DiskGNN: Bridging I/O Efficiency and Model Accuracy for Out-of-Core GNN Training

RENJIE LIU[*], Southern University of Science and Technology, China
YICHUAN WANG[*], UC Berkeley, United States
XIAO YAN[†], Centre for Perceptual and Interactive Intelligence, Hong Kong
HAITIAN JIANG, New York University, United States
ZHENKUN CAI, Amazon, United States
MINJIE WANG, AWS Shanghai AI Lab, China
BO TANG[†], Southern University of Science and Technology, China
JINYANG LI, New York University, United States

Graph neural networks (GNNs) are models specialized for graph data and widely used in applications. To train GNNs on large graphs that exceed CPU memory, several systems have been designed to store data on disk and conduct out-of-core processing. However, these systems suffer from either *read amplification* when conducting random reads for node features that are smaller than a disk page, or *degraded model accuracy* by treating the graph as disconnected partitions. To close this gap, we build *DiskGNN* for high I/O efficiency and fast training without model accuracy degradation. The key technique is *offline sampling*, which decouples *graph sampling* from *model computation*. In particular, by conducting graph sampling *beforehand* for multiple mini-batches, DiskGNN acquires the node features that will be accessed during model computation and conducts pre-processing to pack the node features of each mini-batch contiguously on disk to avoid read amplification for computation. Given the feature access information acquired by offline sampling, DiskGNN also adopts designs including *four-level feature store* to fully utilize the memory hierarchy of GPU and CPU to cache hot node features and reduce disk access, *batched packing* to accelerate feature packing during pre-processing, and *pipelined training* to overlap disk access with other operations. We compare DiskGNN with state-of-the-art out-of-core GNN training systems. The results show that DiskGNN has more than 8× speedup over existing systems while matching their best model accuracy. DiskGNN is open-source at https://github.com/Liu-rj/DiskGNN.

CCS Concepts: • **Computing methodologies → Machine learning**; • **Information systems → Data management systems**.

Additional Key Words and Phrases: Graph Neural Networks, Model Training System, Out-of-core Computing

[*]Renjie Liu and Yichuan Wang contributed equally, work done during Renjie's internship at AWS Shanghai AI Lab.
[†]Dr. Xiao Yan and Dr. Bo Tang are the corresponding authors.

Authors' Contact Information: Renjie Liu, Southern University of Science and Technology, Shenzhen, China, liurj2023@mail.sustech.edu.cn; Yichuan Wang, UC Berkeley, Berkeley, United States, yichuan_wang@berkeley.edu; Xiao Yan, Centre for Perceptual and Interactive Intelligence, Hong Kong, Hong Kong, yanxiaosunny@gmail.com; Haitian Jiang, New York University, New York, United States, haitian.jiang@nyu.edu; Zhenkun Cai, Amazon, Santa Clarla, United States, zkcai@amazon.com; Minjie Wang, AWS Shanghai AI Lab, Shanghai, China, minjiw@amazon.com; Bo Tang, Southern University of Science and Technology, Shenzhen, China, tangb3@sustech.edu.cn; Jinyang Li, New York University, New York, United States, jinyang@cs.nyu.edu.

ACM 2836-6573/2025/2-ART34
https://doi.org/10.1145/3709738

Table 1. Execution statistics of disk-based GNN systems and our DiskGNN on the Ogbn-papers100M graph.

Metrics	Ginex [39]	MariusGNN [52]	DiskGNN (ours)
End-to-end time (hrs)	9.72	3.66	**1.09**
Pre-processing time (hrs)	*1.66*	*0.81*	*0.03*
Training time (hrs)	*8.06*	*2.85*	*1.06*
Avg. epoch time (sec)	580	205	**76.3**
Disk access time (sec)	*412*	*27.1*	*51.2*
Disk access volume (GB)	*484*	*6.46*	*73.9*
Final test accuracy (%)	**65.9**	64.0	**65.9**

ACM Reference Format:
Renjie Liu, Yichuan Wang, Xiao Yan, Haitian Jiang, Zhenkun Cai, Minjie Wang, Bo Tang, and Jinyang Li. 2025. DiskGNN: Bridging I/O Efficiency and Model Accuracy for Out-of-Core GNN Training. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 34 (February 2025), 27 pages. https://doi.org/10.1145/3709738

1 Introduction

Graph data is ubiquitous in domains such as e-commerce [56], finance [11, 61], bio-informatics [44], and social networks [15]. As machine learning models specialized for graph data, graph neural networks (GNNs) achieve high accuracy for various graph tasks (e.g., node classification [25], link prediction [69], and graph clustering [50]) and hence are used in applications like recommendation [63], fraud detection [55], and pharmacy [18]. In particular, GNNs compute an embedding for each node v in the graph by recursively aggregating the input features of v's neighbors. To alleviate the exponential neighbor explosion, *graph sampling* is widely adopted to select some neighbors for each node during aggregation.

Large graphs with millions of nodes and billions of edges are common [19, 24, 26]. They may exceed the main memory capacity and require solutions to scale up GNN training. Some systems (e.g., DistDGL [72], DSP [6], and P3 [13]) conduct distributed training by partitioning the graph data and training computation over multiple machines. However, these distributed solutions are expensive to deploy and suffer from low GPU utilization due to heavy inter-machine communication. Now that solid-state disks (SSDs) are cheap, capacious, and reasonably fast (with a bandwidth of 2-7GB/s), some systems like Ginex [39], GIDS [38], MariusGNN [52], and Helios [47] conduct out-of-core training on a single machine by storing graph data on disk. Compared with distributed systems, these disk-based solutions are more accessible and cost-effective.

Existing disk-based systems and their limitations. Most disk-based systems (e.g., Ginex [39], GIDS [38] and Helios [47]) follow the workflow of in-memory GNN systems (e.g., DGL [9]) and conduct *fine-grained disk access*. For each mini-batch of training, they first perform graph sampling to determine the required graph nodes and corresponding features, and then collect those node features and perform model computation. The difference from in-memory systems is that the node features are read from disk instead of CPU memory. These systems adopt various optimizations (e.g., caching popular node features in CPU memory [39] and using asynchronous I/O for disk access [47]), but a key problem remains with their random small read pattern. To be specific, the node features required by a mini-batch are not contiguously stored on disk, causing each node feature (typically < 512B) to be fetched as a 4KB disk page. This causes substantial *read amplification* and poor I/O efficiency. As shown in Table 1, Ginex spends most of its training time on disk access, and its disk read volume (484GB) is much larger than the sampled node features (73.9GB, as achieved by our DiskGNN).

To avoid read amplification, MariusGNN [52] conducts *block-based disk access* by organizing a graph as node feature partitions and edge partitions, with each partition containing the edges between two node partitions. Pre-processing is conducted before training so that each node and edge partition takes up a consecutive disk region. To conduct training, MariusGNN loads some feature partitions along with the edge partitions between them into CPU memory, treats the induced graph as the complete graph, and regularly swaps the memory-resident partitions. As shown in Table 1, MariusGNN has a small disk traffic because it reads large partitions without read amplification. However, model accuracy is degraded because MariusGNN does not execute the training logic faithfully, i.e., for each graph node v, MariusGNN limits graph sampling to v's memory-resident neighbors while all of v's neighbors should be considered. Training longer cannot compensate for such accuracy degradation as shown by our experiments in § 7.

DiskGNN. The above analysis shows a tension between I/O efficiency and model accuracy in existing systems. As such, we build DiskGNN to achieve both of them based on a new training paradigm called *offline sampling*. In particular, offline sampling decouples the two main stages of GNN training, i.e., graph sampling and model computation, and conducts graph sampling for many mini-batches before model computation. In this way, DiskGNN acquires the node features that will be accessed during model computation and uses the information to adjust the data layout for efficient access as pre-processing. Offline sampling does not degrade model accuracy because it faithfully executes the training logic (as opposed to MariusGNN), i.e., sampling and training maintain their procedures without modification. For the same reason, offline sampling generalizes across different GNN models and graph sampling schemes.

Specifically, we group the node features according to their access frequencies and assign them to a *four-level feature store* that involves GPU memory, CPU memory, and disk, with the principle that more popular node features should be kept in faster storage. More importantly, to avoid read amplification in disk access, we pre-process the node features required by each mini-batch by packing them into a consecutive disk region. The packing scheme could consume large disk space as one node feature may be replicated for different mini-batches. To tackle this problem, we trade off between I/O efficiency and disk space with a hybrid packing strategy, which consists of shared features among mini-batches that use *node reordering* to reduce read amplification and dedicated features for each mini-batch that use the original *consecutive packing*. The rationale of reordering is to place the node features required by a mini-batch adjacent to each other such that they can be read with a small number of disk pages. We also accelerate the pre-processing of DiskGNN using *batched packing*, which reads a large chunk of node features and packs these features for all min-batches each time. This ensures that pre-processing involves only sequential disk access and avoids repetitive data read across mini-batches.

DiskGNN is implemented on top of the Deep Graph Library (DGL) [57], one of the most popular open-source frameworks for graph learning. DiskGNN adopts a pipeline to overlap the disk access of a mini-batch with the model computation of its preceding mini-batches. We also carefully implement the I/O operations of DiskGNN for efficiency and provide simple APIs for usability.

We evaluate DiskGNN on 4 large public graph datasets and 2 predominate GNN model architectures. The results show that DiskGNN consistently yields shorter training time than both Ginex and MarisGNN with an average speedup of 7.5x and 2.5x, respectively. Moreover, DiskGNN matches the model accuracy of Ginex while being significantly more accurate than MarisGNN. We also conduct micro experiments to validate our designs. The results show that the batched packing can accelerate pre-processing by 7.3x, and the training pipeline can speed up training by over 2x.

To summarize, we make the following contributions:

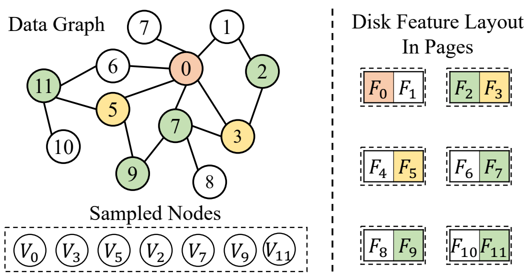

Data Graph
Sampled Nodes
V_0 V_3 V_5 V_2 V_7 V_9 V_{11}
Disk Feature Layout In Pages
F_0 F_1 F_2 F_3
F_4 F_5 F_6 F_7
F_8 F_9 F_{10} F_{11}

Fig. 1. An illustration for node-wise graph sampling. The seed node is v_0, and the sampled 1-hop and 2-hop neighbors are marked in yellow and green, respectively. We assume two node features take up a disk page.

- We observe that existing out-of-core GNN training systems face the tension between I/O efficiency and model accuracy.
- We design DiskGNN to achieve both I/O efficiency and model accuracy with offline sampling, i.e., collecting data access information beforehand to optimize data layout for efficient access.
- We propose a suite of designs tailored for on-disk workloads to make DiskGNN efficient, including a four-level feature store, batched feature packing, and pipelined training.

2 Background on GNN Training

GNN basics. GNN models take a data graph $G = (V, E)$, where V and E are the node set and edge set, and each node $v \in V$ comes with a feature vector h_v^0 that describes its properties. For instance, in the Ogbn-papers100M dataset, each node is a paper, an edge indicates that one paper cites another paper, and the node feature is a 128-dimension float embedding of the paper's title and abstract. A GNN model stacks multiple graph aggregation layers, with each layer aggregating the embeddings of a node's neighbors. Specifically, in the k^{th} layer, the output embedding h_v^k of node v is computed as

$$h_v^k = \sigma\big[W^k \cdot \big(h_v^{k-1} + AGG_k\big(\{h_u^{k-1}, \forall u \in \mathcal{N}(v)\}\big)\big)\big], \tag{1}$$

where h_v^{k-1} and h_u^{k-1} are the embeddings of nodes v and u in the $(k-1)^{\text{th}}$ layer, and set $\mathcal{N}(v)$ contains the neighbors of node v. For the first layer, h_u^0 is the input node feature. $AGG_k(\cdot)$ is the neighbor aggregation function, W^k is a learnable projection matrix, and $\sigma(\cdot)$ is the activation function. By expanding Eq. (1), it can be observed that for a K-layer GNN model, computing the final output embedding h_v^K for node v involves all its K-hop neighbors.

Graph sampling for GNN training. GNN training is usually conducted in mini-batches, where each mini-batch computes the output embeddings for some seed nodes (i.e., nodes with labels) and updates the model according to a loss function that measures the difference between model output and ground truth. Training is said to finish an epoch when all seed nodes are used once, and typically many epochs are required for convergence. As a seed node can have many K-hop neighbors, graph sampling is widely used to reduce training cost by sampling some of the neighbors for computation [8, 17, 20, 66, 68, 73]. For instance, the popular node-wise sampling [17] uses a fan-out vector to specify the number of neighbors to sample and conducts neighbor sampling independently for the nodes in the same layer. For instance, the left plot of Figure 1 uses a fanout of <2,2>, which means that sampling is conducted for 2 steps, and each node samples 2 neighbors for both steps. In the first step, $\{v_3, v_5\}$ are sampled as the neighbors of the seed node v_0; in the second step, v_3 samples its neighbors $\{v_2, v_7\}$ while v_5 samples $\{v_9, v_{11}\}$.

3 Offline Sampling

Challenge of out-of-core training. For graph datasets, node features are usually much larger than graph topology, and thus out-of-core training needs to store (most of) the node features on disk. For each mini-batch, some node features need to be fetched from disk to GPU memory to conduct model computation. For instance, in Figure 1, handling a mini-batch sampled from seed node v_0 requires $\{v_0, v_3, v_5, v_2, v_7, v_9, v_{11}\}$. However, disk is accessed with 4KB page as the minimum granularity but each node feature, usually several hundred bytes (e.g. 512 bytes for Ogbn-papers100M), is smaller than a disk page. As the required node features are not contiguous on disk, many small random reads are used to fetch them, which causes read amplification and makes the disk traffic much larger than the required features (i.e., Ginex in Table 1). The right plot of Figure 1 shows an example where two node features take up a disk page. Training only requires 7 node features but the disk read traffic is 12 node features due to amplification.

Offline sampling for proactive data layout optimization. Each mini-batch of GNN training involves two main steps, i.e., *graph sampling* to determine the computation graph and required node features, and *model computation* to run the computation and update the model. Existing systems couple the two steps for each mini-batch, i.e., they run the mini-batches sequentially, and each mini-batch first conducts graph sampling and then model computation. The insight of offline sampling is that the model computation for a mini-batch does not have to follow immediately after graph sampling; instead, we can sample many mini-batches before running their model computation. This decoupling allows us to collect the features to be accessed beforehand and optimize the data layout in advance for efficient access during model computation. In particular, the following chances emerge.

- *Cache configuration to reduce disk access.* With knowledge of nodes to be accessed by all mini-batches from offline sampling, we can rank the nodes by their access frequencies and cache more popular nodes in faster memory (e.g., GPU memory and CPU memory) to reduce disk access. Such a cache configuration is optimal in that it minimizes the total number of node features fetched from the disk.
- *Feature packing to avoid read amplification.* For each mini-batch, we can first collect all node features it requires and store them contiguously as a disk block (called feature packing) beforehand. During model computation, we can read all these features at once without read amplification.
- *Batched packing for many mini-batches.* It is inefficient to conduct feature packing *individually* for each mini-batch because collecting the node features still requires small random disk reads. However, when handling many mini-batches, most of the node features are accessed by at least one of these mini-batches. This observation allows us to switch the feature packing scheme from *mini-batch oriented* to *feature chunk oriented*. In particular, we can read a large chunk of node features from disk each time, find the features in the chunk that are required by each mini-batch, and append these features to the disk storage of each mini-batch. This is efficient as it involves only large sequential disk access.

DiskGNN exploits the above optimization opportunities enabled by offline sampling. We note that feature packing essentially trades *disk space* for *access efficiency* because a node feature may be required by multiple mini-batches and thus replicated by feature packing. The increased disk space consumption is usually not a problem because disk capacity is cheap. When the space overhead of packing is too large, offline sampling may use *node reordering* [62, 71], a classical technique in graph processing, to reduce (instead of eliminate) read amplification. The rationale is to renumber the graph nodes such that the node features accessed concurrently by one mini-batch are likely to be in the same disk page.

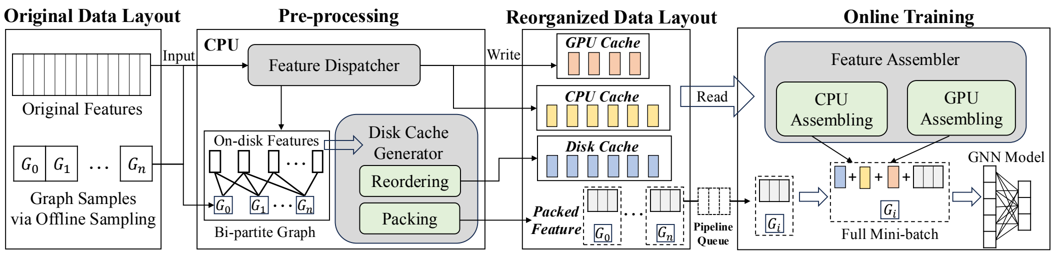


Fig. 2. DiskGNN system architecture and workflow.

Besides disk-based training, offline sampling may also benefit cloud-based training by allowing *flexible instance selection*. This is because public clouds (e.g., AWS [1] and Azure [2]) provide machine instances with different configurations (e.g., memory capacity, CPU choices and GPU presence) and thus different prices; and by decoupling graph sampling and model computation, we can choose different instances for them to reduce monetary cost. In particular, graph sampling conducts small random access to graph topology, and thus it suits an instance with enough CPU memory to hold the graph topology. We may not use GPU because the computation of sampling is lightweight. Model computation involves neural networks and thus suits an instance with GPU. We can reduce the idle time of expensive GPU by conducting graph sampling and packing the node features beforehand on a cheaper CPU instance.

Generality of offline sampling. Offline sampling and hence DiskGNN generalize across different GNN models and graph sampling algorithms because they are treated as black boxes. That is, offline sampling only assumes that graph sampling produces samples and model computation consumes graph samples to update the model, and there are no constraints on the internals of the sampling and training algorithms. Thus, offline sampling also applies to recent GNN models that are popular for link or high-order relation prediction (e.g., SEAL [69], Shadow [67], and GraIL [49]) as they can be regarded as special forms of graph sampling that jointly consider multiple seed nodes.

4 DiskGNN System Overview

DiskGNN implements the ideas discussed in §3, and Figure 2 depicts its workflow. We assume that graph sampling has been conducted to obtain the graph samples of many mini-batches. This can be done by running existing GNN frameworks like DGL [57] and PyG [12]. At initialization, DiskGNN takes the graph samples of many mini-batches and all node features of the data graph as input, and assumes that they are stored on disk. Then, DiskGNN conducts pre-processing to construct a data layout for efficient access during model training. This is achieved by storing the node features in a *four-level feature store* that involves GPU memory, CPU memory, and disk. After pre-processing, DiskGNN runs model training by going over the mini-batches to update the model. For each mini-batch, DiskGNN loads its graph sample from the disk, assembles the required node features from the *four-level feature store*, and feeds these data to the GPU for model computation. The data reading and model computation of different mini-batches are overlapped with pipelining.

Four-level hierarchical feature store. During pre-processing, DiskGNN first collects the access frequencies of all node features. This procedure is lightweight, as it simply keeps a counter for each node and streams the graph samples from disk. Nodes with higher access frequencies are considered more popular and DiskGNN determines the node feature layout according to popularity.

- *GPU cache* stores the most popular node features in GPU memory. GPU cache size is configured by excluding the working memory for model training from overall GPU memory.

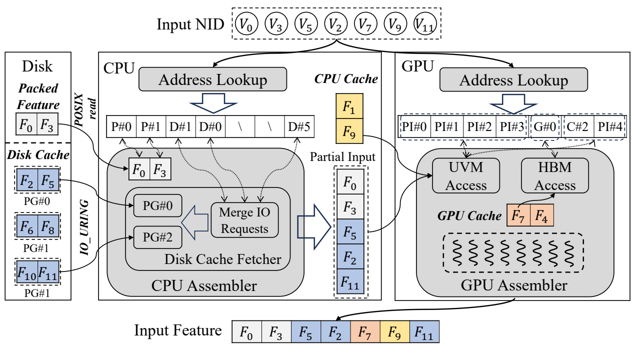


Fig. 3. An example of feature assembling in DiskGNN. G#, C#, D#, P#, and PI# denote that a node feature locates in GPU cache, CPU cache, disk cache, packed chunk, and partial input, respectively. PG refers to disk pages in the disk cache.

- *CPU cache* stores the second most popular node features in CPU memory, and GPU accesses the CPU cache as unified virtual memory (UVM) [35] via PCIe during training. CPU cache size is configured by excluding the memory required to assemble the graph samples and node features of several mini-batches from the overall CPU memory to prepare for training.
- *Disk cache* stores the third most popular node features. As discussed in §3, feature packing may consume large disk space because it can store multiple copies of the same node feature. DiskGNN allows the user to specify the maximum disk space the system can use, and disk cache is activated when feature packing exceeds the space limit. The node features in the disk cache are not replicated, instead, they are shared among mini-batches and laid out on disk by reordering in the hope that the node features required by a mini-batch are stored in a small number of disk pages. As such, the disk cache reduces rather than eliminates read amplification.
- *Packed feature chunk* stores the node features that are not in the above three cache components. For each mini-batch, DiskGNN creates a disk chunk to store its packed node features, and the graph sample of the mini-batch is also kept in the chunk since the node features and graph samples will be fetched together. Reading a packed feature chunk does not have read amplification because each chunk is usually larger than disk page size.

DiskGNN decides whether to use the disk cache by analyzing the access frequencies of the node features. The details of determining the node features to store in the disk cache and node reordering are discussed in §5.1, and how to conduct pre-processing and fill in the four-level feature store efficiently is discussed in § 5.2.

Feature assembling. During training, DiskGNN takes two steps to assemble the node features in the feature store for each min-batch. In the first step, the CPU reads the features in the disk cache and packed feature chunks and prepares them as *partial input* for the GPU. In the second step, the GPU reads the GPU cache, CPU cache, and partial input to obtain all the required features. For both steps, DiskGNN uses *hash maps* to look up the addresses of the required node features. Figure 3 provides a working example of the feature assembling process, where the required node features are $\{0, 3, 5, 2, 7, 9, 11\}$, and nodes $\{1, 9\}$ and $\{7, 4\}$ are stored in the CPU and GPU cache.

For CPU assembling, DiskGNN interprets the node IDs to address their locations on disk and launches disk I/O to read their features from the packed chunks (P#) and disk cache (D#). Packed features $\{0, 3\}$ reside in one feature chunk and are loaded via one disk page. For features $\{2, 5, 11\}$ in the disk cache, DiskGNN first merges the requests pointing to the same disk page to eliminate duplicate accesses and then reads the required pages. In particular, $\{2, 5\}$ are in the same disk page due to node ordering and thus fetched via one page. For GPU assembling, DiskGNN interprets the node IDs to address their locations in CPU and GPU, including partial input (PI#), CPU cache (C#) and GPU cache (G#). To collect all features, DiskGNN launches UVM accesses to load the features for PI# and C#, and directly fetches the features in GPU cache for G#.

5 Key Designs

In this part, we first introduce how DiskGNN configures the disk cache to trade off between I/O efficiency and storage overhead in §5.1. Then, we describe how to fill in the four-level feature store efficiently for pre-processing in §5.2, followed by the training pipeline to overlap computation with disk access in §5.3.

5.1 Segmented Disk Cache

As discussed in §4, feature packing eliminates read amplification but may consume large disk space (e.g., 10x of the original dataset). This is because a node feature can be required by multiple mini-batches (also called graph samples) and thus replicated in their feature chunks. As such, DiskGNN allows users to set a constraint C for the used disk space. To meet the space constraint, DiskGNN stores the disk-resident features in two parts, i.e., *disk cache* shared among mini-batches and *feature chunks* private for individual mini-batches. Space consumption is reduced because features in the disk cache are not replicated. Then, the problems are which features to store in the disk cache and how to organize them for efficient access. That is, we aim to solve the problem:

$$\begin{aligned} \min \quad & \text{I/O} = \textstyle\sum_{i=1}^{n} |P_i| + A_i \cdot |D_i| \\ s.t. \; & \text{Space} = |\mathrm{V_d}| + \textstyle\sum_{i=1}^{n} |P_i| < C \end{aligned} \tag{2}$$

where n is the number of mini-batches; P_i, D_i, and A_i are the contiguous feature chunks, required disk-cached features, and read amplification factor when reading the disk cache for the i-th mini-batch, respectively; $\mathrm{V_d}$ represents the node features in the disk cache. That is, we minimize the total I/O of the mini-batches under the disk space constraint C.

We cannot solve directly Eq. 2 due to the chicken-egg problem, i.e., A_i (i.e., the read amplification factor) should be known to solve $\mathrm{V_d}$ (i.e., nodes in the disk cache) but A_i can only be determined after obtaining $\mathrm{V_d}$ and its disk storage layout. To find the optimal solution of Eq. 2, we may enumerate all possible disk cache configurations, which is infeasible. As such, we opt for an approximate solution to trade for efficency. Eq. 2 reveals the key factors to consder, i.e., (i) reducing read amplitication of the disk cache and (ii) allocating disk space bettwen disk cache and packed node features. For (i), we fill the disk cache with the third most popular node features and determine its layout by node reordering. This is because popular nodes provide sharing chances among graph samples and node reordering reduces read amplification. For (ii), we develop a heuristic to search the space allocation.

Node reordering for disk cache. Suppose that we have decided which features to store in the disk cache, the problem becomes how to store the features to make the read amplification factors (i.e., A_i) small. This can be achieved by reordering the features, and Figure 4 provides an illustration. Specifically, in Figure 4a, the features are arranged according to their IDs, and a mini-batch that requires features of node $\{0, 2, 4, 6\}$ needs to read 4 disk pages due to read amplification; after reordering in Figure 4c, $\{0, 2\}$ and $\{4, 6\}$ are in the same disk pages, and thus the mini-batch only needs to read 2 disk pages. Note that feature reordering is essentially determining an order of the nodes, where computation only involves the IDs of the node features. Therefore, we use node IDs during descriptions of the reordering process afterward.

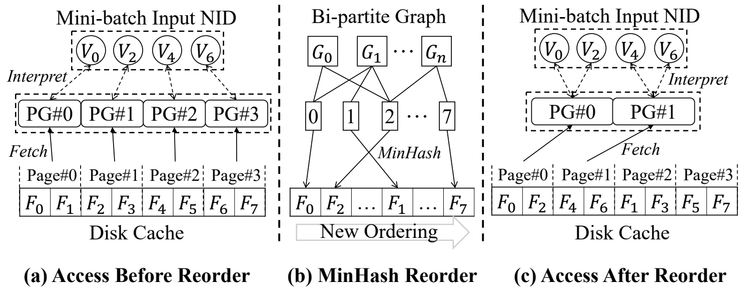


Fig. 4. Feature access pattern before and after reordering. Minhash is used to reorder features in disk cache.

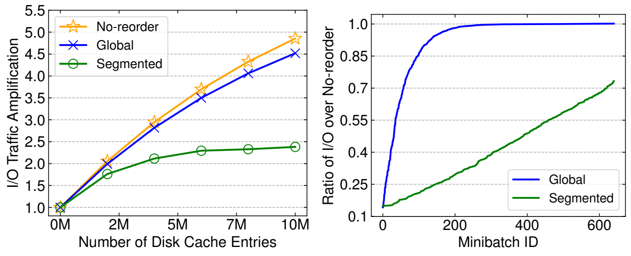
Algorithm 1 Disk Cache Reordering using MinHash

```
Input: $n$ graph samples $\{G_1, \cdots, G_n\}$, node entries in the disk cache $V_d = \{V_{d1}, \ldots, V_{dm}\}$, number of hash functions k
Output: Reordered cached entries $V_r$
 1: Initialize $H \leftarrow \emptyset$, $S \leftarrow$ size $|V_d|$ array filled with inf
 2: for $i \leftarrow 1, \cdots, k$ do                         ▷ Generate k hash functions
 3:     $H \leftarrow H \cup \{ \textsc{Permute}(1, \cdots, n) \}$
 4: for $i \leftarrow 1, \cdots, n$ do                         ▷ Iterate over n graph samples
 5:     $(V_i, E_i) \leftarrow G_i, V_{in} \leftarrow V_i \cap V_d$       ▷ Collect disk-cached nodes
 6:     for $v \in V_{in}$ do                               ▷ Generate hash signature for each node
 7:         for $H_j \in H$ do                              ▷ Iterate over k hash functions
 8:             $S(v) \leftarrow \textsc{Min}(S(v), H_j(i))$          ▷ Calculate MinHash value
 9: $V_r \leftarrow V_d[\textsc{Sort}(S)]$                       ▷ Reorder based on MinHash values
10: Return $V_r$
```

We observe that disk cache reordering resembles the well-known graph reordering problem [62]. In particular, given a graph, graph reordering re-numbers the nodes such that the neighbors of each node have adjacent IDs. One key purpose of graph reordering is to reduce cache miss for in-memory graph processing, i.e., the neighbors of each node spread over fewer cache lines after reordering. For disk cache reordering, we can construct a bipartite graph where each graph sample connects to its required node features and an illustration is provided in Figure 4b; and the goal is to reorder the node features such that the node features required by each graph sample spread over a small number of disk pages. Due to such similarity, we use a graph reordering algorithm for disk cache reordering. Specifically, we choose HashOrder [71] over more complex algorithms (e.g., Gorder [62]) because it is lightweight and shown to produce high-quality ordering.

Algorithm 1 shows how we use HashOrder for disk cache reordering. In particular, HashOrder models a disk-cached node as the set of graph samples that require it and assigns adjacent IDs to two nodes if their graph sample sets are similar. This is achieved using MinHash, where similar sets are more likely to have the same hash value. Lines 2-3 of Algorithm 1 generate k MinHash functions by permuting the IDs of the graph samples, and note that the MinHash value is the minimum over the outputs of the k hash functions (i.e., Line 8); Lines 4-8 compute MinHash values for the nodes in disk cache by going over all graph samples with $S(v)$ keeping the MinHash value of node v; Line 5 collects disk-cached nodes that are required by graph sample G_i; Line 9 sorts the node entries according to their MinHash values.

Segmented disk cache. We observe that reordering the disk cache *globally for all mini-batches* has a limited effect in reducing the read amplification. This is evidenced by Figure 5a, which compares



(a) I/O amplification for an epoch (b) I/O for mini-batches

Fig. 5. Effect of node reordering for global and segmented disk cache on Friendster. (a) I/O amplification for an epoch, and (b) the ratio of I/O over No-reorder for each mini-batch (ranked by the I/O ratio). A segment has 50 min-batches.

No-reorder with *global reordering*. Further examination in Figure 5b shows that only one-tenth of the mini-batches enjoy more than 50% I/O reduction w.r.t. *No-reorder*. These phenomena boil down to two reasons. First, different mini-batches require different node features, and thus it is more difficult to assign a good ordering for the node features when considering more mini-batches. For instance, a mini-batch may prefer to store features of node $\{0, 2\}$ in one disk page but another mini-batch prefers to collocate $\{0, 4\}$. Second, MinHash takes the minimum as the hash values and hence favors the initial mini-batches with small IDs.

The two reasons above motivate us to build a disk cache *locally for some mini-batches* (as opposed to all mini-batches), which we refer to as segmented disk cache. In particular, we divide the mini-batches into *segments* with each segment containing s mini-batches with consecutive IDs, the local disk cache of a segment only considers its constituting mini-batches, and different segments use different disk caches. Figure 5 shows that compared with the global disk cache, the segmented disk cache is much more effective in reducing I/O amplification.

Search for cache configuration. With the segmented disk cache, we need to decide two parameters, i.e., the number of mini-batches in a segment s and the local access frequency threshold m for a node feature to be kept in the disk cache. A small s reduces read amplification as each disk cache considers fewer mini-batches but increases space consumption since there are more segments and hence disk caches. Similarly, a small m reduces read amplification as more features are stored in the packed feature chunks, which do not have read amplification but increase space consumption. Therefore, with a disk space constraint C, we need to balance between s and m for high I/O efficiency.

The straightforward solution is to enumerate all combinations of s and m; and for each combination, we check if it satisfies the space constraint, construct the disk caches, and compute the total I/O traffic of the mini-batches. As we will show in §7, this brute-force search method can take more than an hour. To reduce the search time, we adopt a simple heuristic. In particular, we use $m = 1$ to favor the packed feature chunks in using disk space because they completely eliminate ready amplification. Then, we search for the minimum s that satisfies the space constraint. This is cheap because, for each s, we only need to count the number of features in the disk caches rather than actually reordering the disk cache. The experiments in §7 show that this heuristic takes only a few seconds and produces efficient cache configurations.

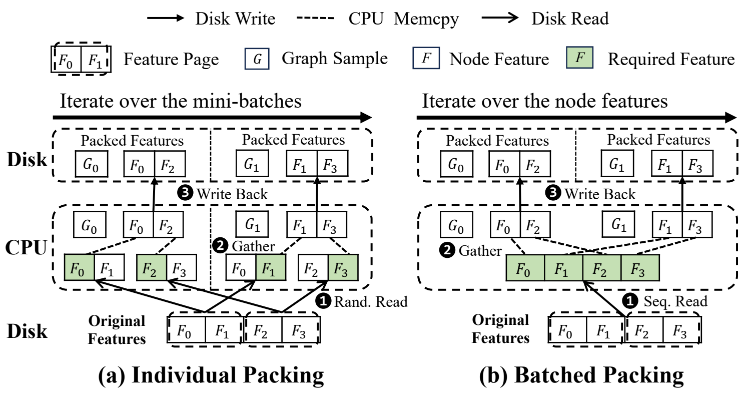


Fig. 6. An example of naive individual feature packing and DiskGNN's efficient batched feature packing.

5.2 Batched Feature Packing

After determining the cache configurations, i.e., where each node feature should be stored in the target data layout, DiskGNN reads the input node features from disk to materialize the target data layout. We first present a naive method for this purpose and then introduce our efficient solution to tackle the limitations of the naive method. In the subsequent discussions, we target at constructing the packed feature chunks for the mini-batches.

Individual packing. A straightforward method is to process the mini-batches sequentially; and for each mini-batch, it first conducts random disk reads to collect the required node features and then writes the node features back as contiguous disk pages. Figure 6a shows an example with two mini-batches. Mini-batch G_0 requires node features {0,2}; the two node features take up only one disk page but two disk pages are read from disk because {0,2} spread over two pages. The case is similar for mini-batch G_1. From the example, we observe that individual packing is inefficient for two reasons. First, each random read targets a single node feature, which is usually smaller than disk page, causing read amplification. Second, if a node feature is required by multiple mini-batches, it will be read from the disk multiple times, i.e., once for each mini-batch. Empirically, we observe that individual packing has a long running time.

Batched packing. We tackle the inefficiencies of individual packing by switching from a mini-batch-oriented view to a node feature-oriented view. We call this method batched packing because it processes all mini-batches concurrently. In particular, we split the node features into logical partitions with consecutive IDs, and the partition size is configured such that each partition fits in CPU memory. In each iteration, we load a partition of consecutive node features from disk, filter out the node features required by each mini-batch, and append these node features to pack feature chunk of the mini-batch. Once all mini-batches are processed, we proceed to the next iteration with the next partition of node features.

Specifically, we determine the size of each feature partition by excluding the working memory of the mini-batches from the total memory. With a total memory of C KB and N mini-batches, the partition size is $C - 4N$ so that each mini-batch can keep a 4KB buffer in memory to collect the features required by it. When there are too many mini-batches, $C - 4N < 0$ is possible. In this case, we can divide training into episodes, where each episode conducts pre-processing and training for some of the mini-batches. However, we did not encounter this case in our experiments.

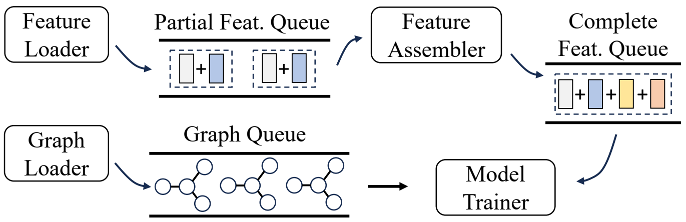


Fig. 7. DiskGNN's training pipeline.

Figure 6b provides an example for batched packing. Node features $\{0, 1, 2, 3\}$ are fetched from disk as one partition, and we write features {0,2} for mini-batch G_0 and features {1,3} for mini-batch G_1. In the example, batched packing reads 2 disk pages while individual packing reads 4 disk pages in Figure 6a. We observe that batched packing is more efficient than individual packing because (i) it involves only large sequential reads (for node feature partitions) and writes (for chunks of node feature) when accessing the disk, and thus is free from read amplification; and (ii) each node feature is read from disk only once. A possible problem with batched feature packing is that one node feature may be read from disk but no mini-batch needs it. However, we observe that almost all node features are required if many mini-batches are considered.

Besides preparing the packed feature chunks, batched packing is also used to fill in the GPU cache, CPU cache, and disk cache by treating them as special mini-batches. During pre-processing, we write both the GPU cache and CPU cache as files on disk to reserve CPU memory for packing. This allows for larger node partitions and accelerates packing. DiskGNN can conduct pre-processing and model training on different machines, since the target data layout is fully materialized on the disk of the pre-processing machine and the training machine can load the data via network.

5.3 Training Pipeline

A naive method to conduct model training is to process the mini-batches sequentially; for each mini-batch, the system first reads its disk-resident node features to CPU memory, then assembles all required features in GPU memory, and finally conducts model computation. This is inefficient because CPU and GPU processing needs to wait for slow disk I/O. Like other out-of-core systems [33, 52], we leverage a pipeline to overlap the computation and I/O of consecutive mini-batches. The key difference from existing systems is that more fine-grained pipeline stages are used to load and assemble node features from our four-level feature store.

Figure 7 depicts the training pipeline of DiskGNN. In particular, DiskGNN divides a mini-batch into four pipeline stages and handles each stage with a worker thread. The workers interact via shared queues following the producer-consumer pattern, and they run different stages of consecutive mini-batches in parallel. The four pipeline stages are ❶ *feature loading*, ❷ *feature assembling*, ❸ *graph loading*, and ❹ model training. Figure 7 shows that the four stages form two separate dependency paths, i.e., ❶→ ❷→❹, and ❸→❹. On the first path, *feature loader* fetches the disk-resident features from the disk cache and packs feature chunks to a consecutive CPU memory region. These features are put into the partial-feature-queue, which is consumed by the *feature assembler* when assembling the features from both CPU and GPU memory for a mini-batch. After assembling, the complete set of features required by a mini-batch are sent to the complete-feature-queue. On the second path, the *graph loader* fetches the computation graph of each mini-batch from disk to the graph-queue in CPU memory. As the last stage for both paths, the *model trainer* retrieves both the computation graph and the complete features of a mini-batch to conduct training computation for this mini-batch.

DiskGNN uses the same ordering to process mini-batches in both pipeline paths, and thus the heads of both the complete-feature-queue and graph-queue correspond to the same mini-batch. As GNN models and graph samples are typically small and can not saturate GPU computation, we run the *model trainer* and *feature assembler* on separate CUDA streams to improve GPU utilization.

With the training pipeline, the four workers can run concurrently to process different mini-batches. For example, when the *model trainer* is conducting computation on mini-batch b, the *graph loader* can fetch the graph sample for mini-batch $b + 1$, the *feature assembler* can also assemble the complete features for mini-batch $b + 1$ in parallel, and the *feature loader* can load the disk-resident features for mini-batch $b + 2$. This allows to overlap SSD reads, CPU to GPU data transfer, and GPU computation such that DiskGNN is bounded by the longest stage rather than the sum of all stages.

6 Implementation

DiskGNN is developed using the C++ library of PyTorch [40] as the backend. We utilize DGL [57], a popular open-source framework for graph learning, to store the graph samples on disk and perform model training. Key implementation details include the following:

- I/O operations: DiskGNN uses pread[29] for sequential disk access to fetch the packed feature chunks and can saturate SSD bandwidth with a single thread. For random disk accesses to fetch features in the disk cache, DiskGNN leverages io_uring [27] and uses 4 threads with each thread holding a ring to launch concurrent I/O requests, as this is observed to optimize read performance. OpenMP[36] is used to execute the 4 threads and subsequent memcpy operations in parallel. For all I/O operations, we use POSIX open [28] as the file descriptor with the O_Direct flag to bypass the OS page cache and directly access the SSD.
- Feature assembling: For feature assembling on GPU, DiskGNN uses Unified Virtual Addressing (UVA) [35] to fetch the node features resident on CPU memory. For the address lookup operations on both CPU and GPU (i.e., to obtain the locations of node features), we prepare interpreted address tables during pre-processing and load them from disk before model training. This saves the interpreting cost during training and avoids repetitive address generation for each graph sample across the epochs.
- Training pipeline: For the producer-consumer-based pipeline, the sizes of all shared queues are set to 2. This is observed to fully overlap the stages and consume a small amount of memory.

Easy-to-use API. DiskGNN allows users to construct an efficient data layout for model training with a single line of Python code.

```python
def DiskGNN_train(dataset_pth : PATH, disk_size : int, cpu_size : int, gpu_size : int, kwargs)
```

In particular, dataset_pth specifies the location of the original graph data and graph samples. DiskGNN first configure the features to store in the CPU cache and GPU cache according to cpu_size and gpu_size, respectively. Then, our heuristic is employed to identify the data layout for the segmented disk cache under the disk space constraint disk_size. Subsequently, batched packing is executed to construct the target data layout. After completing all data orchestration, training can start by integrating the I/O engine, feature assembler, and model trainer.

7 Evaluation

In this part, we conduct extensive experiments to evaluate DiskGNN and compare it with state-of-the-art disk-based GNN training systems. The main observations are that:

Table 2. Graph datasets used in the experiments.

Attributes	Friendster	Papers	MAG240M	IGB260M
Abbr.	FS	PS	MG	IG
Vertex count	66M	111M	244M	269M
Edge count	3.6B	3.3B	3.4B	3.9B
Avg. degree	56.1	30.1	14.2	14.8
Graph size (GB)	28.5	25.9	27.9	30.8
Feature size (GB)	31.3	52.9	117	129
Train nodes (%)	N/A	1.09	0.45	5.06

Table 3. Node access distribution, where nodes are ranked by their access frequencies in the graph samples.

Node Rank	**Access Ratio**			
	FS	**PS**	**MG**	**IG**
<1%	14.1%	43.2%	56.4%	22.5%
1%~5%	25.5%	36.5%	32.3%	30.0%
5%~10%	18.2%	11.2%	7.3%	20.8%
>10%	42.1%	9.1%	4.0%	26.7%

- *DiskGNN consistently yields shorter training time than the baselines while matching the best model accuracy of them.*
- *DiskGNN performs well across different configurations, e.g., for CPU cache size, disk space constraint, and model settings.*
- *The designs of DiskGNN, e.g., node reordering, batched packing, and training pipeline, are effective in improving efficiency.*

7.1 Experiment Settings

Datasets and models. We use the four graph datasets in Table 2 for experiments and refer to them by abbreviations subsequently. These datasets are publicly available and widely used to evaluate GNN models and systems. We follow the common practice to pre-process these datasets. In particular, FS and IG are undirected graphs, and we replace each undirected edges with two directed edges. For PS, we add a reverse edge for each directed edge to enlarge the receptive field of each node during neighbor aggregation. As FS only provides the graph topology (i.e., without node features and labels), we randomly generate a 128-dimension float vector for each node as the feature and select 1% of its nodes as the seed nodes for training by assigning fake labels. Table 3 reports the node access distribution of the graphs during training. The results show that the distributions are skewed for all graphs, with a small portion of nodes dominate the total accesses. However, the skewness is more severe for PS and MG than FS and IG.

We choose two representative GNN model architectures, i.e., GraphSAGE [17] and GAT [51], and adopt their popular hyper-parameter settings. In particular, GraphSAGE uses the mean aggregation function while GAT uses multi-head attention for neighbor aggregation. The hidden embedding dimension of GraphSAGE is 256 while a hidden embedding dimension of 32 and 4 attention heads are used for GAT. Following the open-source implementations of DGL [9], both GraphSAGE and

Table 4. SSD configurations and their comparisons to DRAM.

Storage Type	RD IOPS (4KB)	Bandwidth	Price ($/GB)
DRAM (DDR4)	>10M	25 GB/s	11.13
AWS NVMe	625k	3 GB/s	0.125
AWS gp3	16k	2.5 GB/s	0.08

GAT are set to have 3 layers. For graph sampling, we use node-wise neighbor sampling with a fanout of [10,15,20] for both models, and the mini-batch size is set as 1024.

Baseline systems. We compare DiskGNN with two state-of-the-art disk-based GNN training systems, i.e., Ginex [39] and MariusGNN [52]. As introduced in §1, Ginex reads each disk-resident node feature individually and manages the feature swapping between disk and CPU memory using the Belady's algorithm; MariusGNN organizes the graph into edge chunks and node partitions and samples only the memory-resident node features for training. As such, Ginex and MariusGNN represent two distinct paradigms, i.e., fine-grained and partition-based feature access. We do not compare with Helios [47] and GIDS [38] because Helios is not open-source, and GIDS uses GPU-initiated disk I/O, which is only supported by the latest GPUs. They both adopt the same fine-grained feature access pattern as in Ginex and thus also suffer from read amplification. As a naive baseline, we also adapted DGL to disk-based training (called DGL-OnDisk) by using pread to read node features from the disk.

Platform and metrics. We mainly experiment on a AWS g5.48xlarge instance [3] with a 96-core AMD EPYC 7R32 CPU, 748GB RAM, 2 × 3.8TB AWS NVMe SSD, and an NVIDIA A10G GPU with 24GB memory. To account for the impact of hardware on performance, we also use an AWS gp3 SSD, which is relatively cheaper but has much lower IOPS than the NVMe SSD. The statistics of both SSD types are listed in Table 4. *To simulate the case of large graphs that exceed CPU memory, we set CPU memory constraints for all systems as different proportions of the graph features, with* 10% *by default.* The GPU is connected to the host CPU via PCIe 3.0 with a full bandwidth of 7GBps. The operating system is Ubuntu 20.04, and the software is CUDA 11.7 [34], Python 3.9.18 [42], PyTorch 2.0.1 [43], DGL 1.1.2 [9], and PyG 2.5.0 [41].

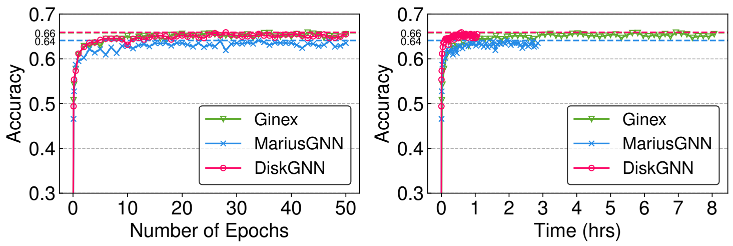
We compare the systems in terms of both model accuracy and training efficiency. For model accuracy, we report the test accuracy at the epoch when the highest validation accuracy is achieved for each system. For PS and MG, we run 50 training epochs to reach convergence. For IG, only 20 epochs are needed as it has more seed nodes. We do not use FS in the accuracy evaluation because it does not provide node features and labels. For training efficiency, we run each system for 5 epochs and record the average time of the latter 4 epochs, leaving the first epoch for warning up. Since Ginex, MariusGNN, and DiskGNNall conduct pre-processing, we also amortize their pre-processing time over all training epochs for an end-to-end comparison. *To make the comparison fair, we ensure that all systems use the same amount of CPU memory.*

7.2 Main Results

Model accuracy. Table 5 reports the final model accuracy and end-to-end running time of the systems. The results show that DiskGNN matches the accuracy of Ginex, while accelerating Ginex and MariusGNN by 2-10x in end-to-end time. The small differences in accuracy of DiskGNN and Ginex is caused by random factors such as parameter initialization and graph sampling. The model accuracy of MariusGNN is noticeably lower than Ginex and DiskGNN. For instance, on the MG

Table 5. Final model accuracy (%) and end-to-end running time (hrs) comparison for the systems.

Systems		Datasets		
		Papers100M	MAG240M	IGB-HOM
SAGE	Ginex	65.85/9.72	**67.90**/8.94	58.96/151
	DiskGNN	**65.91/1.09**	67.79/**0.77**	**59.03/13.7**
	MariusGNN	64.01/3.66	65.84/3.89	58.77/25.5
GAT	Ginex	**65.03**/9.33	66.48/9.20	56.43/143
	DiskGNN	**65.03/1.29**	**66.53/0.99**	**56.69/14.7**
	MariusGNN	OOM/7.37	OOM/4.27	OOM/28.8



(a) Accuracy v.s. epoch (b) Accuracy v.s. training time

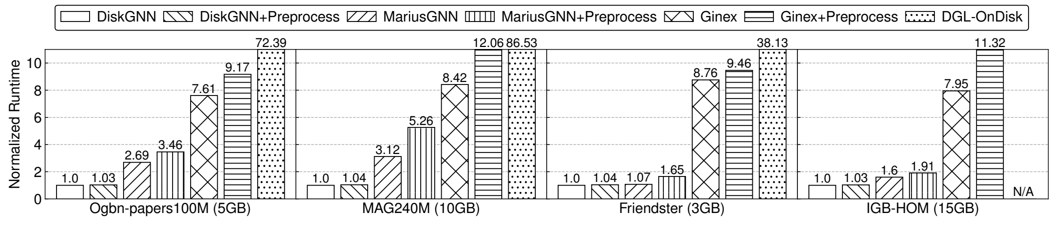
Fig. 8. Accuracy curve for GraphSAGE on PS.

graph and GraphSAGE model, the accuracy degradation of MariusGNN is 2.1%, which is large for GNN models and may be unacceptable for applications such as recommendation. For GAT, MariusGNN runs out-of-memory (OOM) when evaluating the model accuracy for all datasets.

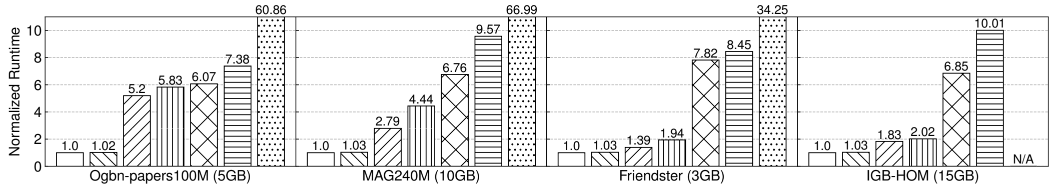
Figure 8 tracks the model accuracy for training GraphSAGE on the PS graph. Figure 8a plots accuracy against epoch count and shows that the accuracy of MariusGNN saturates after some epochs, and training more epochs does not improve its accuracy. Ginex and DiskGNN achieve almost identical accuracy when training for the same number of epochs, suggesting that we can use the epoch time to measure training efficiency. Figure 8b plots accuracy against training time and validates the speedups of DiskGNN over the baselines in end-to-end time reported by Table 5.

Training efficiency. Figure 9 reports the epoch time of the systems. In each case (i.e., model plus dataset), we normalize the results by the epoch time of DiskGNN because the absolute epoch time spans a large range for the datasets, which will make the figure difficult to read. To account for the overhead of pre-processing, we also report DiskGNN + Preprocess, Marius + Preprocess, and Ginex + Preprocess, which amortize the pre-processing time of the systems over the training epochs. For a fair comparison, we revise Ginex to receive the graph samples in advance like DiskGNN. During pre-processing, Ginex computes the cache eviction schedule using Belady's algorithm while Marius organizes the graph into edge chunks and node partitions. DGL-OnDisk can not finish an epoch in 10 hours on IG, and thus we report N/A for it.

Figure 9 shows that DiskGNN consistently outperforms all baseline systems across the four datasets and two GNN models. In particular, the speedup of DiskGNN over Ginex is over 6x in all cases and can be 8.76x at the maximum. This is because Ginex suffers from severe read amplification by reading each node feature individually from disk while DiskGNN enjoys efficient disk access due to its data layout optimizations. Compared with MariusGNN, DiskGNN has about 2x speedup



(a) GraphSAGE



(b) GAT

Fig. 9. Normalized epoch time for training the two GNN models, the epoch time of DiskGNN is set as 1.0.

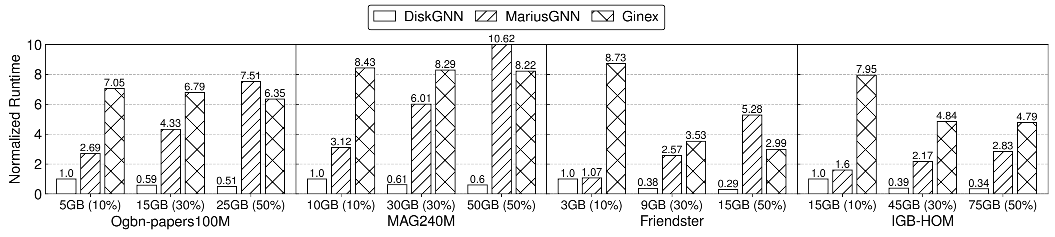


Fig. 10. Normalized epoch time with different memory constraints for training the GraphSAGE model.

in 6 out of the 8 cases, while providing superior model accuracy. The speedup of DiskGNN over MariusGNN is smaller on FS because its popular nodes are less dominant in access (c.f. Table 3), which makes the CPU and GPU cache less effective. Regarding DGL-OnDisk, the speedup of DiskGNN is significant (i.e., 86.53x at the maximum) because DGL-OnDisk needs to read every node feature from disk (without CPU and GPU cache). Considering the pre-processing time, the difference between DiskGNN + Preprocess and DiskGNN is within 5% for all cases, indicating that the pre-processing of DiskGNN is lightweight w.r.t. training. In comparison, MariusGNN and Ginex can have over 40% amortized overhead due to pre-processing.

Memory constraint. Figure 10 reports the epoch time of DiskGNN, MariusGNN, and Ginex for GraphSAGE when changing the CPU memory limit. We set the memory as percentages (i.e., roughly 10%, 30% and 50%) over all node features and also mark the absolute memory size. The results show that DiskGNN trains consistently faster than MariusGNN and Ginex with different memory constraints. DiskGNN observes a diminishing return in epoch time when enlarging the CPU memory, e.g., increasing from 30% to 50% feature size brings a smaller speedup than from 10% to 30% feature size. This is because most accesses to node features are served by the GPU and CPU caches with a reasonable memory size, and thus further increasing the memory is not very effective in reducing disk access. FS and IG observe larger reductions in epoch time than other datasets when increasing CPU memory from 10% to 30%. This is because their node access distributions are

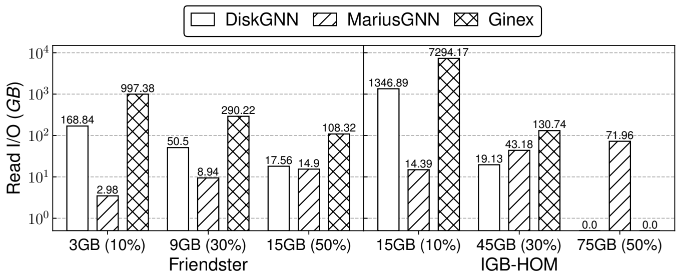


Fig. 11. Disk traffic adjusting the memory constraints.

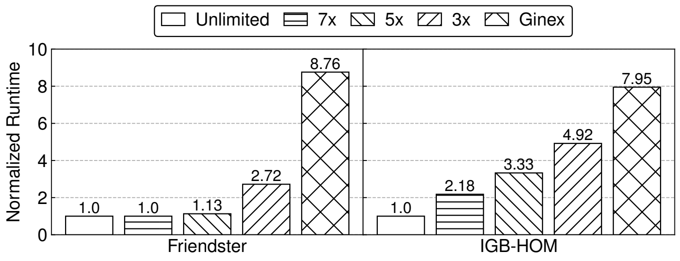


Fig. 12. Normalized epoch time with different disk space constraints for training the GraphSAGE model.

less skewed (i.e., 42.1% and 26.7% for the nodes behind top-10% in Table 3), and thus the increased CPU memory avoids more disk accesses. MariusGNN has longer epoch time at larger memory size because it loads more edge chunks and node partitions from disk to fill in the CPU memory. Moreover, graph sampling for the memory resident partitions will involve more neighbors, leading to longer sampling and model training time as the graph samples involve more nodes.

Disk traffic. To understand the performance of the systems, Figure 11 report the average amount of data they read from disk in an epoch. We include only FS and IG due to the page limit (similarly for some other experiments). The results show that the disk traffic of DiskGNN is less than 1/5 of Ginex in all cases, which explains the speedup of DiskGNN over Ginex. When the memory size is 50% of the node features, DiskGNN and Ginex almost have no disk traffic to read the node features on the IG graph, justifying the diminishing return of increasing CPU memory. Additionally, the disk traffic on IG drops more quickly than FS because the node access distribution is more skewed for IG, and thus more feature accesses are served by GPU and CPU caches. As we have explained, MarisGNN's disk traffic increases with CPU memory because it loads more node partitions and edge chunks. MariusGNN is slower than DiskGNN despite its lower disk traffic because it has other overheads (e.g., inducing a graph between the memory resident edge chunks and graph sampling).

Disk space constraint. Figure 12 reports the epoch time of DiskGNN with different constraints on the disk space usage, which is specified as a multiplier (i.e., 7x, 5x and 3x) of feature size. We include Ginex as a reference, and the results of PS and MG are omitted because they use less disk space than the feature size even if we only use packing to eliminate read amplification. This is because PS and MG have more skewed node access distributions, with the top-1% nodes taking up 43.2% and 56.4% of all access while the proportions are 14.1% and 22.5% for FS and IG. Specifically, when access is more biased towards the hot nodes, memory cache is more effective, and less disk

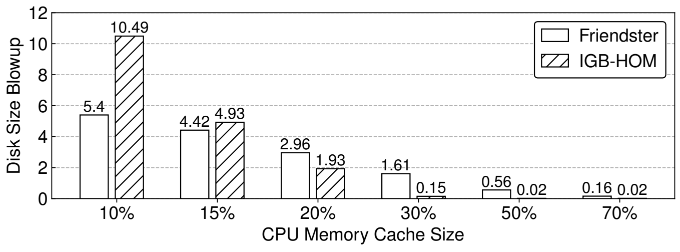


Fig. 13. Disk space usage versus CPU memory cache sizes.

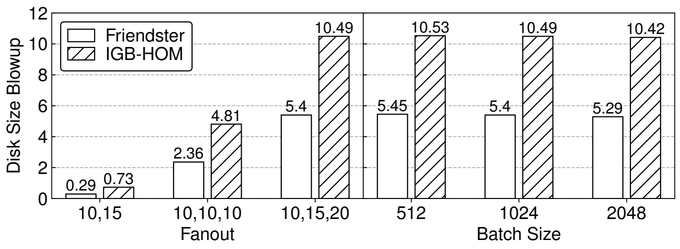


Fig. 14. Disk space usage versus fanouts and batch sizes.

space is required to pack the cold node features. The graph sparsity does not have a direct influence on DiskGNN's disk space usage and this is evidenced by that MG and IG have similar average node degrees in Table 2 (i.e., 14.2 and 14.8) but differs by a large margin in disk space usage. When disk space is unlimited, FS uses 5.39x of the feature size while IG uses 10.19x of the feature size. This explains why FS observes a very small change in epoch time when switching from unlimited to 5x feature size. The results show that DiskGNN has a longer training time when using smaller disk space. This is because DiskGNN stores more node features in the disk cache instead of packed feature chunks, and the disk cache relies on node reordering to reduce read amplification while packed feature chunks eliminate read amplification.

Effect of CPU memory on disk consumption. The maximum disk space DiskGNN will occupy is affected by the memory size. This is because when the GPU and CPU caches hold more node features, fewer node features need to be stored in the packed feature chunks on disk. Figure 13 reports how the maximum disk space usage changes when adjusting the memory cache size, and both disk space and cache memory are relative to the size of the original node features. The results show that the disk space usage reduces quickly when increasing the memory cache size. Specifically, when the memory cache size is over 20% of the node features, the disk space consumption is below 2 times of the node features. The reduction of the disk space usage is more significant on the IG graph because its node accesses are more skewed towards the popular nodes.

Effect of sampling parameters on disk consumption. Figure 14 reports the disk space utilized by DiskGNN when adjusting the fanout and batch size for graph sampling. We keep the batch size as 1024 when changing the fanout while keep the fanout as [10,15,20] when varying the batch size. The results show that enlarging the fanout from [10,15] to [10,15,20] significantly increases disk space. This is because with larger fanouts, each graph sample contains more nodes. When increasing the batch size, disk space remains stable because the seed nodes of a batch take up only a

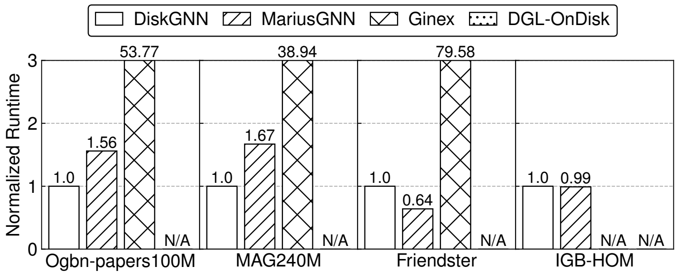


Fig. 15. Normalized epoch time on AWS gp3 SSD.

Table 6. Model accuracy (%) and total time (hrs) comparison on GCN and ShadowGNN.

Systems		Ogbn-papers100M		MAG240M	
		Acc.	Total Time	Acc.	Total Time
GCN	Ginex	64.08	9.78	66.93	9.27
	DiskGNN	64.27	1.33	66.93	0.57
	MariusGNN	63.26	3.62	65.20	3.75
Shadow	Ginex	65.67	15.20	67.79	13.96
	DiskGNN	65.57	7.33	67.72	5.30

small portion of all graph nodes, and thus different seed nodes share a limited number of commonly sampled neighbors. As such, the de-duplication effect is not obvious for large batch sizes.

Another SSD. To investigate the influence of hardware, Figure 15 reports the epoch time of the systems when using the AWS gp3 SSD. Compared with the default NVMe SSD, gp3 SSD has much lower IOPS but the sequential read bandwidth is comparable (c.f. Table 4). DGL-OnDisk can not finish an epoch in 10 hours on all 4 graphs, and so does Ginex for IG. We observe that the epoch time of DiskGNN increases to about 1.2x over the NVMe SSD, which corresponds to the bandwidth degradation of gp3 SSD w.r.t. NVMe SSD. Compared with Ginex, DiskGNN achieves larger speedups when using the gp3 SSD. This is because Ginex heavily relies on small random read to fetch disk resident node features, and thus its performance is bounded by the poor IOPS of AWS gp3 SSD.

Generality of DiskGNN. We experiment with GCN [25] for another GNN model and ShadowGNN [67] for another graph sampling algorithm. Table 6 reports the final model accuracy and end-to-end running time. We use a fanout of [10,10,10] for graph sampling to prevent GPU OOM for ShadowGNN as it induces much larger graph samples than neighbor sampling. Note that MariusGNN does not support ShadowGNN because it customizes data structures to avoid redundant computation for neighbor sampling. The results show that DiskGNN supports both GCN and ShadowGNN and achieves similar accuracy with Ginex. Moreover, DiskGNN yields shorter end-to-end running time than the baselines. The speedup of DiskGNN over Ginex is smaller for ShadowGNN than for GCN because ShadowGNN produces dense graph samples by retrieving all edges between the sampled graph nodes, which makes disk access cost less dominant due to heavier graph loading and model computation.

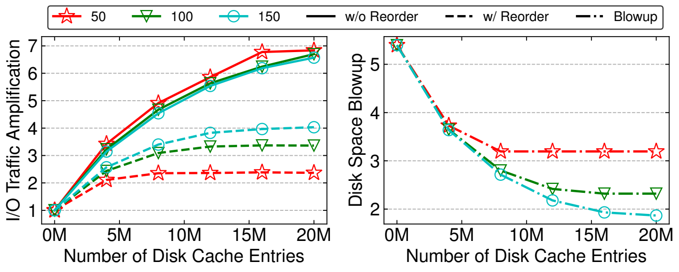


Fig. 16. Effectiveness of segmented disk cache on the FS graph. Left is the disk traffic amplification for feature reading, right is the disk space usage w.r.t. the original features.

Table 7. Efficiency and quality of the heuristic method to search for disk cache configuration compared with the brute-force search. *Blowup* is the disk space to use, *I/O Amp.* is disk traffic amplification, and *Time* is the search time.

Blowup		**Methods**				**Speedup**
		Brute Force		Heuristic		
		I/O Amp.	Time (s)	I/O Amp.	Time (s)	
FS	3x	2.98x	257.0	2.58x	0.99	259.60x
	5x	1.33x	75.00	1.09x	0.18	416.67x
IG	3x	5.10x	6288	4.85x	22.8	275.83x
	5x	4.11x	4755	3.39x	9.76	487.27x
	7x	3.01x	3261	2.28x	4.61	707.52x

7.3 Microbenchmarks

Segmented disk cache with reordering. In Figure 16, we evaluate the effectiveness of the segmented disk cache and node reordering on the FS graph. We use three configurations (i.e., 50, 100, and 150) for the number of min-batches in a segment (i.e., segment size). The left plot suggests that our MinHash-based node reordering is effective in reducing the disk traffic. Moreover, the disk traffic reduction of reordering is larger with a smaller segment size (e.g., 50 versus 150). This is because the reordering suits the min-batches better when considering a small number of mini-batches. However, the right plot shows that using a smaller segment size yields a larger disk space consumption. This is because give a fixed number of mini-batches, a smaller segment size results in more disk caches.

Heuristic search method. Table 7 compares the search time and result quality of our heuristic method to search for disk cache configuration in § 5.1 with the brute-force method. We measure result quality by disk traffic amplification, and lower amplification means higher quality. Brute-force search may have lower result quality than our method because it needs a step size when iterating over the segment size, as checking all possible segment sizes is too costly. This prevents brute-force search from finding the optimal configuration. The results show that our heuristic method reduces the search time of brute-force by over 250x while yielding comparable or even better result quality.

Batched packing. Figure 17 reports the speedup of batched packing over individual packing across the 4 datasets. For both solutions, we mark the feature loading time with shadow. With individual

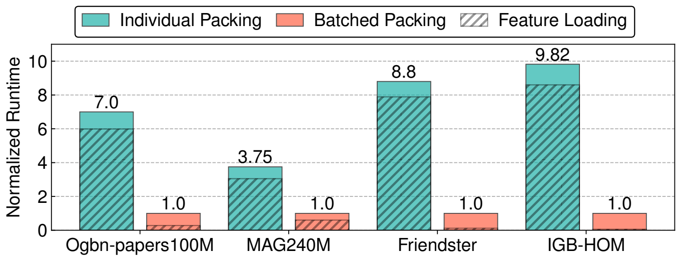


Fig. 17. Normalized running time of naive individual packing and our optimization batched packing for pre-processing.

Table 8. Pipelined training vs. sequential execution.

Mem. cache size		Methods		Speedup
		Sequential (s)	Pipeline (s)	
FS	3GB (10%)	165.3	96.67	1.71x
	9GB (30%)	88.98	36.51	2.44x
	15GB (50%)	54.24	28.12	1.93x
IG	15GB (10%)	1901.82	960.73	1.98x
	45GB (30%)	652.35	312.71	2.09x
	75GB (50%)	641.26	324.61	1.98x

packing, loading time is consistently the bottleneck, constituting between 81% and 90% of the pre-processing time. Batched packing addresses this bottleneck by replacing repetitive random reads of on-disk features with sequential reads without duplication. The speedups of batched feature packing over individual packing are generally larger on FS and IG since the two datasets need to load more disk resident features with a more uniform node access distribution and hence lower CPU cache hit ratio.

Training pipeline. Table 8 validates our producer-consumer-based training pipeline. Sequential execution refers to the implementation that processes the mini-batches sequentially, while pipelined execution overlaps the computation and I/O of different mini-batches. The results show that our training pipeline achieves a maximum speedup of 2.44x and an average speedup of over 2x.

Feature store. Figure 18 evaluates the gain of the four-level feature store for GraphSAGE on FS and IG. We incrementally add the GPU cache, CPU cache, and feature packing to a naive implementation that loads the required node features via random disk access (i.e., Raw). We do not include the disk cache because its gain is validated by Figure 12. The results show that both the GPU cache and CPU cache reduce epoch time, and this is because they reduce disk access. Feature packing significantly reduces epoch time because it eliminates read amplification by packing the node features required by each mini-batch as a contiguous disk block for sequential read.

Training time breakdown. Figure 19 decomposes the training epoch time of the systems into five stages. We omit DGL-onDisk because its epoch time is much longer than the other systems. To eliminate the interference among stages, we disable pipelining and configure the systems to execute the stages sequentially, and thus the total time of the stages is longer than the epoch time reported previously. MariusGNN has a *graph sample* stage while the other systems receive graph

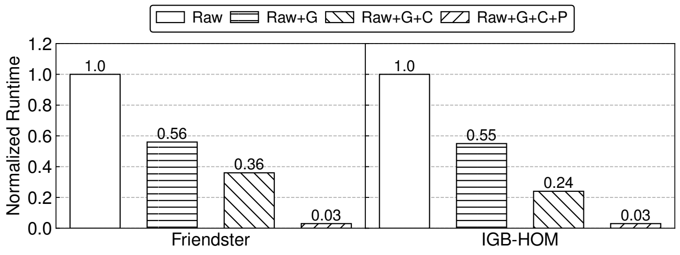


Fig. 18. Normalized epoch time of DiskGNN when incrementally adding the caches. **Raw** is a naive implementation that reads each node feature individually from disk. **G** is for GPU cache, **C** is to CPU cache, and **P** is feature packing.

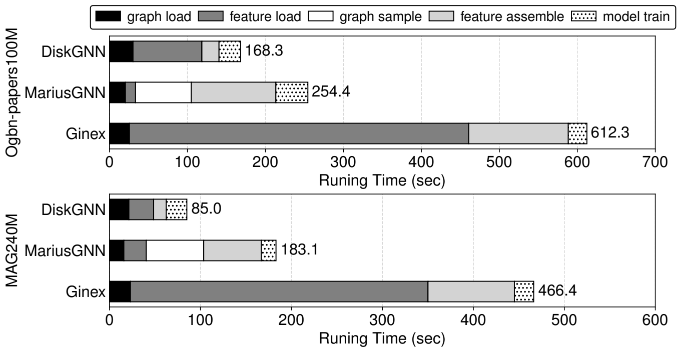


Fig. 19. Decomposing training epoch time for the systems using sequential execution of the stages.

samples from external systems. The results show that DiskGNN has shorter disk access time (i.e., *graph load* and *feature load*) than Ginex due to our optimizations. MariusGNN is fast in disk access (i.e., *feature load*) due to its block-based training scheme but spends a long time for *feature assemble* in memory. For all systems, *model train* takes a short time compared to the other stages. This is because computations are lightweight for GNN models and suggests that using multiple GPUs will not be cost-effective since computation is not the bottleneck.

8 Related Work

Graph learning frameworks. DGL [57] and PyG [12] are two popular graph learning frameworks that provide comprehensive user interfaces to express various GNN algorithms [17, 22, 25, 51] and efficient CPU and GPU operators for graph sampling [7, 8, 20, 31, 66, 70, 73] and model training. We enjoy their developments by building DiskGNN upon DGL. Many systems optimize GPU kernel optimizations for GNN training, e.g., GNNadvisor [59], Graphiler [64], TC-GNN [60], QGTC [58] and gSampler [16]. GNNadvisor and Graphiler explore CUDA kernel management and kernel fusion opportunities in GNN computation to improve GPU utilization. TC-GNN and QGTC translate the sparse GNN workload to dense operators and facilitate GPU Tensor Core Units to speed up the computation. These systems are orthogonal to DiskGNN because they assume that the graph topology and node features fit in CPU memory, while DiskGNN tackles efficient data loading from disk to CPU when the graph is large.

Large-scale GNN training systems. To handle large-scale graphs that can not fit in CPU memory, many systems utilize multiple machines to train the GNN model, with each machine holding a partition of graph data [10, 23]. NeuGraph [32], ROC [21], PipeGCN [54], BNS-GCN [53] and DGCL [5] represent early distributed GNN systems that adopt full-graph training. They compute output embeddings for all graph nodes in each iteration and suffer from high GPU memory consumption to store the intermediate embeddings. Recent distributed systems, such as DistDGL [72], Quiver [48], P3 [13], DSP [6], BGL [30], GNNLab [65] and Legion [46], adopt mini-batch training to process some seed nodes in each iteration and use graph sampling to control the number of neighbors for aggregation. However, they still suffer from the heavy communication costs between machines to exchange node features and intermediate embeddings. Besides, these distributed GNN training systems are expensive because they require a cluster.

Out-of-core processing systems. DiskGNN follows the general design principles of out-of-core processing systems, which include keeping hot data in memory, avoiding random disk access, and overlapping I/O with computation. For instance, LSM [37] avoids random disk writes by sequentially appending log entries and caches recently accessed data in memory. FlashNeuron [4] offloads intermediate model activations to disk in a compressed format and uses prefetching to overlap data movement with computation. FlexGen [45] distributes large model weights to the storage hierarchies (e.g., GPU, CPU, SSD) based on popularity and searches for efficient data layout by solving a linear programming problem. AttentionStore [14] places the KV caches of LLMs in the memory hierarchies based on user access patterns. DiskGNN observes that the node feature accesses of GNN training are specified by the graph samples, which makes it possible to collect access information and apply these design principles. Moreover, we also tailor the system designs for GNN training (e.g., two-step feature assembly, node reordering for disk cache, batched feature packing, and training pipeline).

9 Conclusion

We present DiskGNN, an efficient framework designed to support large-scale GNN training when data is stored on disk. DiskGNN adopts offline sampling as the core design to achieve both I/O efficiency and model accuracy, and the key idea is to conduct graph sampling to collect data access information such that the data layout can be optimized for efficient access. DiskGNN also incorporates a suite of designs, which includes reordering the node features to reduce I/O traffic with better data locality, batched packing to speed up pre-processing by transforming random disk reads into sequential disk reads, and pipelined training to overlap disk access with other operations. Extensive experiments demonstrate that DiskGNN significantly outperforms existing disk-based GNN training systems, showing a speedup of up to 8× over them.

Acknowledgments

Dr. Bo Tang and Renjie Liu were supported by National Science Foundation of China (NSFC No. 62422206) and a research gift from AlayaDB Inc. We thank the anonymous reviewers for their valuable comments and insightful suggestions, AWS Shanghai AI Lab for providing the computing resources and funding, and Dr. Lingfan Yu for helpful discussions in early stages of this project.

References

[1] 2024. AWS. https://aws.amazon.com/. [Online; accessed Apirl-2024].
[2] 2024. Azure. https://azure.microsoft.com. [Online; accessed Apirl-2024].

[3] AWS. 2024. Amazon EC2 G5 instance. https://aws.amazon.com/cn/ec2/instance-types/g5. [Online; accessed April-2024].
[4] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. 2021. FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks. In *FAST*. 387–401.
[5] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *Eurosys*. 130–144.
[6] Zhenkun Cai, Qihui Zhou, Xiao Yan, Da Zheng, Xiang Song, Chenguang Zheng, James Cheng, and George Karypis. 2023. DSP: Efficient GNN training with multiple GPUs. In *PPoPP*. 392–404.
[7] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *ICLR*.
[8] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *ICML*. 941–949.
[9] DGL. 2024. Deep Graph library. https://www.dgl.ai. [Online; accessed Apirl-2024].
[10] Zezhong Ding, Yongan Xiang, Shangyou Wang, Xike Xie, and S. Kevin Zhou. 2024. Play like a Vertex: A Stackelberg Game Approach for Streaming Graph Partitioning.
[11] Fuli Feng, Xiangnan He, Xiang Wang, Cheng Luo, Yiqun Liu, and Tat-Seng Chua. 2019. Temporal Relational Ranking for Stock Prediction. *ACM Trans. Inf. Syst.* (2019), 27:1–27:30.
[12] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR 2019 Workshop on Representation Learning on Graphs and Manifolds*.
[13] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *OSDI*. 551–568.
[14] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *ATC*. 111–126.
[15] Laura Garton, Caroline Haythornthwaite, and Barry Wellman. 1997. Studying Online Social Networks. *J. Comput. Mediat. Commun.* (1997).
[16] Ping Gong, Renjie Liu, Zunyao Mao, Zhenkun Cai, Xiao Yan, Cheng Li, Minjie Wang, and Zhuozhao Li. 2023. GSampler: General and Efficient GPU-Based Graph Sampling for Graph Learning. In *SOSP*. 562–578.
[17] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NeurIPS*. 1024–1034.
[18] Kehang Han, Balaji Lakshminarayanan, and Jeremiah Zhe Liu. 2021. Reliable Graph Neural Networks for Drug Discovery Under Distributional Shift. In *NeurIPS 2021 Workshop on Distribution Shifts: Connecting Methods and Applications*.
[19] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: datasets for machine learning on graphs. In *NeurIPS*.
[20] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling towards Fast Graph Representation Learning. In *NeurIPS*. 4563–4572.
[21] 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 *MLSys*. 187–198.
[22] Haitian Jiang, Renjie Liu, Xiao Yan, Zhenkun Cai, Minjie Wang, and David Wipf. 2023. MuseGNN: Interpretable and Convergent Graph Neural Network Layers at Scale. *arXiv preprint arXiv:2310.12457* (2023).
[23] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* (1998), 359–392.
[24] Arpandeep Khatua, Vikram Sharma Mailthody, Bhagyashree Taleka, Tengfei Ma, Xiang Song, and Wen-mei Hwu. 2023. IGB: Addressing The Gaps In Labeling, Features, Heterogeneity, and Size of Public Graph Datasets for Deep Learning Research. In *KDD*.
[25] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.
[26] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[27] Linux. 2024. io_uring. https://man7.org/linux/man-pages/man7/io_uring.7.html. accessed, April-2024.
[28] Linux. 2024. POSIX open. https://man7.org/linux/man-pages/man2/open.2.html. accessed, April-2024.
[29] Linux. 2024. POSIX pread. https://man7.org/linux/man-pages/man2/pwrite.2.html. accessed, April-2024.
[30] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2023. BGL:GPU-Efficient GNN training by optimizing graph data I/O and preprocessing. In *NSDI*. 103–118.
[31] Ziqi Liu, Zhengwei Wu, Zhiqiang Zhang, Jun Zhou, Shuang Yang, Le Song, and Yuan Qi. 2020. Bandit Samplers for Training Graph Neural Networks. In *NeurIPS*.

[32] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel deep neural network computation on large graphs. In *ATC*. 443–458.

[33] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Marius: Learning Massive Graph Embeddings on a Single Machine. In *OSDI*.

[34] NVIDIA. 2024. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit. [Online; accessed Apirl-2024].

[35] NVIDIA Corporation. 2024. Unified Addressing. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__UNIFIED.html. accessed, April-2024.

[36] OpenMP. 2024. OpenMP. https://www.openmp.org. accessed, April-2024.

[37] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* (1996), 351–385.

[38] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-mei Hwu. 2024. Accelerating Sampling and Aggregation Operations in GNN Frameworks with GPU Initiated Direct Storage Accesses. In *VLDB*. 1227–1240.

[39] Yeonhong Park, Sunhong Min, and Jae W. Lee. 2022. Ginex: SSD-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching. In *VLDB*. 2626–2639.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*.

[41] PyG. 2024. PyTorch Geometric. https://www.pyg.org. [Online; accessed Apirl-2024].

[42] Python. 2024. Python. https://www.python.org/downloads/release/python-398/. [Online; accessed Apirl-2024].

[43] PyTorch. 2024. PyTroch. https://pytorch.org. [Online; accessed Apirl-2024].

[44] Sungmin Rhee, Seokjun Seo, and Sun Kim. 2018. Hybrid Approach of Relation Network and Localized Graph Convolutional Filtering for Breast Cancer Subtype Classification. In *IJCAI*. 3527–3534.

[45] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *ICML*. 31094–31116.

[46] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, et al. 2023. Legion: Automatically Pushing the Envelope of Multi-GPU System for Billion-Scale GNN Training. In *ATC*. 165–179.

[47] Jie Sun, Mo Sun, Zheng Zhang, Jun Xie, Zuocheng Shi, Zihan Yang, Jie Zhang, Fei Wu, and Zeke Wang. 2023. Helios: An Efficient Out-of-core GNN Training System on Terabyte-scale Graphs with In-memory Performance. *arXiv preprint arXiv:2310.00837* (2023).

[48] Zeyuan Tan, Xiulong Yuan, Congjie He, Man-Kit Sit, Guo Li, Xiaoze Liu, Baole Ai, Kai Zeng, Peter Pietzuch, and Luo Mai. 2023. Quiver: Supporting GPUs for Low-Latency, High-Throughput GNN Serving with Workload Awareness. arXiv:2305.10863

[49] Komal Teru, Etienne Denis, and Will Hamilton. 2020. Inductive relation prediction by subgraph reasoning. In *ICML*. 9448–9457.

[50] Anton Tsitsulin, John Palowitch, Bryan Perozzi, and Emmanuel Müller. 2024. Graph clustering with graph neural networks. *The Journal of Machine Learning Research* (2024).

[51] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *ICLR*.

[52] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2023. Mariusgnn: Resource-efficient out-of-core training of graph neural networks. In *Eurosys*. 144–161.

[53] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. 2022. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. In *MLSys*. 673–693.

[54] Cheng Wan, Youjie Li, Cameron R Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. 2022. PipeGCN: Efficient full-graph training of graph convolutional networks with pipelined feature communication. *arXiv preprint arXiv:2203.10428* (2022).

[55] Daixin Wang, Yuan Qi, Jianbin Lin, Peng Cui, Quanhui Jia, Zhen Wang, Yanming Fang, Quan Yu, Jun Zhou, and Shuang Yang. 2019. A Semi-Supervised Graph Attentive Network for Financial Fraud Detection. In *ICDM*. 598–607.

[56] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-Scale Commodity Embedding for E-Commerce Recommendation in Alibaba. In *KDD*. 839–848.

[57] 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. In *Proceedings of the ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[58] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: accelerating quantized graph neural networks via GPU tensor core. In *PPoPP*. 107–119.

[59] 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 *OSDI*. 515–531.
[60] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. 2023. TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs. In *ATC*. 149–164.
[61] Mark Weber, Giacomo Domeniconi, Jie Chen, Daniel Karl I. Weidele, Claudio Bellei, Tom Robinson, and Charles E. Leiserson. 2019. Anti-Money Laundering in Bitcoin: Experimenting with Graph Convolutional Networks for Financial Forensics. *CoRR* (2019).
[62] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *SIGMOD*. 1813–1828.
[63] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2022. Graph Neural Networks in Recommender Systems: A Survey. *ACM Comput. Surv.* (2022).
[64] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing graph neural networks with message passing data flow graph. In *MLSys*. 515–528.
[65] 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 *Eurosys*. 417–434.
[66] Minji Yoon, Théophile Gervet, Baoxu Shi, Sufeng Niu, Qi He, and Jaewon Yang. 2021. Performance-Adaptive Sampling Strategy Towards Fast and Accurate Graph Neural Networks. In *KDD*. 2046–2056.
[67] Hanqing Zeng, Muhan Zhang, Yinglong Xia, Ajitesh Srivastava, Andrey Malevich, Rajgopal Kannan, Viktor Prasanna, Long Jin, and Ren Chen. 2021. Decoupling the Depth and Scope of Graph Neural Networks. In *NeurIPS*.
[68] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2019. Accurate, Efficient and Scalable Graph Embedding. In *IPDPS*. 462–471.
[69] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *NeurIPS*. 5171–5181.
[70] Qingru Zhang, David Wipf, Quan Gan, and Le Song. 2021. A Biased Graph Neural Network Sampler with Near-Optimal Regret. In *NeurIPS*. 8833–8844.
[71] Tianyi Zhang, Aditya Desai, Gaurav Gupta, and Anshumali Shrivastava. 2024. HashOrder: Accelerating Graph Processing Through Hashing-based Reordering.
[72] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *IA3*. IEEE, 36–44.
[73] 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. In *NeurIPS*. 11247–11256.

Received July 2024; revised September 2024; accepted November 2024