MariusGNN: Resource-Efficient Out-of-Core Training of Graph Neural Networks

Roger Waleffe*
University of Wisconsin–Madison

Jason Mohoney
Universtity of Wisconsin–Madison

Theodoros Rekatsinas
ETH Zürich

Shivaram Venkataraman
University of Wisconsin–Madison

Abstract

We study training of Graph Neural Networks (GNNs) for large-scale graphs. We revisit the premise of using distributed training for billion-scale graphs and show that for graphs that fit in main memory or the SSD of a single machine, out-of-core pipelined training with a single GPU can outperform state-of-the-art (SoTA) multi-GPU solutions. We introduce MariusGNN, the first system that utilizes the entire storage hierarchy—including disk—for GNN training. MariusGNN introduces a series of data organization and algorithmic contributions that 1) minimize the end-to-end time required for training and 2) ensure that models learned with disk-based training exhibit accuracy similar to those fully trained in memory. We evaluate MariusGNN against SoTA systems for learning GNN models and find that single-GPU training in MariusGNN achieves the same level of accuracy up to 8× faster than multi-GPU training in these systems, thus, introducing an order of magnitude monetary cost reduction. MariusGNN is open-sourced at www.marius-project.org.

---

*Corresponding author: waleffe@wisc.edu

Table 1. Many of the large graphs that GNNs are applied to can fit in main memory or the disk of a single commodity machine (e.g., AWS P3 GPU instances range from 61-488GB of CPU memory and contain up to 16TB of disk storage). The first five graphs are publicly available.

Graph	Nodes	Edges	Feat. Dim	Memory (GB)		
				Edges	Feat.	Tot.
Papers100M [17]	111M	1.62B	128	13	57	70
Mag240M-Cites [16]	122M	1.30B	768	10	375	385
Freebase86M [12]	86M	338M	100	4	69	73
WikiKG90Mv2 [16]	91M	601M	100	7	73	80
Hyperlink 2012 [30]	3.5B	128B	50	2k	1.4k	3.4k
Facebook15 [6]	1.4B	1T	100	8k	560	8.5k

1 Introduction

Graph Neural Networks (GNNs) have emerged as an important method for machine learning (ML) [3]. A diverse array of models [2, 13, 41, 46] have been introduced to apply ML over protein structures [37], social networks [9], and knowledge graphs [34]. These models achieve state-of-the-art (SoTA) accuracy for *node classification* and *link prediction* [22, 36, 49] by encoding data dependencies that arise in graphs.

However, resource-efficient GNN training over large-scale graphs has been highlighted as a major challenge in the literature [3, 11, 39, 52]. Graphs used in production settings contain millions of nodes and billions of edges. In addition, their nodes and edges are associated with feature vectors that form the inputs to GNNs. This data can require hundreds of GBs of storage [18, 28] (Table 1), which exceeds the capacity of GPU accelerators (e.g., 16GB on NVIDIA V100s). Moreover, the node representations at internal GNN layers depend on the nodes' multi-hop neighborhood, thus, a GNN dataflow graph can scale exponentially as more layers are added in a GNN. These two characteristics—graph storage sizes and GNN neighborhood dependencies—necessitate the use of mini-batch training coupled with multi-hop neighbor sampling for learning GNNs over large-scale inputs [8, 31].

To perform training over large graphs, SoTA systems, such as Deep Graph Library (DGL) [43] and PyTorch Geometric (PyG) [10], rely on CPU memory for graph storage and use distributed mini-batch training either over multiple GPUs or multiple machines. However, distributed solutions introduce deployment and maintenance overheads [29] and in

many cases obtain sub-linear speedups [52]. For example, we find that DGL yields only a 2.2× speedup when using eight instead of one GPU (see Section 7.2). As a result, current distributed systems can incur increased monetary costs due to underutilization of allocated hardware.

We revisit GNN training and ask: *When is distributed GNN training needed?* We argue that for graphs that fit in memory or the SSD of a single machine (Table 1), *out-of-core pipelined training on a single machine can outperform SoTA distributed systems.* We introduce MariusGNN, a system that utilizes the entire memory hierarchy—including disk—for GNN training over large-scale graphs. While prior work including GraphChi [23], Mosaic [28], and COST [29], have proposed using a single machine with disk for processing large graphs, they focus on graph analytics workloads such as PageRank. More recent graph ML systems such as Marius [33] and PyTorch BigGraph [24] utilize disk, but do not contain multi-hop neighborhood samplers that are necessary for GNNs and thus only support specialized link prediction models for knowledge graph embeddings. In contrast, MariusGNN supports disk-based training of general k-hop sampling-based GNN models (subsuming prior graph ML systems) for both node classification and link prediction.

With experiments over four datasets using popular GNN architectures, we show that MariusGNN's disk-based, single-GPU training can be 8× faster than eight-GPU deployments of SoTA systems. This improvement yields monetary cost reductions of an order of magnitude. We find that for graphs where SoTA 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 4). Moreover, we show that single-GPU 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).* 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.* Achieving these results required innovation and careful engineering along two dimensions: 1) optimized sampling to minimize the exponential costs of mini batch construction and 2) novel policies for transferring and processing training data that can maximize disk throughput while ensuring model accuracy is comparable to full in-memory training.

Despite recent efforts for efficient sampling when constructing a mini batch [4, 5, 35, 48, 54], we find that existing sampling algorithms in SoTA systems bottleneck pipelined training and lead to the underutilization of GPUs. For example, using PyG's CPU-based neighborhood sampler for a three-layer GNN on a graph with 100M nodes takes 1200ms while the GPU-based operations for training take only 170ms (Section 7.4). These timings arise because current approaches for sampling multi-hop neighborhoods resample one-hop

neighbors for graph nodes multiple times while constructing a single GNN dataflow graph (e.g., Figure 1). Due to the graph structure, the same nodes can appear repeatedly both within a layer and across different layers of the dataflow graph. When a node appears multiple times in the same layer, existing systems (e.g., DGL [43]) only sample the one-hop neighbors of this node once, yet when a node appears in separate layers, the one-hop neighbors of this node are resampled each time (e.g., node A in Figure 1). Such resampling leads to redundant computation and data transfers and can limit pipeline throughput when training multi-layer GNNs.

In MariusGNN, we introduce a new data structure to minimize redundant computation and data transfers during sampling (Section 4). We term the new data structure *DENSE*, as it uses a Delta Encoding of Neighborhood SamplEs for nodes in successive GNN layers. DENSE allows us to reuse sampled one-hop neighbors to construct the input of different GNN layers and enables in-CPU sampling that is up to 14× faster than SoTA systems (Section 7.4). 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.

Disk-based execution in MariusGNN requires that the graph be split into partitions and subsets of these partitions be transferred to CPU memory for mini-batch training. We study partition replacement policies and mini batch construction policies for both link prediction and node classification.

For link prediction, we show that prior SoTA partition replacement policies, which greedily minimize IO, lead to biased models and harm accuracy when training GNNs. For example, when we implement the Buffer-aware Edge Traversal Algorithm (BETA) from prior work [33] in MariusGNN, we find that the accuracy of learned GNN models drops by up to 16% on common benchmarks when compared to training with the entire graph in main memory. We study the reason behind this drop (Section 5.1) and find that existing policies lead to consecutive mini batches containing correlated training examples, a property that conflicts with the independently distributed assumption of ML training data.

To close the accuracy gap between in-memory and out-of-core training while minimizing disk IO, we introduce a new policy termed COMET (COrrelation Minimizing Edge Traversal). First, COMET separates the granularity of data storage and access from data transfer by utilizing two levels of partitioning (physical and logical partitions). Second, it decouples mini batch generation from partition replacement, allowing for randomized deferred processing of training examples, thus, further shuffling the order in which training examples are processed. To maximize throughput and accuracy, we provide automated rules for setting COMET's hyperparameters. Using COMET, we are able to reduce the gap between the link prediction accuracy of disk-based training and that of training with the full graph in memory by up to 80% compared to prior work (Section 7.5).
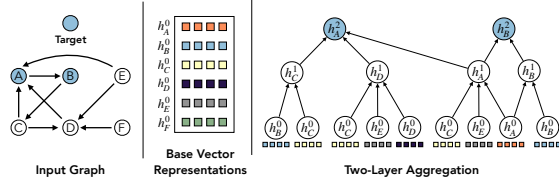
Figure 1. Example two-layer GNN aggregation for nodes {A, B} using a sample of their two-hop incoming neighborhood.

For node classification, we find that simple data transfer policies can be effective. In large-scale graphs [16, 17], it is often the case that the set of nodes labeled for training is only 1-10% of all graph vertices. Based on this information, we show that a policy that caches all labeled nodes in main memory and transfers random partitions from disk at the beginning of every epoch yields high-accuracy models for all the benchmark graphs we consider (Section 7.5).

2 Preliminaries on Training GNNs

We focus on training GNNs for node classification and link prediction. Given a graph $G = (V, E)$ with nodes V and edges E, node classification assigns a class label to a node $v \in V$ given its local information (e.g., local features of the node) and information from its neighborhood in G. Link prediction considers a pair of nodes $(v_1, v_2) \in V \times V$ and predicts if the two nodes should be connected via an edge. Link prediction also uses the local features of nodes v_1 and v_2 and information from their neighborhoods to obtain a prediction.

Local information for each node in G is modeled with a *base vector representation* that encodes local node features. This representation can either be fixed (e.g., the conference of a paper in a citation graph) or *learned* (e.g., a learned representation of the type of an entity node in a knowledge graph). For a node $v \in V$, we denote this base vector representation h_v^0. Base representations are stored in a *lookup table*.

To combine local information with neighborhood information, GNN layers aggregate the local representation of each node with the representations of its neighbors. Given a node v, the k−th layer of a multi-layer GNN represents node v with a vector h_v^k that is defined recursively as $h_v^k = AGG(h_v^{(k-1)}, \{h_u^{(k-1)} : u \in N_v\})$ where AGG is an aggregation function with learnable parameters (e.g., a weighted sum) and N_v denotes the nodes in the one-hop neighborhood of v; The k-th layer aggregates information from the k-hop neighborhood of node v. An example is shown in Figure 1. To perform node classification, h_v^k can be fed into a fully-connected and softmax layer, 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* [3]) to obtain the prediction.

GNN training is typically performed using mini-batch gradient descent. For node classification, training examples correspond to *target nodes* together with the true class label for each node. For link prediction, training examples are pairs of target nodes with binary labels (i.e., edge or not). In

a k-layer GNN the target nodes correspond to the nodes in the last (k-th) layer. To compute its output, a k-layer GNN requires that a mini batch include the base representation of all nodes in the k-hop neighborhood of the target nodes. However, k-hop neighborhoods can grow exponentially and the storage required for the base representations can exceed GPU memory if the entire neighborhood of each target node is used. For this reason, *sampling* of the k-hop neighborhood, typically performed in CPU, is required.

Given a mini batch, a *forward pass* is performed on the GPU to compute h_v^k and the loss and gradients needed to update the model parameters. The updates for learned base representations are also transferred to the CPU and the corresponding entries of the lookup table are updated. The time for sampling the k-hop neighborhood can dominate the time for the forward pass, thus leading to GPU underutilization [21] and lower training throughput.

For large-scale graphs, the storage for the graph structure and the base representations of the nodes may exceed CPU memory (Table 1). In this case, mini batch construction needs to be performed over a subgraph of the original graph that is loaded in main memory. This operation requires the original graph be split into *partitions*. These partitions can either be loaded in different machines to perform distributed training or stored on disk and brought into main memory to perform training over the corresponding subgraph. MariusGNN uses disk-based training. We opt for *pipelined disk-based training* [33] where a buffer is used to hold partitions in main memory. In-buffer partitions are used to construct mini batches. To achieve high accuracy, we need to iterate over all available training examples. To this end, pipelined disk-based training requires swapping partitions between the buffer and disk according to a *partition replacement policy*. For high throughput this policy should also minimize IO. Prior work on the Marius system [33] introduced the SoTA greedy BETA policy which achieved near minimal IO when iterating over all graph edges. While BETA allowed Marius to train *decoder-only* knowledge graph embedding models for link prediction, Marius does not support multi-hop neighborhood sampling, and thus cannot train k-layer GNNs for $k > 0$, or node classification tasks. Moreover, when BETA is implemented in MariusGNN and applied to k-layer GNN training it leads to reduced model accuracy.

3 Training GNNs in MariusGNN

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

MariusGNN represents a graph as an edge list. In addition, base vector representations 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
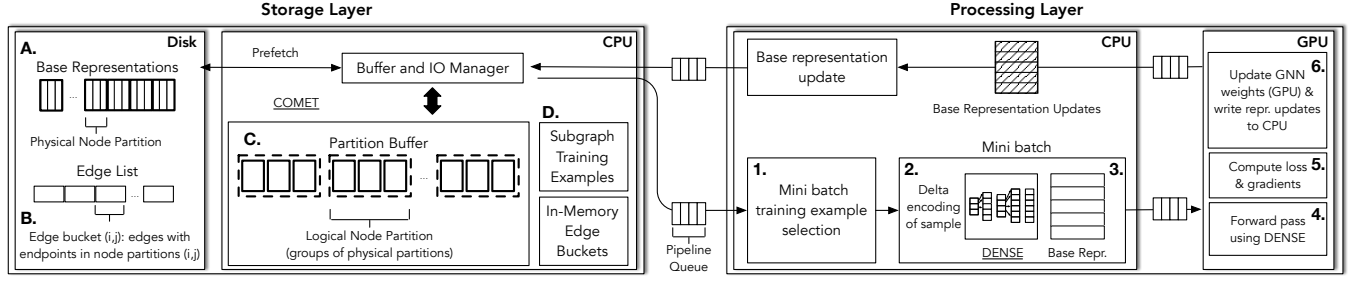
Figure 2. MariusGNN System Diagram. The lifecycle of a mini batch consists of Steps 1-6. The storage layer replacement policy (e.g., COMET) periodically updates graph partitions in memory during training (Steps A-D).

partition i and a destination node in partition j. Edges in each edge bucket are stored sequentially on disk.

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 5.1). Given the logical partitions, let S_i be a set of logical partitions such that all corresponding physical partitions fit in memory. 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.

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. Thus, for link prediction, pairs of node partitions that contain valid edges 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 policy to generate S and X. COMET ensures that all valid edges of 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 5.1. For node classification, MariusGNN uses a simple policy that caches all nodes used as training examples in memory to achieve both high accuracy and throughput (Section 5.2). 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 sequence S to determine which node

partitions and edge buckets to bring into the buffer and in what order. MariusGNN uses a buffer with capacity of c physical node 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, the storage layer passes X_i to the processing layer.

The processing layer generates mini batches from X_i in a random order. MariusGNN performs multi-hop neighborhood sampling using the DENSE data structure to construct a mini batch (Section 4). 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 base vector representations 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 base vector representations are transferred back to CPU memory and used to update the node representations in the partition buffer. As done in prior work [33], we perform all data transfers in a pipelined manner as shown in Figure 2.

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.

4 The DENSE Data Structure

DENSE is designed to 1) minimize redundant computation in multi-hop sampling and 2) to allow for efficient GNN forward passes using dense kernels for linear algebra operations.

4.1 Multi-hop Neighborhood Samples With DENSE

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 1, we showed how to compute the representation of target node A after two layers. To compute h_A^2
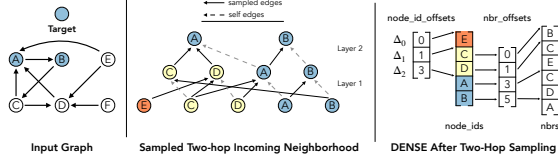
Figure 3. Example two-hop neighborhood sample for target nodes $\{A, B\}$ and the corresponding DENSE data structure.

Algorithm 1: Multi-hop Neighborhood Sampling

Input: 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)

---

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 requires traversing the one-hop neighborhood of node A multiple times, introducing redundant computation.

In MariusGNN, 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 $P \subseteq N$. In MariusGNN, 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, MariusGNN 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. DENSE consists of three Δ's: $\Delta_2 = \{A, B\}$, $\Delta_1 = \{C, D\}$, $\Delta_0 = \{E\}$. Notice that unlike in Figure 1, here, the sampled one-hop neighbors of A, $\{C, D\}$ are used in both GNN layers.

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, MariusGNN samples 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.

MariusGNN performs one-hop sampling using CPU multi-threading. We store two sorted versions of the in-memory edge list containing all edges between the node partitions currently in memory: 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, MariusGNN 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, MariusGNN computes 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 MariusGNN 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 discuss the implication of this reduced randomness on the accuracy of GNN training in Section 7.2 but find that training with DENSE can reach comparable accuracy to existing systems.

4.2 Forward Pass Computation With DENSE

After sampling is completed, MariusGNN transfers DENSE to the GPU so it can be used to compute the GNN forward pass. We also transfer an array H^0 containing the base representations 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 base representation for each node ID in the DENSE nbrs array.

To complete a forward pass over a k-layer GNN, we iterate over each layer $i \in [1 \ldots k]$ 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, 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 ith 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, node E and the neighbors of $\{C, D\}$ are not needed after layer one.

DENSE allows MariusGNN 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 kth 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 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).

5 Policies for Disk-Based GNN Training

We discuss the partition replacement policies used by MariusGNN for disk-based training. As described in Section 3, MariusGNN uses different policies for link prediction and node classification. We discuss each in turn.

Algorithm 2: On GPU DENSE Update After Layer i

Input: 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

Input: 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

5.1 Policies for Link Prediction

MariusGNN uses the COMET policy to maximize throughput while maintaining high accuracy when training link prediction models. Before we introduce COMET, we discuss why policies that only optimize for throughput lead to biased training and hence harm the accuracy of the learned models.

Greedy policies (e.g., BETA [33]) 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 $S = \{S_1, S_2, \ldots\}$ to be the sequence of partition sets which will be loaded into memory during one epoch and $X = \{X_1, X_2, \ldots\}$ to be the sequence of training examples used to generate mini batches for each $S_i \in S$ (Section 3). 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 4. As highlighted in the introduction, 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 7.5, 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
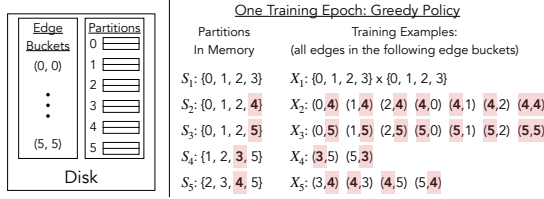
Figure 4. Greedy sequence of partitions in memory S and training examples X that are correlated. E.g., the examples in X_2 all come from edge buckets containing partition four.
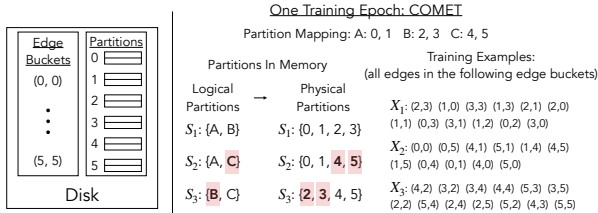

Figure 5. Partition and training example sequences generated by COMET to minimize training example correlation.

each epoch while simultaneously minimizing IO. To increase randomness, we design COMET around two mechanisms: 1) a two-level (logical and physical) partitioning scheme and 2) randomized generation of training examples. Following the architecture from Section 3, 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 3). COMET then generates $S = \{S_1, S_2, \dots\}$ by greedily swapping one logical partition between S_i and S_{i+1} such that all pairs of partitions (and thus pairs of nodes) 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 6, 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 [33]. 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 4). For these cases, training on X_i completes before S_{i+1} is loaded leading to IO bottlenecks.

5.2 Policies for Node Classification

For node classification we find that a simple replacement policy is 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). We find that in large-scale graphs [16, 17], it is often the case that the training nodes make up only one to ten percent of all graph vertices. As such, the base representations 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 base representations in CPU memory. While prior works have also used static caching for GNN training [26, 47], 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: Rather than randomly partitioning the graph, we assign all training nodes sequentially to the first k physical partitions. Non-training nodes are assigned to physical partitions randomly 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). If $k \geq c$, MariusGNN uses random partitioning and COMET but generates S as follows: replace a random logical partition in memory with a random one from disk that has not appeared in memory until all partitions have appeared in the buffer. In the future, we plan to study how this approach compares with other schemes [27] for dynamically caching training examples in CPU memory.

6 Auto-tuning Rules For Disk 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 5.1). This problem is general to ML workloads [7, 14, 15]. We design B to capture the extent to which the sequence of training examples generated by COMET exhibits this phenomenon. Figure 6a illustrates the dependency between B and model accuracy. The depicted results correspond to empirical measurements over a benchmark model (GraphSage [13]) and dataset (FB15k-237 [40]). 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 5.1). 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 $d_i = max_{v1,v2 \in V}(t_i^{v1} - t_i^{v2})$. 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 6c shows that B decreases with increasing physical partitions. The observed trends can be characterized by the equation $B = O(p^{-\alpha_1})$ for some constant $\alpha_1 > 1$. Figure 6b 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 $B = O(l^{\alpha_2})$ 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 4.1 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 6b 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: $IO = O(l^{-\alpha_3})$ for $\alpha_3 > 1$ and $|S| = O(l)$. 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 $T = O(l)$.

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/writes to small random reads/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 $T = O(1)$ for $p \leq \alpha_4$ and $O(p)$ for $p > \alpha_4$, with α_4 representing the number of partitions which cause the smallest disk reads to 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 base vector
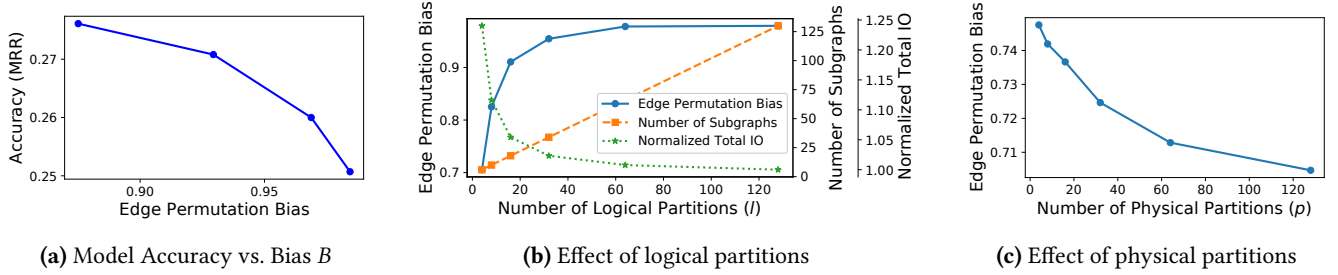
(a) Model Accuracy vs. Bias B

(b) Effect of logical partitions

(c) Effect of physical partitions

Figure 6. Empirical measurements on the effect of COMET parameters using GraphSage on FB15k-237.

representation of each node is of 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 node representations 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: $B = O(p^{-\alpha_1})$ for some constant $\alpha_1 > 1$ and $T = O(1)$ for $p \leq \alpha_4$ and $O(p)$ 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 (Section 4.1) 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 $B = O(l^{\alpha_2})$ for $0 < \alpha_2 < 1$ and $T = O(l)$. 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 $l = 2 * p/c$.

7 Evaluation

We implemented MariusGNN in 16k lines of C++ and 5k lines of Python. We evaluate MariusGNN on four large-scale graphs (see Table 1 for dataset statistics), including two from the OGB large-scale challenge [16], and compare against the popular SoTA GNN systems DGL and PyG. We show that:

1. MariusGNN reaches the same level of accuracy 2-8× faster and 8-64× cheaper than DGL and PyG on all datasets for both node classification and link prediction.
2. DENSE allows MariusGNN to reduce mini-batch sampling and compute times by up to 14× and 8× respectively.
3. MariusGNN enables cheap and efficient training of GNNs using disk storage. COMET yields runtime and accuracy improvements compared to SoTA methods. Additionally, MariusGNN allows us to train GNNs for node classification when datasets exceed commodity main memory.
4. MariusGNN auto-tuning rules for COMET yield configurations that achieve the highest throughput and model quality, lowering the deployment burden for training.

7.1 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) (Section 7.2). 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/backward pass computation in these systems and MariusGNN (Section 7.4). Furthermore, we compare the multi-hop sampling time in MariusGNN to the SoTA sampling implementation in NextDoor [19] (Section 7.4). 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.

While MariusGNN uses techniques such as a partition buffer and CPU-GPU pipelining, which are employed in prior disk-based graph systems (e.g., Marius [33] and Py-Torch BigGraph [24]), we do not compare against these systems directly as they do not support GNN computation. The primary reason for this is that neither system contains the neighborhood sampling algorithms and data structures needed to support multi-layer GNNs. As described in this work, MariusGNN addresses the limitation of these systems by developing an efficient neighborhood sampler (DENSE). Moreover, while both MariusGNN and Marius use a partition buffer for disk-based training (Marius' disk-based training approach was shown to outperform that of PyTorch Big-Graph), each system uses a different partition replacement policy. In Section 7.5 we compare the COMET policy developed in MariusGNN for high throughput and high accuracy disk-based GNN training with Marius' partition replacement policy (by implementing the later in MariusGNN).

Hardware Setup We evaluate all systems using AWS P3 instances (Table 2). We use an EBS volume with 1GBps of bandwidth and 10000 IOPS as disk storage. We report results for MariusGNN using two hardware configurations: one for disk-based training (M-GNN$_{Disk}$) and one for training with the full graph in memory (M-GNN$_{Mem}$). For the former, we minimize training costs by using the P3.2xLarge machine—an instance that does not have enough CPU memory to store any of the large-scale graphs in Table 1. For the latter, we use the cheapest P3 instance which has enough RAM for training

Table 2. 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

(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 M-GNN$_{Mem}$. 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) [16, 17]. 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) [13] GNN on both datasets, a common choice for these graphs [21, 52]. 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) [16]. As a second graph for large-scale link prediction, we use Freebase86M (FB) [51]. 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 [41] 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 [24, 33, 51] using the DistMult [46] score function and train all systems for a fixed number of epochs: five 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.

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 [13, 16, 17] 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. MariusGNN disk-based training hyperparameters are set using the auto-tuning rules (see Section 6). The specific hyperparameters used for each experiment can be found in our artifact (see the Appendix).

7.2 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 3-5 and Figure 7. 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. We report two configurations for MariusGNN—one with graph data stored in main memory (M-GNN$_{Mem}$) and one using disk-based training (M-GNN$_{Disk}$). 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. Moreover, cost reductions for all experiments are *at least* 8×. Differences in training time and cost can be almost two orders of magnitude: Baseline systems can take six days and $1720 dollars for training (see training on Wiki in Table 4) yet MariusGNN needs only eight hours or $36 dollars for the same dataset.

Node Classification We focus on end-to-end results for node classification in more detail (Table 3). 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. We show the time-to-accuracy on Papers100M in Figure 7. Not only does MariusGNN train 4× faster than baseline systems per epoch, but it also lowers the time-to-accuracy by the same factor. 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
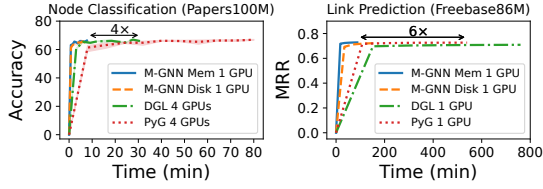
Figure 7. Time-to-accuracy for MariusGNN, DGL, and PyG. MariusGNN reaches the same level of accuracy 4-6× faster.

and GPU-based GNN computation in MariusGNN compared to baseline systems (evaluated in Section 7.4). Second, while both DGL and PyG require multi-GPU machines on AWS due to CPU memory requirements, 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).

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

MariusGNN can train the same GraphSage models for node classification using disk-based training on a single AWS P3.2xLarge machine, leading to 16× and 64× cheaper learning 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 link prediction. End-to-end results for all systems on Freebase86M and WikiKG90Mv2 are reported in Table 4. Time-to-accuracy on Freebase86M is shown in Figure 7. MariusGNN in-memory training is 6× and 7× faster than the best 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 5. Interestingly,

Table 3. 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.

	Epoch (min.)		Accuracy		Cost ($/epoch)	
Dataset	Papers	Mag	Papers	Mag	Papers	Mag
M-GNN$_{Mem}$	**0.77**	2.57	66.38	63.17	0.16	1.05
M-GNN$_{Disk}$	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

Table 4. 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× faster and 13-18× cheaper. (OOT: out of time)

	Epoch (min.)		MRR		Cost ($/epoch)	
Dataset	FB	Wiki	FB	Wiki	FB	Wiki
M-GNN$_{Mem}$	**17.5**	**46.6**	.7285	.4655	3.57	9.38
M-GNN$_{Disk}$	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

Table 5. 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 construction result in similar training time and cost on GraphSage and the more computationally expensive GAT.

	Epoch (min.)		MRR		Cost ($/epoch)	
Model	GS	GAT	GS	GAT	GS	GAT
M-GNN$_{Mem}$	**17.5**	**52.6**	.7285	.7331	3.57	10.7
M-GNN$_{Disk}$	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

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 sampling operations rather than GPU computation.

COMET allows MariusGNN to train the same models for link prediction on a 3× cheaper P3.2xLarge machine 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 embeddings 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 the introduction, 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 (Wiki).

Table 6. Comparison of the time required for mini-batch neighborhood sampling, GPU-based computation, and the number of nodes/edges sampled per mini batch in MariusGNN, DGL, and PyG for GraphSage GNNs of varying depth on Papers100M. 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)					Number of Nodes/Edges Sampled Per Mini Batch				
#Layers	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
M-GNN	**1.4**	**18**	**103**	**401**	**1.8k**	4	**6.1**	**21**	**153**	OOM	12k/13k	136k/181k	1M/2M	6M/17M	23M/91M
DGL	5.7	28	376	5.4k	49k	4.7	29	215	1231	OOM	13k/20k	182k/278k	2M/4M	9M/37M	33M/222M
PyG	2.2	59	1227	19k	96k	**3.2**	13	168	OOM	OOM	13k/20k	178k/258k	2M/4M	9M/32M	31M/174M

We evaluate COMET in more detail in Section 7.5 and show that it trains faster while improving accuracy compared to SoTA policies for seven model/dataset combinations.

7.3 Extreme Scale GNN Training With One GPU

A core motivation of our work is to investigate if distributed GNN training [50, 52] is necessary or if the resources in a single machine can be used efficiently to scale GNN training. To evaluate this, we stress-test MariusGNN with respect to graph size: We consider the task of learning vector representations 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). 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 representations, 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 OGB large-scale challenge [16] (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 distributed training over large clusters and to understand the power of optimized single machine deployments.

7.4 Effect of DENSE on Training

We have shown that end-to-end training in MariusGNN is faster than existing systems. The 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/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

Table 7. Comparison of the time required for GPU-based mini-batch neighborhood sampling in MariusGNN and NextDoor for GraphSage GNNs of varying depth on Live-Journal. 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

mini batch for 1) CPU neighborhood sampling and 2) GPU training and compare those times against the corresponding methods used in DGL and PyG. For these experiments, we use a GraphSage GNN on the 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 6. 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 6). 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 6, we report the average number of unique nodes and edges sampled per mini batch for each system. DENSE allows for mini batch construction using fewer samples than baselines. For example, constructing a three-hop neighborhood in MariusGNN 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 Marius-GNN, DGL, and PyG are all the same order of magnitude, yet MariusGNN 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 the multi-hop sampling in MariusGNN to the SoTA accelerated sampling implementation of NextDoor [19]. 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 the paper), MariusGNN also includes support for GPU-based multi-hop sampling 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/69M edges) [25]. For each layer, we sample 20 outgoing neighbors per system.

GPU-based sampling times for MariusGNN and NextDoor are shown in Table 7. 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 7 shows that redundant sampling can bottleneck even the most optimized sampling implementations. At the same time, DENSE avoids this bottleneck and can scale to five-layer GNNs with little sampling overhead.

For graphs that exceed GPU memory, the open-source release of NextDoor is unable to perform multi-hop sampling. In this setting, MariusGNN uses mixed CPU-GPU training with CPU-based neighborhood sampling (evaluated above).

7.5 Evaluating COMET for Disk-Based Training

In Section 7.2, we showed that COMET allows for disk-based GNN training on the link prediction task 7.5-18× cheaper

Table 8. COMET versus the SoTA BETA policy [33] 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. (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

than DGL and PyG. We now evaluate COMET in more detail and compare to the SoTA greedy policy from Marius [33] called 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. Since Marius does not support GNNs, we implement BETA in MariusGNN. We use the GraphSage and GAT GNNs on the graphs FB15k-237 [40], 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 BETA on the specialized decoder-only models supported by Marius. 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 6. 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 8. 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 example, COMET
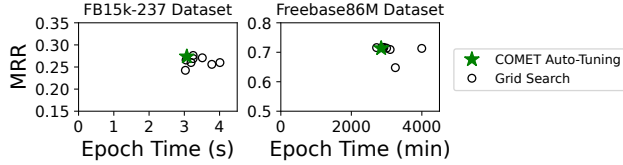
Figure 8. MRR and runtime for GraphSage GNN disk-based training with COMET using different hyperparameters. The auto-tuning rules used by MariusGNN are near-optimal.

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 hide IO for the latter part.

7.6 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 6 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 8. 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.

8 Related Work

Systems for ML over Graph Data Many systems support GPU training of GNNs [8, 11, 20, 21, 26, 45, 53]. Two such popular systems are DGL [43] and PyTorch Geometric [10]. 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 [8, 21, 26, 31, 32]. Additional works focus on optimized GPU kernels [44]. In general, these works focus on orthogonal challenges of GNN training than those discussed here and these ideas can be incorporated into MariusGNN. Finally, there are works that focus on scaling the training of non-GNN based graph ML models [1, 24, 33, 51]. For example, Marius [33] utilizes pipelined training to achieve state-of-the-art throughput for specialized knowledge graph embeddings.

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 [11, 20, 42, 52, 53]. Recent work introduces DistDGLv2 as a distributed version of DGL [52] and utilizes METIS partitioning,

co-location of data with mini batch computation, and asynchronous mini batch preparation to scale training. Other works distribute training in a serverless manner [39]. In MariusGNN, we use a disk-based approach to scaling beyond CPU memory. On Papers100M, MariusGNN is 6.7× cheaper than DistDGLv2 based on the reported cost to 66% accuracy for each system. Other works have previously supported disk-based training for link prediction using non-GNN models [24, 33, 38]. MariusGNN provides disk-based GNN support for both node classification and link prediction.

Neighborhood Sampling Many works focus on reducing the overhead of neighborhood sampling. Initial approaches sample a fixed number of neighbors per node [13], while follow-up works sample a fixed number of neighbors per layer [4, 54]. Other works decouple the sampling frequency from the mini batch frequency [35]. MariusGNN focuses 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 [5, 48]. These contributions can be incorporated in MariusGNN and are orthogonal to our study. Finally, recent works utilize GPUs to speed up sampling [8, 19]. MariusGNN supports GPU-based sampling but uses CPU-based sampling to scale to large graphs.

9 Conclusion

This paper introduced MariusGNN, a system for pipelined mini-batch training of GNNs in a single machine. We showed that MariusGNN with one GPU can achieve the same level of model accuracy up to 8× faster than existing systems using eight GPUs. To achieve these results, we introduced the DENSE data structure to minimize the redundancy of multi-hop sampling and the two-level COMET replacement policy for disk-based training. Overall, our results highlight the need to optimize single GPU implementations of ML systems before resorting to multi-GPU approaches.

Acknowledgments

References

[1] Taha Atahan Akyildiz, Amro Alabsi Aljundi, and Kamer Kaya. 2020. GOSH: Embedding Big Graphs on Small Hardware. In *49th International Conference on Parallel Processing - ICPP* (Edmonton, AB, Canada) *(ICPP '20)*. Association for Computing Machinery, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/3404397.3404456

[2] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Data. In *Advances in Neural Information Processing Systems*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.), Vol. 26. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf

[3] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. 2021. Machine Learning on Graphs: A Model and Comprehensive Taxonomy. arXiv:2005.03675 [cs.LG]

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

[5] Wei-Lin Chiang, 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* (Jul 2019). https://doi.org/10.1145/3292500.3330925

[6] Avery Ching, 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 (2015), 1804–1815.

[7] Christopher M De Sa. 2020. Random Reshuffling is Not Always Better. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 5957–5967. https://proceedings.neurips.cc/paper/2020/file/42299f06ee419aa5d9d07798b56779e2-Paper.pdf

[8] Jialin Dong, 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*. Association for Computing Machinery, 289–299.

[9] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *The World Wide Web Conference*. 417–426.

[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

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

[12] Google. 2018. Freebase Data Dumps. https://developers.google.com/freebase.

[13] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf

[14] Jeff Haochen and Suvrit Sra. 2019. Random Shuffling Beats SGD after Finite Epochs. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 2624–2633. https://proceedings.mlr.press/v97/haochen19a.html

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

[16] Weihua Hu, 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)*. https://openreview.net/forum?id=qkcLxoC52kL

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

[18] Ihab F Ilyas, 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*.

[19] Abhinav Jangda, 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.

[20] Zhihao Jia, 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*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 187–198. https://proceedings.mlsys.org/paper/2020/file/fe9fc289c3ff0af142b6d3bead98a923-Paper.pdf

[21] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. 2022. Accelerating Training and Inference of Graph Neural Networks with Fast Sampling and Pipelining. *Proceedings of Machine Learning and Systems* 4 (2022).

[22] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[23] Aapo Kyrola, 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)*. USENIX Association, Hollywood, CA, 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[24] Adam Lerer, 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 (2019), 120–131.

[25] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

[26] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 401–415.

[27] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2021. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *arXiv preprint arXiv:2112.08541* (2021).

[28] Steffen Maass, 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* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 527–543. https://doi.org/10.1145/3064176.3064191

[29] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! But at what {COST}?. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

[30] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. 2014. Web Data Commons - Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph/.

[31] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021.

Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2087–2100. https://doi.org/10.14778/3476249.3476264

[32] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen mei Hwu. 2021. PyTorch-Direct: Enabling GPU Centric Data Access for Very Large Graph Neural Network Training with Irregular Accesses. arXiv:2101.07956 [cs.LG]

[33] Jason Mohoney, 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.

[34] Namyong Park, Andrey Kan, Xin Luna Dong, Tong Zhao, and Christos Faloutsos. 2019. Estimating node importance in knowledge graphs using graph neural networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 596–606.

[35] Morteza Ramezani, 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*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 18482–18492. https://proceedings.neurips.cc/paper/2020/file/d714d2c5a796d5814c565d78dd16188d-Paper.pdf

[36] Chao Shang and Jie Chen. 2021. Discrete Graph Structure Learning for Forecasting Multiple Time Series. In *Proceedings of International Conference on Learning Representations*.

[37] Alexey Strokach, David Becerra, Carles Corbi-Verge, Albert Perez-Riba, and Philip M. Kim. 2020. Fast and Flexible Protein Design Using Deep Graph Neural Networks. *Cell Systems* 11, 4 (2020), 402–411.e4. https://doi.org/10.1016/j.cels.2020.08.016

[38] Ding Sun, Zhen Huang, Dongsheng Li, Xiangyu Ye, and Yilin Wang. 2021. Improved Partitioning Graph Embedding Framework for Small Cluster. In *Knowledge Science, Engineering and Management*, Han Qiu, Cheng Zhang, Zongming Fei, Meikang Qiu, and Sun-Yuan Kung (Eds.). Springer International Publishing, Cham, 203–215.

[39] John Thorpe, 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)*. USENIX Association, 495–514. https://www.usenix.org/conference/osdi21/presentation/thorpe

[40] Kristina Toutanova, 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*. Association for Computational Linguistics, Lisbon, Portugal, 1499–1509. https://doi.org/10.18653/v1/D15-1174

[41] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

[42] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. 2021. FlexGraph: a flexible and efficient distributed framework for GNN training. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 67–82.

[43] Minjie Wang, 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 (2019). http://arxiv.org/abs/1909.01315

[44] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 515–531. https://www.usenix.org/conference/osdi21/presentation/wang-yuke

[45] Zonghan Wu, 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 (2021), 4–24. https://doi.org/10.1109/TNNLS.2020.2978386

[46] Bishan Yang, 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* (2014).

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

[48] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJe8pkHFwS

[49] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *Advances in Neural Information Processing Systems* 31 (2018), 5165–5175.

[50] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE Computer Society, Los Alamitos, CA, USA, 36–44. https://doi.ieeecomputersociety.org/10.1109/IA351965.2020.00011

[51] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. 2020. 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.

[52] Da Zheng, Xiang Song, Chengru Yang, Qidong Su, Minjie Wang, Chao Ma, and George Karypis. 2022. Distributed Hybrid CPU and GPU training for Graph Neural Networks on Billion-Scale Graphs. arXiv:2112.15345 [cs.DC]

[53] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2094–2105. https://doi.org/10.14778/3352063.3352127

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

A   Artifact Appendix

A.1   Abstract

We have released the artifact for reproducing the paper's experimental results in our GitHub repository artifact branch (`eurosys_2023_artifact`). The artifact includes the MariusGNN source code, Deep Graph Library and PyTorch Geometric baseline implementations, configuration files, and Python scripts to execute the experiments reported in the paper for all three systems. The source code and experiment configurations can be used to study implementation details that were not mentioned in the paper for brevity. Details on how to use the artifact can be found in the following sections and in the GitHub README file.

A.2   Description & Requirements

A.2.1   How to access
The artifact is available at the following GitHub url: `MariusGNN GitHub Artifact`.

A.2.2   Hardware dependencies
Artifact hardware dependencies and the specific hardware used for each experiment reported in the paper are provided in the GitHub repository README file. Paper experiments were run on AWS P3 GPU machines to support training over large-scale graphs, but MariusGNN and the artifact can be run on CPU only hardware as well.

A.2.3   Software dependencies
Artifact software dependencies and the specific versions used for the paper experiments are listed in the GitHub repository README file under the 'Build Information and Environment' heading.

A.2.4   Benchmarks
The artifact uses publicly available GNN benchmark datasets (graphs): FB15k-237, OGBN-Arxiv, OGBN-Papers100M, OGB-Mag240M, Freebase86M, and OGB-WikiKG90Mv2. Scripts to download and preprocess all datasets are included in the artifact and fully automated.

A.3   Set-up

Installation and configuration steps required to prepare the artifact environment are described in the 'Getting Started' section of the GitHub repository README file. In the section titled 'End-to-End Docker Installation' we have provided a Dockerfile and instructions which include all dependencies and commands necessary to install and build the artifact.

A.4   Evaluation workflow

In this section, we provide a brief outline of the artifact functionality and intended use of this artifact to reproduce the paper's main claims. Reproducing all experiments reported in the paper requires significant compute resources: The experiments in the paper were run on AWS P3 GPU machines and we estimate that reproducing all the claims may cost approximately $5000. *Thus, we have organized our artifact according to 1) a 'minimal working example' to demonstrate artifact functionality and 2) a list of experiments for reproducing the numbers reported in the paper and the estimated cost of each experiment.* We first highlight the minimal working example that can establish artifact functionality, then list the paper's major claims and how each are supported by the artifact experiments.

A.4.1   Artifact Functionality
To demonstrate how the artifact can be used to produce experimental results comparing MariusGNN, DGL, and PyG, we include a set of minimal working examples described in the GitHub README file under the section 'Artifact Minimal Working Example (Functionality)'. These examples consist of Python scripts to train GNNs using each system for the tasks of link prediction (on the FB15k-237 graph) and node classification (on the OGBN-Arxiv graph). The Python scripts follow the same format as those used to create the paper's experiments, but operate on small graphs (rather than the large-scale graphs used in the paper) and can be run without the need for GPUs.

A.4.2   Major Claims
The major experimental claims made in the paper are:

- (C1): For the task of node classification, MariusGNN can train GNNs over large-scale graphs 3-8× faster and up to 64× cheaper than competing systems while reaching comparable accuracy. This is proven by the experiment (E1) described in Section 7.2 with results reported in Table 3.
- (C2): For the task of link prediction, MariusGNN can train GNNs over large-scale graphs 6× faster and 13-18× cheaper than competing systems while reaching comparable accuracy. This is proven by the experiment (E2) described in Section 7.2 with results reported in Table 4.
- (C3): A key reason for the improved end-to-end performance of MariusGNN is faster neighborhood sampling and GNN forward pass computation. Using the DENSE data structure, in MariusGNN sampling is 14× faster and GPU computation is 8× faster for a four-layer GNN when compared to baseline systems. This is proven by the experiment (E3) described in Section 7.4 with results reported in Table 6.
- (C4): The COMET partition replacement policy for disk-based link prediction leads to simultaneously faster training and higher MRR compared to existing SoTA policies. This is proven by the experiment (E4) described in Section 7.5 with results reported in Table 8.

A.4.3   Experiments
The artifact contains a table in the GitHub repository README file under 'Reproducing Experimental Results' which describes what experiment to run to reproduce the above main claims. Each experiment in the table also contains the required hardware, cost/time estimates, a short explanation, the expected results, and an explicit link to one of the above four claims as well as the corresponding table in the paper. All experiments are run in the same way as the minimal working example and as described in the

'Artifact Documentation: Running Experiments' section of the README. That is, the experiment name is provided to the `run_experiment.py` script with any additional desired arguments.