PCGraph: Accelerating GNN Inference on Large Graphs via Partition Caching

Lizhi Zhang¹, Zhiquan Lai¹, Yu Tang¹, Dongsheng Li¹, Feng Liu¹, Xiaochun Luo²
¹ National Laboratory for Parallel and Distributed Processing (PDL), Computer College, National University of Defence Technology, Changsha, Hunan, China
² PLA News Media Center, Beijing, China
Email: zhanglizhi15@nudt.edu.cn

Abstract—Graph neural networks (GNNs) have been emerging as powerful learning tools for unstructured data and successfully applied to many graph-based application domains. Sampling-based GNN inference is commonly adopted in existing graph learning frameworks to handle large-scale graphs. However, this approach is restricted by the problems of redundant vertex embedding computation in GPU and inefficient loading of vertex features from CPU to GPU. In this paper, we propose PCGraph, a system that supports adaptive GNN inference and feature partition caching. PCGraph significantly reduces the vertex embedding computation time by adaptive GNN inference technology, which selects the optimal inference algorithm and minimizes vertex embedding computation. PCGraph also reduces the redundant data transfer between the CPU and GPU by partition the target vertices and caching their corresponding partitions in turn. We evaluate PCGraph against two state-of-the-art industrial GNN frameworks, i.e., PyG and DGL, on a diverse array of benchmarks. Experimental results show that PCGraph reduces up to 99% vertex embedding computation and 98.5% data loading time, and achieves up to 360× performance speedup over the state-of-the-art baselines.

Index Terms—graph neural networks, inference, embedding computation, feature caching, pipeline parallel

I. INTRODUCTION

As a kind of unstructured data, graphs have been widely used in real-world applications [1], including social networks [2], knowledge graphs [3], physical systems [4], and recommendation systems [5], [6]. Graph neural networks (GNNs) [7]–[11] have emerged as powerful tools to model graph-structured data and explore complex connections between elements.

GNN inference is the key to applying the trained GNN models to real-world scenarios, and it is also a necessary stage to test the training quality of GNNs. However, performing GNN inference on industrial-scale graphs is a complex problem. The time of GNN reference is usually much longer than GNN training. In the era of big data, graphs in the real world could reach the scale of billions. For example, Microsoft Academic Graph (MAG) [12] constructs a heterogeneous graph with 111 million vertices and 3.23 billion edges. Facebook maintains a social network graph with nearly 2 billion users [13]. It is challenging to infer GNNs on such giant graphs as one batch, which would exceed GPU memory capacity. Therefore, the sampling-based inference method [14]–[16] is commonly discussed for efficient large graphs inference, which reduces memory footprint while maintaining convergence accuracy. Unfortunately, the sampling-based GNN inference faces the problems of redundancy vertex embedding computation and feature data loading. Due to the clustering characteristics of the real-world graph, different vertices in the graph may share many common neighbors [17]. Therefore, the existing GNN inference frameworks usually repeatedly load features and compute the embeddings of some common neighbor vertices, leading to redundancy and unnecessary data transfer and computation in GNN inference.

Directly apply multi-layer sampling graph inference, which is the same as the graph training, will lead to the repeated generation of embeddings between mini-batches. DGL [18] and PyG [19] adopt hierarchical sampling inference to divide the trained model into layers. Only when the vertex embedding computation of one layer is completed will the following layer’s calculation be performed. There is no repeated generation of vertex embeddings in the algorithm, and inference can achieve better efficiency in some tasks. However, the hierarchical sampling inference increases mini-batch sampling and data exchanges between CPU and GPU. Moreover, for industrial-scale graphs, this inference technology will compute embeddings of irrelevant vertices, resulting in a waste of computing resources. HAG [17] eliminates redundant computation and data transfers in GNN inference by a new GNN graph representation, but complex search algorithms will consume much time. ROC [20] improves GNN inference performance in a distributed cluster, which has higher restrictions on hardware conditions.

To reduce the time consumed by feature data loading, NGra [21] and NeuGraph [22] overlap data transfer and model computation through a pipeline scheduling to hide the data transfer latency. However, due to its irregular chunk size, the data loading time cannot be well hidden. PyG [19] caches all feature data on the GPU memory before GNN computation to reduce data loading time during inference. However, this method is only suitable for small datasets and cannot complete the reference of real-world graphs. PaGraph [23] adopts a computation-aware feature caching strategy that can reduce data loading time. Still, due to limited GPU memory and limited cache hit rate, PaGraph cannot sufficiently reduce the data loading time.

Generally, GNN inference runs in a pipeline parallel with...
mini-batch sampling, data loading, and vertex embedding computation. This paper focuses on accelerating the vertex embedding computation and data loading of the GNN inference over large graphs. Our key ideas are minimizing the vertex embedding computation by an adaptive GNN inference technique and reducing the data loading time via feature partition caching.

Firstly, we observed that hierarchical sampling inference would compute the embedding of many irrelevant vertices when there are few target vertices. Still, multi-layer sampling inference can only calculate the embedding related to them, significantly reducing the vertex embedding computation. However, for GNN inference with more target vertices, multi-layer sampling inference will have many repeated vertex embeddings between batches. Hierarchical sampling inference only computes once for each embedding, which can better reduce the vertex embedding computation. As our first contribution, we introduce an adaptive GNN inference technique, which can select hierarchical sampling reference or multi-layer sampling reference according to the computation pattern of different inference tasks to minimize the embeddings computation. According to these two inference algorithms, we propose an adaptive coefficient, which serves as the basis for selecting inference inference algorithms.

Secondly, we found that some vertices between different batches are duplicated, resulting in repeated feature loading of these vertices. Because the GNNs are relatively simple and occupy much less GPU memory. We can use the rest of GPU memory to cache vertex features. However, for giant graphs in the real world, the features of the whole graph cannot be fully cached. We also notice that in GNN reference, the order of target vertices is fixed, and the neighborhood of target vertices can be determined in advance. As our second contribution, we propose a feature partition caching strategy for giant graphs to achieve efficient data loading. We divide the target vertices and cache the relevant feature data in advance when inferring each part of vertices. After the computation, replace it with the subsequent partition of feature data. This feature partition caching strategy dramatically reduces cached graph features, and the vertices in each batch can match with cached features in GPU memory.

We incorporate the above design ideas into PCGraph, a sampling-based GNN framework targeting fast inference for large graphs, implemented on top of Deep Graph Library (DGL). We evaluate the performance of PCGraph by executing three typical GNN algorithms on three representative datasets, namely, ogbn-papers100M, ogbn-products, and Reddit. Experimental results show that PCGraph reduces up to 99% vertex embedding computation and 98.5% data loading time, and achieves up to 360× performance speedup over the state-of-the-art baselines.

The remainder of this paper is organized as follows. Section II introduces the background and challenges of traditional GNN inference. We present the PCGraph inference methods in Section III. The evaluation and analyses of the critical technologies to GNN inference are presented in Section IV.

Finally, we conclude the paper in Section V.

II. BACKGROUND AND MOTIVATION

This section begins with a brief background on GNNs, followed by an introduction to multi-layer sampling inference and hierarchical sampling inference, and then discusses the limitations of existing approaches to motivate this work.

A. Graph Neural Networks

Graph Neural Network [7]–[11] is a representative work in deep learning. Training neural networks on graph data have been widely used because its accuracy is much higher than traditional multilayer perception [24]. GNNs mean that information is propagated through graphs and transformed through neural networks. GNN layers generate intermediate embeddings by aggregating the information from in-edge neighbors of the target vertex. Then use the neural network to convert the aggregated features into output features, which will be sent to the next layer as input features. After superimposing several GNN layers, the final embeddings are obtained, which integrates the whole receptive field of the target vertex. Specifically, the graph neural networks iteratively update the vertex representation according to:

$$\hat{h}_i^{l+1} = \frac{1}{|N_i|} \sum_{j \in N_i} h_j^l$$
$$h_i^{l+1} = \sigma \left( W^{l} \hat{h}_i^{l+1} \right)$$

(1)

Where $\hat{h}_i^{l+1}$ is the information of neighbors aggregated by vertex $i$, $N_i$ is the vertex set connected with vertex $i$, $|N_i|$ represents the in-edge number of vertex $i$, $h_i^{l+1}$ is the embedding of vertex $i$ in the $(l+1)$th layer, $\sigma(\cdot)$ denotes the nonlinear activation function. $W^l$ are the learnable parameters of layer $l$.

The number of neighbor hops aggregated by the target vertex is related to the number of the GNN layers. In a $l$-layer GNN, vertices are able to collect information from $l$-hop in-neighbors.
B. GNN inference

The goal of GNN inference is to compute the \( l \)-layer vertex embeddings for downstream vertex prediction or graph classification. When performing inference, it is usually better to truly aggregate all neighbors to get rid of the randomness introduced by sampling [18]. However, full-graph forward propagation is usually infeasible on GPU due to limited memory and slow on CPU due to slow computation. There are two algorithms of GNN inference with limited GPU memory via mini-batch and neighborhood sampling, namely multi-layer algorithms of GNN inference with limited GPU memory and slow on CPU due to slow computation. There are two algorithms of GNN inference with limited GPU memory via mini-batch and neighborhood sampling, namely multi-layer sampling inference and hierarchical sampling inference.

Multi-layer Sampling Inference. In this inference algorithm, every GNN inference iteration includes three major phases, as shown in Figure 1, namely mini-batch sampling (1), data loading (5), and vertex embedding computation (4). The sampling step samples vertex to generate a mini-batch (1) and gather features of all vertices in the mini-batch (2). When the GNNs have multiple layers, the sub-graph sampling repeats recursively; that is, from a sampled neighbor vertex, it continues sampling its neighbors. The number of recursions is determined by the number of GNN layers. In the data loading step, data is transferred from the host memory to the GPU through PCIe to prepare features for GPU computation (5). Finally, the GPU performs the embedding computation over the loaded data samples with the trained model weights (4). Specifically, the embedding computation is based on the number of model weights layers, and the calculation is being performed forward on the sampled subgraph until it outputs the final embedding.

Hierarchical Sampling Inference. The hierarchical sampling inference is different from the multi-layer sampling inference, as the embeddings of all vertices should be computed by layer. Specifically, for a particular GNN layer, hierarchical sampling inference calculates the embeddings of all vertices in mini-batches. Therefore, the inference algorithm will have an outer loop iterating over the layers and an inner loop iterating over the mini-batches of vertices. Figure 2 depicts the steps of such an inference algorithm. In the sampling step, only all the 1-hop neighbors of the target vertex are sampled (1). These vertices and their corresponding features (2) form a mini-batch. The data is transferred from the host memory to the GPU through PCIe in the data loading step (5). Then the GPU performs the embedding computation over the loaded data samples with one layer of trained model weights (4). After computing the embeddings of a batch, it is written back from GPU to CPU (5). When all embeddings of the dataset are calculated, the new embeddings are represented as new feature inputs to update the embeddings of the next layer (6).

C. Challenges of GNN Inference.

GNN inference based on CPU-GPU architectures has severe data loading and embedding vertex computation problems that need to be resolved.

Redundant vertex access pattern. Due to the vertex dependency nature of real-world graphs, different vertices in a graph may share many common neighbors. For example, in webpage graphs, different websites under the same domain generally have some common links. As another example, in recommendation systems, users in the same group may have interests in common items. This joint neighbor is because a vertex in a graph is likely to be connected with multiple vertices, making it possible to select multiple times by different mini-batches. This redundant vertex access pattern leads to redundancy in vertex data loading and vertex embedding computation [17], [25].

The inefficiency in data loading. The redundant vertex access pattern exacerbates the data loading burden. As shown in Figure 3, in a 2-layer GNN computation, to obtain the embeddings of \( A \) and \( B \), it is necessary to load the data of \( A \), \( B \), and \( C \) many times. As a result, the loaded vertices are dozens of times that of the original dataset for each epoch. Since the GPU can only start training after the data is loaded, there is a long waiting time before the vertex embedding calculation. PaGraph [23] and PyTorch-Direct [26] find that data loading time from CPU to GPU usually dominates the end-to-end GNN training time and incurs significant time and resource overheads. PyG [19] adopts an entire dataset cache strategy to solve the redundancy of data loading, but this is not suitable for large datasets and cannot be applied to the real world. Therefore, it is necessary to propose a caching technology suitable for giant graph datasets.

The redundancy in vertex embedding computation. The redundancy of vertex access will also exacerbate the redundant vertex embedding computation. As shown in Figure 3, the GNN computes the final embeddings of \( A \) and \( B \) by aggregating the previous-layer embeddings of neighbors.
Because vertices in the input graph share common neighbors, the GNN performs redundant embedding computation of \{A, B, C\} twice. DGL [18] and PyG [19] avoid this redundant vertex embedding computation through hierarchical sampling inference. Hierarchical sampling reference only samples the 1-hop neighbors to compute a layer of embeddings. This method avoids the vertex embedding computation between overlapped neighborhoods, but it may result in another form of computational redundancy caused by the generation of irrelevant vertex embeddings. Hierarchical sampling inference needs to compute the embedding of all vertices in the graph, but the embedding of some vertices is not necessary to generate the target vertex embedding. Therefore, we need to analyze these two redundant vertex embedding computation modes to find the optimal vertex embedding computation algorithm.

III. PCGraph

Motivated by the challenges discussed in Section II-C, we propose PCGraph, a novel system to enable fast vertex embedding computation and data loading on GNN inference over large graphs. We design PCGraph with two key ideas: 1) an adaptive GNN inference technology to reduce the vertex embedding computation, 2) a feature partition caching strategy in a single machine for reducing the data loaded from CPU to GPU.

A. Adaptive GNN Inference

By introducing Section II-B, we analyze that although hierarchical sampling inference does not compute repeated embeddings, it may generate embedding of irrelevant vertices for large-scale graphs. Another problem of hierarchical sampling reference is that it is necessary to record all vertex embeddings in each layer, and the occupation of host memory cannot be ignored. Inference on vast datasets may lead to an overflow of host memory. Although multi-layer sampling reference does not produce irrelevant vertex embeddings, the batch of multi-layer sampling is large in scale. There are repeated vertex embeddings among batches, which will also lead to redundancy of computation. This section analyzes the vertex embedding computation and data transfer required by the two inference algorithms to find the optimal solution.

Vertex embedding computation. To ensure that GNN inference technology has better applicability to different graphs, PCGraph can adapt to appropriate inference algorithms according to the differences of inference tasks to minimize the redundancy of vertex embeddings computation. We define the adaptive coefficient of a graph \(g\) as the criterion for selecting the optimal inference algorithm, which expressed as:

\[
Adap(g) = \frac{v \sum_{i=0}^{l-1} d_v^i}{V \ast l}
\]  

(2)

Where \(v\) represents the number of target vertex in the graph. Because the sampled vertex in multi-layer sampling increase exponentially with the number of layers, it is not feasible to use all neighbors sampling. We set the sampled neighbors in each layer as \(d_v\), \(d_v\) denotes the average degree (number of neighboring vertices) of the target vertex. \(V\) indicates the number of vertices in the whole graph, and \(l\) is the number of layers of GNN. The numerator \(v \sum_{i=0}^{l-1} d_v^i\) on the right side of Equation 2 is the number of vertex embeddings that needs to be calculated in multi-layer sampling inference, where \(d_v^i\) is the number of \(i\)-hop neighbors of the target vertex and \(vd_v^i\) represents vertex embeddings that need to be calculated at \((l - i)th\) layer. The denominator \(V \ast l\) on the right side of Equation 2 represents the total number of vertex embeddings generated in hierarchical sampling inference. Due to each layer needs to generate the embedding of all vertices, the vertex embedding computation of one layer is \(V\).

The adaptive coefficient of a graph controls inference and denotes the ratio of computation required by multi-layer sampling inference and hierarchical sampling inference. If \(Adap(g) < 1\), PCGraph selects multi-layer sampling inference because it has fewer embedding computations in this case. Otherwise, PCGraph adopts the hierarchical sampling inference.

Loaded feature data. Although adaptive inference technology reduces the vertex embedding computation, there is inefficient data loading in both inference algorithms. In multi-layer sampling inference, only the features of the source vertex in each batch need to be loaded. Thus, when the average degree is \(d_v\), the total loaded feature data for \(v\) target vertices with the inference of \(l\)-layer GNN in this inference algorithm is:

\[
M(data) = vd_v^l
\]  

(3)

As for hierarchical sampling inference, it is necessary to sample the 1-hop neighbors of all vertices in the graph \((d_V\) denotes the average degrees of the whole graph.) and calculate their embeddings by GNN layer. Thus, the loaded data in one inference epoch is:

\[
H(data) = lVd_V
\]  

(4)

We adopt a feature partition caching strategy to reduce the redundant data loaded in the inference process to improve inference efficiency. We will introduce this method in Section III-B. Because of avoiding a large amount of data loading, we usually only consider the embedding computation of the graph for adaptive inference algorithms rather than the data loading.

B. Feature Partition Caching

Caching policy. To avoid redundant data loading from CPU to GPU, data caching is a widely discussed technology. Because the GNNs are relatively simple and occupy much less GPU memory. The current work leverage the rest of GPU memory to cache the vertex features and directly fetch the feature data from the GPU during model computation without frequent access to the host memory. PyG [19] caches the whole graph features in the GPU memory, which has high volume requirements for GPU memory and cannot fully cache the features of giant graphs. The current design of the GNN system [23] caches features that may be frequently accessed in GPU. Some features can be directly fetched from GPU if they are cached; yet, there are still some missed features that need
to be queried from the host memory. This feature loss is due to the GPU memory cannot cache all the required features.

We introduce feature partition caching to PCGraph, a feature caching strategy with a 100% hit rate in GPU to address data collection inefficiency. PCGraph loads different partitions of features to GPU memory in turn during the inference process to ensure that the feature of each batch can be accessed in the GPU. Unlike the training algorithm requiring a randomly shuffled sequence of training samples, it is impossible to predicate the vertices in each mini-batch at runtime. The sequence of the vertices in the inference algorithm is fixed, enabling us to determine the vertex range of each batch in reference. We only need to sample the neighbors of the target vertex in order and convert the aggregated information to obtain the vertex embeddings. Therefore, it is not hard to foretell which vertex will be accessed at the next mini-batch.

Usually, static caching is more efficient and straightforward, but GPU memory cannot cache all the feature data. Thus, a dynamic caching policy is favored. PCGraph dynamically schedules feature partition to GPU memory according to the range of target vertices during reference. To do so, we can divide the target vertex into several partitions in advance and obtain all l-hop neighbors of the target vertex in each partition. When inferring a specific part of the target vertex, the corresponding neighbor features are cached in GPU memory beforehand. Before the inference of target vertices in the next partition, PCGraph clears the cached feature data and loads the next partition into GPU memory.

**Partition policy.** To realize the efficient inference of giant graphs, we should first partition the feature data in advance. We determine the number of vertex partitions by checking the remaining available GPU memory space during inference and allocating the available GPU memory for caching partitions in turn. When the GPU free memory is small, the target vertex needs to be divided into more partitions to accommodate the free memory. More partitions mean more data loading and processing are required during inference, thus reducing inference efficiency.

When randomly partition the target vertex, we observe an inefficient caching phenomenon, i.e., the neighborhood expansion of the target vertex in each partition is vast, which leads to too much memory occupation. Taking two training vertices as an example, we can see from the left of Figure 4 that the target vertices (green vertices) are scattered in the graph with random partition, and there are many expanded neighbors (gray vertices). However, as shown in the right of Figure 4, the target vertices of cluster partition are concentrated in similar clusters, and the extended neighbors are also limited to the same cluster. Therefore, the cluster partition can limit the size of the graph partitions, and large graphs can be partitioned with a smaller number of partitions while satisfying the free GPU memory. Specifically, we adopt cluster-based graph partition(such as Metis [27] and Graclus [28]) to minimize the cut edges; thus, the target vertex in each partition can better concentrate in similar clusters.

**Fast feature collection.** Because searching features by vertices number in a batch requires traversing all cached data, the sequential lookup is time-consuming. Therefore, besides a consecutive space for caching feature data in GPU memory, we also organize the vertex metadata into a global index table to answer where the queried vertex is located for fast feature lookup. Each time a feature partition is cached, PCGraph establishes a global index table, in which the key corresponds to the global vertex number and value indicates the position of vertex feature in cached data. Figure 5 shows the fast feature collection: ① Get the input vertex number of the batch. ② Find the location of the local vertex from the global index table. ③ Obtain input features from locally cached data. In this way, we can achieve fast feature queries from cached data through the global index table. This size of the global index table is far less than the cached feature data, which only occupies little GPU memory.

**Multi-layer sampling inference with partition caching.**

Algorithm 1 presents the steps of multi-layer sampling inference with feature partition caching. First, we partition the target vertex into n parts (line 1), and obtain the L-hop neighborhood of each part (line 2, 3). Then, we load the features of the target vertex partition into GPU memory in turn (line 5). For the target vertices in each partition, we sample the L-hop neighborhood of the target vertex with a certain batchsize (line 7). After that, we collect feature data from the cached data, which forms input data with sampled batches (line 8). Finally, GPU performs forward vertex embedding computation over the input data with the trained model weight by layer until the final vertex embedding is output(line 10). When the inference of the target vertex in the first partition

---

![Fig. 4. The comparison of neighborhood expansion between random partition and cluster partition.](image)

![Fig. 5. Fast feature collection with global index table.](image)
is finished, GPU will clear the cached data partition and load the following data partition from the host memory to continue inference.

Hierarchical sampling inference with partition caching.
Algorithm 2 depicts the steps of hierarchical sampling inference with feature partition caching. In the first step, we partition the whole vertex into $n$ parts (line 1), and obtain the 1-hop neighborhood of each part (line 2, 3). Then, we cache the features corresponding to the 1-hop neighbors of the first vertex partition in turn into GPU memory (line 6). For the vertices in each partition, CPU samples the 1-hop neighbors of the vertex with a certain batchsize (line 8), and the GPU collects the corresponding features from the cached data to form the input data (line 9). Next, GPU calculates the first layer embedding of the vertex by using the first layer weight of the model (line 10). After the vertex embedding of the batch is computed, it will be written back to the intermediate variable in the host memory (line 11). GPU will clear the cached data partition and load the following data partition from the host memory to continue inference when the inference of the vertex in the first partition is finished. When the intermediate variable in the host memory records the embedding of all target vertices in the first GNN layer, the embeddings are represented as new features (line 12) to compute the embeddings of the next layer with the model weight of the next layer.

IV. EVALUATION

A. Experiment Setup

In this section, focusing on vertex classification, we will evaluate the efficiency and performance of the proposed method through experiments on several GNN models and datasets.

Datasets. The statistics of datasets used in the experiments are listed in Table I. The ogbn-papers100M dataset is a directed citation graph of 111 million papers indexed by MAG [12]. Each vertex is a paper, and each directed edge indicates that one paper cites another one. Overall, this dataset is orders-of-magnitude larger than any existing vertex classification datasets. The ogbn-products dataset is a graph that represents an Amazon products co-purchasing network [29]–[31]. The vertex represents products sold on Amazon, and edges between two products indicate they are purchased together. Reddit is a dataset an online discussion forum [16], [32], with each vertex being a post and each edge being a comment between posts. Unless otherwise stated, all experiments use the same train/validation/test splits as prior works [31], [33], [34].

When performing hierarchical sampling inference, it is usually better to truly aggregate over all neighbors to get rid of the randomness introduced by sampling. However, due to the exponentially expanded neighborhood in graph sampling, complete neighbor sampling for multi-layer sampling inference is inefficient. Thus, we use the average degree of each dataset as the sampled neighbors. We further show in Section IV-F that this average degree neighbor sampling does not impact the inference accuracy.

Environments. All the experiments are conducted on a machine with 1 NVIDIA 3090 GPU (24GB memory), 2 24-core Intel Xeon CPUs (2.40GHz), and 512GB DDR4 host memory. The machine is installed with Ubuntu 18.04, CUDA library v11.2. Our experiments are carried out on PyTorch [35], an open-source Python machine learning library. We compare the training speed with Deep Graph Library (DGLv0.5.3) [18] and PyTorch Geometry (PyG v1.6.3) [19], two of the state-of-the-art GNN training frameworks.

GNN models. The following three representative GNN models are used in our experiments: GraphSAGE [16], GCN [36], and GAT [37]. Specifically, we train GCN with neighborhood sampling due to exponentially extended neighborhood and limited GPU memory. We run mini-batch training with
the same hyperparameters on three frameworks to have a fair comparison among PCGraph, DGL, and PyG. This experiment uses 3-layer GNNs for training and inference, and the sampling batchsize is 1000. The hidden sizes of GraphSAGE, GCN, and GAT are 128, 128, and 32, respectively.

B. Overall Performance

First, we evaluate the overall inference performance by comparing it with state-of-the-art implementations of GNN on PyG and DGL. Figure 6 shows the end-to-end inference performance of GraphSAGE, GCN and GAT on different datasets. Overall, PCGraph inference speedup from $2.3 \times$ (GAT) to $2.8 \times$ (GraphSAGE) for ogbn-products and from $4.4 \times$ (GCN) to $5 \times$ (GraphSAGE) for Reddit, compared to DGL. In addition, PCGraph achieves an average $1.9 \times$ speedup (up to $2.1 \times$), and an average $3.6 \times$ speedup (up to $3.8 \times$) over PyG on ogbn-products and Reddit, respectively. We notice that PCGraph achieves an average speedup of $360 \times$ on ogbn-paper100M, compared with DGL. This significant speedup is because compared with the vast scale of the ogbn-papers100M, the number of target vertices is small, and there are large redundant computations in DGL inference. However, PCGraph can select appropriate inference techniques to compute only the relevant vertex embeddings, thereby significantly reducing embedding computations. We also observed that the inference of PyG on ogbn-papers100M has an out-of-memory error. This memory overflow problem is because PyG needs to record intermediate variables in the host memory during the inference process and cache the original feature data in the GPU memory.

C. Time Breakdown

To further explore the data loading and vertex embedding computation overhead reduced by PCGraph, we break down the GNN training time into CPU mini-batch sampling time, CPU-GPU data loading time, and GPU computation time. Figure 7 shows the overall inference time breakdown corresponding to results in Figure 6. We show that the data loading time of PCGraph is reduced by 93.5% to 99% across all datasets, compared with DGL. PCGraph also reduces its data loading time by 85% to 91.3% on ogbn-products and Reddit, compared with PyG.

We observe that the vertex embedding computation time of PCGraph on ogbn-papers100M is greatly reduced, but the time on the other two datasets is not much different compared to PyG and DGL. According to Equation 2, this difference is because PCGraph adopts the multi-layer sampling inference for ogbn-papers100M, dramatically reducing the number of vertices needed for embedding computation. For the other two datasets, PCGraph uses the same inference algorithm as DGL and PyG. The main reason for the time reduction is that the feature partition caching reduces the data loading time. Since the PCGraph completes the feature collection on GPU, which alleviates the time-consuming sampling of CPU, we also observe the sampling time reduction of PCGraph on ogbn-product and Reddit datasets.

D. Impact of Inference Algorithm

We further evaluate the performance of multi-layer sampling inference and hierarchical sampling inference on different datasets to show the effectiveness of adaptive GNN inference technology. Table II lists the parameters of each dataset in Equation 2, and Table III computes the vertex embedding and the loaded data required by the two inference algorithms. We observe that the vertex embedding computation of multi-layer sampling inference on the ogbn-papers100M is negligible, while the calculation of hierarchical sampling inference on the other two datasets is minor. Different inference algorithms have significant differences in embedding computation and data loading. Therefore, PCGraph can select the optimal inference algorithm according to different graph tasks.
Figure 8 compares the inference performance between hierarchical sampling inference and multi-layer sampling inference without any optimization. Inferring the ogbn-papers100M, multi-layer sampling is $360 \times$ faster than hierarchical sampling inference on average. However, hierarchical sampling inference is $6.7 \times$ faster than multi-layer sampling inference on average when inference on ogbn-products. Because the embedding computation and data loading of multi-layer sampling inference is less on ogbn-papers100M, the speedups of this algorithm are very significant. Although the embedding calculation required by hierarchical sampling inference is much less than that of multi-layer sampling inference on the Reddit dataset, the actual inference performance of the two is not much different. This is because hierarchical sampling inference involves a lot of data loading, which impairs the inference performance.

**E. Effectiveness of Partition Caching**

Next, to prove the effectiveness of feature partition caching, we apply this strategy to multi-layer sampling inference and hierarchical sampling inference, respectively. From Table IV, we can see that the feature partition caching can reduce the loaded data by 83.5%, 99%, and 98.5% for multi-layer sampling inference on ogbn-papers100M, ogbn-products, and Reddit, respectively. For hierarchical sampling inference, the data loading reduction of the three datasets by feature partition caching is 94%, 98% and, 98.5%, respectively.

Figure 9 and Figure 10 show the speedup of hierarchical sampling inference and multi-layer sampling inference using feature partition caching. Multi-layer sampling inference with feature partition caching achieves an average $1.5 \times$ speedup and an average $1.7 \times$ speedup over multi-layer sampling inference without optimizations on ogbn-products and Reddit. In addition, hierarchical sampling inference achieves inference speedups from $1.6 \times$ (ogbn-products) to $4.7 \times$ (Reddit) with feature partition caching, compared with no optimization on data loading. We notice that feature partition caching accelerates hierarchical sampling inference more significantly than multi-layer sampling inference. This difference is because hierarchical sampling inference spends more time and resources on data loading, which PCGraph can optimize. It is worth noting that using feature partition caching on ogbn-papers100M reduces the performance of multi-layer sampling inference. This deterioration is due to the scale of ogbn-papers100M is vast, but the target vertices are relatively few; thus, the benefits brought by feature caching can not match the cost of building a global index table shown in Figure 5.

**F. Inference Convergence**

To confirm the correctness of our implementation, we evaluate the inference accuracy of training GraphSAGE with PyG, DGL, and PCGraph over ogbn-papers100M, ogbn-products, and Reddit datasets on a single GPU. As shown in Figure 11, inference of GraphSAGE converges to approximately the same accuracy as PyG and DGL within the same number of epochs on different datasets. Hence, the model effectiveness is guaranteed, verifying that our system can accelerate the real-world graphs training without damaging convergence.

**V. CONCLUSION**

In this paper, we present PCGraph for efficient GNN inference. We analyze the amount of vertex embedding computations and data loading times required by multi-layer sampling inference and hierarchical sampling inference. Then we adopt an adaptive GNN inference technology to select the optimal inference algorithm, which can minimize the vertex embedding computation. Besides, we design a feature partition caching strategy. With this strategy, we could accelerate feature data loading during GNN inference. We demonstrate that PCGraph achieves a higher inference speed over PyG and DGL.
on various benchmarks. Our experimental results show that PCGraph achieves up to 99% vertex embedding computation and 98.5% data loading time reduction. PCGraph gets up to 360× performance speedup over the state-of-the-art baselines.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China under Grant No. 62025208 and 61932001.

REFERENCES