PaGraph: Scaling GNN Training on Large Graphs via Computation-aware Caching

Zhiqi Lin[1], Cheng Li[1,2], Youshan Miao[3], Yunxin Liu[3], Yinlong Xu[1,2]
University of Science and Technology of China[1]
Anhui Province Key Laboratory of High Performance Computing[2], Microsoft Research[3]

ABSTRACT

Emerging graph neural networks (GNNs) have extended the successes of deep learning techniques against datasets like images and texts to more complex graph-structured data. By leveraging GPU accelerators, existing frameworks combine both mini-batch and sampling for effective and efficient model training on large graphs. However, this setup faces a scalability issue since loading rich vertices features from CPU to GPU through a limited bandwidth link usually dominates the training cycle. In this paper, we propose PaGraph, a system that supports general and efficient sampling-based GNN training on single-server with multi-GPU. PaGraph significantly reduces the data loading time by exploiting available GPU resources to keep frequently accessed graph data with a *cache*. It also embodies a lightweight yet effective caching policy that takes into account graph structural information and data access patterns of sampling-based GNN training simultaneously. Furthermore, to scale out on multiple GPUs, PaGraph develops a fast GNN-computation-aware partition algorithm to avoid cross-partition access during data parallel training and achieves better cache efficiency. Evaluations on two representative GNN models, GCN and GraphSAGE, show that PaGraph achieves up to 96.8% data loading time reductions and up to 4.8× performance speedup over the state-of-the-art baselines. Together with preprocessing optimization, PaGraph further delivers up to 16.0× end-to-end speedup.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; **Machine learning**.

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421281

KEYWORDS

graph neural network, cache, large graph, graph partition, multi-GPU

ACM Reference Format:
Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, Yinlong Xu . 2020. PaGraph: Scaling GNN Training on Large Graphs via Computation-aware Caching. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421281

1 INTRODUCTION

Recently, graph neural networks (GNNs) [16, 22, 45, 52] have been proposed to extend deep neural networks from handling unstructured data such as images and texts to structured graph data, and have been successfully applied to various important tasks including vertex classification [16, 22, 45], link prediction [14], relation extraction [51] and neural machine translation [3]. However, on the one hand, traditional graph processing engines like PowerGraph [15] lack auto-differentiation and tensor-based abstractions. On the other hand, current popular deep learning frameworks like TensorFlow [1] have insufficient support on vertex programming abstractions and graph operation primitives. To bridge the gap between these two fields and support GNNs, new systems like DGL [47], Python Geometric [13], MindSpore GraphEngine [32] and NeuGraph [29] were proposed to provide convenient and efficient graph operation primitives compatible with GNNs.

Many real-world graphs are at *giant* scale. For instance, a recent snapshot of the Facebook friendship graph is comprised of 700 million vertices and more than 100 billion edges [5]. Additional to the graph structure data, the high-dimensional features (typically ranging from 300 to 600) associated with vertices and edges lead to greater computation and storage complexity. Under the time and resource constraints, it would be no longer efficient or even feasible to make a full giant graph train totally as a batch. So, a typical practice is sampling [6, 16, 17], which repeatly samples subgraphs from the original graph as the input of a mini-batch, reducing the single mini-batch computation, while still converging to expected accuracy.

Unfortunately, the sampling-based GNN training faces a data loading problem, i.e., the data movement from the host memory to the GPU memory, especially for loading vertex features (e.g., user description, paper semantic embeddings [43]). We have observed 74% of training time spent on data loading. The major reason is that in a GNN model, comparing with the huge data size, it often uses deep neural networks with low computational complexity. So in GPU, the GNN computation takes much shorter time than the data loading. What's even worse, when multiple GPUs within a single physical machine are used to speed up the training, the demand for data samples loaded from CPU to GPU grows proportionally. There are some other optimized strategies, such as preprocessing [7], which is used to prune the GNN models for better training performance. However, even with these optimized strategies, data movement still dominates the training process.

In this paper, we focus on accelerating the sampling-based GNN training over large graphs on a multi-GPU machine. Our key idea is to reduce the data movement between CPU and GPUs. This work is mainly based on the following observations. First, due to referencing vertices along graph structure in GNN computation, different training iterations may use overlapped mini-batch data and exhibit a redundant vertex access pattern. As shown in the following case study (§2.3), comparing with vertices number of the target graph, it can load up to more than 4× the number of vertices during an epoch. Second, in each iteration, the training computation only needs to keep the corresponding sampled subgraphs to current mini-batch, which only consumes a small fraction, e.g., no more than 10% of GPU memory. Therefore, the rest of GPU memory can be used to cache the features of most frequently visit vertices for free, avoiding repeatedly loading from host memory. However, the current design of sampling-based training frameworks hasn't explored the availability of spare GPU memory to achieve it.

Introducing caching to accelerate sampling-based GNN training faces the following system challenges. First, due to the random accessing patterns by the sampling nature, it's hard to predict which graph data will be visited in the next mini-batch. Instead of trying to predict the next visited graph data, we leverage the fact that the vertices with higher out-degrees have a higher probability to be sampled into a mini-batch. Following this fact, as our first contribution, we adopt a static caching strategy to keep frequently accessed graph data in GPU and introduce a new caching-enabled data loading mechanism. When a sampled mini-batch arrives at GPU, the required feature data will be then fetched from both local GPU cache and the host memory which managed by original Graph Storage Server.

Second, the current system design balances the training workloads across multiple GPUs while sharing a single copy of full graph data among them. This single graph parallelism makes the above caching solution inefficient. To avoid this inefficiency, as our second contribution, we adapt "data parallelism" to sampling-based GNN training with caching in a single multi-GPU machine, where each GPU works on its own graph partition. Clearly, the benefits of data parallelism are that data locality can be improved and the number of cached vertices in total will be increased. To achieve this, we design a novel GNN-aware graph partition algorithm, which differs from the traditional general-purpose graph partition algorithms [2, 21] in two aspects. First, to balance the training workload between GPUs, we have to ensure each partition contains a similar number of training target vertices that are correlated to workload, rather than arbitrary vertices. Second, to avoid cross-partition visits when sampling, for every train vertex, we replicate all its L-hop reachable neighbors in its partition. Since L is often not greater than 2 [16, 22, 45] and our partition algorithm is highly optimized, the resulted redundancy is acceptable.

We incorporate the above design ideas into PaGraph, a novel sampling-based GNN framework targeting multi-GPU training over large graphs, implemented on top of the Deep Graph Library (DGL) [47] and PyTorch [38]. We train two famous and commonly evaluated GNNs of GCN [22] and GraphSAGE [16] on large real-world graphs. Experimental results show that PaGraph achieves up to 96.8% data loading time reductions for each training epoch and up to 4.8× speedup over DGL, while converging to approximately the same accuracy within the same number of epochs. Together with preprocessing optimization, PaGraph even achieves up to 16.0× speedup over DGL.

2 BACKGROUND AND MOTIVATION

In this section, we begin with a brief background on GNNs, followed by an introduction to sampling-based GNN training, and then discuss the limitations of existing approaches to motivate this work.

2.1 Graph Neural Networks

In this work, we target attributed graphs, where vertices or edges are associated with a large number of features (often more than hundreds), in addition to the graph structural information. A GNN model often consists of multiple layers. Figure 1 shows the architecture of a two-layer GNN model, where the blue and green color represents the first and second layer, respectively.

The computation crossing different layers follows a traditional *vertex-centric graph iterative processing* model. At each layer, every vertex follows its incoming edges to aggregate features from neighbors (thin arrows), then uses neural networks to transform the features into an output feature

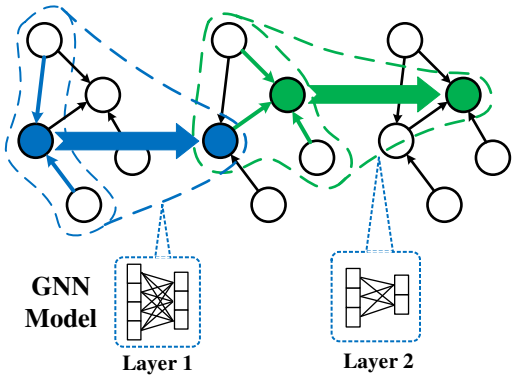


Figure 1: A two-layer GNN architecture.

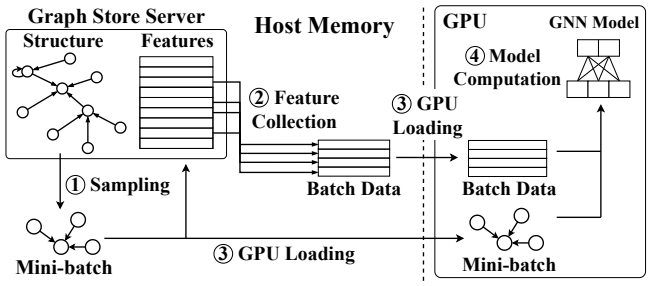


Figure 2: Sampling-based GNN training pipeline, where data loading corresponds to stages ② - ③.

(thick arrows), which will be fed into the next layer as input feature [16]. Within the same layer, all the vertices share the same aggregation neural network and transformation neural network. A single GNN layer could only achieve information passing from direct (1-hop) in-neighbor vertices. With 2 GNN layers, by ingesting the previous layer's output into the next layer as input, we could connect 2-hop in-neighbors. Similarly, in a *L-layer* GNN, vertices are able to collect information from *L-hop* in-neighbors. The output of the last layer could be used to enhance tasks such as vertex classification [16, 22], or used as embeddings for relation extraction [51].

2.2 Sampling-based Training with GPU

Training a GNN model to reach a desirable accuracy often requires a few tens of *epochs*, each of which is defined as a full scanning of all train vertices of a target graph. An epoch consists of a sequence of iterations, during each a mini-batch of train vertices will be randomly selected for evaluating and updating that model. However, unlike training data such as images and sentences where each data sample is independent, graph data is highly structural connected. As a result, training on a vertex requires not only loading the vertex associated data, but also the data of its linked edges and

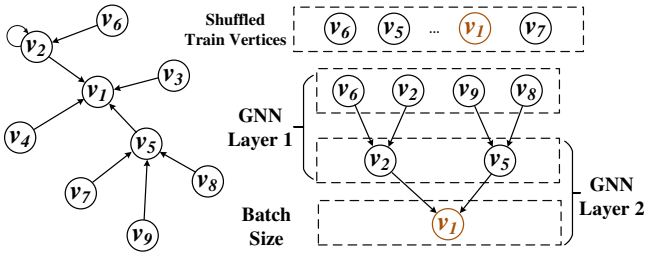


Figure 3: Sampling a mini-batch for a 2-layer GNN. (sampling neighbor = 2, mini-batch size = 1)

connected vertices, making the data loading in GNN quite different from traditional machine learning training.

The unique data access pattern of GNN computation makes it hard to efficiently handle the exponential growth of the number of vertices and their associated features for training w.r.t the number of GNN layers. To address this issue, instead, sampling, as a typical optimization solution, is widely adopted [6, 7, 16, 17]. Sampling-based training selects a limited number of a mini-batch's layer-hop reachable neighbors, and achieves almost the same training accuracy with much less computational cost [7].

Figure 2 diagrams the workflow of sampling-based GNN training over GPU, which is the de facto standard accelerator for deep learning training in almost every single framework [1, 8, 11, 32, 38]. There is a global Graph Store Server, which manages the full graph structure along with feature data in CPU memory. Every training iteration involves three major steps, namely, sampling(①), data loading(②-③) and model computation(④). In each iteration, a *Data Sampler* randomly collects a number (mini-batch size) of train vertices, and then traverses the graph structure and samples their *L-hop* neighbor vertices, to form the input data samples(①). In a 2-neighbor sampling, as shown in Figure 3, for train vertex v_1, it selects v_2, v_5 from its 1-hop in-neighbors. Then, it samples 2-hop in-neighbors v_6, v_2, v_9, v_8 from the selected 1-hop vertices v_2, v_5's direct neighbors. The data loading step prepares the feature data for GPU computation. Here we would emphasize that data loading is not just transferring data from CPU to GPU via PCIe. Instead, the data loading procedure goes through two stages as follows. A *Data Loader* picks up a mini-batch and queries the graph store to gather the features of all vertices from that batch (②), and loads these samples into GPU memory via a PCIe link (③). Finally, a *Trainer* at CPU launches the GPU kernels to perform GNN computation on GPU over the loaded data samples (④). The process will run iteratively until the target model converges.

Dataset	V#	Loaded V#	Volume (GB)
reddit	232K	959K	2.16
wikitalk	2.39M	3.93M	8.81
livejournal	4.04M	16.30M	36.56
lj-link	5.20M	21.16M	47.45
lj-large	10.69M	31.42M	47.06
enwiki	12.15M	43.88M	65.38

Table 1: Loaded number of vertex and corresponding data volume during one epoch. V# denotes vertex number.

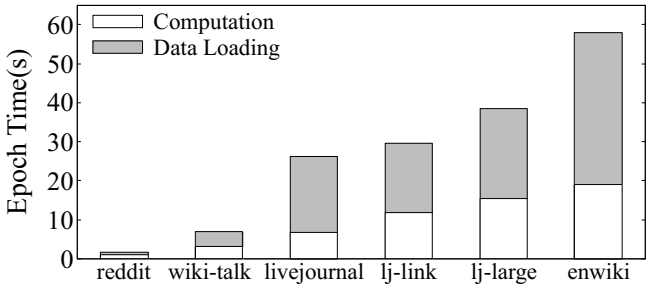


Figure 4: Data loading and computation time in an epoch[1].

Utilization	GraphSAGE	GCN
Computation	21.22%	17.12%
Memory	7.17%	7.94%

Table 2: GPU resource utilization of different models.

2.3 Problems and Opportunities

Thanks to the powerful computational resource, GPU evidently speeds up the GNN training process. Our experiments show that when training the GCN [22] model over the livejournal [50] dataset, a single 1080Ti GPU provides 3× the performance of the CPU-only (16 cores) setup. And the performance gap would be gradually amplified as more GPUs are involved. However, there is still a large room for improving GNN training with GPUs. In particular, sampling-based GNN training over GPU suffers from a severe data loading problem that needs to be resolved.

The data loading inefficiency. Throughout experiments with 6 real-world graphs, reddit [16], wiki-talk [27], livejournal [50], lj-link [23], lj-large [33] and enwiki [24], we find that data loading from CPU to GPU usually dominates the end-to-end GNN training time. In the experiments, we adopt a representative GNN model, a 2-layer GCN [22], for performance understanding. We take the widely-used neighbor sampling to create mini-batches of vertices for training. As suggested by existing work [7], the neighbor sampling method here selects 2 neighbors for each vertex. Figure 4 summarizes the training epoch time over different graphs, and also shows the time break down into data loading and computation. Note that we omit the sampling overhead since sampling runs faster than and is overlapped with data loading. Clearly, across all the graph datasets, data loading takes much longer time than computation. For example, GCN on livejournal spends 74% of the end-to-end training time on data loading. This situation will become worse, when multiple GPUs are used to collectively train a shared model.

With a deep understanding of results, we identify the following factors that seriously slow down the whole training process. Meanwhile, they also provide us the opportunities to further improve the performance of sampling-based GNN training with GPUs.

Redundant vertex access pattern. We continue to understand the total amount of data sent from CPU to GPU for completing a training epoch. Table 1 summaries the number of vertices loaded during an epoch, which can be up to more than 4 times the total number of vertices. It indicates that some vertices are loaded multiple times. This is because that a vertex in a graph is likely to be connected with multiple train vertices, which make it have chances to be selected multiple times by different mini-batches. For example, the redundant vertex access pattern exacerbates the data loading burden, and creates data loading of tens of gigabytes for each epoch as shown in Table 1.

GPU resource underutilized. Since the data loading phase precedes the GPU training phase, we further explore the negative impacts of the data loading bottleneck. We summarize the resource utilization in Table 2 where two GNN models (GCN and GraphSAGE) are being trained. Surprisingly, only around 20% of the GPU computational resources are in-use, with even less memory consumption, e.g., less than 10%. This is because the mini-batch data that CPU sent to GPU is not sufficient to fully explore the hardware parallelism in GPU. In the meanwhile, the GPU is idle waiting for training data samples to arrive at most time.

High CPU contention. We further break down the time spent in different stages of the data loading process and find that the feature collection phase is CPU-intensive and takes much longer than the CPU-GPU data movement. This surprising result leads to: (1) with a single GPU, we achieved about 8GB/s PCIe bandwidth utilization, while the average utilization is lower; (2) with multiple GPUs, concurrent workers for collecting features would contend CPU resources with samplers, e.g., the time for sampling and feature collection increased by 88% and 59% over the 1-GPU case, respectively,

[1]Average time is collected by running DGL with a GTX-1080Ti GPU

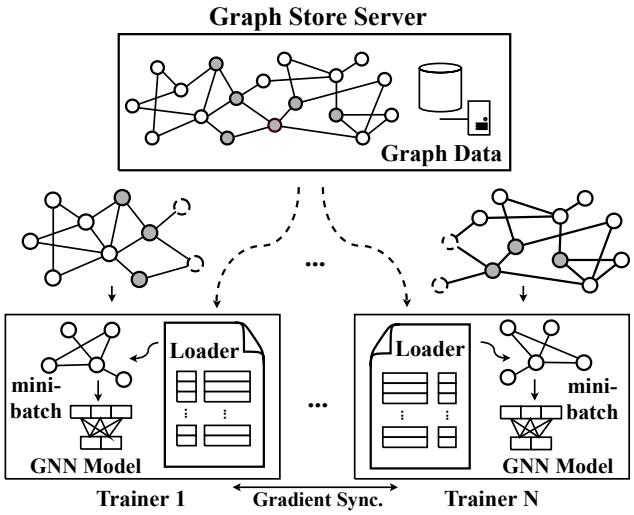


Figure 5: Overall architecture of PaGraph. ***For simplicity, directions of edges in graphs are omitted.***

while the GPU computation time remains unchanged. In the 4-GPU case, the maximum PCIe bandwidth utilization drops to half and the average is even worse. This indicates that the CPU capacity cannot cope with the GPU computation demands, given the large amount of data required by each iteration.

Summary. The multiple accesses to vertices and the spare GPU memory space mentioned above provide us the chance to cache as many vertices as possible in GPU memory for cheap re-usage. This caching solution can reduce the amount of data loaded from host to GPU, which significantly reduces the CPU contentions, shortens the time spent on collecting features and PCIe transferring, and eventually improves training speed. Therefore, in this work, we aim to introduce caching to sampling-based GNN training over GPUs for performance gains.

3 PAGRAPH

Motivated by the experimental results presented in Section 2.3, we propose PaGraph, a novel system to enable fast sampling-based GNN data parallel training on large graphs. We introduce two key techniques to PaGraph: 1) a GNN computation-aware caching mechanism for reducing the data loaded from CPU to GPU, and 2) a cache-friendly data parallel training method to scale GNN training on multiple GPUs.

Figure 5 shows the overall architecture of PaGraph. For a single machine with N GPUs, the whole graph is partitioned into N sub-graphs. The graph structural information of these partitions together with the vertex feature data are stored in a global *Graph Store Server* in the CPU shared memory. There

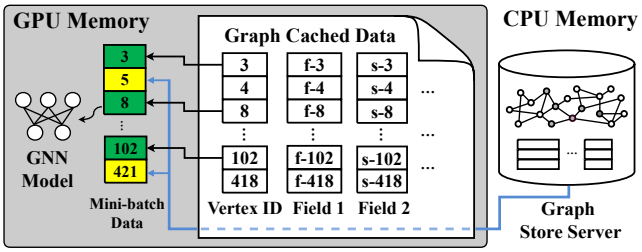


Figure 6: New data loading with caching.

are N independent *Trainers*. Each *Trainer* is responsible for training computation on a dedicated GPU. It replicates the target GNN model, and only consumes its own data partition. Each GPU contains a *cache*, which keeps features of frequently visited vertices.

During each training iteration, each *Trainer* receives the graph structural information of a mini-batch produced by a *Sampler*. As shown in Figure 6, it also gets the associated vertex features gathered by a *Data Loader* by either directly fetching from its own GPU *cache* if they are cached (marked as green), or querying the *Graph Store Server* for missing features (marked as yellow). The former case saves the data copying overhead from CPU to GPU, while the latter one cannot. After that, the structural data, and both the cached and loaded (from host) vertex features of that mini-batch are fed into the GNN model to compute gradients. *Trainers* do not interact with each other except synchronizing locally produced gradients at the end of each iteration among peers to update the model parameters.

3.1 GNN Computation-aware Caching

Caching policy. To generate better models, for each epoch, most training algorithms require a randomly shuffled sequence of training samples, which makes it impossible to predicate the vertices in each mini-batch at runtime. The neighbors of a vertex are also randomly selected and thus unpredictable as well during training. Therefore, it's hard to foretell which vertex is most likely to be accessed at the next mini-batch. Even though, due to the unique access patterns of the neighbor-sampling method, the out-degree of a vertex indicates the probability for it being selected throughout the whole epoch. This says that with a higher out-degree, a vertex is more likely to be an in-neighbor of other vertices, and thus is more likely to be sampled in a mini-batch. Thus, it is sufficient to select high out-degree vertices for filling up the cache.

Usually, a dynamic caching policy is favored. However, it is not suitable in our situation where the *cache* is inside GPU. This is because GPU cannot work stand-alone, and all computations performed at GPU must be assembled into

GPU kernels and launched by CPU. Most current GNNs are lightweight [52], and hence graph data swapping between CPU memory and GPU memory has intolerable overhead during training (see §2.3). Therefore, instead of making on-the-fly decisions on what to be cached, e.g., LRU [9], we use static caching to avoid the overhead of dynamic data swapping. To do so, we can pre-sort vertices by out-degree offline, and at runtime select top high out-degree vertices to fill up the GPU cache. We leave the exploration of efficient dynamic caching policies as future work. Though it is simple, as shown in §5, this static caching policy effectively achieves a high cache hit ratio.

Cache memory space. To avoid resource contention with the high priority training computation, we need to estimate the maximum amount of available GPU memory for the cache allocation. To achieve this, we leverage the fact that the memory consumption is similar across training iterations. This is because the sampling-based mini-batch training uses almost the same amount of data samples as input and performs almost the same amount of computation to train a shared GNN model for each iteration. As a result, it is sufficient to decide the right cache size via a one-time sampling of GPU memory usage. In more details, right after the training of the first mini-batch, we check the size of free GPU memory during training and allocate the available GPU memory for caching graph data accordingly (see §4 for more details).

Data management. In the GPU *cache*, we manage the cached vertex features by maintaining two separated spaces. First, we allocate consecutive memory blocks for feature data. The cached feature data of vertices is organized as several large $[N, K_i]$ matrices, where N denotes the number of the cached vertices, and K_i is the dimension of features under the i-th feature-name fields. Second, to enable fast lookup, we organize the vertex meta data into a hash table to answer whether the queried vertex is cached and where it locates for later retrieval. The meta data is far less than the cached feature data, e.g., no more than 50 MB for a partition with 10 million vertices.

Discussions. Less skewed graph would limit the efficiency of the cache optimization. However, many real-world graphs including the evaluated ones are power-law graphs [12, 15, 34], and exhibit high skewness. Therefore, our solution could benefit common GNN training jobs. For those with less skewed graph, one possible solution would be combining both prefetching and dynamic caching to achieve good cache efficiency. We leave this exploration as future work.

3.2 Data Parallel Training and Partition

The current design of GNN systems such as DGL, balance computation across multiple GPUs but make them share a single copy of graph data [10]. When directly applying

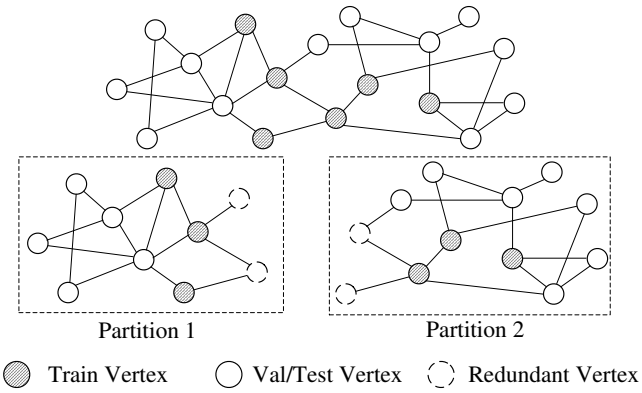


Figure 7: Self-reliance graph partition for a one-layer GNN model. *For simplicity, the edge directions are omitted.*

the above GNN-aware caching method to this setting, we observe a cache inefficiency phenomena, i.e., the cache hit ratio keeps decreasing with the increasing number of GPUs. This is because that the single graph serves the data visiting locality for parallel *Trainers* on multiple GPUs, and thus all GPU *caches* would keep similar vertices.

To address this cache inefficiency, we introduce "data parallelism" to PaGraph, which has been widely applied to leverage multiple GPUs to train neural network models efficiently. In our system, rather than accessing to a shared graph, a Trainer consumes its own data partition (subgraph), performs the training computation to get local gradients, and then exchanges gradients among peers to update its model replica synchronously. Clearly, the benefits of data parallelism are that data locality can be improved and the number of cached vertices in total will be increased. To make this happen, although there exist numerous graph partition algorithms [2, 15, 21], we still need to design a new one to meet the following two goals specific to data parallel GNN training. First, it should keep computation balanced across different Trainers, as unbalanced computation may result in different number of mini-batches per epoch for different Trainers. This will break gradient synchronization and get training stuck. Second, it needs to avoid cross-partition accesses from different Trainers as possible.

Computation balance. To achieve computation balance across different Trainers, all the partitions should have a similar number of train vertices. Assume that we need K partitions. We scan the whole train vertex set, and iteratively assign the scanned vertex to one of K partitions. During every iteration t, a train vertex v_t is assigned with a K dimension score vector, where the i-th element represents the feasibility for assigning the vertex to the i-th partition for $i \in [1, K]$. The score is computed by Eq.(1).

$$score_{v_t}^{(i)} = |TV_i \cap IN(V_t)| \cdot \frac{TV_{avg} - |TV_i|}{|PV_i|}, \quad (1)$$

TV_i represents the train vertex set already assigned to the i-th partition. $IN(V_t)$ denotes the L-hop in-neighbor set of train vertex v_t. PV_i controls the workload balance, and denotes the total number of vertices in the i-th partition, including the replicated vertices. v_t is most likely to be assigned to a partition which has smallest PV. TV_{avg} is the expected number of train vertices in the final i-th partition. To achieve computation balance, we set TV_{avg} as $\frac{|TV|}{K}$, which indicates that all partitions will get almost the same number of train vertices.

Self-reliance. For the queries including the edges across different partitions, they must be forwarded to the Graph Store Server to get a full set of neighbors. Inspired by [15, 39], PaGraph introduces minimum extra vertices and edges in each partition to deal with cross-partition edges. Figure 7 demonstrates how to partition a graph with self-reliance for a one-layer GNN model. The shaded vertices denote the train vertices, the white vertices are val/test vertices, and the dashed vertices are the introduced redundant vertices.

For each partition, PaGraph extends the sub-graph with redundant vertices and edges to include all the neighbor vertices of required hops during sampling. For GNN models with L GNN layers, we will include L-hops in-neighbor vertices for each train vertex, e.g., a one-layer GNN model only requires to include direct in-neighbors of each train vertex. For the extended vertices, PaGraph only brings in necessary edges to satisfy the required message flow during training. Note that the extended vertices may include train vertices. These extended train vertices are regarded as mirrors [15] and will not be trained. In this way, partitions are independent from each other. Each Trainer can sample mini-batches completely from its own sub-graph without accessing global graph structure.

4 IMPLEMENTATION

We built PaGraph on the top of the Deep Graph Library (DGL (v0.4)) [47] and PyTorch (v1.3) [38]. To demonstrate the generality of the PaGraph design, we are implementing its core ideas in MindSpore [32], an open source deep learning framework that enables GNN training. Since the gradient synchronization is not a primary concern in GNN training, we just adopt an existing solution, NCCL RingAllReduce, to implement data parallel GNN training [36]. We use the Graph Store Server implemented by DGL to store graph structural data and feature data in CPU shared memory[2]. We store the full graph structure as adjacency matrix of CSC format [42].

[2]For larger graph that cannot reside in the main memory, we leave the tiered storage for solving this problem in future work

Algorithm 1: Computation-balanced and cross-partition access free graph partition

Input: graph $\mathcal{G}$, train vertex set TV, number of hops L, partition number K
Output: graph partition $\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_k$

for $i \leftarrow 1$ *to* K **do**
  $\mathcal{G}_i \leftarrow \varnothing$ // Initialization
for *each train vertex* $v_t \in TV$ **do**
  $IN(v_t) \leftarrow$ IN-NEIGHBOR$(v_t, \mathcal{G}, L)$
  $score_{v_t} \leftarrow$ SCORE$(v_t, IN(v_t))$
  $ind \leftarrow \arg\max_{i \in [1,K]} \{score_{v_t}^{(i)}\}$
  $\mathcal{G}_{ind} \leftarrow \mathcal{G}_{ind} \cup \{v_t\} \cup \{IN(v_t)\}$
return $\{\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_K\}$

CSC format aggregates in-neighbor sets of each vertex inside a consecutive memory space and thus allows fast access to fetch them. The overall implementation consists of 1.47K lines of code in Python, where 293 lines are for the *Data Loader*, 436 lines correspond to the partition algorithm, and 737 lines are changes made to the original sampling training workflow.

Cache initialization and maintenance. We extend *Data Loader* to incorporate simple Python APIs for fetching required data from local GPU memory or CPU memory, as demonstrated in Figure 8. First, we initialize *Data Loader* by connecting it to Graph Store Server and making it be aware of data parallel training configurations, e.g., how many GPUs are used, which GPU it serves for, etc. During the first mini-batch training, *Data Loader* will check the total GPU memory, denoted as *total_mem*, and the peak GPU memory allocated by PyTorch, denoted by *used_mem*. We also preserve a certain size of GPU memory unused for memory fluctuations during training (denoted as *preserved_mem*), which we found 1.0 GB is enough in practice. After that, we call *loader.auto_cache*, which calculates the size of available memory and assigns this amount of memory to cache by subtracting *used_mem* and *preserved_mem* from *total_mem*. Once the cache is allocated, *Data Loader* continuously loads into *cache* features (specified by *field_names*) of vertices with top high out-degrees until *cache* is full. To reduce the time cost of this initialization process, we offline analyze the sub-graph structure, and rank vertices based on their out-degrees. From the second iteration on, *Data Loader* fetches data from Graph Store Server and local GPU *cache* by calling *loader.fetch_data*, which is parameterized by *mini_batch*. This new loading flow is transparent to training jobs.

Offline graph partition. Algorithm 1 presents our GNN-specific partitioning method for splitting large graphs to fit caching-enabled data parallel training. We implement our partition algorithm based on Linear Deterministic Greedy

(LDG)[2], a stream-based partition solution. We only evaluate the train vertices (line 3), assign them to the target partition with index *ind* (line 6), and include their L-hop in-neighbors (line 7) into a partition. Preparing partitions for GNN model training is an offline job before training. Although this step introduces extra time cost and consumes additional resources, it is acceptable due to the following reasons. First, the partition is a one-time offline job, and thus will not block the training process. Second, recent studies [19, 28] find that a number of machine learning jobs are generated by parameter tuning experiments and they share most configurations include dataset and partitions. Therefore, this one-time job can be beneficial across these parameter tuning experiments.

To reduce the storage burden of partitions, we remove the redundant vertices and edges which don't make contributions during training. For a given L-layer graph neural network, we exam whether a val/test vertex is out of L-hops away from all train vertices. If so, we remove this vertex and its associated edges from the sub-graph. Also we remove redundant edges to avoid inefficient message flow. Undirected edges will be converted into directed edges if one of the message flow direction is unnecessary. With the refinement of graph structure, graph data associated with redundant vertices and edges is also removed.

Resource isolation. GNN libraries are usually implemented on top of deep learning frameworks and add to these frameworks additional graph storage and sampling features. In the implementation, both sampling and data loading are placed in a single process and simultaneously use OpenMP [37] to perform concurrent jobs, such as pre-fetching multiple mini-batches, gathering the mini-batch data and copying it into a consecutive memory space, etc. We observed interference between sampling and data loading with a single process, where both of them compete CPU resources. This interference also slows down the frequency of the kernel launching from CPU host to GPU device. To eliminate this resource contention, in our implementation, we make sampling and data loading use separate processes, and tune the OpenMP configurations to balance CPU resources between them.

Local shuffling. To achieve better empirical performance [41] in data parallel training, data samples need to be shuffled. *Shuffling* can be done across partitions (globally) or within each partition (locally), where the former case may lead to faster convergence, but is more costly than the latter one. However, local shuffling is widely adopted in real-world practice, and recent studies show that it can still perform well with slightly slower convergence [31]. Therefore, PaGraph locks the assigned partition for each Trainer and shuffles the accessing order of data samples in that partition before every epoch begins. We further show in §5.7 that local shuffling does not impact both the model accuracy and the convergence speed.

```
import PaGraph.storage as storage
# create loader
loader = storage.GraphCacheServer(
    graph_store_server, num_nodes,
    node_mapping, trainer_id)
# auto cache graph data
loader.auto_cache(graph, field_names)
# auto fetch data from CPU and GPU
loader.fetch_data(mini_batch)
```

Figure 8: APIs to extend *Data Loader* for cache-enabled data loading.

Discussions. Currently, PaGraph works on a single multi-GPU server, but the core idea of caching and graph partition can be directly applied to distributed GNN training to leverage even more GPUs for handling larger graphs that cannot be fitting into the memory of a single server. The only thing that needs to carefully design in that extension is to efficiently synchronize gradients among servers, so as to avoid the synchronization bottleneck.

5 EVALUATION

We intensively evaluate the performance of PaGraph using representative GNN models and real-world datasets. In particular, we explore the single GPU performance, the scaling efficiency achieved using multiple GPUs, and also conduct a breakdown analysis on the cache performance under different conditions and quality of our partition algorithms.

5.1 Experimental Setup

Environments. We deploy experiments on a multi-GPU server which consists of dual Intel Xeon E5-2620v4 CPUs, 512GB DDR4 main memory and 4 NVIDIA GTX 1080Ti (11GB memory) GPUs with no NVLink connections. The machine is installed with CentOS 7.6, CUDA library v10.1, DGL [47] v0.4, PyTorch [38] v1.3.

Datasets. We use six real-world graph datasets listed in Table 3 for evaluation, including reddit [16] social network, wiki-talk [27] page edition history network, three variants of the livejournal [23, 33, 50] communication network (livejournal, lj-link, lj-large), and enwiki [24] wikipedia links network. The "feature" column represents the dimension of vertex features, and the "label" column shows the number of vertex classes. Following the setting of Reddit in [16], we split the vertices in each dataset into train, val, and test vertex categories with a 65:10:25 ratio.

Dataset	vertex#	edge#	feature	label
reddit	232.96K	114.61M	602	41
wiki-talk	2.39M	10.04M	600	60
livejournal	4.04M	69.46M	600	60
lj-link	5.20M	103.55M	600	60
lj-large	10.69M	224.61M	400	60
enwiki	12.15M	756.28M	400	60

Table 3: Statistics of datasets. (K: thousand. M: million)

Model	Layer#	Agg	Apply	Hidden
GCN	2	sum	FC	32
GraphSAGE	2	mean	FC	16

Table 4: Model architectures. (FC: fully connected layer. Agg: type of aggregating operation. Hidden: hidden embedding dimension of the first layer output.)

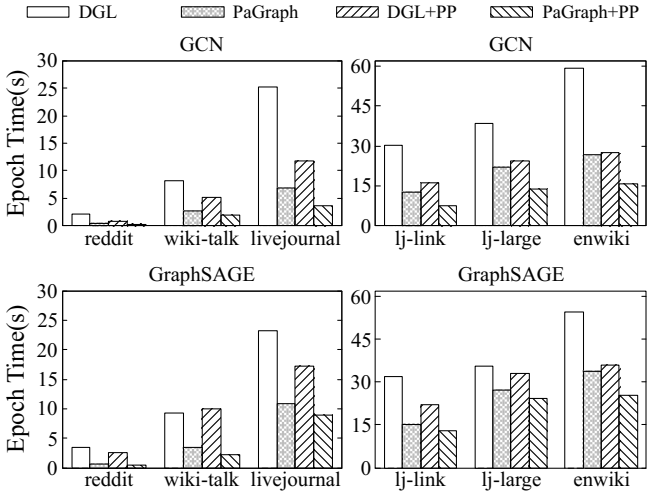


Figure 9: Training performance of GCN and GraphSAGE on single GPU. (PP: preprocessing optimization)

Sampling methods. We adopt Neighbor Sampling (NS) [16] combined with Skip Connection [17] to evaluate PaGraph. A recent work[7] already shows neighbor-based sampling training can achieve comparable model performance with full graph training, where only two neighbors are sampled at each neighborhood layer. Thus, in our evaluation, we choose to follow their suggestion that two random neighbors will be sampled for a vertex in each layer to form a mini-batch. We set the training batch size as 6000 for the whole of evaluation, following the suggestion from DGL [10].

Models. We use two representative GNN models, Graph Convolutional Network (GCN) [22] and GraphSAGE [16] to evaluate PaGraph. We adopt the same model architectures and configurations from [16, 22], with more details in Table 4. *Graph Convolutional Network (GCN)* generalizes the convolution operation on graphs as follows. Each vertex in a GCN layer aggregates the features of its neighbor vertices using a *sum* operation. Then the aggregated feature goes through a fully-connected layer and a ReLU activation to generate output representations. GCN has shown its excellence in tasks such as classifications [22] and neural machine translation [3]. *GraphSAGE* [16] is an inductive learning model to learn the different aggregation functions on different number of hops. There are four aggregation types in GraphSAGE: GCN(sum), Mean, LSTM and Pooling. We choose to present the experimental results of the GraphSAGE-Mean model since all four aggregators show the similar execution cost [16].

Baselines. We deploy the original DGL as the first baseline to compare to PaGraph. In addition, we run an advanced DGL where the preprocessing [7] optimization is enabled as another baseline, denoted as "DGL+PP". The idea behind preprocessing is to remove the first layer of GNN model by aggregating the corresponding features offline. Even though, the data loading still dominates the performance of preprocessing-enabled training and makes it fail to take most advantage of this optimization. Unless stated otherwise, all the performance numbers are the average of results from 10 epochs.

5.2 Single GPU Performance

We firstly evaluate the training performance of PaGraph on a single GPU. Figure 9 shows the training performance of GCN and GraphSAGE on different datasets. Overall, PaGraph achieves training performance speedups from 1.7× (lj-large) to 3.7× (reddit) for GCN and from 2.1× (enwiki) to 4.8× (reddit) for GraphSAGE, compared to DGL. We observe that the combination of preprocessing and DGL (DGL+PP) behaves differently across the two GNN models, i.e., the performance speedup of GCN achieved by preprocessing is better than GraphSAGE. This is due to the different forwarding procedures used in GCN and GraphSAGE. During forwarding in GCN, each vertex aggregates the features from its neighbors and sum-reduces these features into one single feature. Unlike GCN, for each vertex in the corresponding mini-batch, GraphSAGE has to save its own feature and the aggregated feature from its neighbors. This leads GraphSAGE to more CPU-GPU data transfer compared with GCN. In contrast, we further observe that our optimizations in PaGraph can better exploit the potential of the preprocessing optimization, other than the vanilla DGL. For instance, PaGraph+PP improves the performance of DGL+PP by 1.4× to 3.0× and 1.4× to 6.9× for both GCN and GraphSAGE across 6 datasets.

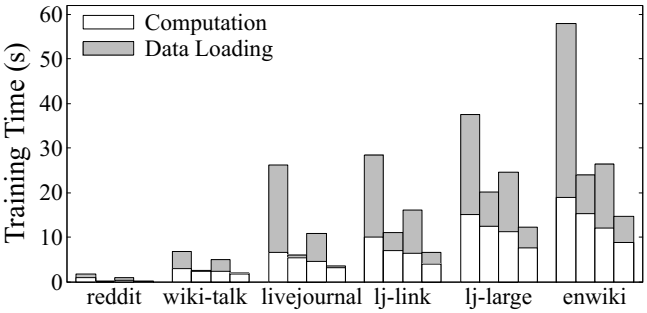


Figure 10: Breakdown of GCN training time on single GPU. Each bar cluster from left to right represents DGL, PaGraph, DGL+PP and PaGraph+PP.

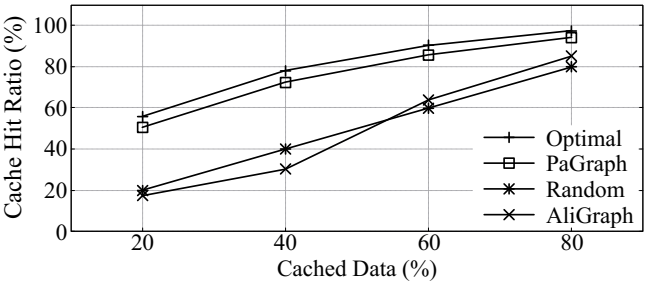


Figure 11: Cache policy comparison with different cache capacity. "Optimal" represents the ideal cache hit ratio. (Dataset: livejournal)

Training time breakdown. To further explore the data loading overhead reduced by PaGraph, we break down the GCN training time on both DGL and PaGraph into GPU computation time and CPU-GPU data loading time. We collect such system statistics using nvprof [35] and PyTorch Profiler [40]. Figure 10 shows the breakdown results corresponding to experiments related to GCN in Figure 9. In this clustered bar figure, from left to right each bar cluster presents the result of DGL, DGL+PP, PaGraph and PaGraph+PP, respectively. We show that although preprocessing saves both computation and data loading, it still suffers from data loading bottleneck, which occupies more than 50% of training time. PaGraph achieves lower cost of both data loading and computation. Data loading time of PaGraph is reduced by 77.4% to 96.8% across all datasets, compared with DGL. Similarly, PaGraph+PP also reduces its data loading time by 60.0% to 93.5%, compared with DGL+PP. Interestingly, the computation time is slightly reduced due to resource isolation implementation, where CPU resources are better leveraged and interactions between CPUs and GPUs are more smooth. We also observe the similar trends from the GraphSAGE experiments.

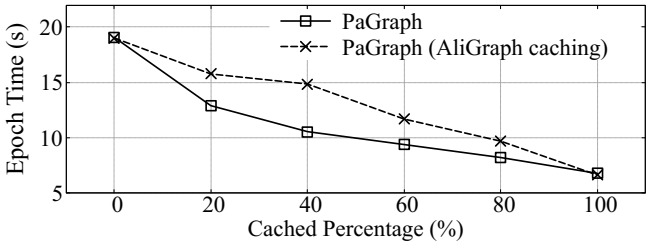


Figure 12: Training performance of PaGraph at different cached percentage. (Dataset: livejournal)

5.3 Effectiveness of Caching Policy

Next, we compare our static cache policy with the policy presented in AliGraph [53], which supports GNN training across multiple CPU machines. It reduces communication cost between training tasks and the remote storage system by caching a vertex at local if its in-degree to out-degree ratio exceeds a threshold. We also compare to the random strategy, which randomly keeps vertices inside Loader. To help understand how different cache policies perform, we derive the best cache hit ratio which can obtained if all subsequent vertex visits can be absorbed in cache. We denote this best cache hit ratio as "optimal". The derivation is done by analyzing the visiting trace of training. We did not directly compare PaGraph to AliGraph, since the open-source version of AliGraph did not include the cache code, and it was built atop of TensorFlow other than PyTorch and designed for CPU machines. To make a fair comparison, we follow the description in its paper to implement the AliGraph caching policy in PaGraph.

Figure 11 shows the cache hit ratio under different cached ratio using a single GPU, respectively. We observe that when only 20% of graph are cached, we can achieve more than 50% hit ratio, which is more than 200% of the performance of other polices. More interestingly, it shows that our caching policy is not complicated, but is incredibly effective, very close to the optimal case. We also achieve the similar close-optimal cache hit ratio on other datasets. Furthermore, we explore the performance implications of the different cache hit ratios achieved by both PaGraph and AliGraph. As shown in Figure 12, PaGraph achieves reasonable speedup as cached percentage increasing and its caching policy achieves up to 1.4× performance speedup than AliGraph.

We also observe that with our caching policy, PaGraph achieve huge data loading reductions on large graphs, compared to DGL. Table 5 shows the average reduced data loading volume of each epoch in each Trainer during GCN training. Overall, we achieve 91.8%, 80.9% and 81.0% loading reductions for lj-link, lj-large and enwiki datasets, respectively.

Dataset	DGL(GB)	PaGraph(GB)	PaGraph Cached
lj-link	46.17	3.80 (91.8%↓)	81%
lj-large	46.82	8.94 (80.9%↓)	58%
enwiki	65.38	12.42 (81.0%↓)	54%

Table 5: Average loaded data volume in one epoch of DGL and PaGraph on one GPU.

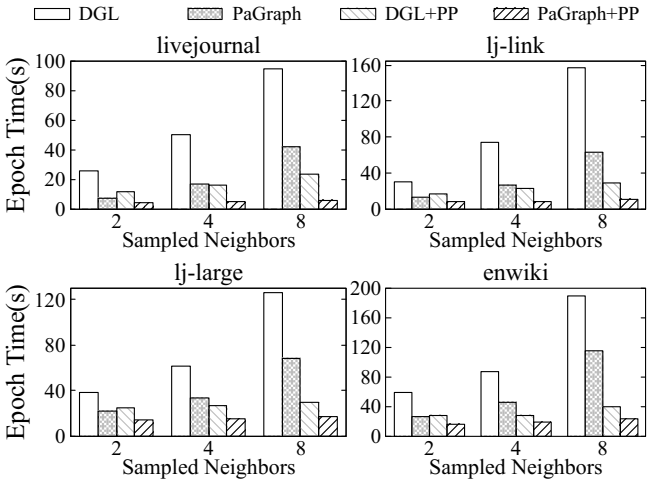


Figure 13: Multi-neighbor training performance.

5.4 Impact of Multi-neighbors

To verify the robustness of PaGraph, we evaluate the performance speedup w.r.t the number of sampled neighbors. Figure 13 shows training performance with different number of sampled neighbors on DGL, DGL+PP, PaGraph and PaGraph+PP. As the number of sampled neighbor increases, PaGraph combined with preprocessing can achieve even magnitude speedups (e.g., 16.0× of livejournal and 15.1× of lj-link at 8 sampled neighbors). Although preprocessing reduces GNN computation, it still suffers from data loading bottleneck. With combination of PaGraph and preprocessing, we can alleviate data loading cost as well as computation overhead, and thus achieve an approximately product-effect speedup.

5.5 Multi-GPU Performance

We evaluate the scalability with different number of GPUs. Figure 14 shows the experimental results of training GCN and GraphSAGE against two real-world datasets with a different number of GPUs. In general, PaGraph out-performs DGL and shows better scalability when GPU number increases. In a deep learning task, linear speedup is usually the upper bound. However, in PaGraph, the performance of multiple GPUs shows super-linear speedup, e.g., throughput on 4-GPU is 4.9× of which on the one of a single GPU when training

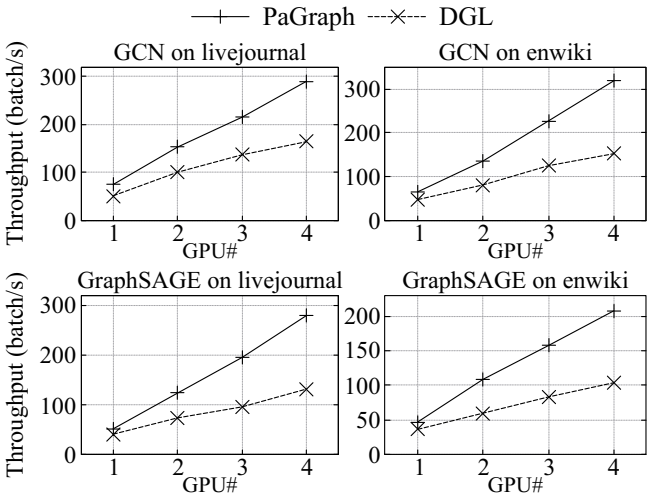


Figure 14: GCN and GraphSAGE scalability of PaGraph and DGL on 2 datasets. (Preprocessing enabled)

GCN on enwiki. This is because multiple GPUs provide more available memory for caching, thus can achieve a higher cache hit ratio and lower data loading cost. Therefore, with the linear increase of computation resources, the data loading overhead for each Trainer decreases, leading to super-linear speedup. To confirm this, we also test the performance of GCN on enwiki with a total cache size fixed 6 GB on four GPUs. It shows that the speedup on 4-GPU is only 3.7×, 23% lower than the speedup achieved without the limitation of cache size.

We also evaluate the performance of PaGraph on multiple GPUs to show its effectiveness on different algorithm optimizations. As shown in Figure 15, when using 2 GPUs, PaGraph achieves the performance of 1.8× to 4.5× over DGL across all datasets. With preprocessing optimization, PaGraph can achieve speedups from 2.1× to 8.5× over DGL with preprocessing.

5.6 Implications of Graph Partition

To validate the benefit of data parallel training enabled by graph partition, we first adapt DGL to use our caching mechanism, denoted as "DGL+Cache", and then compare PaGraph with DGL+Cache, where the only difference is that only the former uses partitioned graph data while the latter uses non-partitioned full graph. Figure 16a illustrates the cache hit ratio under different number of GPUs. Since Trainers in DGL+Cache shares the same global Graph Store which maintains the full graph structure, the cached part inside the GPU cache accessed by each Trainer is the same. However, within PaGraph, as the number of Trainer goes up, each Trainer consumes vertices sampled from a smaller set of train vertices, and exhibits better data locality. Therefore,

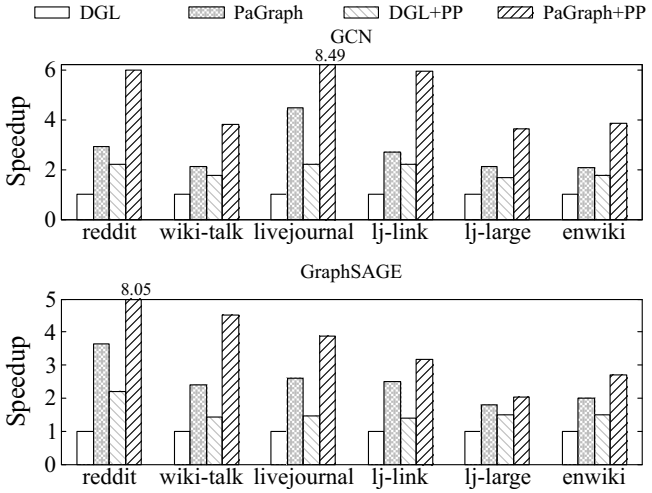
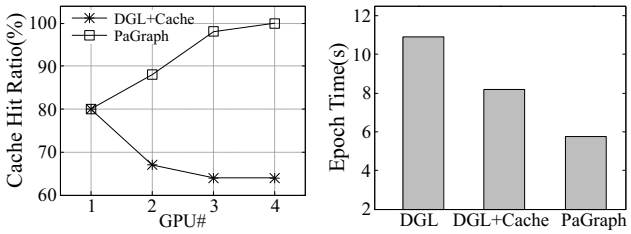


Figure 15: Training speedup with different optimizations of GCN and GraphSAGE on different datasets (2 GPUs used).



(a) Cache hit comparison. **(b) Performance comparison.**

Figure 16: Implications of graph partition in terms of cache hit ratio and performance of training GCN over the enwiki dataset.

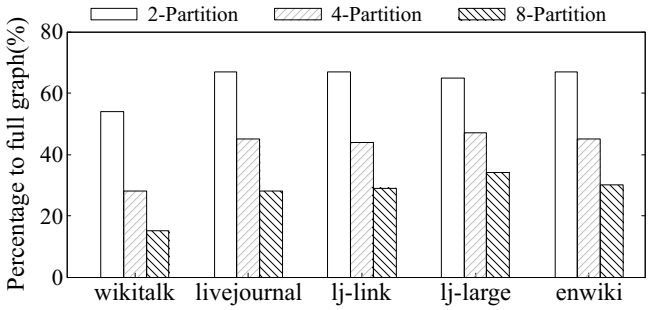


Figure 17: The proportion of vertices in each partition on average, compared to the full graph.

Dataset	livejournal	lj-link	lj-large	enwiki
Time (min)	6.0	8.1	17.1	32.2

Table 6: Partition time cost.

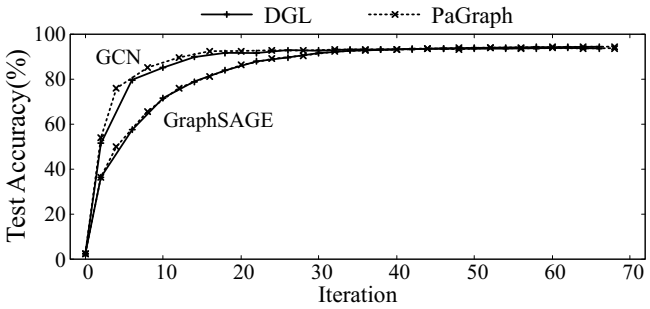


Figure 18: Test accuracy of PaGraph and DGL during 4-GPU training. (Dataset: Reddit)

the cache hit ratio achieved by PaGraph keeps increasing when more GPUs are in-use, while DGL+Cache observes a decreasing cache hit ratio. This difference can be directly translated into performance gaps shown in Figure 16b. With partition, PaGraph outperforms DGL+Cache and DGL by 40% and 130%, respectively.

However, the improvement of graph partitioning comes at prices, namely, redundancy across partitions and offline time cost. Next, we quantitatively analyze their impacts. Figure 17 shows the proportion of vertices in a single partition to the whole graph, when the number of partitions varies. The y-axis presents the proportion occupied by each partition relative to the full graph. As the partition number increases, the vertex number of each partition decreases. With our algorithm, we bring a small proportion of redundant vertices to each partition in a 8-partition configuration, accounting for 2.5% to 21.5% of full graph size. This cost is still affordable, given the training speedups achieved using partitions. Second, Table 6 shows the overall partition cost of four larger datasets out of the total 6, ranging from 6.0 to 32.2 minutes. We believe this is acceptable, since partition is an one-time task performed offline and its cost could be amortized among multiple epochs and repeated training for hyper-parameter tuning [49].

5.7 Training Convergence

To confirm the correctness of our implementation as well as the shuffling impact, we evaluate the test accuracy of training the two exercised GNN models with PaGraph and DGL over the reddit dataset on 4 GPUs. DGL adopts global shuffling while PaGraph adopts local shuffling. As shown in Figure 18, on both model GCN and GraphSAGE, PaGraph converges to approximately the same accuracy as the original DGL within the same number of computation iterations. We did

not conduct similar experiments for datasets other than Reddit because features of vertices in those datasets are randomly initialized.

6 RELATED WORK

Frameworks for GNN training. Deep learning frameworks like PyTorch [38], MXNet [8] and TensorFlow [1] have been widely adopted in both academia and industry. However, these frameworks don't provide enough graph operations required for GNN. Thus, in recent years, it drives the birth of a few specialized frameworks [4, 13, 18, 26, 29, 44, 47, 53] designed for handling graph neural networks. They all borrow and extend traditional graph processing primitives and vertex-programming abstractions into current deep learning frameworks. For instance, NeuGraph [29] develops graph operation primitives by Scatter-ApplyEdge-Gather-ApplyVertex (SAGA) to provide efficient and convenient programming models.

Full graph vs. sampling training. The full graph training trains graph neural network model with whole graph data in every forward-backward propagation, while the sampling training only trains part of vertices and edges in every forward-backward propagation. To deal with the large graphs which can not be fully filled into GPU, PBG [26] and NeuGraph [29] split the full graph into chunks and iteratively loads each chunk and its vertex data into CPU and GPU for full graph computation, respectively. While for sampling training, DGL [47] locates the full graph and its data in the CPU shared memory and only loads the required vertex and edge data into GPU memory at the beginning of every forward-backward propagation. For large graphs, recent work [7] already shows the sampling training can achieve faster convergence several times than full graph training with similar final model performance. AliGraph is a framework supporting sampling-based GNN training on CPU platforms, rather than exploring the potentials of GPUs [53].

Caching. Caching graph data benefits many graph processing tasks [18, 25, 53]. Pre-select [25] applies a static caching policy to speed up BFS-like computations. AliGraph [53] caches vertices and their data for distributed computing scenarios to avoid communication cost between trainer and remote storage. However, as we have verified in §5.3, its caching policy is not suitable for neighbor-based sampling characteristics. ROC [18] targets full graph training and explores an efficient strategy for swapping the intermediate results of GNN model between CPU and GPU by leveraging partition and caching. We differ from ROC as handling intermediate results is not the primary concern under sampling-based GNN training.

Graph partition. Partitioning graphs [2, 15, 20, 21, 30, 46] is widely adopted in distributed computations. PowerGraph [15] designs a vertex-cut partition algorithm for power-law graphs. NeuGraph [29] leverages Kernighan-Lin [21] to partition graphs into chunks with different sparsity level. [44] further proposes a GNN-specific graph partition algorithm to reduce the communication overhead among multiple machines. All the partition algorithms mentioned above are designed for non-GNN tasks or full graph GNN training. Our partition algorithm is designed for sampling training, and cooperated with caching.

Other optimizations. Gorder [48] designs a general graph re-ordering method to accelerate graph processing tasks. Norder [25] proposes a tailored re-ordering technique for BFS-like tasks. We have implemented these techniques in PaGraph but found there is no significant improvement on either partition quality or training performance.

7 CONCLUSION

Graph neural network is an emerging trend in deep learning. To accelerate training performance, we present PaGraph, a general sampling-based training scheme that leverages the combination of GNN computation-aware caching and graph partition. We implement PaGraph on DGL and PyTorch, and evaluate it with 2 representative GNN models over 6 graph datasets. Experimental results show that PaGraph reduces up to 96.8% data transfer and reaches up to 4.8× performance improvements, comparing to DGL. With preprocessing, PaGraph can even achieve a 16.0× performance speedup.

ACKNOWLEDGEMENTS

We sincerely thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by National Nature Science Foundation of China through grant No. 61802358 and 61772486, and sponsored by Huawei's joint innovation project.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.

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

[3] Joost Bastings, Ivan Titov, Wilker Aziz, Diego Marcheggiani, and Khalil Simaan. 2017. Graph Convolutional Encoders for Syntax-aware Neural Machine Translation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1957–1967.

[4] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. 2018. Relational

inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261* (2018).

[5] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW)*. ACM Press, Manhattan, USA, 595–601.

[6] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations (ICLR)*.

[7] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *International Conference on Machine Learning (ICML)*. 941–949.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *Proceedings of the Neural Information Processing Systems, Workshop on Machine Learning Systems (NeurIPS)*.

[9] Marek Chrobak and John Noga. 1999. LRU is better than FIFO. *Algorithmica* 23, 2 (1999), 180–185.

[10] DGL Team. 2019. DGL Large-Scale Training Tutorial. https://docs.dgl.ai/tutorials/models/5_giant_graph/2_giant.html. accessed, Jan-2020.

[11] DGL Team. 2019. DGL Sampling Training Examples. https://github.com/dmlc/dgl/tree/master/examples/pytorch/sampling. accessed, May-2020.

[12] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On power-law relationships of the internet topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Commu- nication (SIGCOMM)*. ACM New York, NY, USA, 251–262.

[13] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[14] Victor Garcia and Joan Bruna. 2018. Few-shot learning with graph neural networks. In *International Conference on Learning Representations (ICLR)*.

[15] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 17–30.

[16] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*. 1024–1034.

[17] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive sampling towards fast graph representation learning. In *Advances in Neural Information Processing Systems (NeurIPS)*. 4558–4567.

[18] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with ROC. In *Proceedings of the Machine Learning and Systems (MLSys)*. 187–198.

[19] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. 2019. The Case for Unifying Data Loading in Machine Learning Clusters. In *Proceedings of the USENIX HotCloud*.

[20] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[21] Brian W Kernighan and Shen Lin. 1970. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal* 49, 2 (1970), 291–307.

[22] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*.

[23] KONECT. 2017. LiveJournal links network dataset – KONECT. http://konect.uni-koblenz.de/networks/livejournal-links

[24] KONECT. 2017. Wikipedia links, English network dataset – KONECT. http://konect.uni-koblenz.de/networks/wikipedia_link_en

[25] Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam H Noh, and Jiwon Seo. 2019. Pre-select static caching and neighborhood ordering for BFS-like algorithms on disk-based graph engines. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*. 459–474.

[26] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. In *Proceedings of the Machine Learning and Systems (MLSys)*. Palo Alto, CA, USA.

[27] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Predicting positive and negative links in online social networks. In *Proceedings of the 19th international conference on World wide web (WWW)*. 641–650.

[28] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits. *CoRR* abs/1603.06560 (2016). arXiv:1603.06560

[29] 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 *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*. 443–458.

[30] Christian Mayer, Muhammad Adnan Tariq, Ruben Mayer, and Kurt Rothermel. 2018. Graph: Traffic-aware graph processing. *IEEE Transactions on Parallel and Distributed Systems* 29, 6 (2018), 1289–1302.

[31] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. 2017. Convergence analysis of distributed stochastic gradient descent with shuffling. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*.

[32] MindSpore Team. 2020. Welcome to the Model Zoo for MindSpore. https://github.com/mindspore-ai/mindspore/tree/master/model_zoo/official. Accessed, Sep-2020.

[33] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proceedings of the Internet Measurement Conference (IMC)*.

[34] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary physics* 46, 5 (2005), 323–351.

[35] NVIDIA Corporation. 2007. Nvidia nvprof. https://docs.nvidia.com/cuda/profiler-users-guide/index.html. accessed, Jan-2020.

[36] NVIDIA Corporation. 2019. NVIDIA collective communications library (NCCL). https://developer.nvidia.com/nccl. [Online; accessed: 20-December-2019].

[37] OpenMP Team. 2012. The OpenMP API specification for parallel programming. https://www.openmp.org. accessed, May-2020.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the Advances in Neural Information Processing Systems 32 (NeurIPS)*. 8024–8035.

[39] Josep M Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. 2011. The little engine (s) that could: scaling online social networks. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 375–386.

[40] PyTorch Team. 2017. PyTorch Profiler. https://pytorch.org/tutorials/recipes/recipes/profiler.html. accessed, May-2020.

[41] Benjamin Recht and Christopher Ré. 2012. Toward a noncommutative arithmetic-geometric mean inequality: conjectures, case-studies, and consequences. In *Proceedings of the Conference on Learning Theory (COLT)*. 11–1.

[42] SciPy Team. 2008. SciPy: Sparse Matrix CSC format. http://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csc_matrix.html. accessed, May-2020.

[43] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine* 29, 3 (2008), 93–93.

[44] Alok Tripathy, Katherine Yelick, and Aydin Buluc. 2020. Reducing Communication in Graph Neural Network Training. *arXiv preprint arXiv:2005.03300* (2020).

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

[46] Shiv Verma, Luke M Leslie, Yosub Shin, and Indranil Gupta. 2017. An experimental comparison of partitioning strategies in distributed graph processing. In *Proceedings of the VLDB Endowment*. VLDB Endowment, 493–504.

[47] 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*.

[48] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1813–1828.

[49] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, 595–610.

[50] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems (KAIS)* 42, 1 (2015), 181–213.

[51] Yuhao Zhang, Peng Qi, and Christopher D Manning. 2018. Graph Convolution over Pruned Dependency Trees Improves Relation Extraction. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2205–2215.

[52] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2018. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434* (2018).

[53] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. In *Proceedings of the VLDB Endowment*, Vol. 12. VLDB Endowment, 2094–2105. https://doi.org/10.14778/3352063.3352127