in Section 2.3, performing training over correlated examples reduces randomness in the order edges are processed each epoch and conflicts with the independently distributed assumption of ML training data. In Section 5.5.4, we show that using a greedy policy leads to accuracy degradation compared to training with the full graph in memory.

COMET addresses the above shortcoming by introducing randomness in the order that training examples are processed each epoch while simultaneously minimizing IO. To increase randomness, we design COMET around

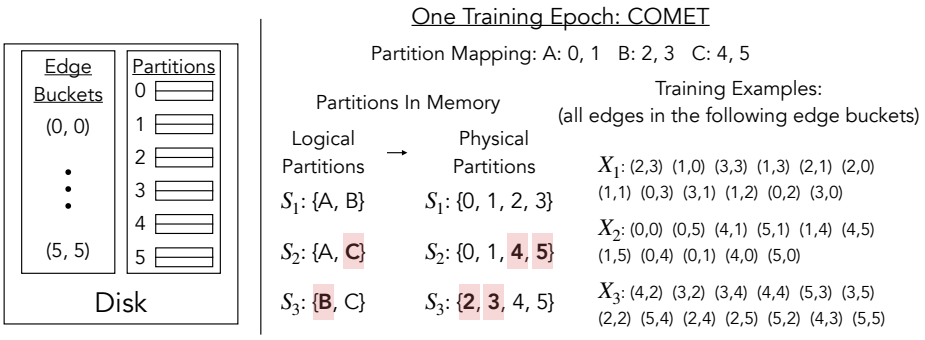


Figure 5.6: Partition and training example sequences generated by COMET to minimize training example correlation.

two mechanisms: 1) a two-level (logical and physical) partitioning scheme and 2) randomized generation of training examples. Following the architecture from Section 5.1, the first mechanism generates the sequence of partition sets S for one epoch and the second mechanism generates the sequence of training examples X.

To decouple data storage and access from data transfer, COMET uses physical partitions on disk but transfers groups of physical partitions—called logical partitions—between disk and CPU memory. At the beginning of each epoch, physical partitions are randomly grouped into logical partitions (without data movement) (Section 5.1). COMET then generates $S = \{S_1, S_2, \dots\}$ by greedily swapping (according to BETA) one logical partition between S_i and S_{i+1} such that all pairs of partitions (and thus edges) appear in at least one S_i with minimal IO. By utilizing logical partitions, COMET allows MariusGNN to improve randomness by utilizing small physical partitions—which fix fewer nodes together in a partition for the whole training process—yet also use large logical partitions to increase turnover rate of graph data between each S_i. In Section 5.4, we analyze how to best set the number of physical and logical partitions to simultaneously minimize IO and maximize accuracy.

Instead of using logical partitions to increase randomness in the sequence

of partitions sets S, an alternative design would be to create a new greedy algorithm to minimize IO while considering multiple physical partition swaps at once. In MariusGNN we opt to use logical partitions for the following reasons. First, by swapping one logical partition at a time MariusGNN can utilize existing one-swap greedy algorithms that have been shown to minimize total epoch IO near the theoretical lower bound (i.e., BETA). Thus, a multi-swap greedy algorithm can at best provide little IO benefit. Finally, allowing for multiple swaps at once exponentially increases the number of swap choices to consider between each set of partitions S_i and S_{i+1}, making it challenging to develop an efficient multi-swap algorithm to generate S. Thus, we focus on using a two-level partitioning scheme in MariusGNN.

Beyond introducing randomness in S, COMET also injects randomness in the sequence of training examples X used to create mini batches. Given that S is generated at the beginning of each epoch, MariusGNN performs the following optimization: For each pair of partitions (i, j) in the graph, MariusGNN identifies all partition sets $S_{(i,j)} \subseteq S$ that contain both i and j. COMET then picks one S_* at random from $S_{(i,j)}$ and assigns the training examples corresponding to pairs of nodes between these two partitions—the edges in edge bucket (i, j)—to X_*. This random assignment allows for the deferred processing of training examples rather than greedily processing all new examples immediately upon arrival in CPU memory. Beyond shuffling training examples, this deferred execution scheme also balances the workload across each X_i—each X_i contains in expectation the same number of training examples. When prefetching is used to mask the IO latency required to load S_{i+1} during mini-batch training on S_i, balanced workloads enable consistent overlapping of IO and compute. In contrast, greedy policies generate unbalanced workloads where some X_i contain very few training examples (e.g., Figure 5.5). For these cases, training on X_i completes before S_{i+1} is loaded leading to IO bottlenecks.

5.3.2 Policies for Node Classification

For node classification we find that a simple replacement policy is generally sufficient to maximize throughput without harming accuracy. To iterate over all training examples during each epoch, we require that all labeled graph nodes in the training set—called the *training nodes*—appear in memory at least once (in at least one S_i) (this is in contrast with link prediction, where we aim to iterate over all edges each epoch). We find that in large-scale graphs (Hu et al., 2021, 2020), it is often the case that the training nodes make up only one to ten percent of all graph vertices. As such, the feature vectors for the training nodes can fit in CPU memory, even when the storage overhead for the full graph is many times larger. When this observation holds, for disk-based node classification, MariusGNN performs static caching of the training nodes and their feature vectors in CPU memory. While prior works have also used static caching for GNN training (Lin et al., 2020; Yang et al., 2022), these approaches focus on caching hot vertices in GPU memory to minimize CPU to GPU data transfer rather than caching training examples in CPU memory to minimize disk to CPU transfers.

More specifically, we perform disk-based node classification as follows: We assign all training nodes sequentially to the first k physical partitions. Non-training nodes are assigned to the remainin $p-k$ physical partitions as before. We generate one set of partitions to be loaded into memory each epoch $S = \{S_0\}$. S_0 contains the k partitions with training nodes together with $c-k$ other randomly chosen physical partitions (buffer capacity c). By construction, all training nodes are assigned to create mini batches in X_0. This policy leads to zero partition swaps (IO) during an epoch (IO does occur between epochs), but assumes that k is less than c (all training examples can fit in CPU memory). In the future, we plan to study how this approach compares with other schemes (Liu et al., 2023) for dynamically caching training examples in CPU memory.

If $k \geq c$, MariusGNN has two options: First, we can partition the graph

Algorithm 7 Node Classification Partition Replacement Policy

Require: parts: logical partitions $[0 \dots l-1]$
c_l: number of logical partitions which can fit in memory
$S_1^l = \texttt{ran\_perm}(\text{parts})[0 : c_l]$
disk = parts $\setminus S_1^l$
$i = 1$
while disk.`size`() > 0 **do**
  $i = i + 1$
  $S_i^l = \texttt{copy}(S_{i-1}^l)$
  $r_1 = \texttt{randInt}(c_l)$
  $r_2 = \texttt{randInt}(\text{disk.}\texttt{size}())$
  $S_i^l[r_1] = \text{disk}[r_2]$
  disk.`pop`(r_2)
return $\{S_1^l \dots S_i^l\}$

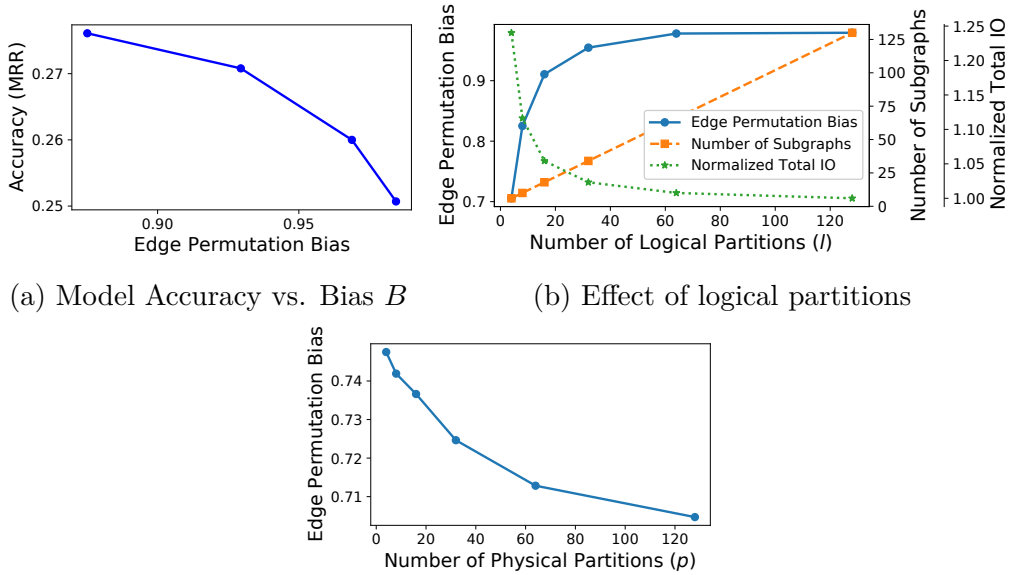
into p physical partitions as usual (in which case the training nodes will be dispersed across all partitions) and use COMET to generate the sequence of partition buffers S for training, but then generate X by randomly assigning training nodes to each X_i rather than edges. COMET, however, generates S with a stronger requirement than needed for node classification; it ensures all pairs of partitions (and thus edges) appear in memory each epoch, rather than just all partitions (and thus nodes and by extension training nodes). As such, we can ensure the latter, which is sufficient for node classification, with less IO according to the following policy to generate S (Algorithm 7): We replace a random logical partition in memory (as before, the replacement policy operates on logical partitions and MariusGNN maps them to their corresponding physical partitions) with a random one from disk that has not appeared in memory until all partitions have appeared in the buffer. For the goal of bringing each partition into memory, this replacement policy is guaranteed to have minimal disk-to-CPU IO as partitions are read from disk exactly once. For this policy, X can be generated as above for COMET.

5.4 Hyperparameter Auto-Tuning Rules For Disk-Based Training

MariusGNN provides auto-tuning for 1) the number of physical partitions p, 2) the number of logical partitions l, and 3) the buffer capacity c to minimize training time and maximize model accuracy out of the box. We focus on p and l since maximizing c best approximates training with the full graph in memory and thus leads to better runtime and accuracy. We first connect p and l to model accuracy, then focus on their effect on runtime, and conclude by using this information to present auto-tuning rules (for p, l, and c).

Effect of p and l on Accuracy To study the effect p and l have on model accuracy, we introduce a proxy metric which we term the *Edge Permutation Bias* B. Recall that model quality can degrade if training consecutively iterates over correlated examples (Section 5.3.1). This problem is general to ML workloads (Haochen and Sra, 2019; De Sa, 2020; Hofmann et al., 2015). We design B to capture the extent to which the sequence of training examples generated by COMET exhibits this phenomenon. Figure 5.7a illustrates the dependency between B and model accuracy. The depicted results correspond to empirical measurements over a benchmark model (GraphSage (Hamilton et al., 2017)) and dataset (FB15k-237 (Toutanova et al., 2015)). We find the same behavior to hold across settings.

We now define B. Let $X = \{X_1 \dots X_n\}$ be the sequence of *edge bucket sets* X_i assigned as training examples for each partition set S_i (Section 5.3.1). If X_i contains edges that focus on a small subset of nodes, then we have the undesired correlation described above. We empirically measure this occurrence as follows: Let V be the set of nodes in the graph. For each node $v \in V$ we keep a tally t_i^v as we iterate over X which measures how many edges we have seen containing this node after each X_i. Tallies are cumulative and we assume a uniform degree distribution. We normalize such that $t_n^v = 1$. This implies $t_i^v \in [0, 1]$. After each X_i we calculate



(a) Model Accuracy vs. Bias B

(b) Effect of logical partitions

(c) Effect of physical partitions

Figure 5.7: Empirical measurements on the effect of COMET hyperparameters using GraphSage on FB15k-237.

$d_i = max_{v1,v2 \in V}(t_i^{v1} - t_i^{v2})$. Given this, $B = max_i d_i \in [0, 1]$.

We are interested in how evenly the tallies are incremented during a training epoch. Biased assignments will lead to processing many edges for a subset of nodes at once while ignoring the remaining graph vertices. This leads to high variance in model gradients across the epoch. With this in mind, $B = z$ means that $z + a$ percent of the edges containing a certain node have been processed before a percent of the edges of another node have been processed (for $a \in [0, 1 - z]$).

Figure 5.7c shows that B decreases with increasing physical partitions. The observed trends can be characterized by the equation $B = \mathcal{O}(p^{-\alpha_1})$ for some constant $\alpha_1 > 1$. Figure 5.7b shows the effect of the number of logical partitions on B. Decreasing the number of logical partitions decreases the Edge Permutation Bias roughly according to $B = \mathcal{O}(l^{\alpha_2})$ for $0 < \alpha_2 < 1$. While we do not provide a closed-form characterization of B

as a function of p and l we find that the aforementioned trends hold across datasets. Moreover, describing the limiting behavior of B with respect to p and l suffices to obtain a concrete methodology for optimizing these hyperparameters (described below).

Effect of p and l on Training Time We now focus on how p and l affect the per-epoch runtime T. To do so, we analyze three metrics that influence training time: 1) the total IO in terms of bytes transferred from disk to CPU memory (IO), 2) the number of partition sets generated per epoch ($|S|$), and 3) the smallest size of disk reads (R) in bytes. As Quantities 1 and 2 increase, the total training time increases—recall from Section 3.3.1 that preparing each S_i for training requires creating single-hop sampling data structures—while a decrease in Quantity 3 leads to an increased runtime.

By construction in COMET, the number of logical partitions affects Quantities 1 and 2. Figure 5.7b shows that as l increases the total IO decreases and the number of partition sets per-epoch increases. The limiting behavior of IO and $|S|$ with respect to l is: $IO = \mathcal{O}(l^{-\alpha_3})$ for $\alpha_3 > 1$ and $|S| = \mathcal{O}(l)$. For the purposes of this work, we assume that the training time is dominated by the number of partition sets ($|S|$) per epoch instead of the total IO for two reasons: First, the relative difference between the best and worst IO is usually only between 5-25 percent and second, prefetching can overlap IO with compute. Thus, we take the training time $T = \mathcal{O}(l)$.

The training time T is also affected by the number of physical partitions p through Quantity 3. As p increases, the size of each partition decreases linearly and the size of each edge bucket decreases quadratically—the smaller of these quantities is the smallest disk read size R. As a result, disk access transitions from large sequential reads and writes to small random reads and writes with increasing p. Given the hardware constraints of block storage, the latter can become a bottleneck, particularly when read sizes R are less than the disk block size D. Thus, we model the affect of p on training time according to $T = \mathcal{O}(1)$ for $p \leq \alpha_4$ and $\mathcal{O}(p)$ for $p > \alpha_4$, with α_4 representing

the number of partitions which cause the smallest disk reads to be equal the block size D.

Methodology for Setting Hyperparameters Given the effect of p and l on accuracy and training time described above, together with the desire to maximize the buffer capacity c, we now present rules for setting the COMET hyperparameters. We assume a graph $G = (V, E)$, that the feature vectors of each node have dimension d, that CPU memory is of capacity CPU bytes, and that the disk block size is D. First, we calculate the total overhead of storing all features as $NO = |V| * d * 4$ bytes (using floating point numbers). Likewise, the edge overhead EO can be calculated from $|E|$ and the number of bytes per edge. Then, the overhead of each node partition is $PO = NO/p$ and the expected size of each edge bucket is $EBO = EO/p^2$.

With the above definitions, p affects the Edge Permutation Bias B and training time T as follows: $B = \mathcal{O}(p^{-\alpha_1})$ for some constant $\alpha_1 > 1$ and $T = \mathcal{O}(1)$ for $p \leq \alpha_4$ and $\mathcal{O}(p)$ for $p > \alpha_4$ with $\alpha_4 = min(NO/D, \sqrt{EO/D})$. Thus, to minimize B without increasing T, we set $p = \alpha_4$. We then maximize c such that $c * PO + 2 * c^2 * EBO + F < CPU$. The edge term is multiplied by two because MariusGNN utilizes two sorted versions of the edge list for neighborhood sampling (Section 3.3.1) and we leave some extra CPU space for working memory (fudge factor F). Finally, B and T are affected by the number of logical partitions according to $B = \mathcal{O}(l^{\alpha_2})$ for $0 < \alpha_2 < 1$ and $T = \mathcal{O}(l)$. As such, we minimize both by minimizing l. COMET imposes the constraint that the number of logical partitions in the buffer $c_l \geq 2$ and that $p/c = l/c_l$. Therefore $l = 2 * p/c$.

5.5 Results: Disk-Based Training in MariusGNN

We implement disk-based training in MariusGNN. Together with the code for mixed CPU-GPU mini-batch training, MariusGNN totals 16k lines of C++ and 5k lines of Python. We evaluate disk-based training on four large-

scale graphs (see Table 1.1 for dataset statistics) using the same setting as Section 3.4 and compare against the results when using state-of-the-art GNN systems (DGL and PyG) as well as MariusGNN with the full graph in memory. We show that:

1. MariusGNN disk-based training reaches the same level of accuracy up to 2-8× faster and 8-64× cheaper than DGL and PyG on all datasets for both node classification and link prediction.

2. MariusGNN enables cheap and efficient training of GNNs using disk storage. COMET yields fast runtime and high accuracy for link prediction. Additionally, MariusGNN allows us to train GNNs for node classification when datasets exceed commodity main memory.

3. MariusGNN auto-tuning rules for COMET yield configurations that achieve the highest throughput and model quality, lowering the deployment burden for training.

5.5.1 Experimental Setup

We highlight the setup used for disk-based experiments. All other experimental details remain the same as described in Section 3.4.1.

Baselines We compare end-to-end GNN training over large-scale graphs in MariusGNN against DGL 0.7 and PyG 2.0.3 (late 2021 releases).

Hardware Setup As before, we evaluate disk-based training using AWS P3 instances (Table 3.1). We use an EBS volume with 1GBps of bandwidth and 10000 IOPS as disk storage. In this section, we report results for MariusGNN using two hardware configurations: one for disk-based training (MariusGNN$_{Disk}$) and one for training with the full graph in memory (MariusGNN$_{Mem}$; these results are the same as reported for MariusGNN in Section 3.4). For disk-based training, we minimize training costs by using only a single P3.2xLarge machine—an instance that does not have enough CPU memory to store any of the large-scale graphs used in

these experiments (having only 61GB of memory). For the latter, recall that we use the cheapest P3 instance which has enough RAM for training with the full graph in memory (either a P3.8xLarge or P3.16xLarge). Baseline systems do not support training if graph data does not fit in CPU memory. Thus, for each graph, we report results for DGL and PyG using the same P3 instance that was used for MariusGNN$_{Mem}$. Recall also that we allow baseline systems to use the maximum number of GPUs they support and available in the instance. *MariusGNN$_{Disk}$ and MariusGNN$_{Mem}$ uses only one GPU for all experiments.*

Datasets, Models, and Metrics For node classification and link prediction, we use the same datasets, models, and metrics described in Section 3.4.1

Hyperparameters We use the same hyperparameters as described in Section 3.4.1. MariusGNN disk-based training hyperparameters are set using the auto-tuning rules presented in Section 5.4.

5.5.2 End-to-End System Comparisons

We discuss end-to-end training results for MariusGNN disk-based training compared to MariusGNN, DGL, and PyG with the full graph in memory on node classification and link prediction tasks.

Results are reported in Tables 5.1-5.3 and Figure 5.8. As in Section 3.4.2, for each experiment we train all systems for the same fixed number of epochs and measure 1) the per-epoch runtime, 2) model accuracy or MRR, and 3) the monetary cost per epoch based on AWS pricing. However, we now report two configurations for MariusGNN—one with graph data stored in main memory (MariusGNN$_{Mem}$) and one using disk-based training (MariusGNN$_{Disk}$). Next, we highlight key takeaways among all end-to-end results before focusing on each setting (Table) in more detail.

Key Takeaway MariusGNN disk-based training is the faster *and* cheaper training option to comparable accuracy compared to baseline systems (DGL and PyG) for all dataset and model combinations on both learning tasks.

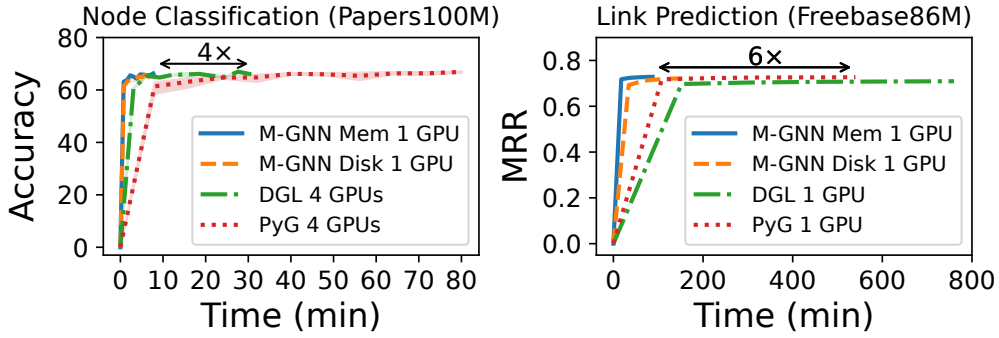


Figure 5.8: Time-to-accuracy for MariusGNN, DGL, and PyG. MariusGNN reaches the same level of accuracy 4-6× faster and remains faster than baseline systems even when using disk-based training.

Table 5.1: MariusGNN, DGL, and PyG for node classification on large-scale graphs using a GraphSage GNN. Using a single GPU, MariusGNN can reach the same level of accuracy as multi-GPU baselines 3-8× faster and up to 64× cheaper by leveraging disk-based training.

	Epoch (min.)		Accuracy		Cost ($/epoch)	
Dataset	Papers	Mag	Papers	Mag	Papers	Mag
MariusGNN$_{Mem}$	**0.77**	2.57	66.38	63.17	0.16	1.05
MariusGNN$_{Disk}$	0.83	**0.94**	66.03	62.53	**0.04**	**0.05**
DGL	3.07	7.83	66.98	63.73	0.63	3.19
PyG	8.01	19	66.93	63.47	1.63	7.75

Moreover, cost reductions for all experiments are *at least* 8×. Disk-based training can lead to differences in cost of almost two orders of magnitude: Baseline systems can take six days and $1720 dollars for training (see training on Wiki in Table 5.2), yet MariusGNN disk-based training needs only $36 dollars for the same dataset. These improvements allow MariusGNN to efficiently scale GNN training to graphs more than two orders of magnitude larger than existing benchmark datasets.

Node Classification We focus on end-to-end disk-based training results for node classification in more detail (Table 5.1). We show the time-to-accuracy

Table 5.2: MariusGNN, DGL, and PyG for link prediction on large-scale graphs. All systems use a GraphSage GNN and one GPU. MariusGNN reaches comparable accuracy to baselines 6-7× faster and 13-18× cheaper using disk-based training. (OOT: out of time)

	Epoch (min.)		MRR		Cost ($/epoch)	
Dataset	FB	Wiki	FB	Wiki	FB	Wiki
MariusGNN$_{Mem}$	**17.5**	**46.6**	.7285	.4655	3.57	9.38
MariusGNN$_{Disk}$	34.2	69.9	.7216	.4156	**1.74**	**3.56**
DGL	152	844	.7091	OOT	31.0	172
PyG	108	312	.7267	.4683	22.0	63.6

on Papers100M in Figure 5.8. MariusGNN can train the same GraphSage models for node classification as baseline systems using disk-based training on a single AWS P3.2xLarge machine with one GPU, while still training 4× and 8× faster than DGL (the fastest baseline). These timings occur even as DGL stores the full graph in CPU memory and uses four and eight GPUs on Papers100M and Mag240M-Cites respectively; recall that while both DGL and PyG support multi-GPU training, they both underutilize the additional compute resources: DGL and PyG four-GPU training on Papers100M are only 1.4× and 1.1× faster than their single-GPU performance respectively, and DGL eight-GPU training on Mag240M-Cites is only 2.2× faster than with one-GPU (single-GPU baselines not reported in Table 5.1). *The combination of a cheaper machine, baseline systems scaling sublinearly, and faster epoch runtimes lead MariusGNN disk-based training to be 16× and 64× cheaper on the two graphs.* Moreover, disk-based node classification in MariusGNN can actually be faster than in-memory training (e.g., Mag240M-Cites). This occurs because neighborhood sampling operations are performed over in-memory subgraphs, leading to fewer returned neighbors and smaller mini batches. While this can improve throughput, it can also introduce slight accuracy reductions (e.g., 63.17 to 62.53).

Link Prediction We now focus on end-to-end disk-based training for

Table 5.3: Comparison of GraphSage (GS) and GAT GNN training in MariusGNN, DGL, and PyG for link prediction on Freebase86M. Baselines bottlenecked by CPU-based mini batch preparation result in similar training time and cost on GraphSage and the more computationally expensive GAT.

	Epoch (min.)		MRR		Cost ($/epoch)	
Model	GS	GAT	GS	GAT	GS	GAT
MariusGNN$_{Mem}$	**17.5**	**52.6**	.7285	.7331	3.57	10.7
MariusGNN$_{Disk}$	34.2	56.9	.7216	.7251	**1.74**	**2.90**
DGL	152	151	.7091	.6516	31.0	30.8
PyG	108	107	.7267	.7252	22.0	21.8

link prediction. Results for all systems on Freebase86M and WikiKG90Mv2 are reported in Table 5.2 and 5.3. Time-to-accuracy on Freebase86M is shown in Figure 5.8. COMET allows MariusGNN to train the same models for link prediction on a 3× cheaper (P3.2xLarge) machine than baseline systems by utilizing disk storage. For this task, disk-based training in MariusGNN is slower than in-memory training for two reasons: 1) COMET requires performing disk IO during every epoch, and 2) the P3.2xLarge machine has 4× fewer CPU resources available for reading/writing feature vectors to main memory. Yet, epoch runtimes remain 1.9×-4.5× faster than baseline systems, yielding cost reductions of 7.5-18×. As described in Section 5.3, achieving high-accuracy disk-based GNN models for link prediction is a key challenge. On Freebase86M, COMET allows MariusGNN to reach comparable model quality to the in-memory setting. Yet, recovering in-memory accuracy remains an open problem for some datasets (e.g., Wiki). We evaluate COMET in more detail in Section 5.5.4.

5.5.3 Extreme Scale GNN Training With One GPU

A core motivation of our work is to use the resources in a single machine efficiently to scale GNN training to massive graphs. To evaluate our ability to achieve this goal, we stress-test MariusGNN with respect to graph size:

We consider the task of learning feature vectors for link prediction over the *entire hyperlink graph from the Common Crawl 2012 web corpus*, a graph with 3.5 billion nodes (web pages) and 128 billion edges (hyperlinks between pages) (Table 1.1). We use MariusGNN disk-based training on an AWS P3.2xLarge instance with one GPU, 60GB of RAM, and 4TB of SSD storage. To learn the features, we use a GraphSage GNN with 10 neighbors, the DistMult score function with 500 negative samples, and an embedding dimension of 50. We find that MariusGNN is able to train this GNN model over the hyperlink graph—a graph with 210× more edges than the largest graph in the Open Graph Benchmark large-scale challenge Hu et al. (2021) (WikiKG90Mv2)—while maintaining a throughput of 194k edges/sec, leading to a monetary cost of only $564 per epoch. Thus, MariusGNN costs only 3.3× more per epoch while training on the hyperlink graph compared to baseline systems training on WikiKG90Mv2. This experiment presents an initial large-scale benchmark that can be used by the community to measure the cost of training over large graphs and to understand the power of optimized single machine deployments.

5.5.4 Evaluating COMET for Disk-Based Training

In Section 5.5.2, we showed that COMET allows for disk-based GNN training on the link prediction task 7.5-18× cheaper than DGL and PyG. We now evaluate COMET in more detail and compare to BETA. We perform disk-based training using both methods and measure 1) the per-epoch runtime and 2) the disk-based model accuracy (using MRR). We also report MRR for in-memory training as a baseline for disk-based MRR. We use GraphSage and GAT GNNs on the graphs FB15k-237 Toutanova et al. (2015), Freebase86M, and WikiKG90Mv2. We include FB15k-237 (14541 nodes, 272115 edges) to measure the bias present in disk-based training policies while utilizing all neighbors for GNN aggregation and all negatives for computing MRR (as opposed to using neighbor/negative sampling for large graphs). We also use the DistMult knowledge graph embedding model to compare COMET and

Table 5.4: COMET versus BETA for disk-based link prediction using GraphSage (GS) and GAT GNNs as well as the DistMult (DM) knowledge graph embedding model. COMET leads to simultaneously faster training and higher MRR (accuracy). (237: FB15k-237; Epoch time in seconds for 237)

Model	Graph	Mem MRR	Disk-Based MRR		Epoch (min.)	
			COMET	BETA	COMET	BETA
DM	237	.2533	**.2659**	.2431	**1.78**	1.95
DM	FB	.7249	**.7220**	.7189	**13.73**	17.51
DM	Wiki	.3941	**.4071**	.3951	**22.54**	27.75
GS	237	.2825	**.2736**	.2369	**3.07**	3.28
GS	FB	.7342	**.7123**	.6976	**47.45**	50.08
GS	Wiki	.4658	.4078	.4080	**76.66**	82.34
GAT	237	.2869	**.2341**	.2076	**3.51**	3.90
GAT	FB	.7418	**.7053**	.6860	**42.01**	46.02

BETA on decoder-only models. We utilize a buffer capacity that can store 1/4 of all partitions in memory. We enable prefetching to overlap IO with computation. COMET hyperparameters are set as described in Section 5.4. For BETA, which has no auto-tuning rules, we manually tune the number of partitions for best performance.

We report the runtime and MRR for all models and datasets in Table 5.4. While the BETA policy achieves near in-memory MRR for the specialized DistMult model, MRR drops by up to 16% for GraphSage and GAT GNNs. By promoting mini-batch randomness, COMET reduces the gap to in-memory training for GNN models by up to 80%. Moreover, COMET actually results in improved disk-based MRR for DistMult as well. Overall, COMET results in higher MRR compared to BETA for seven of the eight model/dataset combinations. Yet completely recovering the in-memory MRR for disk-based link prediction remains a challenge (e.g., GAT on FB15k-237 or GS on Wiki) and area of interest for future work.

While COMET allows for higher MRR compared to BETA, it also simultaneously allows for faster training. In particular, epoch time is reduced for DistMult—a less compute intensive model—which is IO bottlenecked. For

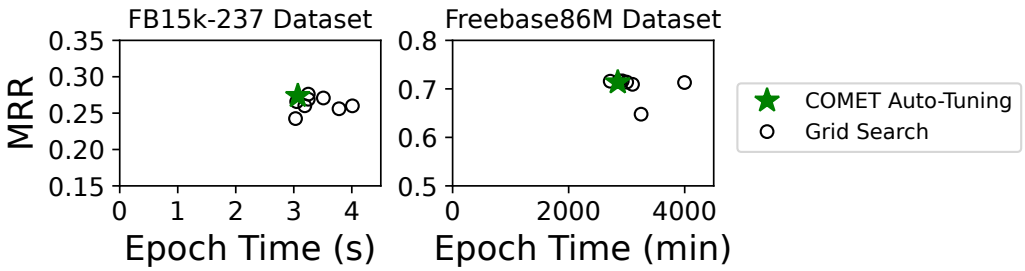


Figure 5.9: MRR and runtime for GraphSage GNN disk-based training with COMET using different hyperparameters. The auto-tuning rules used by MariusGNN are near-optimal.

example, COMET is 1.28× faster than BETA for DistMult on Freebase86M. In this setting, prefetching to overlapping IO with computation is needed for high throughput. While both COMET and BETA minimize IO, by decoupling mini batch generation from partition replacement and allowing for the deferred processing of training examples, COMET evenly distributes mini batches and IO across each epoch. This is in contrast to the greedy BETA policy which results in most mini batch processing occurring during the early part of each epoch, leaving little computation to overlap with IO for the latter part of training.

5.5.5 Evaluating COMET Auto-Tuning Rules

We evaluate the effectiveness of the parameter auto-tuning rules used in MariusGNN for disk-based training. To this end, we measure the runtime and MRR of COMET obtained when training uses the rules described in Section 5.4 and compare against the runtime and MRR obtained for each configuration in a hyperparameter scan. We use a GraphSage GNN and train on two datasets (FB15k-237 and Freebase86M). Results are shown in Figure 5.9. The auto-tuning rules used by MariusGNN lead to a hyperparameter setting that achieves near-optimal runtime and MRR simultaneously, eliminating the need for expensive hyperparameter search.

5.6 Summary

In this Chapter we focused on disk-based training in MariusGNN in which graph partitions are stored on disk and brought into CPU memory for training. We described partition replacement policies for swapping partitions between CPU memory and disk that allow MariusGNN to iterate over all training examples in the graph with minimal IO (e.g., BETA) and with as much randomness as possible (e.g., COMET). We also presented automated rules for setting the hyperparameters involved in disk-based training. Finally, we showed experimentally that disk-based training can reach comparable accuracy to training with the full graph in memory in existing systems, while training faster and for orders of magnitude less monetary cost—Disk-based training provides the cheapest option to scale to large graphs by using the entire memory hierarchy on a given machine instead of requiring additional machines to scale training.

6 SCALABLE DISTRIBUTED GNN TRAINING

We now focus on scalable, cost-effective, distributed GNN training over large graphs using common cloud offerings; our goal is to enable GNN training over multiple GPUs that are all fully utilized, to linearly reduce runtimes compared to single-GPU deployments and avoid paying for idle GPUs.

To achieve this goal, we introduce Armada, a new end-to-end system for large-scale distributed GNN training. Armada builds on the contributions of previous chapters (e.g., disk-based training (Section 5), mixed CPU-GPU pipelining (Section 3.1), DENSE multi-hop neighborhood sampling (Seciton 3.3)) and introduces a disaggregated architecture that supports scaling each part of the GNN workload (storage, mini batch preparation, and mini batch computation; see Sections 2.2 and 2.3) independently. We provide an overview of Armada in Section 6.1. Then in Section 6.2, we discuss important optimizations in Armada to minimize communication between and within disaggregated components and ensure that each layer in the architecture can scale independently without communication bottlenecks.

Finally, we show in Section 6.3, that Armada's disaggregated architecture allows for optimized resource utilization. Specifically, Armada can eliminate the bottleneck of mini batch preparation in existing state-of-the-art distributed systems (Section 2.3) and ensure that GPUs remain saturated with mini batches during training. In the same setting for which existing systems achieve only 2.3× and 1.7× speedup with eight instead of one GPU, Armada achieves a 7.5× speedup. This improved scaling leads to runtime improvements of up to 4.8× and monetary cost reductions up to 3.4× compared to state-of-the-art systems.

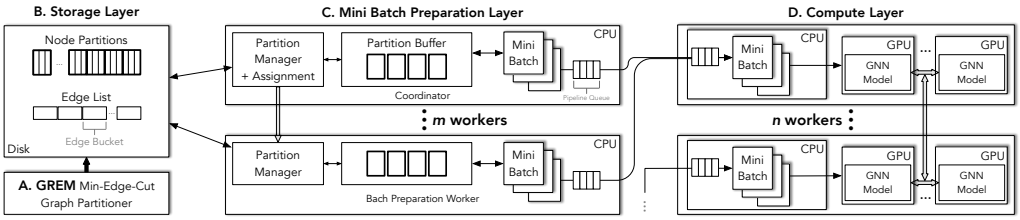


Figure 6.1: Armada system diagram. **A.** Graph data is partitioned using GREM (Section 4.1) and then **B.** stored on disk in the storage layer. **C.** A disaggregated mini batch preparation layer loads graph partitions into memory and prepares mini batches for workers in the compute layer. **D.** The compute workers process these mini batches on GPUs and periodically synchronize dense model parameters.

6.1 Overview: Armada's Disaggregated Architecture

Armada addresses the bottleneck of mini batch preparation in large-scale distributed GNN training (Section 2.3) by employing a disaggregated architecture (Figure 6.1) that separates graph storage from machines used for training, uses a set of cheap CPU-only machines to perform mini batch preparation, and uses a set of GPU machines to perform mini batch computation. Concretely, Armada consists of four components; we next discuss the responsibilities of each during training, before describing optimizations to reduce communication across the architecture in Section 6.2.

6.1.1 Disaggregated Architecture

GREM Partitioning Layer Given an input graph (stored on disk) in the form of an edge list and a set of features for each node, Armada partitions the nodes into a set of p physical partitions. To do so, Armada employs GREM, a min-edge-cut partitioning algorithm that can scale to massive graphs using a single commonly available machine (Chapter 4). GREM returns a label for each node specifying its partition, and saves this

information to disk. Armada then uses this information to rearrange the edge list and feature vectors in preparation for training according to the format expected by the storage layer.

Storage Layer The storage layer in Armada can store the partitioned graph using a variety of common backends (e.g., AWS S3, EBS, HDFS). As for disk-based training in MariusGNN (Section 5.1), we store the feature vectors for each node in a partition sequentially and group the graph edges into buckets: For a pair of node partitions (i, j), edge bucket (i, j) contains all edges from nodes in i to nodes in j. All edges in each bucket are then stored sequentially as a list. This format allows sets of partitions and the edges between them to be accessed using only sequential file reads/writes.

Mini Batch Preparation Layer Given the partitioned graph in the storage layer, the mini batch preparation layer is responsible for preparing mini batches for training. It consists of a distributed set of workers running on cheap CPU-only machines. Each worker reads a set of partitions (and the edges between them) from the storage layer into memory. The specific partition assignment for each machine is made by a designated worker, called the *coordinator*, according to a partition assignment policy (Section 6.2). After loading their assigned partitions, workers construct batches for training. Armada supports both 1) local construction, where machines prepare batches using only the data in their own CPU memory, leading to zero communication across machines, and 2) distributed construction, where multi-hop neighborhoods are sampled across the whole graph in the aggregate CPU memory of the layer, by communicating across workers as needed. To minimize communication between workers in the latter setting, Armada relies on the min-edge-cut partitioning returned by GREM and supports replicating high degree nodes on each worker (Section 6.2). Once batches are prepared, each worker pushes them to a specified (when configuring Armada) worker in the compute layer. To minimize the data transferred to the compute workers for each batch, Armada uses *mini batch grouping*—batches

destined for different GPUs on the same machine are grouped together for transmission to enable greater data compression (Section 6.2).

Because the mini batch preparation layer is disaggregated, the number of workers can be chosen independently from the other layers in Armada, allowing workers to be allocated based on specific workloads and deployment scenarios. In particular, for workloads bottlenecked by mini batch preparation, Armada can allocate enough CPU resources to ensure that all compute workers (GPUs) remain saturated with computation. Additionally, for massive graphs, Armada can rely on the storage layer for primary graph storage, rather than on the CPU memory of the batch preparation (or compute) layer, providing the option for lower cost, memory-efficient training deployments (as in disk-based training). In this case (i.e., the full graph does not fit in the aggregate CPU memory), Armada's coordinator instructs workers to swap graph partitions to and from the storage layer as needed to ensure that all partitions (and thus nodes) still appear in memory at least once per epoch (Section 6.2).

Compute Layer Armada's compute layer consists of a set of machines with attached GPU(s) and is responsible for model computation. Compute workers listen for mini batches from specified worker(s) in the mini batch preparation layer and then perform the GNN forward/backward pass on received batches in parallel. To minimize the amount of data sent for each mini batch, compute workers in Armada also maintain a *feature cache* of frequently accessed features in their local CPU memory (Section 6.2). GNN model parameters are replicated across GPUs and model gradients are synchronized before each parameter update. If applicable, gradients for learnable feature vectors are transferred to the CPU memory of the compute worker and then sent back to the corresponding batch worker so it can update its partitions in memory.

6.1.2 Mini-Batch Training in Armada

We now discuss one epoch of mini batch GNN training in Armada. Each training round (epoch) in Armada begins with all graph data stored in the storage layer. The coordinating batch construction worker then decides which partitions should be loaded into memory on each batch construction worker in order for training to proceed. By default, the coordinator ensures that all partitions are loaded into memory on at least one worker at least once per epoch, even when the full graph does not fit in the aggregate CPU memory of all batch workers (see Section 6.2). More specifically, as for disk-based training (Section 5.1), the coordinator defines a sequence $S^i = \{S^i_1, S^i_2, \dots\}$ for each batch construction worker i, with each S^i_j specifying a set of partitions. This sequence defines which partitions each batch construction worker should load into memory and in what order during training. The coordinator also defines $X^i = \{X^i_1, X^i_2, \dots\}$ for each worker i, with each X^i_j specifying a subset of training examples (e.g., graph nodes; see Section 2.2) in S^i_j, such that when S^i_j is in memory, all and only training examples in X^i_j are used to generate mini batches for training. Unique assignment ensures that training examples are processed by only one worker once per epoch.

Given these definitions, training proceeds as follows: each batch construction worker i loads S^i_1 into memory from the storage layer (when a set of c partitions are loaded, the edges in all c^2 edge buckets between these partitions are also loaded into memory, together inducing an *in-memory subgraph*) and then uses the training examples in X^i_1 to generate mini batches for training. Armada generates mini batches from X^i_1 in a random order and performs multi-hop neighborhood sampling over graph nodes and edges currently in CPU memory on worker i; that is, only over the nodes and corresponding edges in S^i_j, or over all graph nodes and edges currently in CPU memory across all workers. After neighborhood sampling, batch construction workers load the corresponding features for graph nodes in the mini batch before transferring the mini batch to the one of the workers in

the compute layer for computation. Each GPU on the compute workers then uses its local copy of the GNN model to compute the forward and backward pass for mini batches in parallel. After gradient synchronization, each GPU updates its dense GNN parameters for its respective GNN model.

Once training completes on X_1^i, the batch construction worker updates the partitions in memory from S_1^i to S_2^i by swapping partitions as necessary to and from the storage layer. One epoch completes when all batch construction workers have exhausted their sequences S and X.

6.2 Disaggregated Training - Implementation Details

Given a min-edge-cut partitioned graph, Armada employs a disaggregated architecture to enable cost-effective, distributed GNN training (Section 6.1). We now describe important design details that allow Armada to independently scale each layer and minimize communication in this architecture.

Partition Assignment to Batch Construction Workers We first discuss how Armada's designated worker, called the *coordinator*, assigns partitions in the storage layer to machines in the mini batch preparation layer in order to complete one round of training. To support scaling each of these two layers independently, we require that the algorithm has the following minimum guarantee: all partitions (and thus graph nodes) must appear in memory at least once per epoch, regardless of whether the full graph (all partitions) fits in the aggregate CPU memory of the batch construction workers or not[1] (this requirement is similar to those for disk-based training).

By default, the coordinator assigns partitions to workers as follows: First, the partitions are randomly split into disjoint subsets, one for each worker. This split occurs without data movement (only a mapping is maintained on

[1]If the full graph does not fit in the aggregate CPU memory of the batch construction workers, then neighborhood sampling cannot be done over the full graph, but can instead be done over the entire subgraph in the aggregate CPU memory of the layer.

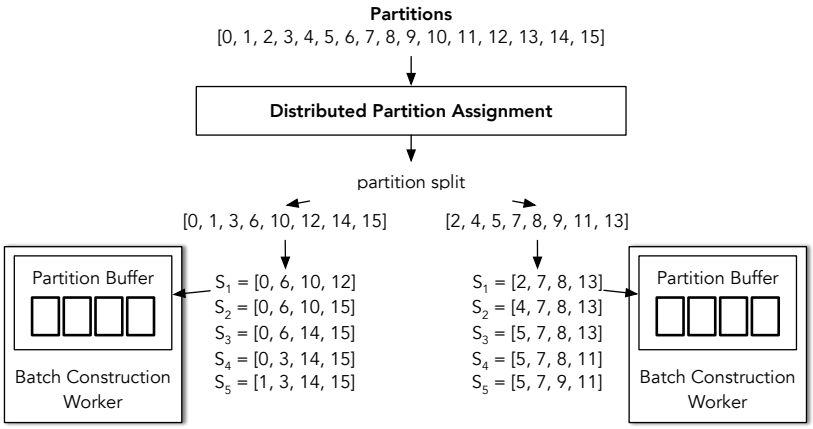


Figure 6.2: Example of the default assignment of graph partitions to batch construction workers during training in Armada. The example uses 16 partitions and two batch construction workers, each of which can fit four partitions in CPU memory at once.

the coordinator). Given an assigned set of partitions for each worker, the coordinator then decides the order in which each worker should load these partitions into memory. The coordinator first assigns each worker to load as many random partitions from its disjoint subset into memory as possible and then to swap any remaining partitions into memory one by one in a random order. We show a simple example of this algorithm in Figure 6.2. Note that this policy is equivalent to running the policy for disk-based node classification (Algorithm 7) on each machine's assigned subset of partitions. As for disk-based training (Section 5.3), to improve end-model accuracy when all partitions are unable to fit in CPU memory, Armada supports logical partitions and randomly assigning training examples to increase data shuffling within and across epochs.

The default policy, while simple, satisfies two desired properties: 1) opportunities for parallelism are maximized, as each batch worker operates on a disjoint set of partitions, and 2) each partition is read from the storage layer exactly once, ensuring all nodes appear in memory with minimal I/O between the two layers. Armada, however, can easily support other partition

assignment policies. In particular, to further minimize communication between workers due to cross-machine neighborhood sampling (in addition to min-edge-cut partitioning), Armada supports partial or even entire (memory permitting) feature replication across workers, as done in prior work (Cao et al., 2023; Kaler et al., 2023). In this case, the nodes to be replicated are placed in a special partition that is assigned to, and kept in memory, on *all* workers. Min-edge-cut partitioning and randomized partition assignment are then used on the remaining nodes.

Finally, we remark that while the default policy ensures that all nodes will appear in memory at least once each epoch, it does not ensure that all edges will appear in memory each epoch. With min-edge-cut partitioning, however, most of the graph edges have their source and destination node in the same partition, implying that most graph edges will appear in memory each epoch (all partitions are guaranteed to appear in memory). If a guarantee on edges is required, there are two options: 1) a partition assignment policy that ensures all partition *pairs* will appear together in memory at least once each epoch (and thus all edges will appear in memory) can be used (e.g., a distributed version of COMET (Section 5.3)), or 2) batch construction workers can be allocated as needed to store the full graph in memory.

Mini Batch Grouping We next discuss mini batch grouping, the first of two techniques used by Armada to minimize data transfer between batch preparation and compute workers and improve throughput.

Mini batch grouping applies when a batch construction worker is responsible for sending data to a compute worker that contains multiple GPUs. In this case, the batch construction worker must prepare and transfer one mini batch per GPU for each training iteration (such that each GPU can process a batch in parallel). Armada groups these mini batches into a global batch, as mini batches contained in a global batch may require the same nodes. This allows Armada to optimize feature loading and transfer: Armada loads and transfers the feature vectors for the unique nodes in a global batch only

once and copies them between GPUs as needed. For compute workers with 8 GPUs, we find mini batch grouping reduces batch preparation time by 1.13× and transfer time by 1.72×, increasing overall throughput by 1.15× on the common OGBN-Papers100M graph used in the experiments (Section 6.3).

Compute Worker Feature Caching Finally, Armada can further minimize communication between the batch preparation and compute layers by caching feature vectors for frequently accessed nodes locally on compute workers (in CPU memory). In this case, Armada needs to send only the non-cached features for each (global) batch between layers. Mini batches are then augmented as needed with the additional feature vectors once they are received in CPU memory by compute workers, before being transferred to the GPU(s) for training. Batch construction workers remain informed of the cache contents by listening to and acknowledging messages from the compute workers that describe planned updates. By default, compute workers use a simple LRU cache policy for eviction.

6.3 Results: Disaggregated Training in Armada

We evaluate Armada's disaggregated architecture on common large-scale graphs and compare against the popular state-of-the-art GNN systems DGL (version 1.1) (Wang et al., 2019; Zheng et al., 2020a), Salient++ (Kaler et al., 2023), and our own MariusGNN (Chapter 3 and 5). Our experiments show that disaggregation allows Armada to achieve scalable, cost-effective GNN training—we achieve a 7.5× speedup when using eight instead of one GPU when existing state-of-the-art systems yield 2.3× speedup at best.

6.3.1 Experimental Setup

We start by discussing the setup used in our experiments.

Armada and Baseline Details MariusGNN supports only single-GPU training, thus we modify it to support multi-GPU training using a standard

distributed data parallel architecture. We report results for two versions of Armada: 1) Armada and 2) Armada - Aggregated. The former uses the disaggregated architecture described throughout the paper, while the latter uses the CPUs on the GPU machine(s) used for training to prepare batches. The two versions allow us to directly evaluate the benefit of disaggregation. For partitioning, we use GREM with Armada and place one partition on each batch construction worker; for baselines, we use their default partitioning plus feature replication and caching (which are METIS-based).

Hardware Setup We partition and train all systems using AWS machines. To measure scalability, we use p3.16xlarge instances with eight NVIDIA V100 GPUs and vary how many GPUs are available to each system. These machines contain 64 vCPUs, 488 GiB of CPU memory, and 128 GiB of aggregate GPU memory. For Armada, we use additional m6a.16xlarge machines for mini batch preparation. These machines have 64 vCPUs and 256 GiB of CPU memory.

Datasets, Models, and Metrics We report results using Open Graph Benchmark (OGB) datasets (Hu et al., 2020, 2021); we use OGBN-Papers100M (111M nodes, 1.6B edges) and OGB-WikiKG90Mv2 (91M nodes, 601M edges) for large-scale studies, and OGBN-Products (2.5M nodes, 62M edges) plus FB15K-237 (Toutanova et al., 2015) (14.5K nodes, 272K edges) for microbenchmarks. We train a three-layer GraphSage GNN on these datasets with two different hidden sizes: 256 (*GraphSage-Small*) and 1024 (*GraphSage-Large*). The former allows us to run experiments using a data-bound model while the latter aims to represent a compute-bound model. In both cases, we use 30, 20, and 10 neighbors per layer sampled from both incoming and outgoing edges, as done in (Waleffe et al., 2023). For GNN training, we run for 10 epochs and measure runtime and monetary cost. We do not include the time to partition when reporting GNN training times. (we report partitioning time independently). We average experiments over three runs.

Hyperparameters We use the same hyperparameters for GNN model

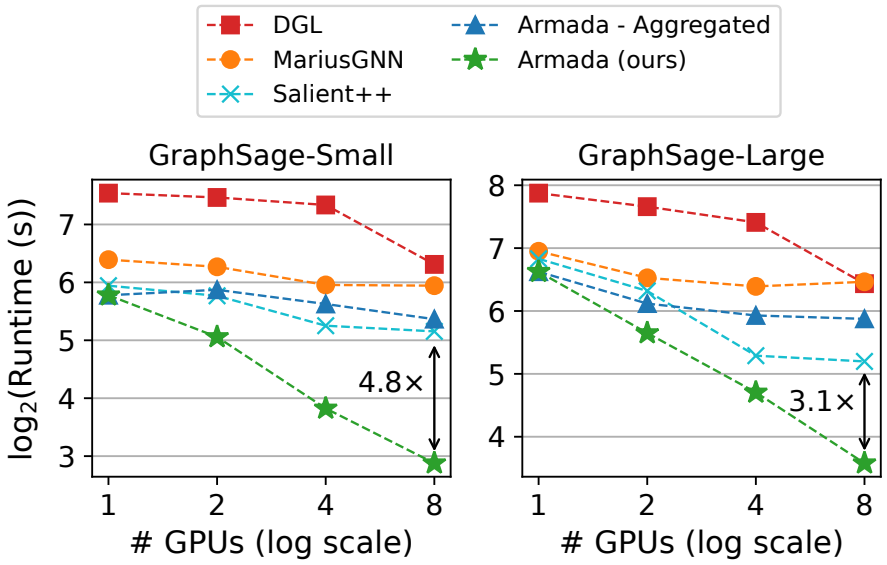


Figure 6.3: Epoch runtime versus number of GPUs for DGL, MariusGNN, Salient++, and Armada using two different GraphSage GNNs on the OGBN-Papers100M dataset. Disaggregation allows Armada to scale linearly with respect to the number of GPUs.

architecture and training across systems (e.g., model hidden dimension, number of neighbors, batch size, etc.). These hyperparameters are chosen based on values from prior works (Hu et al., 2020; Waleffe et al., 2023). For hyperparameters specific to the throughput of each system (e.g., the number neighborhood sampling workers), we manually tune them and select the best configuration.

6.3.2 GNN Training: System Comparisons

Given a partitioned graph, we now evaluate Armada's disaggregated architecture for GNN training. Runtime and cost per epoch for two models on OGBN-Papers100M with Armada and existing systems is shown in Table 6.1 and 6.2 . We plot the runtime versus the number of GPUs in Figure 6.3 to show the scaling of each system. For these experiments, all systems sample

Table 6.1: Runtime of DGL, MariusGNN+DDP, Salient++, and Armada on OGBN-Papers100M using a (left) GraphSage-Small and (right) GraphSage-Large GNN. With disaggregated mini batch preparation, Armada can scale training from one to eight GPUs while existing systems can not. Armada uses 0, 1, 2, and 4 disaggregated CPU-only batch construction workers for 1-, 2-, 4-, and 8-GPU training respectively. Relative improvement compared to single-GPU training for each system is shown in parentheses.

	GraphSage-Small: Epoch Runtime (s)				GraphSage-Large: Epoch Runtime (s)				Acc
# GPUs	1	2	4	8	1	2	4	8	-
DGL	186	177	161	79.4 (2.3×)	235	202	170	86.5 (2.7×)	67.4
M-GNN	84.0	77.1	62.1	61.5 (1.4×)	124	92.3	83.9	88.3 (1.4×)	67.1
Salient++	61.5	54.5	38.1	35.6 (1.7×)	114	79.8	39.1	36.7 (3.1×)	68.2
Armada	**54.9**	**33.3**	**14.2**	**7.35 (7.5×)**	**98.7**	**50.2**	**26.1**	**12.0 (8.2×)**	67.2

Table 6.2: Training cost of DGL, MariusGNN+DDP, Salient++, and Armada for the experiments in Table 6.1. The extra machines used for disaggregated mini batch preparation in Armada are cheap compared to the GPU-based machines used for model computation and do not prevent reductions in total training cost. Relative cost reduction compared to single-GPU training for each system is shown in parentheses.

	GraphSage-Small: Epoch Cost ($)				GraphSage-Large: Epoch Cost ($)			
# GPUs	1	2	4	8	1	2	4	8
DGL	1.26	1.20	1.09	0.54 (2.3×)	1.60	1.37	1.16	0.59 (2.7×)
MariusGNN	0.57	0.52	0.42	0.42 (1.4×)	0.84	0.63	0.57	0.60 (1.4×)
Salient++	0.42	0.37	0.26	0.24 (1.7×)	0.78	0.54	0.27	0.25 (3.1×)
Armada	**0.37**	**0.25**	**0.12**	**0.07 (5.3×)**	**0.67**	**0.38**	**0.22**	**0.12 (5.6×)**

neighbors across the whole graph, and thus reach similar accuracy (e.g., see Table 6.1 right). Next, we highlight key takeaways from these experiments before focusing on the results for existing systems and Armada in detail.

Key Takeaways Across experiments and GPU counts, Armada is the fastest and cheapest option; runtime and cost reductions are up to 4.8× and 3.4× versus existing systems. Armada is also the only system that can effectively scale training to multiple accelerators: the best existing system achieves at most a 3.1× speedup when moving from one to eight GPUs, yet Armada achieves an 8× speedup in the same setting.

Existing Systems We find that existing systems are unable to effectively scale GNN training across multiple compute resources (i.e., GPUs). For GraphSage-Small, the most scalable system (DGL) achieves only a 2.3× speedup when moving from one to eight GPUs. Salient++, the fastest baseline, achieves only 1.7× speedup. With a more compute-intensive model (GraphSage-Large), baseline systems are able to scale better—e.g., Salient++ achieves a 3.1× speedup—but they still suffer from sublinear speedups as a result of CPU-based mini batch preparation bottlenecks (Section 2.3).

Armada: The Benefit of Disaggregation Armada, however, achieves near-perfect scalability. For GraphSage-Small and -Large respectively, Armada achieves a 7.5× and 8× speedup when moving from one to eight GPUs. The key reason Armada can scale linearly is because of its disaggregated architecture. In particular, Armada can scale the number of CPU resources used to parallelize mini batch preparation independently from the number of GPUs used for training, ensuring that even as the number of GPUs increases, they all remain saturated with computation (Section 6.1). The effect of disaggregation is evident by comparing Armada to Armada - Aggregated in Figure 6.3. We also show the benefit of disaggregation in Figure 6.4; we report the epoch runtime in Armada when training GraphSage-Large on OGBN-Papers100M with eight GPUs and a varying number of disaggregated batch construction workers. Figure 6.4 shows that as the number of CPU resources used for mini batch preparation increases (by adding additional batch construction workers), the runtime decreases until the accelerators are fully saturated and the epoch runtime plateaus (as it becomes bottlenecked by GPU-based computation rather than CPU-based mini batch preparation).

Although the additional machines needed for batch preparation incur additional cost, these machines are cheaper than the GPU machines used for computation. Thus, Armada is still able to achieve total training cost reductions; we achieve a 5.3× and 5.6× reduction in cost when using eight instead of one GPU for GraphSage-Small and -Large respectively. We

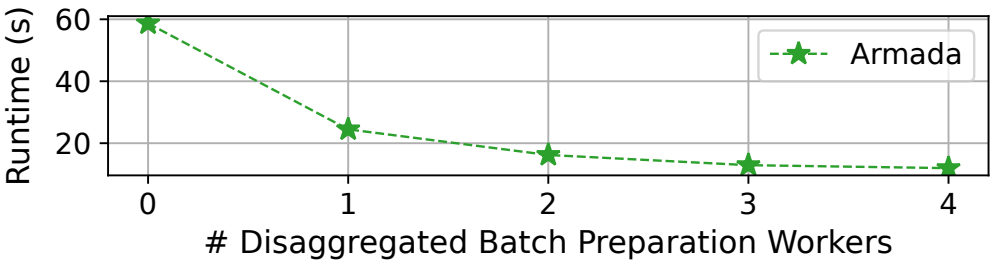


Figure 6.4: Epoch runtime in Armada when training GraphSage-Large on OGBN-Papers100M with eight GPUs and a varying number of disaggregated batch preparation workers; independently scaling these workers allows Armada to minimize runtime.

hypothesize that the cost of multi-GPU training could be further reduced for Armada by using cheaper AWS machines for mini batch preparation.

Summary Armada's disaggregated architecture allows resource utilization to be optimized in the presence of GNN workload imbalance, leading to linear scaling and cost-effective distributed GNN training over large-scale graphs in commonly available cloud hardware.

7 CONCLUSION

In this chapter, we conclude this dissertation by summarizing the key ideas.

The goal of this dissertation is to develop cost-effective, scalable GNN training over large graphs with billions of nodes and edges and potentially TBs of high-dimensional feature data. To move towards this goal, we proposed a sequence of algorithmic and systems contributions, each progressively building on the last.

First, in Chapter 2, we discussed necessary background on GNNs and described GNN mini-batch training. We highlighted that GNN training over large graphs is challenging because 1) the storage overhead for feature vectors associated with graph nodes (which can be learned) necessitates that these features are stored off device (e.g., in CPU memory) and 2) the node representations at internal GNN layers depend on feature vectors in nodes' multi-hop neighborhoods. We showed that these challenges can lead to low GPU utilization, sublinear scaling, and higher than necessary training cost and runtime over massive graphs as a result of data movement overheads arising from the need to transfer feature vectors to and from GPU memory and CPU bottlenecks arising from multi-hop neighborhood sampling.

Then, in Chapter 3, we presented techniques to maximize GPU utilization during mixed CPU-GPU mini-batch training on a single machine. We presented a pipelined architecture to overlap data preparation and data movement with computation. We showed, however, that this pipelined architecture can leaded to reduce model accuracy as a result of asynchrony; thus, we introduced OAC, a new policy for pipelined training that ensures equivalence to synchronous one by one execution, even as multiple mini batches are prepared and transferred in parallel. To minimize the overhead of CPU-based multi-hop neighborhood sampling in this setting, we also introduced DENSE, a new data structure and algorithm for minimizing the redundant computation and data access present when constructing multi-hop

neighborhoods. We implemented these contributions in MariusGNN and showed that they enable sampling that is up to 14× faster and end-to-end training that is up to 4× faster than existing state-of-the-art systems, even as these systems use four GPUs and MariusGNN uses only one.

Next, in Chapter 4, we focused on efficient min-edge-cut graph partitioning as a prerequisite to enable training GNNs over large graphs which exceed the CPU memory capacity of a single machine. We introduced GREM, a novel memory-efficient min-edge-cut partitioning algorithm that builds on existing streaming greedy approaches, but continuously refines prior vertex assignments rather than freezing them after an initial greedy selection. Compared to METIS, widely used by existing state-of-the-art systems for GNN training, we showed that GREM can reach comparable quality but with 8-65× less memory and 8-46× faster.

Given a partitioned graph, in Chapter 5, we showed how to use cheap and high capacity disk storage to scale GNN training to graphs which don't fit in CPU memory. We developed partition replacement policies for loading and swapping graph partitions between disk and CPU memory such that 1) the entire graph appears in memory with near-minimal IO and 2) training in this setting leads to models with high accuracy. We implemented these policies and disk-based training in MariusGNN. We showed that disk-based training can reach the same level of accuracy up to 8× faster than existing systems and enables scaling to large graphs that do not fit in memory using just a single, cheap machine, leading to up to 64× monetary cost reductions compared to existing systems which require paying for expensive machines with additional CPU memory.

Finally, in Chapter 6, we focused on scalable distributed GNN training. We introduced Armada, a new system for multi-GPU and multi-machine training. We showed that by leveraging a disaggregated architecture, Armada can independently allocate graph storage, CPU resources used for GNN neighborhood sampling, and GPU resources use for model computation in

order to improve efficiency and ensure that expensive GPUs remain saturated with computation. We showed that this design allows Armada to achieve a 7.5× speedup when using eight instead of one GPU in the same setting for which the best existing system achieves only a 2.3× speedup.

Overall, this thesis highlights the promise of new algorithms and systems to democratize large-scale GNN training and the need to progressively optimize ML systems, from single-GPU implementations to multi-GPU deployments, in order to minimize cost and maximize scalability. To this end, many of the techniques and learnings included in this work can be generalized beyond GNN training: mixed CPU-GPU pipelining and OAC can be applied to any ML system which uses GPUs for computation but stores learnable parameters off device, and GREM can be used to reduce the computational requirements of min-edge-cut graph partitioning across a diverse array of applications. Finally, the key ideas behind DENSE (to minimize redundant computation), disk-based training (to leverage the entire memory hierarchy with minimal IO), and Armada (disaggregation to independently scale each part of the workload) can also be applied to other settings in ML and computer science more generally. We leave such research and exploration to future work.

REFERENCES

Abbas, Zainab, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming graph partitioning: an experimental study. *Proceedings of the VLDB Endowment* 11(11):1590–1603.

Akyildiz, Taha Atahan, Amro Alabsi Aljundi, and Kamer Kaya. 2020. Gosh: Embedding big graphs on small hardware. In *49th international conference on parallel processing - icpp*. ICPP '20, New York, NY, USA: Association for Computing Machinery.

Alistarh, Dan, Jennifer Iglesias, and Milan Vojnovic. 2015. Streaming min-max hypergraph partitioning. *Advances in Neural Information Processing Systems* 28.

Andreev, Konstantin, and Harald Räcke. 2004. Balanced graph partitioning. In *Proceedings of the sixteenth annual acm symposium on parallelism in algorithms and architectures*, 120–124.

Belady, Laszlo A. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5(2):78–101.

Brohee, Sylvain, and Jacques Van Helden. 2006. Evaluation of clustering algorithms for protein-protein interaction networks. *BMC bioinformatics* 7(1):488.

Brown, Tom, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33:1877–1901.

Cao, Kaidi, Rui Deng, Shirley Wu, Edward W Huang, Karthik Subbian, and Jure Leskovec. 2023. Communication-free distributed gnn training with vertex cut. *arXiv preprint arXiv:2308.03209*.

Chami, Ines, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. 2021. Machine learning on graphs: A model and comprehensive taxonomy. 2005.03675.

Chen, Jie, Tengfei Ma, and Cao Xiao. 2018. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*.

Chiang, Wei-Lin, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

Ching, Avery, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8(12):1804–1815.

De Sa, Christopher M. 2020. Random reshuffling is not always better. In *Advances in neural information processing systems*, ed. H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, vol. 33, 5957–5967. Curran Associates, Inc.

Derrow-Pinion, Austin, Jennifer She, David Wong, Oliver Lange, Todd Hester, Luis Perez, Marc Nunkesser, Seongjae Lee, Xueying Guo, Brett Wiltshire, et al. 2021. Eta prediction with graph neural networks in google maps. In *Proceedings of the 30th acm international conference on information & knowledge management*, 3767–3776.

Dong, Jialin, Da Zheng, Lin F Yang, and George Karypis. 2021. Global neighbor sampling for mixed cpu-gpu training on giant graphs. In *27th acm sigkdd conference on knowledge discovery and data mining, kdd 2021*, 289–299. Association for Computing Machinery.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Min-

derer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*.

Fairchild, Kimm, Steven E Poltrock, and George W Furnas. 1988. Graphic representations of large knowledge bases. *Cognitive science and its applications for human-computer interaction* 201.

Faraj, Marcelo Fonseca, and Christian Schulz. 2022. Buffered streaming graph partitioning. *ACM Journal of Experimental Algorithmics* 27:1–26.

Fey, Matthias, and Jan Eric Lenssen. 2019. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*.

Gandhi, Swapnil, and Anand Padmanabha Iyer. 2021. P3: Distributed deep graph learning at scale. In *15th USENIX symposium on operating systems design and implementation (OSDI 21)*, 551–568. USENIX Association.

Google. 2018. Freebase data dumps. https://developers.google.com/freebase.

Graur, Dan, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. 2022. Cachew: Machine learning input data processing as a service. In *2022 usenix annual technical conference (usenix atc 22)*, 689–706.

Hamilton, Will, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, ed. I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, vol. 30. Curran Associates, Inc.

Haochen, Jeff, and Suvrit Sra. 2019. Random shuffling beats SGD after finite epochs. In *Proceedings of the 36th international conference on machine*

learning, ed. Kamalika Chaudhuri and Ruslan Salakhutdinov, vol. 97 of *Proceedings of Machine Learning Research*, 2624–2633. PMLR.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 770–778.

Hellerstein, Joseph M, Michael Stonebraker, and James Hamilton. 2007. *Architecture of a database system*. Now Publishers Inc.

Hersbach, Hans, Bill Bell, Paul Berrisford, Shoji Hirahara, András Horányi, Joaquín Muñoz-Sabater, Julien Nicolas, Carole Peubey, Raluca Radu, Dinand Schepers, et al. 2020. The era5 global reanalysis. *Quarterly Journal of the Royal Meteorological Society* 146(730):1999–2049.

Hilbert, David. 1891. Über die stetige abbildung einer line auf ein flächenstück. *Mathematische Annalen* 38(3):459–460.

Hofmann, Thomas, Aurelien Lucchi, Simon Lacoste-Julien, and Brian McWilliams. 2015. Variance reduced stochastic gradient descent with neighbors. *Advances in Neural Information Processing Systems* 28.

Hu, Weihua, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A large-scale challenge for machine learning on graphs. In *Thirty-fifth conference on neural information processing systems datasets and benchmarks track (round 2)*.

Hu, Weihua, 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* 33:22118–22133.

Huang, Gao, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 4700–4708.

Ilyas, Ihab F, Theodoros Rekatsinas, Vishnu Konda, Jeffrey Pound, Xiaoguang Qi, and Mohamed Soliman. 2022. Saga: A platform for continuous construction and serving of knowledge at scale. In *Sigmod 2022*.

Jain, Sachin, Chaitanya Swamy, and K Balaji. 1998. Greedy algorithms for k-way graph partitioning. In *the 6th international conference on advanced computing*, 100. Citeseer.

Jangda, Abhinav, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using gpus. In *Proceedings of the sixteenth european conference on computer systems*, 311–326.

Jia, Zhihao, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. In *Proceedings of machine learning and systems*, ed. I. Dhillon, D. Papailiopoulos, and V. Sze, vol. 2, 187–198.

Jin, Xin, Zhihao Bai, Zhen Zhang, Yibo Zhu, Yinmin Zhong, and Xuanzhe Liu. 2024. Distmind: Efficient resource disaggregation for deep learning workloads. *IEEE/ACM Transactions on Networking*.

Jumper, John, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. 2021. Highly accurate protein structure prediction with alphafold. *Nature* 596(7873):583–589.

Kaler, Tim, Alexandros Iliopoulos, Philip Murzynowski, Tao Schardl, Charles E Leiserson, and Jie Chen. 2023. Communication-efficient graph neural networks with probabilistic neighborhood expansion analysis and caching. *Proceedings of Machine Learning and Systems* 5.

Kaler, Tim, 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. *Proceedings of Machine Learning and Systems* 4:172–189.

Karypis, George, and Vipin Kumar. 1997. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices.

Kipf, Thomas N, and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Kloeckner, Andreas, Matt Wala, Nathan Hartland, Albert Danial, Fritz Obermeyer, Cathy Wu, and Christoph Gohlke. 2022. PyMetis.

Kung, Hsiang-Tsung, and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6(2):213–226.

Kwak, Haewoon, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on world wide web*, 591–600. WWW '10.

Kyrola, Aapo, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale graph computation on just a PC. In *10th usenix symposium on operating systems design and implementation (osdi 12)*, 31–46. Hollywood, CA: USENIX Association.

Lam, Remi, Alvaro Sanchez-Gonzalez, Matthew Willson, Peter Wirnsberger, Meire Fortunato, Alexander Pritzel, Suman Ravuri, Timo Ewalds, Ferran Alet, Zach Eaton-Rosen, et al. 2022. Graphcast: Learning skillful medium-range global weather forecasting. *arXiv preprint arXiv:2212.12794*.

Lerer, Adam, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-biggraph: A large scale graph embedding system. *Proceedings of Machine Learning and Systems* 1:120–131.

Leskovec, Jure, and Andrej Krevl. 2014. Snap datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

Lin, Haiyang, Mingyu Yan, Xiaochun Ye, Dongrui Fan, Shirui Pan, Wenguang Chen, and Yuan Xie. 2023. A comprehensive survey on distributed training of graph neural networks. *Proceedings of the IEEE*.

Lin, Zhiqi, 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*, 401–415.

Liu, Tianfeng, 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)*, 103–118.

Maass, Steffen, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the twelfth european conference on computer systems*, 527–543. EuroSys '17, New York, NY, USA: Association for Computing Machinery.

McSherry, Frank, Michael Isard, and Derek G Murray. 2015. Scalability! but at what {COST}? In *15th workshop on hot topics in operating systems (hotos {XV})*.

Merkel, Nikolai, Daniel Stoll, Ruben Mayer, and Hans-Arno Jacobsen. 2023. An experimental comparison of partitioning strategies for distributed graph neural network training. *arXiv preprint arXiv:2308.15602*.

Meusel, Robert, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. 2014. Web data commons - hyperlink graphs. http://webdatacommons.org/hyperlinkgraph/.

Min, Seung Won, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021a. Large graph convolutional network training with gpu-oriented data communication architecture. *Proc. VLDB Endow.* 14(11):2087–2100.

Min, Seung Won, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen mei Hwu. 2021b. Pytorch-direct: Enabling gpu centric data access for very large graph neural network training with irregular accesses. `2101.07956`.

Mohoney, Jason, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Marius: Learning massive graph embeddings on a single machine. In *15th usenix symposium on operating systems design and implementation (osdi 21)*, 533–549.

Nickolls, John, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue* 6(2): 40–53.

Niu, Feng, Benjamin Recht, Christopher Ré, and Stephen J Wright. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *arXiv preprint arXiv:1106.5730*.

OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Patwary, Md Anwarul Kaium, Saurabh Garg, and Byeong Kang. 2019. Window-based streaming graph partitioning algorithm. In *Proceedings of the australasian computer science week multiconference*, 1–10.

Petroni, Fabio, 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*, 243–252.

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1(8):9.

Ramakrishnan, Raghu, Johannes Gehrke, and Johannes Gehrke. 2003. *Database management systems*, vol. 3. McGraw-Hill New York.

Ramezani, Morteza, Weilin Cong, Mehrdad Mahdavi, Anand Sivasubramaniam, and Mahmut Kandemir. 2020. Gcn meets gpu: Decoupling "when to sample"from "how to sample". In *Advances in neural information processing systems*, ed. H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, vol. 33, 18482–18492. Curran Associates, Inc.

Shao, Yingxia, Hongzheng Li, Xizhi Gu, Hongbo Yin, Yawen Li, Xupeng Miao, Wentao Zhang, Bin Cui, and Lei Chen. 2024. Distributed graph neural network training: A survey. *ACM Computing Surveys* 56(8):1–39.

Stanton, Isabelle. 2014. Streaming balanced graph partitioning algorithms for random graphs. In *Proceedings of the twenty-fifth annual acm-siam symposium on discrete algorithms*, 1287–1301. SIAM.

Stanton, Isabelle, 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*, 1222–1230.

Sun, Ding, Zhen Huang, Dongsheng Li, Xiangyu Ye, and Yilin Wang. 2021. Improved partitioning graph embedding framework for small cluster. In *Knowledge science, engineering and management*, ed. Han Qiu, Cheng

Zhang, Zongming Fei, Meikang Qiu, and Sun-Yuan Kung, 203–215. Cham: Springer International Publishing.

Thorpe, John, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th usenix symposium on operating systems design and implementation (osdi 21)*, 495–514. USENIX Association.

Toutanova, Kristina, Danqi Chen, Patrick Pantel, Hoifung Poon, Pallavi Choudhury, and Michael Gamon. 2015. Representing text for joint embedding of text and knowledge bases. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, 1499–1509. Lisbon, Portugal: Association for Computational Linguistics.

Tsourakakis, Charalampos, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th acm international conference on web search and data mining*, 333–342.

Tu, Stephen, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the twenty-fourth acm symposium on operating systems principles*, 18–32.

Veličković, Petar, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *International conference on learning representations*.

Waleffe, Roger, and Jason Mohoney. 2024. Optimistic asynchrony control: Achieving synchronous convergence with asynchronous throughput for embedding model training. In *2nd workshop on advancing neural network*

training: Computational efficiency, scalability, and resource optimization (want@ icml 2024).

Waleffe, Roger, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2023. Mariusgnn: Resource-efficient out-of-core training of graph neural networks. In *Proceedings of the eighteenth european conference on computer systems*, 144–161.

Waleffe, Roger, Devesh Sarda, Jason Mohoney, Emmanouil-Vasileios Vlatakis-Gkaragkounis, Theodoros Rekatsinas, and Shivaram Venkataraman. 2024. Armada: Memory-efficient distributed training of large-scale graph neural networks. In *Under submission*.

Wang, Lei, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. 2021a. Flexgraph: a flexible and efficient distributed framework for gnn training. In *Proceedings of the sixteenth european conference on computer systems*, 67–82.

Wang, Minjie, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR* abs/1909.01315.

Wang, Yuke, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021b. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In *15th usenix symposium on operating systems design and implementation (osdi 21)*, 515–531. USENIX Association.

Wu, Zonghan, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* 32(1):4–24.

Yang, Bishan, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2014. Embedding entities and relations for learning and inference in knowledge bases. *arXiv preprint arXiv:1412.6575*.

Yang, Jianbang, 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.

Zafarani, Reza, Mohammad Ali Abbasi, and Huan Liu. 2014. *Social media mining: an introduction*. Cambridge University Press.

Zeng, Hanqing, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. Graphsaint: Graph sampling based inductive learning method. In *International conference on learning representations*.

Zhang, Wei, Yong Chen, and Dong Dai. 2018. Akin: A streaming graph partitioning algorithm for distributed graph storage systems. In *2018 18th ieee/acm international symposium on cluster, cloud and grid computing (ccgrid)*, 183–192. IEEE.

Zheng, D., C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis. 2020a. Distdgl: Distributed graph neural network training for billion-scale graphs. In *2020 ieee/acm 10th workshop on irregular applications: Architectures and algorithms (ia3)*, 36–44. Los Alamitos, CA, USA: IEEE Computer Society.

Zheng, Da, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. 2020b. Dgl-ke: Training knowledge graph embeddings at scale. In *Proceedings of the 43rd international acm sigir conference on research and development in information retrieval*, 739–748.

Zheng, Da, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In *Proceedings of the 28th acm sigkdd conference on knowledge discovery and data mining*, 4582–4591.

Zhu, Rong, 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):2094–2105.

Zou, Difan, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-dependent importance sampling for training deep and large graph convolutional networks. *Advances in neural information processing systems* 32.