Two-level Graph Caching for Expediting Distributed GNN Training

Zhe Zhang*, Ziyue Luo*, Chuan Wu*
*Department of Computer Science, The University of Hong Kong, Email: {zzhang2, zyluo, cwu}@cs.hku.hk

Abstract—**Graph Neural Networks (GNNs) are increasingly popular due to excellent performance on learning graph-structured data in various domains. With fast expanding graph sizes and feature dimensions, distributed GNN training has been adopted, with multiple concurrent workers learning on different portions of a large graph. It has been observed that a main bottleneck in distributed GNN training lies in graph feature fetching across servers, which dominates the training time of each training iteration at each worker. This paper studies efficient feature caching on each worker to minimize feature fetching overhead, in order to expedite distributed GNN training. Current distributed GNN training systems largely adopt static caching of fixed neighbor nodes. We propose a novel two-level dynamic cache design exploiting both GPU memory and host memory at each worker, and design efficient two-level dynamic caching algorithms based on online optimization and a lookahead batching mechanism. Our dynamic caching algorithms consider node requesting probabilities and heterogeneous feature fetching costs from different servers, achieving an $O(\log^3 k)$ competitive ratio in terms of overall feature-fetching communication cost (where k is the cache capacity). We evaluate practical performance of our caching design with testbed experiments, and show that our design achieves up to 5.4x convergence speed-up.**

I. Introduction

Graph Neural Networks (GNNs) [1–3] have been increasingly adopted to encode complex features from graph-structured data, for various downstream tasks such as user interaction analysis in social networks [4], protein interface prediction for drug discovery [5] and traffic prediction in intelligent transportation systems [6]. Graphs in many real-world systems are large, e.g., the citation graph has more than 120 million nodes (published papers before 2019) [7] with sophisticated citation connections. Graph size and feature dimensions are also fast expanding, e.g., the social network of Friendster [8] contains more than 1 billion edges and the user features in Facebook's ego data [9] have more than 250 dimensions. A large graph cannot be held entirely in the memory of a single modern server and is often partitioned among multiple servers for distributed GNN training.

In distributed GNN training, multiple workers are deployed on the servers storing the graph partitions; each worker fetches node features from the local graph partition or other servers, and trains a local copy of the GNN that aggregates the features to generate low-dimensional graph representations (aka embeddings). It has been observed that inter-machine communication for feature fetching dominates the training time, i.e., occupying up to 80% of the time in each training iteration [10]. Efficient feature fetching is hence the key to improve the performance of distributed GNN training. Existing efforts on reducing feature communication time can be categorized into two camps: (i) Transmission scheduling that pipelines feature fetching and GNN computation [11, 12]. (ii) Communication traffic reduction with better graph partition and caching. The current distributed GNN training frameworks (e.g., DistDGL [11], SAR [13], Dorylus [14]) partition the input graph using the METIS algorithm [15], minimizing cross-partition edges. They also adopt one-hop neighbor caching (meta-data such as node IDs only, instead of the node features), or static feature caching for a few hops of neighbors according to node degrees or sampling history [16–18].

We tackle the cross-server feature communication bottleneck in distributed GNN training from the perspective of efficient feature caching on different servers. With static feature caching, cache hit (i.e., features of a sampled node are cached locally) heavily relies on the graph sampling method and the batch size (i.e., the number of training nodes used in each iteration) in GNN training. Static caching used in the current distributed GNN training frameworks is oblivious to the potentially different feature fetching costs (in case of cache misses) from servers of different inter-connection bandwidth [16]. Besides, existing caching designs only consider one-level cache using the GPU [17, 19], not fully exploiting available device or host memory capacities.

We propose a novel two-level cache design for distributed GNN training, that exploits caching in both GPU memory and host memory, caters to node requesting dynamics on different servers and addresses potentially heterogeneous feature fetching costs among servers. The goal is to minimize the feature fetching overhead and expedite distributed GNN training substantially. Our main contributions are summarized as follows:

▷ We design a dynamic two-level cache where the top-tier cache is in the GPU memory and the second-tier cache is in the host memory. To compose local training batches, a worker (running on GPU(s)) fetches features of the training nodes and their L-hop neighbors from the GPU memory, the host memory, or other servers. The top-tier GPU cache stores the current training batch, and evicts previously cached nodes for hosting newly sampled nodes. The host memory caches nodes fetched from other servers that are evicted from the GPU memory, besides consistently storing nodes of the graph partition assigned to this server.

▷ We design an efficient online algorithm for two-level dynamic caching, such that the nodes evicted by top-tier GPU cache can be handled by the second-tier CPU cache, to effectively minimize feature fetching costs during iterative GNN training. To our best knowledge, we are the first to propose two-level dynamic caching with a lookahead mechanism, exploiting sampling, feature fetching and training parallelization in distributed GNN training. Other highlights of the algorithm design include the novel parallel probabilistic eviction trails in deciding node evictions, which plays a key role in calculating online caching policy fleetly.

▷ We adopt potential function analysis and carefully prove that our online two-level caching algorithm achieves an $O(\log^3 k)$ competitive ratio in terms of overall feature-fetching communication cost (k is the maximum between GPU cache capacity and CPU cache capacity), as compared to the offline optimum with full knowledge of future node requests in distributed GNN training. We show low time complexity for running our online caching algorithm at each worker during distributed GNN training as well.

▷ Our caching design was evaluated through practical performance testing by training representative GNN models on various state-of-the-art large graph datasets under different settings. The results of our experiments demonstrate that distributed GNN training with our dynamic feature caching converges $3.37\times$ faster in homogeneous cross-machine bandwidth settings and achieves up to a $5.40\times$ speed-up in heterogeneous cross-machine bandwidth settings.

II. Background and Related Work

A. Graph Neural Network and Distributed GNN training

Graph Neural Network (GNN) [1–3] is a type of neural networks (NN) that encodes graph-structured data. For a L-layer GNN, it takes the features of L-hop neighbors of training nodes as input, and produces embeddings of the training nodes through operations such as graph convolution [3] or graph attention [2]. The embeddings are then fed into a downstream model (e.g., another NN) for tasks such as node classification and edge prediction [1, 5].

Given a graph $G = (V, E)$, we denote the input feature of node v by $h_v^{(0)}$, the neighbors of node v by $\mathcal{N}(v)$, and the edge feature between node v and its neighbor u by e_{vu}. Then the embedding of vertex v at layer l is calculated through $h_v^{(l+1)} = \sigma(h_v^{(l)}, \text{Agg}_{u\in\mathcal{N}(v)} f(h_v^{(l)}, h_u^{(l)}, e_{vu}))$, where $\sigma(\cdot)$ and $f(\cdot)$ are trainable GNN model parameters and Agg is the aggregator (e.g., min, max). There are two types of GNN training. (i) Full-batch training [16]: the entire graph is used for GNN training and node embedding generation. (ii) Mini-batch training [1]: in each training iteration, a set of training nodes are selected; for each training node, at most f_l neighbors (*fanout*) in its l-th neighbor hop are sampled, $\forall l \in [1, L]$, and a subgraph is constructed which serves as one training sample; the multiple training samples corresponding to the training nodes constitute a training batch, which is used for GNN training. Full-batch training can easily suffer from GPU memory overflow and is not suitable for training on larger graphs [19]. We focus on mini-batch training.

In distributed GNN training, the large input graph is typically partitioned among multiple servers using algorithms such as METIS [15], to minimize cross-partition edges and balance the workload among partitions. Multiple samplers and trainers run on the servers for training batch sampling and GNN training. Various neighbor sampling methods have been adopted, including uniform sampling [1], random walk or importance-based sampling [20]. In each training iteration, each trainer carries out forward computation with the GNN using a training batch, and then backward propagation to derive gradients of the model (i.e., gradients of $\sigma(\cdot)$ and $f(\cdot)$). Gradients produced on different trainers are then synchronized to update the model parameters globally.

B. Graph Data Cache in Existing GNN Training Systems

To mitigate communication, recent GNN systems adopt graph data caching, which stores graph structure and features in host/GPU memory of each server. Pagraph [16] runs GNN training on multiple GPUs in a single server and adopts static GPU caching of hot nodes with high degrees. Yang *et al.* [17] found that this static caching is only effective when uniform sampling or degree-based importance sampling is used on power-law input graphs. They instead cache nodes with the highest request numbers (i.e., those sampled for GNN training for most times) in previous training epochs (an epoch includes multiple training iterations during which all training nodes are trained once) in GPUs, and the caches remain unchanged in each epoch. Liu *et al.* [19] co-design a dynamic caching policy and the neighbor sampling order. Their design improves caching performance when used in conjunction with their specific sampling method, but cannot generalize to different sampling methods. We seek a dynamic caching design that does not rely on particular sampling methods.

C. Theoretical Results of Caching

One-level online caching [21, 22] is a well-studied problem with $O(k)$-competitive deterministic algorithms and $O(\log k)$-competitive randomized algorithms, where k is the cache capacity. The widely used Least Recently Used (LRU) cache is proved to be $O(k)$-competitive [23]. Bansal *et al.* introduce an $O(\log^2 k)$-competitive algorithm [24] for the generalized cache problem where cached items have various sizes and fetching costs, which is later improved to $O(\log k)$ by Adamaszek *et al.* [22]. Recently, a number of studies combine traditional one-level caching algorithms with black-box predictions using machine learning (ML) models [25, 26]. Lykouris *et al.* [25] and Bansal *et al.* [26] utilize future item-usage predictions in learning-augmented cache design; competitive ratios of such caching algorithms are positively correlated to the prediction error and upper-bounded by $O(\log k)$. All those algorithms only consider a single-level cache.

To our best knowledge, the recent work from Bansal *et al.* [27] is the only theoretical study on multi-level caching, and presents an $O(\log^2 k)$ algorithm for a p-level cache with different priorities for different cache levels. In their design, each item is cached as several copies across different cache levels and requests of different priorities are served by caches of different levels. The setting is different from graph data caching in GNN training, where node requests are not differentiated in terms of priorities but may incur different fetching costs, and the consumers (trainers on GPUs) always retrieve node features from the high-level cache (in GPU memory). Bansal *et al.*'s work does not consider different fetching costs for cache replacements, and their algorithm is complex to be applied in practical distributed GNN training, due to requiring multiple rounds of item eviction probability updates and evictions to meet cache capacity constraints.

III. System Model

A. Distributed GNN Training System

We consider a distributed GNN training system that learns over a large graph on M servers. The graph is divided into M partitions using the METIS algorithm [15] (as in DistDGL [11]) and each partition is assigned to be stored in the host memory of one of the M servers. Without loss of generality, we consider a simplified system model where each server hosts one worker, with one GPU memory and one host memory. Given the fast communication among GPUs in the same server, this can be considered as an abstraction that treats the potentially multiple GPUs on a server as one.

The worker on each server contains a sampler, a fetcher and a trainer. In each training iteration, the sampler selects a number of training nodes in the graph partition stored on this server, samples L-hop neighbor nodes of the training nodes with a given sampling method [1] and constructs a batch of subgraph training samples (containing meta-data of the nodes only, such as node IDs). The sampler typically runs on the CPU [28]. The fetcher retrieves features of nodes in the batch from the GPU memory, the host memory and other servers (depending on where the features are cached). The trainer utilizes those features to train the local GNN model, and then exchanges model gradients/parameters with other trainers using a communication primitive such as AllReduce [29].

We adopt a pipeline design across the sampler, the fetcher and the trainer on each worker, as illustrated in Fig. 1. The sampler feeds the sampled batch (which contains only meta-data of the nodes) to a sampled batch queue. The fetcher reads the batch from the sampled batch queue, fetches features of nodes in the batch from the GPU cache, the CPU cache and other servers to prepare a training batch (which contains the full features of sampled nodes), and feeds it to a training batch queue (in GPU). The fetcher then updates the GPU and CPU caches, i.e., evicting some nodes from the respective cache and replacing them with those in the current batch or evicted from the GPU. The trainer reads training batches from the training batch queue and uses them for GNN training. The two queues allow sampling, feature fetching and training to be carried out in a pipeline manner: the sampler can sample the next batch without waiting for the fetcher or the trainer and the fetcher can retrieve features for the next batch after it has prepared the previous training batch. In this way, sampling, feature fetching and training times overlap and the end-to-end GNN training time is reduced.

We further enable one-batch lookahead for GPU caching: After the fetcher has fed one training batch to the training batch queue and before it updates the GPU cache with this batch's features, it reads the next batch's meta data from the sampled batch queue and sets the eviction probability of nodes in the GPU cache, which belong to the next batch, to zero, prohibiting those nodes from being evicted. Then the fetcher updates the GPU cache with the current batch's features and starts to fetch features for the next batch.

Table I: Notation Table

Notation	Description
$x(i,j)$	1 if node i is evicted from GPU cache after j-th request
$q(i,j)$	1 if node i is evicted from CPU cache after j-th request
$r(i,t)$	# of times of node i requested by time t
w_1	cost for fetching node from host memory to GPU
$w(i)$	cost for fetching node i from other server
N	set of nodes in the whole graph
T	total number of training iterations
A	set of nodes in the local graph partition
$S(t)$	training batch for training iteration t
$B(t)$	set of nodes ever cached in GPU cache by t
k_x, k_y	size of GPU cache and CPU cache

B. Two-level Cache Design

We exploit both the GPU memory and the host memory on each worker for feature caching, and design a two-level cache. The top-tier cache, referred to as the GPU cache, is inside the GPU memory (with capacity k_x, excluding the memory used for storing GNN model parameters, intermediate results, the batch queues, etc.), and the second-tier cache, referred to as the CPU cache, is in the host memory (with capacity k_y, excluding the memory used for storing assigned graph partition and by the sampler, etc.). The GPU cache stores features of nodes that are requested recently and have higher probabilities to be requested in the near future. The CPU cache receives the nodes evicted from the GPU cache, and evicts nodes that have been copied to the GPU cache or with a lower expected cost to be fetched again (when the CPU cache is full). An illustration of our two-level cache is in Fig. 2.

C. Two-level online caching problem

We design the caching strategies for our two-level caches by formulating and solving an online caching optimization problem. The decisions are on which nodes to be evicted from the GPU cache and the CPU cache, respectively, when new node features are to be cached and the respective cache is full. The goal is to minimize the overall feature fetching cost to compose the training batches during T iterations of GNN training at each worker (where T is potentially a large number).

Consider a given worker in the distributed GNN training system. Let binary variable $x(i,j)$ denote whether node i is evicted from its GPU cache after it has been sampled into j training batches in the past, and binary variable $q(i,j)$ represent whether node i is evicted from the CPU cache after it has been sampled into j training batches. Let A be the set of nodes in the graph partition assigned to this worker and stored in its host memory (which will not be cached in the CPU cache). N is the set of all nodes in the whole input graph. For a node i sampled into a training batch, its fetching cost is zero if it is in the GPU cache, w_1 if its features are cached in the host memory (in the CPU cache or local graph partition), and $w(i)$ if to be fetched from another server. $w(i)$ is larger if the graph partition containing node i is on a server with lower inter-connection bandwidth with the current worker (server).

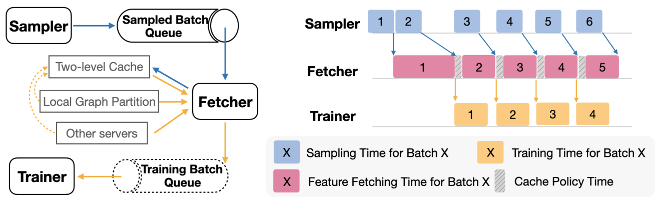


Figure 1: The processing pipeline of distributed GNN training

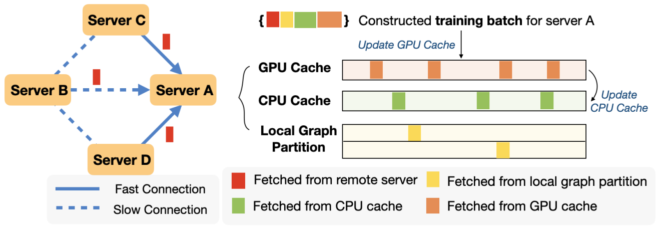


Figure 2: Two-level cache design

Let $S(t)$ be the training batch in training iteration t. For each node i in the input graph, $r(i,t)$ denotes the number of times that node i has been sampled in and before training iteration t. For a node in the training batch $S(t)$, we have $r(i,t) = r(i,t-1)+1$. Let $B(t)$ denote the set of nodes that have ever been cached in the GPU cache in or before training iteration t. At the beginning of the first training iteration, the GPU cache and the CPU cache are filled with features of randomly selected nodes from the local graph partition. The two-level caching optimization problem is formulated as follows. Notation is summarized in Table I.

$$\text{minimize} \sum_{i\in A}\sum_{j=1}^{r(i,T)} w_1 x(i,j)+ \sum_{i\in N\backslash A}\sum_{j=1}^{r(i,T)}((w(i)-w_1)\cdot q(i,j)+w(i)\cdot x(i,j)) \quad (1)$$

subject to

$$\sum_{i\in B(t)\backslash S(t)} x(i,r(i,t)) \geq |B(t)|-k_x, \ \forall 1\leq t\leq T \quad (1a)$$

$$\sum_{i\in N\backslash A} q(i,r(i,t)) \geq |N\backslash A|-k_y, \quad \forall 1\leq t\leq T \quad (1b)$$

$$q(i,j)+x(i,j)\geq 1, \ \forall i\in N\backslash A, \ 1\leq j\leq r(i,T) \quad (1c)$$

$$q(i,j)\in\{0,1\}, \ \forall i\in N\backslash A, \ 1\leq j\leq r(i,T) \quad (1d)$$

$$x(i,j)\in\{0,1\}, \ \forall i\in N, 1\leq j\leq r(i,T) \quad (1e)$$

$$r(i,t)\geq 0, \ \forall i\in N, 1\leq t\leq T \quad (1f)$$

$$B(t)=\{i: r(i,t)\geq 1\}, \ \forall 1\leq t\leq T \quad (1g)$$

The objective function in (1) represents the overall fetching cost to compose T training iterations at the worker: the first part is the total feature fetching cost for sampled nodes that belong to the graph partition A; the second part represents the fetching cost of nodes in other graph partitions. For a node i in A, $w_1 x(i,j)$ indicates the cost for fetching the node again from the host memory into a training batch after its j-th request (the time when it has been sampled in j previous batches): the cost is w_1 if the node is evicted from the GPU cache after its j-th request ($x(i,j)=1$), and zero if the node is not evicted and hence resides in the GPU cache after its j-th request ($x(i,j)=0$). For a node i in another graph partition ($N\backslash A$), it costs $(w_1-w(i))(1-q(i,j))+w(i)x(i,j)$ to fetch the node again into a training batch after its j-th request: the cost is $w(i)$ if the node has been evicted from both the GPU and CPU caches after its j-th request ($q(i,j)=x(i,j)=1$) and needs to be fetched from the server storing its graph partition, w_1 if the node is evicted from the GPU cache but is in the CPU cache ($x(i,j)=1, q(i,j)=0$), and zero if it is not evicted from the GPU cache ($x(i,j)=0, q(i,j)=1$). Since $(w_1-w(i))(1-q(i,j))+w(i)x(i,j)=(w_1-w(i))-(w_1-w(i))q(i,j)+w(i)x(i,j)$ and the constant term, $(w_1-w(i))$, does not influence the caching strategy, we omit it in the second term in the objective function.

Constraints (1a) and (1b) ensure that the capacities of the GPU cache and the CPU cache are obeyed. To derive (1a), first we have $\sum_{i\in B(t)}(1-x(i,r(i,t)))\leq k_x$ which guarantees that the node feature size in the GPU cache is less than k_x at training iteration t. For a node i sampled into the training batch $S(t)$ in t, we have $x(i,r(i,t))=0$, and thus $|B(t)|-\sum_{i\in B(t)}x(i,r(i,t))\leq k_x$ leads to $|B(t)|-\sum_{i\in B(t)\backslash S(t)}x(i,r(i,t))\leq k_x$. Hence we have $\sum_{i\in B(t)\backslash S(t)}x(i,r(i,t))\geq |B(t)|-k_x$, that is, there should be at least $|B(t)|-k_x$ nodes evicted from the GPU cache at time t. Similarly, to derive (1b), we have $\sum_{i\in N\backslash A}(1-q(i,r(i,t)))\leq k_y$ which guarantees that the node feature size in the CPU cache is less than k_y by training iteration t. Then we have $|N\backslash A|-\sum_{i\in N\backslash A}q(i,r(i,t))\leq k_y$ and hence $\sum_{i\in N\backslash A}q(i,r(i,t))\geq |N\backslash A|-k_y$, i.e., there should be at least $|N\backslash A|-k_y$ nodes evicted from the CPU cache. Constraint (1c) ensures that at most one copy of each node is cached in the CPU cache or the GPU cache at any time.

The caching optimization problem in (1) is an integer linear program. In the next section, we design an online caching algorithm to solve it, deciding cache eviction strategies during iterative GNN training.

IV. Two-Level Online Caching Algorithm

A. The online caching algorithm

Inspired by the caching algorithm in Bansal *et al.*'s work [27], we design an efficient dynamic caching algorithm that each worker carries out during its iterative GNN training, to decide node replacements in its two-level cache in each training iteration. The algorithm is given in Alg. 1. For notation simplicity, we use $x(i)$ and $q(i)$ to represent $x(i,r(i,t))$ and $q(i,r(i,t))$ in the algorithm, which indicate the eviction probability of node i's features from the GPU cache and from the CPU cache, respectively.

For each node in the training batch $S(t)$, we set its eviction probability from the GPU cache to zero ($x(i)=0$) and also set $q(i)=1$ to satisfy the exclusive caching constraint in (1c) (Lines 6-9). With the one-batch lookahead mechanism, we also set the eviction probability of nodes cached in the GPU and appearing in the next sampled batch to zero (Line 13).

We then decide the nodes to evict from the GPU cache to allow GPU caching of nodes in $S(t)$, that are not cached in the GPU yet. We increase the eviction probability $x(i)$ for all nodes inside the GPU cache that are not sampled in $S(t)$, i.e., $i \in N \backslash S(t)$ and $x(i) < 1$, with a step decided by parameters α and β (Line 16). $\alpha \geq 0$ and $\beta \geq 0$ can be set by grid search on a small set of candidate parameter pairs that lead to the highest GPU cache hit ratio, by running a few training iterations. Then we simulate node i's eviction with probability $\gamma_x \cdot x(i)$, where γ_x is a scalar chosen uniformly randomly in the range $[O(1), O(\log k_x)]$: we sample a random variable z uniformly randomly over $[0,1]$; if $z \leq \gamma_x \cdot x(i)$, we consider node i as evicted (i.e., increment the corresponding eviction count in vector V_x), as the probability of $z \leq \gamma_x \cdot x(i)$ is exactly $\min\{1, \gamma_x \cdot x(i)\}$. We simulate eviction of nodes for multiple times (five times as in our experiments) with independently randomly generated scaling factors γ_x, and record the number of times that a node in the GPU cache is evicted in those eviction trials in vector V_x. The rationale of scaling the eviction probability x_i by γ_x (chosen in $[O(1), O(\log k_x)]$) is to ensure that an appropriate number of nodes are potentially evicted (related to the competitive ratio achievable). Then we choose the top $b+c$ nodes with the largest eviction counts in V_x (by calling a TopK function in Line 29), where $b+c$ is the required number of nodes to evict from the GPU cache (with b and c being the number of nodes in $S(t)$ which are cached in the host memory and fetched from other servers, respectively), and replace those $b+c$ nodes in the GPU cache with the $b+c$ nodes in $S(t)$ which were not cached in the GPU (Lines 30-33). We assume that the GPU cache can hold at least b+c nodes here to simplify the analysis. It can be extended to the case where the GPU cache is smaller by randomly selecting a subset of the $b+c$ nodes to cache.

Next, we decide nodes to evict from the CPU cache, for the CPU cache to hold evicted nodes from the GPU. We remove from the set of nodes evicted from the GPU, E, the nodes that belong to the local graph partition (Line 34), which are always in the host memory. We increase the eviction probability $q(i)$ of nodes in the CPU cache by an amount inversely proportional to the fetching cost $w(i) - w_1$ (Line 36), such that the node with a larger fetching cost has a lower probability to be evicted. For example, a node belonging to a graph partition stored in another server with low inter-connection bandwidth with the current worker is less likely to be evicted. Then, similar to the eviction simulation of nodes in the GPU cache, we simulate node eviction from the CPU cache with scaled eviction probabilities, and record in vector V_y the eviction times of nodes in the CPU cache over multiple eviction trials. We choose the top $|E|$ nodes with the largest eviction counts in V_y and replace them in the CPU cache with nodes in E evicted from the GPU (Lines 47-51).

B. Competitiveness of the online caching algorithm

We next show that our online caching Alg. 1 achieves an $O(\log^3 k)$ competitive ratio, where $k = \max\{k_x, k_y\}$, the larger between the capacities of the GPU cache and the CPU cache. To prove this, we introduce an online fractional caching algorithm that solves the optimization problem in (1) by relaxing the integrity constraints (1d) and (1e) and allowing features of a sampled node to be partially cached in/evicted from the respective cache, and analyze the competitive ratio of the online fractional algorithm; then we connect the performance of Alg. 1 with that of the online fractional algorithm and show the competitive ratio of Alg. 1.

The online fractional caching algorithm for each worker in each training iteration includes three main steps:

Step (i): Set eviction fractions $x(i) = 0$ and $q(i) = 1$ for nodes in the training batch, i.e., $\forall i \in S(t)$.

Step (ii): Evict enough nodes from the GPU cache for accommodating nodes in the training batch (that are not cached in the GPU cache), by updating the eviction fraction $x(i)$ for all nodes inside the GPU cache and not in the training batch by an amount proportional to the current eviction fraction: $x(i) = \max\{1, x(i) + \frac{x(i)+1/k_x}{w_1}\Delta x\}$, where Δx is a small update step. We keep updating $x(i)$'s until the GPU cache capacity constraint (1a) is satisfied.

Step (iii): Evict enough nodes from the CPU cache to store nodes evicted by the GPU cache which are not in the local graph partition, by updating eviction fractions for nodes in the CPU cache by an amount proportional to the current eviction fraction and inversely proportional to the fetching cost of the node: $q(i) = \max\{1, q(i) + \frac{q(i)+1/k_y}{w(i)-w_1}\Delta q\}$, where Δq is a small update step. We keep increasing the eviction fractions until the CPU cache capacity constraint (1b) is satisfied.

We prove that the online fractional algorithm achieves a competitive ratio of $O(\log k)$, computed by dividing the objective value of (1) achieved by the fractional solutions by the offline optimal objective value of (1) with optimal integer solutions.

Theorem 1. *Assuming $w_1 \leq w(i) \leq mw_1$ (where m is an upper bound on $\frac{w(i)}{w_1}$) and $k_y \geq k_x$, the online fractional algorithm is $O(\log k)$-competitive in solving the two-level caching problem in (1), where $k = \max\{k_x, k_y\}$.*

Proof. We provide a proof sketch here and leave the full proof in the technical report. We use a potential function inspired by Bansal *et al.* [27] to prove the competitive ratio:

$$\Delta \text{On} + \Delta \Phi \leq O(\log k) \Delta \text{OPT} + const \quad (3)$$

where On is the objective value (1) of the online fractional algorithm, OPT is the objective value (1) of the offline optimal algorithm (with integer solutions), and *const* is independent of the sequence of training batches. ΔOn and ΔOPT represent the change of the objective value upon node eviction with the online fractional algorithm and the offline optimal algorithm, respectively. $\Delta\Phi$ is the change of the potential function value with node eviction conducted by online fractional algorithm or offline optimal algorithm. The potential function Φ is defined as

$$\Phi = 2\sum_{i \in N}(w_1 v_x(i) \ln \frac{1+1/k_x}{x(i)+1/k_x} + (w(i)-w_1)v_q(i) \ln \frac{1+1/k_y}{q(i)+1/k_y}) \quad (4)$$

where $v_x(i)$ and $v_q(i)$ denote the decisions from the OPT algorithm. $v_x(i) = 1$ if OPT evicts node i from the GPU cache,

Algorithm 1 Dynamic two-level caching algorithm at each worker in training iteration t

```
1:  Input $S(t)$: sampled nodes in $t$
2:  Output updated cache states with $x(i), i \in N$ (probability of evicting node $i$ from GPU cache) and $q(i), i \in N \backslash A$ (probability of evicting node $i$ from CPU cache)
3:  $b :=$ number of nodes in $S(t)$ and cached in CPU
4:  $c :=$ number of nodes in $S(t)$ and fetched from other servers
5:  for each node $i \in S(t)$ do
6:      if node $i$ is in the GPU cache, keep it there
7:      if node $i$ is in the CPU cache, evict it from the CPU cache and prepare to cache it into the GPU
8:      if node $i$ is fetched from another server, prepare to cache it into the GPU
9:      $x(i) := 0$, $q(i) := 1$
10: end for
11: // one-batch lookahead
12: for each node $i \in$ sampled batch of $t+1$ and $x(i) < 1$ do
13:     $x(i) := 0$
14: end for
15: for each node $i \in N \backslash S(t)$ and $x(i) < 1$ do
16:     $x(i) := \min\{1, x(i) + \alpha \cdot (x(i) + \beta)\}$
17: end for
18: $V_x := [0]*k_x$ // a $k_x$-dimensional vector recording eviction counts of each node in the GPU cache
19: $V_y := [0]*k_y$ // a $k_y$-dimensional vector recording eviction counts of each node in the CPU cache
20: do parallel
21:     $\gamma_x \sim$ Uniform$[O(1), O(\log k_x)]$
22:     for each node $i \in N \backslash S(t)$ and $x(i) < 1$ do
23:         $z \sim$ Uniform$[0, 1]$
24:         if $z \leq \gamma_x \cdot x(i)$ then
25:             $V_x(i) := V_x(i) + 1$
26:         end if
27:     end for
28: end parallel
29: $E :=$ TopK$(V_x, b + c)$ // Nodes to be evicted from GPU
30: for each node $i$ in E do
31:     $x(i) := 1$, $q(i) := 0$
32: end for
33: Replace nodes in $E$ in GPU cache by sampled nodes in $S(t)$ that were not in the GPU cache
34: $E := E \wedge (N \backslash A)$ // remove nodes in local graph partition from E
35: for each node $i \in N \backslash A$ and $q(i) < 1$ do
36:     $q(i) = \min\{1, q(i) + \frac{\min_{j \in N \backslash A} w(j) - w_1}{w(i) - w_1}\}$
37: end for
38: do parallel
39:     $\gamma_y \sim$ Uniform$[O(1), O(\log k_y)]$
40:     for each node $i \in N \backslash A$ and $q(i) < 1$ do
41:         $z \sim$ Uniform$[0, 1]$
42:         if $z \leq \gamma_y \cdot q(i)$ then
43:             $V_y(i) := V_y(i) + 1$
44:         end if
45:     end for
46: end parallel
47: $F :=$ TopK$(V_y, |E|)$ // Nodes to be evicted from CPU
48: for each node $i$ in $F$ do
49:     $q(i) := 1$
50: end for
51: Replace nodes in $F$ in the CPU cache with nodes in $E$.
```

and 0, otherwise. $v_q(i) = 1$ if OPT removes node i from the CPU cache, and 0, otherwise. $x(i)$ and $q(i)$ are the decisions of online fractional algorithm. The main idea of the proof is to show that inequality (3) holds with all node evictions conducted by online fractional algorithm or offline optimal algorithm. Then it leads to $\Delta\text{On} \leq O(\log k)\Delta\text{OPT}$ and summing up over all training iterations, we get $\text{On} \leq O(\log k)\text{OPT}$ and complete our proof of the $O(\log k)$ competitive ratio. ☐

Theorem 2. *When the eviction fractions of nodes from the GPU and CPU caches increase by at least $1/\log k$ in each step (ii) and each step (iii) of the fractional algorithm (e.g., by setting $\Delta x = \Delta q = 1/\log k$), the objective value of (1), i.e., the overall feature fetching cost, incurred by Alg. 1 is at most $O(\log^2 k)$ times that of the online fractional algorithm.*

Proof. We give the proof sketch here and leave the full proof in our technical report. The main idea is that when the eviction probability of nodes from the CPU and GPU caches increases at least $1/\log k$ in each training iteration in the fractional algorithm, the expected fetching cost of our algorithm is at most $O(\log k)$ times the expected cost of the fractional algorithm. Evicting nodes with scale γ_x leads to γ_x times the cost without the scale factor, and the expected fetching cost without scale factor is exactly at most $O(\log k)$ times the fetching cost of fractional algorithm. With γ_x bounded by $O(\log k)$, the expected fetching cost of our algorithm is at most $O(\log^2 k)$ times that of the fractional algorithm. Our lookahead reduces the fetching cost of the next batch $S(t+1)$, thus reducing the total expected fetching cost. Therefore, the expected cost induced by Alg. 1 is at most $O(\log^2 k)$ times that of the fractional algorithm. ☐

Combining Theorem 1 and Theorem 2, we can readily show that Alg. 1 is $O(\log^3 k)$-competitive.

Theorem 3. *The two-level online caching algorithm in Alg. 1 is $O(\log^3 k)$-competitive in solving problem (1).*

We also show the time complexity of running our online caching algorithm.

Theorem 4. *Alg. 1 produces feasible cache update strategies in each training iteration with time complexity $O(k \log k)$.*

Proof. In each training iteration, Alg. 1 evicts $b+c$ nodes from the GPU cache, exactly the number of nodes that are requested and not cached in GPU yet, and $|E|$ nodes from the CPU cache, exactly the number of nodes that are evicted from GPU and not in the local graph partition. Therefore, Alg. 1 computes feasible cache update solutions. In each training iteration, Alg. 1 updates cache probabilities with $O(k)$ time complexity and chooses the top $b+c$ or $|E|$ entries with $O(k \log k)$ time complexity from a vector of size k_x or k_y. Therefore, the total time complexity is $O(k \log k)$ per training iteration. The algorithm is run on GPU for parallel computation of cache update, eviction trials and TopK operations. □

V. Experimental Evaluation

A. Methodology

Implementation. We implement a distributed GNN training system with the two-level cache based on the popular distributed GNN training framework, DGL 0.8.2 [28]. The caching strategies are implemented using Python 3.9.5 and PyTorch 1.10.2 [30]. We modify the sampler component to separate the sampling batches and training batches and add a fetcher to fetch features of nodes in the sampled batches. We implement the two-level cache between the sampler and trainer as shown in Sec. III. We also modify the training batch loader provided by DGL to a data loader reading batches from training batch queue. Besides, we reuse the DGL's neighbor sampling component, graph store and execution runtime [28]. We use Ring AllReduce [29] to exchange model parameters with primitives provided by the gloo backend of PyTorch [30].

Testbed. We run distributed GNN training on 4 fully-connected servers, each with one 1080Ti 11GB GPU, one Intel Xeon E5-1660 v4 (3.20 GHz) CPU and 48GB host memory. We limit the server NIC bandwidth to 1Gbps, 10Gbps and 20Gbps in our experiments.

Workload. We train three representative GNN models, GraphSAGE [1], GAT [2] and GCN [3] on the ogbn-products dataset (containing 2.4 million nodes, 61.9 million edges and 100 feature dimensions) [31] and the reddit dataset (containing 0.2 million nodes, 114.6 million edges and 602 feature dimensions) [1]. The GraphSAGE model uses a hidden size of 256 and the mean aggregator to aggregate neighbor features. In the GAT model, we use a hidden size of 128, 4 heads for each layer and 0.6 dropouts for both feature and attention layers. The GCN model has a hidden size of 256. We use the METIS algorithm [15] to partition the graph and the Adam optimizer [32] with a learning rate of 0.001 for GNN training. We adopt uniform neighbor sampling [1], with fan-outs of 5 and 10 for two-layer models, 5,10 and 15 for three-layer models.

Baselines. We compare our design with five baselines: (1) *Direct Fetch*, which does not cache any nodes in both GPU and CPU; (2) *Static caching of hot nodes*: cache nodes with the largest degrees in GPU; (3) *Static caching of neighbor nodes*: cache nodes in GPU that are 1-hop neighbors of nodes in the local graph partition; (4) *One-level LRU*: enable only the GPU cache which evicts nodes that are least recently requested; (5) *Two-level LRU*: enable both GPU cache and CPU cache using LRU, with the GPU cache evicting nodes to the CPU cache.

Settings. We set the fetching cost $w(i)$ according to our profiled fetching overhead among servers: 5, 3 and 1 from a server of 1 Gbps, 10 Gbps and 20Gbps bandwidth, respectively. We choose γ among 0.3, 0.5, 1, 2, and 5. We use grid search to find appropriate α and β (in Alg. 1) under different cache sizes and batch sizes, e.g., $\alpha = 1.9$ and $\beta = 0.01$ for cache size 500,000 and batch size 1024. We experiment on both a homogeneous server bandwidth setting (all servers use 20Gbps NIC bandwidth) and heterogeneous bandwidth settings (1Gbps bandwidth at one server and 20Gbps at the other three servers, unless stated otherwise).

B. GNN Training Convergence

We first compare GNN training convergence time with different baselines in both homogeneous and heterogeneous settings. The training batch size (number of training nodes in each iteration) at each worker is 1024 and both GPU and CPU cache sizes are $500,000$ (the number of cached nodes, with a 400-byte feature per node for ogbn-products). Table II gives the model training time to reach the target training accuracy for different GNN models with two or three layers. Fig. 3 further shows detailed model training convergence under the heterogeneous setting. GNN training using our caching strategies achieves up to 3.37× (5.40×) speed-up in the homogeneous (heterogeneous) setting, compared to training without caching (direct fetch), and outperforms all other baselines with $1.06\times$ to $3.03\times$ speed-up. Dynamic caching strategies (LRU and ours) in general perform better than static caching due to higher cache hit rates. One-level LRU and two-level LRU underperform static caching in some cases when training a two-layer model. It is because relatively low feature fetching cost is incurred when training a two-layer model, and dynamic caching has a higher policy time for cache update (i.e., the time for computing caching decisions and updating the cache entries), which offsets the benefit brought by the higher cache hit rate. Our algorithm achieves higher speed-up in the heterogeneous setting than in the homogeneous setting. The larger speed-up results from our lookahead mechanism and efficient weighted caching strategy, which gives nodes to be fetched from other servers with lower inter-connection bandwidth lower eviction probabilities from the CPU cache.

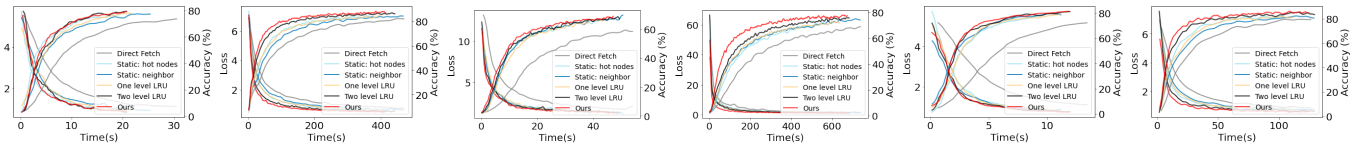
We further train the 3-layer GraphSAGE model on two datasets and in three heterogeneous settings as shown in Table III (e.g., (1,20,20,20) Gbps indicate one server with 1 Gbps bandwidth and three servers with 20 Gbps bandwidth). We use a batch size of 1024 and a GPU/CPU cache size of 500,000 or 50,000 for ogbn-products dataset and reddit dataset, respectively, since reddit has a relatively small number of nodes. Our caching design achieves 1.28× to 2.19× faster training convergence as compared to static caching with hot nodes.

C. Cache hit rate and average feature fetch time

We train the 3-layer GraphSAGE model [1] with 100 iterations (batches), and obtain the hit rate of our two-level cache (the ratio of the number of nodes that are cached in the GPU or CPU caches divided by the number of nodes in the training batch) and the average feature fetching time in each iteration (the time from when the fetcher obtains the sampled batch to when it sends the training batch to the training batch queue). We vary the training batch size per worker and GPU cache size, and set the CPU cache size to the same as the GPU cache size for all two-level caches.

Table II: GNN Training Convergence Time (seconds). $\times$ indicates the speed-up computed by dividing the respective convergence time by the convergence time achieved with our caching algorithm.

Dataset		ogbn-products											
Model		GraphSAGE				GAT				GCN			
# of Layers / Convergence Accuracy		2 / 85%		3 / 90%		2 / 77%		3 / 80%		2 / 89%		3 / 90%	
Homo-geneous	Direct Fetch	15.83	1.47$\times$	228.38	2.40$\times$	79.23	1.79$\times$	395.388	2.17$\times$	46.85	1.51$\times$	345.56	3.37$\times$
	Static (Hot Node)	13.56	1.26$\times$	115.03	1.21$\times$	54.14	1.23$\times$	256.95	1.41$\times$	33.71	1.08$\times$	116.83	1.14$\times$
	Static (Neighbor)	13.84	1.28$\times$	168.43	1.77$\times$	57.39	1.30$\times$	282.20	1.53$\times$	43.09	1.39$\times$	135.38	1.32$\times$
	One-level LRU	11.86	1.10$\times$	137.01	1.44$\times$	52.42	1.19$\times$	278.61	1.53$\times$	35.40	1.14$\times$	170.29	1.66$\times$
	Two-level LRU	12.71	1.18$\times$	130.54	1.37$\times$	66.35	1.50$\times$	195.79	1.07$\times$	40.27	1.30$\times$	156.54	1.53$\times$
	Ours	10.78	/	95.06	/	44.15	/	182.60	/	31.08	/	102.57	/
Hetero-geneous	Direct Fetch	90.89	2.42$\times$	1604.52	4.15$\times$	338.49	2.52$\times$	1761.79	3.29$\times$	259.26	3.98$\times$	1972.20	5.40$\times$
	Static (Hot Node)	42.77	1.14$\times$	789.29	2.04$\times$	189.04	1.41$\times$	1084.76	2.03$\times$	104.12	1.57$\times$	934.88	2.56$\times$
	Static (Neighbor)	61.59	1.64$\times$	1169.61	3.03$\times$	196.82	1.46$\times$	1088.24	2.03$\times$	143.41	2.16$\times$	958.81	2.62$\times$
	One-level LRU	41.56	1.10$\times$	855.39	2.21$\times$	146.15	1.09$\times$	834.61	1.56$\times$	95.88	1.44$\times$	1012.07	2.77$\times$
	Two-level LRU	43.10	1.15$\times$	686.29	1.78$\times$	142.17	1.06$\times$	783.69	1.46$\times$	98.39	1.48$\times$	521.67	1.43$\times$
	Ours	37.51	/	386.20	/	134.46	/	535.43	/	66.51	/	365.32	/



(a) 2-layer GraphSAGE (b) 3-layer GraphSAGE (c) 2-layer GAT (d) 3-layer GAT (e) 2-layer GCN (f) 3-layer GCN

Figure 3: GNN Training Convergence (with Gaussian filter $\sigma = 2$): heterogeneous server bandwidths

Table III: Convergence Time (seconds) of 3-layer GraphSAGE on different datasets and heterogeneous settings

Dataset	ogbn-products		reddit	
Convergence Accuracy	90%		95%	
Settings	Static(Hot Node)	Ours	Static(Hot Node)	Ours
(1,20,20,20) Gbps	789.29	386.20	841.72	656.32
(1,10,20,20) Gbps	858.74	391.47	888.89	677.17
(1,10,10,20) Gbps	959.80	462.84	938.95	716.38

a) Homogeneous server bandwidths: Fig. 4 shows that our algorithm with one batch lookahead increases the hit rate by up to 32%, 41%, and 11% as compared to the static-hot caching, one-level LRU and two-level LRU, respectively. Caching with lookahead achieves 7% higher cache hit rate than without lookahead. Fig. 5 shows that our algorithm saves 67%, 69% and 36% feature fetching time per training iteration as compared to static-hot caching, one-level LRU and two-level LRU, respectively, and up to 21% with lookahead than without.

b) Heterogeneous server bandwidths: Our further experiments on heterogeneous server bandwidths also show that our algorithm with lookahead achieves up to 28%, 37% and 8% hit rate increase as compared to static-hot cache, one-level LRU and two-level LRU, respectively. The lookahead mechanism enables up to 4% higher hit rate than without. Besides, our cache saves 91%, 92% and 76.6% feature fetching time per iteration compared to static-hot, one-level LRU and two-level LRU, respectively, and up to 49% with lookahead than without. Our algorithm achieves higher time reduction in the heterogeneous setting, due to retaining more nodes fetched through costly inter-server connection in the cache saves more feature re-fetching time. The result figures in the heterogeneous setting are included in our technical report due to space limit.

D. Cache Policy Time

We compare the policy time which includes the caching decision computation time and cache entry update time per training iteration under different batches and cache sizes, when training the 3-layer GraphSAGE in the heterogeneous setting. Fig. 6 shows that with the increase of the batch size, the policy time increases sub-linearly, as the number of cache entries to be updated increases. When increasing the cache size, increase of the policy time with dynamic caching strategies slows down. This is because when the cache gets larger, the cache hit rate is higher and the increase of cache entries that require updating becomes smaller. Although the policy time of dynamic caching is non-trivial as compared to static caching (which is zero since no cache update is needed), dynamic caching reduces the end-to-end model convergence time since it largely reduces the feature fetching time and the cache update can overlap with model training.

E. Impact of CPU cache size

We train the 3-layer GraphSAGE with a batch size of 1024 and a GPU cache size of 50,000 in the homogeneous setting, and vary the CPU cache size from 50,000 to 1,000,000. In Fig. 7, we observe that the cache hit rate increases and the feature fetch time decreases when the CPU cache gets larger. The marginal reduction of fetch time per unit CPU-cache-size increase decreases. The policy time for updating the CPU cache increases, and the increase is small compared to the reduction of the fetch time. According to these observations, a larger CPU cache is more beneficial.

F. Impact of lookahead batch number

We train the 3-layer GraphSAGE with both GPU and CPU cache sizes fixed to 500,000 in the homogeneous setting, and vary the number of lookahead batches. When we look ahead h sampled batches in each training iteration, we reduce the GPU eviction probability of node i by $1/2^{h-1}$ if it appears in the h-th lookahead batch (Line 13 in Alg. 1). For example, if the node appears in the next batch (i.e., first lookahead batch), then its eviction probability is set to 0; if the node appears

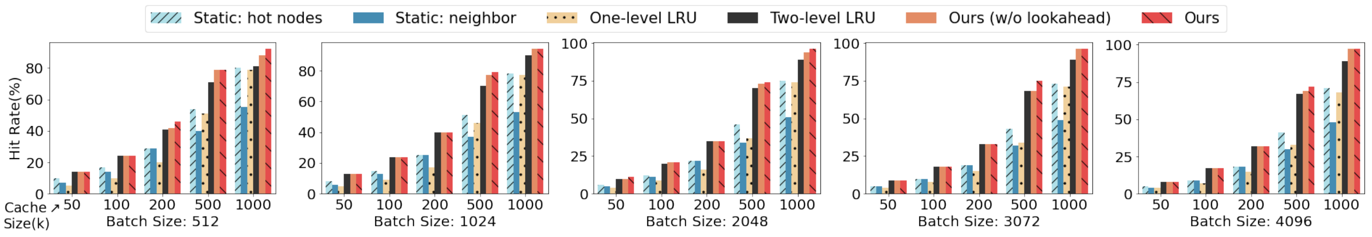


Figure 4: Cache hit rate under different cache sizes and batch sizes: homogeneous server bandwidths.

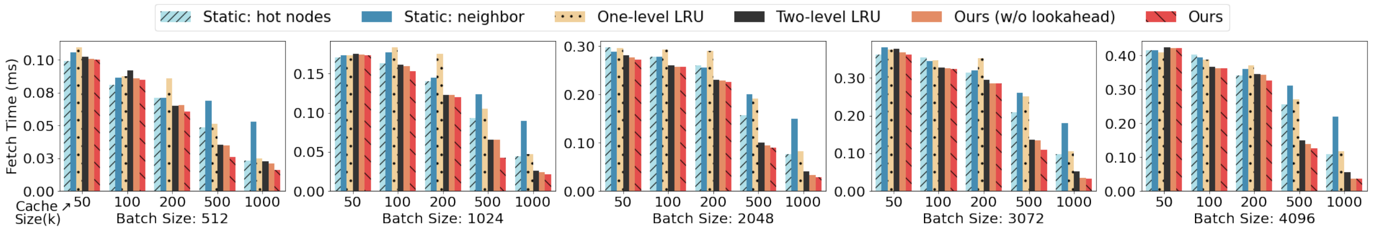


Figure 5: Average feature fetch time under different cache sizes and batch sizes: homogeneous server bandwidths.

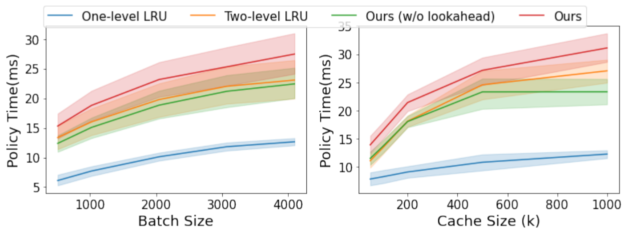


Figure 6: Cache policy time

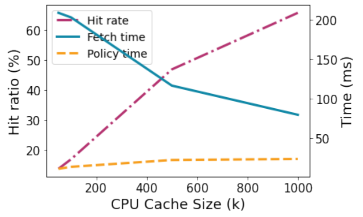


Figure 7: Hit rate and feature fetch time: different CPU cache sizes

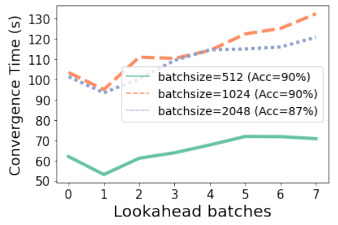


Figure 8: Convergence time: different lookahead batch numbers

in the second and third lookahead batches given its previous eviction probability p, then its eviction probability becomes $p' = p - 1/2p - 1/4p$. Fig. 8 shows the model convergence time to reach the respective target model accuracy. We see that model training with one-batch lookahead leads to the best performance. The convergence time increases with more lookahead batches because sampling of more future batches takes more time, leading to delay of cache update by the fetcher and the trainer training (i.e., less efficient sampler/fetcher/trainer pipelining), and processing multiple lookahead batches leads to higher cache policy time by the fetcher as well.

G. Comparison with offline optimal caching

We compare the average hit rate and one epoch training time with the offline optimal two-level caching of solving problem (1) exactly (by pre-sampling the batches in one training epoch and evicting nodes that will be requested in the farthest future - shown to be optimal in [33]). We train the 3-layer GAT model in the homogeneous setting. One epoch includes 95 batches (aka training iterations) when the batch size is 512 and 47 batches when the batch size is 1024 on ogbn-products dataset. We vary the GPU cache size and set the CPU cache size to the same as GPU cache size. As Fig. 9 shows, our algorithm achieves lower hit rates than the offline optimal cache, but better training speeds in most cases. It is because we overlap sampling, feature fetching and GNN computation, while the offline optimal cache requires nontrivial time for pre-sampling and constructing the look-up table for nodes' farthest request time. Besides, the offline optimal cache occupies much more space in GPU memory than ours due to the look-up table.

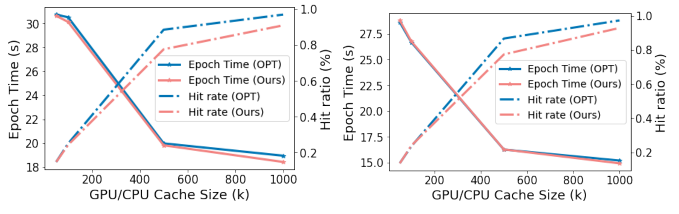

(a) Batch size: 512 (b) Batch size: 1024

Figure 9: Cache hit rate and epoch time compared to offline optimum: training 3-layer GAT in homogeneous setting

VI. Conclusion

This paper proposes an efficient two-level dynamic cache in distributed GNN training, exploiting both GPU memory and host memory for feature fetching expedition. We carefully design sampling, feature fetching and training parallelization, and decide the caching strategies based on online optimization and a lookahead mechanism. Our online two-level caching algorithm considers heterogeneous feature fetching costs from different servers and leverages parallel probabilistic node eviction trials to achieve fast online caching policies. We prove an $O(\log^3 k)$ competitive ratio of our online algorithm. Testbed experiments show that our design achieves up to 5.4× convergence speed-up on representative GNN training workloads under various fetching cost settings.

Acknowledgment

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17204619, 17208920 and 17207621.

References

[1] W. L. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 2017.

[2] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[3] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *ICLR 2017*. OpenReview.net, 2017.

[4] H. Kwak, C. Lee, H. Park, and S. B. Moon, "What is twitter, a social network or a news media?" in *WWW 2010*, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds. ACM, 2010, pp. 591–600.

[5] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur, "Protein interface prediction using graph convolutional networks," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 2017, pp. 6530–6539.

[6] L. Zhao, Y. Song, C. Zhang, Y. Liu, P. Wang, T. Lin, M. Deng, and H. Li, "T-GCN: A temporal graph convolutional network for traffic prediction," *IEEE Trans. Intell. Transp. Syst.*, vol. 21, no. 9, pp. 3848–3858, 2020.

[7] M. Fire and C. Guestrin, "Over-optimization of academic publishing metrics: observing Goodhart's Law in action," *GigaScience*, vol. 8, no. 6, 05 2019, giz053. [Online]. Available: https://doi.org/10.1093/gigascience/giz053

[8] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowl. Inf. Syst.*, vol. 42, no. 1, pp. 181–213, 2015.

[9] J. J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 548–556.

[10] S. Gandhi and A. P. Iyer, "P3: distributed deep graph learning at scale," in *OSDI 2021*, 2021.

[11] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "Distdgl: Distributed graph neural network training for billion-scale graphs," in *10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms, IA3 2020, Atlanta, GA, USA, November 11, 2020*. IEEE, 2020, pp. 36–44.

[12] T. Kaler, N. Stathas, A. Ouyang, A. Iliopoulos, T. B. Schardl, C. E. Leiserson, and J. Chen, "Accelerating training and inference of graph neural networks with fast sampling and pipelining," in *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, D. Marculescu, Y. Chi, and C. Wu, Eds. mlsys.org, 2022.

[13] H. Mostafa, "Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs," in *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, D. Marculescu, Y. Chi, and C. Wu, Eds. mlsys.org, 2022.

[14] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu, "Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads," in *OSDI*. USENIX Association, 2021, pp. 495–514.

[15] G. Karypis and V. Kumar., "Metis–unstructured graph partitioning and sparse matrix ordering system," in *Technical Report*, 1995.

[16] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "Pagraph: Scaling GNN training on large graphs via computation-aware caching," in *SoCC '20*. ACM, 2020.

[17] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou, "GNNLab: a factored system for sample-based GNN training over gpus," in *EuroSys '22*. ACM, 2022.

[18] J. Dong, D. Zheng, L. F. Yang, and G. Karypis, "Global neighbor sampling for mixed CPU-GPU training on giant graphs," in *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, F. Zhu, B. C. Ooi, and C. Miao, Eds. ACM, 2021, pp. 289–299.

[19] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo, "BGL: gpu-efficient GNN training by optimizing graph data I/O and preprocessing," in *NSDI'23*, 2023.

[20] G. Nikolentzos and M. Vazirgiannis, "Random walk graph neural networks," in *NeurIPS 2020*, 2020.

[21] N. Bansal, N. Buchbinder, and J. Naor, "A primal-dual randomized algorithm for weighted paging," *J. ACM*, vol. 59, no. 4, pp. 19:1–19:24, 2012.

[22] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke, "An $O(\log k)$-competitive algorithm for generalized caching," *ACM Trans. Algorithms*, vol. 15, no. 1, pp. 6:1–6:18, 2019.

[23] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Commun. ACM*, vol. 28, no. 2, pp. 202–208, 1985.

[24] N. Bansal, N. Buchbinder, and J. Naor, "Randomized competitive algorithms for generalized caching," *SIAM J. Comput.*, vol. 41, no. 2, pp. 391–414, 2012.

[25] T. Lykouris and S. Vassilvitskii, "Competitive caching with machine learned advice," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 3302–3311.

[26] N. Bansal, C. Coester, R. Kumar, M. Purohit, and E. Vee, "Learning-augmented weighted paging," in *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, J. S. Naor and N. Buchbinder, Eds. SIAM, 2022, pp. 67–89.

[27] N. Bansal, J. S. Naor, and O. Talmon, "Efficient online weighted multi-level paging," in *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, K. Agrawal and Y. Azar, Eds. ACM, 2021, pp. 94–104.

[28] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv:1909.01315*, 2019.

[29] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *J. Parallel Distributed Comput.*, vol. 69, no. 2, pp. 117–124, 2009.

[30] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS '19*, 2019.

[31] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *CoRR*, vol. abs/2005.00687, 2020.

[32] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[33] L. A. Belady, "A study of replacement algorithms for virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966. [Online]. Available: https://doi.org/10.1147/sj.52.0078