TurboGNN: Improving the End-to-end Performance for Sampling-based GNN Training on GPUs

Wenchao Wu, Xuanhua Shi, *Senior Member, IEEE,* Ligang He, *Member, IEEE,* Hai Jin, *Fellow, IEEE,*

Abstract—*Graph Neural Networks* (GNN) have evolved as powerful models for graph representation learning. Sampling-based training methods have been introduced to train large graphs without compromising accuracy. However, it is challenging for the existing GNN systems to effectively utilize multi-core accelerators, especially GPUs, due to a large number of atomic operations and unbalanced workload originating from the serial execution of multiple GNN processing stages. In this paper, we propose a combination of optimization techniques to accelerate the end-to-end performance of the sampling-based GNN training process. Specifically, we propose an adaptive share memory-based sampling technique and a degree-guided thread block scheduling strategy to optimize the graph sampling. Further, based on the observations of resource demand in different training stages, we propose an asynchronous pipeline-based scheduling method, which accelerates the GNN training by decoupling different training stages into a pipeline and therefore improves the GPU resource utilization significantly. The experimental results show that compared with the existing work, the proposed methods can achieve up to 5.6X performance speedup in the end-to-end performance.

Index Terms—Graph Neural Networks, parallelism optimization, scheduling, GPU

1 INTRODUCTION

Because of the ability to learn both the structure and attributes of the graphs at the same time, *Graph neural networks* (GNN) is widely used in many fields such as node classification and link prediction in recommendation systems [1], social networks [2], biomedical science [3], knowledge graph [4], [5]. Since training GNN is a very time- and resource-consuming task [6], general-purpose *graphics processing units* (GPUs) are often used to accelerate the training process. Several general GNN learning frameworks have been developed, such as [7], [8], [9].

The core computations in GNN training come from the continuous information gathering from neighboring vertices and updating the vertices' feature vectors through a neural network.Multiple such layers can be stacked to aggregate multiple hop messages. Usually, a GNN has 2-3 layers. Challenges still remain to train GNN efficiently. First, many real-world social graphs are of huge size with rich attribute information. For example, ogbn-paper100M [10] has 111M vertexes, 1.6B edges with 53GB of vertex feature, while the memory capacity of commercially available GPUs is usually tens of GB, (e.g., 16GB for NVIDIA P100 GPU). Second, the multi-layer stacking structure like a deep learning network increases the memory footprint of the full graph-based training. To solve the scalability problem, the method of *sampling-based training* is proposed [11]. In the sampling-based training, subgraphs are extracted by starting from the training vertexes and continuously sampling the neighboring vertices within L-hops. A fixed number of neighbors are selected (sampled) in each layer based on specific sampling strategies such as random, weighted, and random walk. The sampling can reduce both computations and memory requirements in one iteration. Finally, all the training vertices are processed in mini-batches, and the model parameters are updated iteratively until the model converges.

In the sampling-based training, the existing systems such as DGL [7], PYG [9]and PinSage [1] adopt the hybrid CPU+GPU mode. In this mode, the whole training process is divided into three stages: i) subgraphs sampling, ii) feature extraction and transmission, and iii) the actual GNN training. First, the structure and feature data of the graph are stored in the CPU memory. The CPU is responsible for sampling the graph and generating subgraphs for training. Next, the sampled subgraphs and the collected features are transferred to the GPU, where the GNN training is performed. As the CPU memory capacity is usually much larger than that of GPUs, this mode can support the training of huge graphs in the single- or multi-GPU setting. However, the feature transmission and the CPU-based sampling may become the performance bottleneck due to the low PCIe bandwidth but fast GPU training, which leads to low GPU utilization since the GPU may have to wait for sampled subgraphs.

To address the data transfer bottleneck, the GPU-based feature caching [12] and the UMA (*unified virtual memory*)-based feature fetching techniques [13] are proposed. The feature cache policy takes advantage of different probabilities that a vertex is sampled. PaGraph [12] proposes the degree-

- *Wenchao Wu, Xuanhua Shi, and Hai Jin are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {wcwu, xhshi, hjin}@hust.edu.cn.*
- *Ligang He is with the Department of Computer Science, University of Warwick, Coventry, CV4 7AL, United Kingdom. E-mail:Ligang.He@warwick.ac.uk.*

Xuanhua Shi is the corresponding author (e-mail: xhshi@hust.edu.cn)

guided feature caching policy to cache the nodes with a high out-degree in the GPU memory in advance, assuming those vertices are more likely to be frequently sampled. Pytorch-direct [13] proposed the UMA-based, GPU-oriented communication kernel to fetch the features, which can improve the transfer efficiency significantly. To relieve the bottleneck of CPU sampling, the previous work [7] proposed a GPU-based sampling algorithm based on the fact that the topological data of a graph only account for a small portion of the entire graph data (comparing to the feature data of a graph). Therefore the topological data can be easily transferred to the GPU, which then utilizes its vast parallel power to perform fast sampling.

The optimization techniques discussed above aim to address the problems in the first two stages of the entire GNN training process, i.e., sub-sampling and data transfer. However, we found that although the subgraph sampling is moved from CPU to GPU in the previous work, there is still big room to improve the efficiency of the GPU sampling (as an essential stage of the GNN training, the GPU sampling stage currently occupies 45% of the processing time on average in an iteration). Moreover, none of the previous works attempts to improve the end-to-end performance of the GNN training which is affected by all the processing stages including sampling, feature extraction, and training. Our studies show that there exist inefficient operation scheduling and executions, which affect the end-to-end performance. Specifically, multiple threads often have to save the sampling results to the same location in global memory. To avoid data race between threads, the existing GPU sampling algorithms run a large number of atomic operations, which is very inefficient and expensive [28]. In addition, it does not take into account the unbalanced distribution in the degrees of the sampled vertices, which may result in low bandwidth utilization and severe load imbalance. Moreover, the existing methods ignore the difference in GPU resource requirements between the GPU-based subgraph sampling stage and the actual GNN training stages. They adopt a *serial execution* approach to scheduling the relevant operations, which we found may lead to long end-to-end training time and low resource utilization.

To address the inefficiency caused by the atomic operations during subgraph sampling, we propose an optimization method to transfer the atomic operations in global memory to shared memory. However, when the atomic operations are performed in shared memory, a consequent problem arises: the shared memory contention may increase, which in turn compromises the concurrency degree of thread blocks since the shared memory is shared by the thread blocks running in an SM (Streaming Multiprocessor). To address this problem, we propose an adaptive, degree-guided policy when applying the optimization method. Namely, as the GNN training progresses, only the atomic operations on the vertices with high degrees, which are regarded as being more "valuable", are transferred to the shared memory. This novel policy improves the performance without compromising the concurrency.

Further, we propose the new scheduling policies to address the problem of resource demand imbalance between the graph sampling stage and the GNN training stage. The scheduling optimization is two-fold. On the one hand, we propose a degree-based task assignment policy to run thread blocks. On the other hand, we propose a novel asynchronous, pipeline-based GNN training method based on our observations of the different but complementary nature of resource demands in different GNN training stages and also workload imbalance within the same stage. The proposed asynchronous, pipeline-based scheduling method can fully overlap the executions of different operations (no matter from different stages or the same stage) by organizing them into a pipeline-based execution, which improves the GPU resource utilization significantly. In summary, there are the following contributions in this paper.

First, we find the inefficiency in the existing GPU-based sampling method due to the high cost of atomic operations in global memory and the workload imbalance between thread blocks. Subsequently, we find the difference in GPU resource demand between the GPU-based sampling stage and the GNN training stage. The serial scheduling adopted by the existing methods does not take into account such a difference and may cause low GPU resource utilization.

Second, to address the inefficiency of memory access in the existing GPU-based subgraph sampling, we develop an optimization method to reduce the atomic operations in global memory by adaptively selecting the atomic operations on high-degree nodes and placing them in the shared memory. This way, the inefficiency of atomic operations is mitigated while maintaining a high degree of thread block concurrency.

Third, to address the problem of workload imbalance, we develop the scheduling policies to i) optimize the task assignment for thread blocks for the sampling kernel and ii) organize the operations in different stages into a pipeline and run the operations simultaneously for the whole iteration, which improves the overall resource utilization significantly.

Finally, we implement the above optimization techniques into DGL and develop an efficient GNN learning framework called TurboGNN. We have conducted extensive experiments. The results show that TurboGNN can improve the end-to-end training performance by up to 5.6x.

The rest of this paper is organized as follows. The background information related to this work is presented in Section 2. The proposed optimization techniques as well as the architecture and implementation of TurboGNN are detailed in Section 3. The experimental results of TurboGNN are discussed in Section 4. Related work is discussed in Section 5. Finally, the conclusions and future work are presented in Section 6.

2 BACKGROUND

2.1 GNN

For a graph $G = (V, E)$, $v \in V$ represents a vertex (node) in the graph G with the feature set denoted by f_v, and the edge between two vertices represents the relationship between them. A GNN model learns the high-dimensional feature representation of each node by gathering the information from its neighboring nodes in the previous layer and updating its feature vector following the topology of the deep network (such as Multilayer Perceptrons) iteratively. The core computations in a GNN layer can be divided into two stages: message aggregation and feature transformation, which can be modeled by formula 1, where h_v^k is the feature vector of vertex v in layer k and $\mathcal{N}(v)$ is the set of neighboring vertices of vertex v. The difference between the GNN models mainly lies in the aggregation and the updating functions (e.g., AGGREGATE and UPDATE in formula 1). Many such layers can be stacked to enhance the distance of information extraction and model accuracy.

$$\begin{aligned} a_v^{(k)} &= \text{AGGREGATE}^{(k)}\left(\left\{h_u^{(k-1)} \mid u \in \mathcal{N}(v)\right\}\right) \\ h_v^{(k)} &= \text{UPDATE}^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right) \end{aligned} \tag{1}$$

2.2 Sampling-based training

In the layer-stacked GNN training model, processing every node in a graph is not viable for large graphs due to the large computation and memory footprint. Also, not every node in a graph is labeled for computing the loss and the gradient and updating the model parameters. Inspired by mini-batch-based training in deep learning, sampling-based training is introduced [11]. At each iteration, the nodes in a mini-batch are randomly shuffled and selected as seed nodes in training. Starting from the seed nodes, a fixed number of neighboring nodes are sampled from all neighbors. This process iterates for K times in a K-layer GNN. In each iteration, a subgraph is generated (e.g., bipartite graph blocks in DGL). A subgraph consists of the destination vertices, which are the source vertices from the last iteration, and their neighbors that are sampled (the sampled neighbors become the destination vertices of the subgraph in the next iteration). This way, each vertex has the same number of edges in the subgraphs, which reduces the computation complexity in GNN training and improves the regularity of the message aggregation for the subgraphs. It has been shown that the sampling-based methods can achieve accuracy competitive with the training of the full graph [14], [15], [16].

In the sampling-based training, the hybrid CPU+GPU computation mode is widely adopted in previous GNN systems [1], [7], [9]. In this mode, an iteration in the GNN training can be further divided into three stages: i) CPU sampling, ii) subgraph and feature transfer; iii) GNN training. The time spent by the CPU in sampling the subgraphs and in transferring the high-dimensional vertex features dominates the entire GNN training process, which forms a severe performance bottleneck [12], [13]. Usually, a GNN uses a shallow network structure with less than four layers. Given the limited bandwidth of PCIe (usually less than 16GB/s) and the large amount of feature data that needs to be transferred (e.g., 53GB for the ogbn-paer100M graph), the huge communication cost can hardly be hidden by GNN computations in such shallow network structures.

To reduce the communication cost, the GPU-based caching technique is proposed in a system called PaGraph [12]. PaGraph caches as many features of the high-degree vertices as possible in the GPU memory.The caching technique has been shown to be effective, especially on large graphs where the node degree follows the power-law distribution [17].

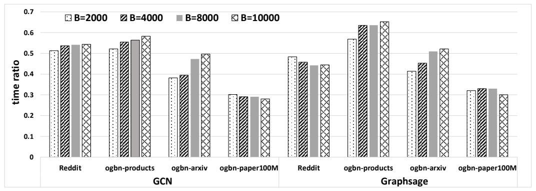


Fig. 1: The proportion of sampling time in the overall time of an iteration. The X-axis is different models, batch size (denoted by B), and datasets, and the Y-axis is the ratio of the GPU sampling time to the overall iteration time.

2.3 GPU based sampling

Since the CPU-based subgraph sampling (the supply of the subgraphs) cannot keep up with the consumption of subgraphs in GPU training, especially in multiple GPU settings, the GPU-based sampling is introduced in the GNN systems such as in the latest version of DGL [7]. The data describing the graph structure will be loaded into the GPU memory before training. In each iteration, a mini-batch is sampled on the GPU. Next, the CPU extracts the features of the sampled vertices and transfers them to the GPU memory. Finally, the GPU trains a GNN model with the sampled vertices and features. Usually, the graph structure data is much less than the vertex feature data, and therefore can be easily stored in the GPU memory, while the feature data are cached in the rest of the GPU memory as much as possible.

3 METHODOLOGY

As discussed in the introduction, the end-to-end GNN training process include three stages: sampling, feature transfer and training. In this section, we first elaborate on the problems in the end-to-end training process, which are also the motivation of this work. Then we propose a combination of optimization methods for improving the end-to-end performance of GNN training.

3.1 Motivation

To demonstrate the problems in the existing GNN training systems, we integrate the GPU-based sampling and graph caching technique proposed by PaGraph [12] into DGL(V0.8.1) [7] to build a state-of-the-art system (called DGL-SOTA in this paper).

We conducted the experiments with two typical GNN networks (GCN [18] and GraphSage [11]) on four datasets (Reddit [19], ogbn-products [10], ognb-arxiv [10], ogbn-paper100M [10]). More detailed configurations are presented in section 4. Though DGL-SOTA achieves significant performance improvement compared with DGL equipped with GPU-based sampling when the feature cannot be fully stored in GPU memory (e.g., an average of 2.5X speedup on the above two models and the ogbn-paper100M dataset), we still found the following issues.

Inefficient GPU-based sampling stage due to load imbalance and a large number of global atomic operations. Fig. 1 shows the experimental results with regard to GPU-based sampling. It can be seen that the sampling time occupies 28%-65% of the whole iteration time. We examined

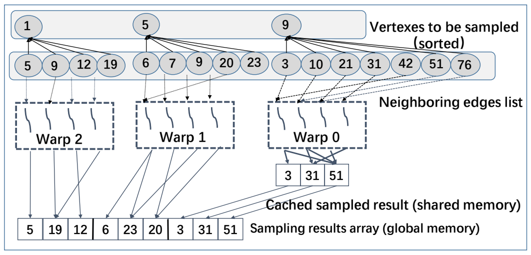

Vertexes to be sampled (sorted)
Neighboring edges list
Warp 2
Warp 1
Warp 0
Cached sampled result (shared memory)
Sampling results array (global memory)

Fig. 2: The sampling process

the sampling implementation on DGL(V0.8.1) in detail. It is based on the Reservoir algorithm [20] to select the k-hop random neighboring vertices. The sampling process is illustrated in Fig. 2. The sampling task for each vertex (such as vertices 1, 5 and 9) is assigned to a GPU warp to increase the efficiency of the memory access to the edge lists and reduce the thread divergence. The 32 threads in a warp sample a fixed number of edges (the number is 3 in Fig. 2) for the sampled vertex. The complexity of the sampling task for each warp is related to the degree of the sampled vertex being assigned, which may cause a severe load imbalance when a power-law graph is processed. Moreover, the threads in a warp need to concurrently modify a global array that stores the sampling results. This introduces a large number of global atomic operations in order to maintain the data consistency, which is expensive and inefficient [28]. As the NVIDIA Visual Profiler [21] shows in our experiments, the average memory bandwidth is only 40% of the peak value. Therefore, it is critical to avoid the overhead of global atomic operators and achieve workload balance in a sampling kernel. We propose three optimization techniques in section 3.2 to address these issues.

Low GPU resource utilization in the end-to-end training process due to serial executions of the three GNN stages in the same iteration. We used the NVIDIA Visual Profiler [21] to obtain the resource usage of GPUs in a whole training epoch. The results show that the GPU resource utilization is low (the detailed experimental results are presented in section 4). For example, in the experiments with the GraphSage on the ogbn-paper100M dataset with a batch size of 4000, the average GPU utilization is no greater than 53%, while on the ogbn-products dataset it is less than 63%. Note that for the ogbn-products dataset, the entire graph and features can be cached in the GPU memory, which eliminates the data transferring cost completely. Even so, the GPU utilization is still unsatisfactory. Increasing the batch size cannot solve this problem, but may even reduce the convergence speed [22]. Therefore, we try to tackle this issue from another direction, which is to optimize the scheduling policies of GPU operations to achieve the objective of improving the average GPU utilization in the whole GNN training process without compromising the training accuracy. In particular, we propose an asynchronous pipeline scheme to schedule and execute multiple stages concurrently (in section 3.3).

In summary, in this paper we aim to improve the performance of the end-to-end GNN training process, which includes the sampling stage, feature transfer stage and the training stage. We identified two causes in the existing GNN systems that affect the end-to-end performance: i) inefficient sampling stage itself due to expensive atomic operations and workload imbalance among threads, and ii) the inefficient serial executions of the three stages. To improve the performance of the sampling stage, we propose a share memory-based data placement policy to address the problem brought by atomic operations. However, we found that using too much shared memory will affect the concurrency of threads. To address this issue, we further propose the degree-guided adaptive shared memory optimization technique. Moreover, to address the problem of workload imbalance among threads, we propose the degree-guided thread block scheduling method. To overcome the inefficiency caused by the serial executions of the three stages, we propose an asynchronous pipeline-based scheduling method to process different stages in different iterations concurrently. In the following sections, we will present each optimization technique in detail.

3.2 Adaptive Shared Memory-based Sampling

Share Memory-based Data Placement. The existing sampling algorithm incurs a large number of atomic operators in the global memory to avoid the data race among threads. Specifically, to prevent the threads from producing repeated sampling results, the threads in a warp first generate a random number modulo edge index to get a position index, num, if num is less than the *fanout* (the number of neighbors to be sampled for each vertex), then the edge is chosen to insert to the result array indexed by num. Since different threads in the warp may get the same index, the $AtomicMax$ function is required to ensure consistency when multiple threads write the data concurrently at the same location in global memory. However, the atomic operators in global memory are costly [28]. The number of such operations is related to the number of iterations in the innermost loop for the edges of nodes, which is further proportional to the degree of the nodes. Even worse, a power-law graph has a skewed degree distribution, and the high-degree vertices can introduce high costs of atomic operations on the global memory, which leads to low utilization of memory bandwidth and severe workload imbalance.

To reduce the amount of global atomic operations, we propose a shared memory-based data placement policy. Specifically, we use the shared memory, which is software-controlled and has much higher bandwidth and lower latency, as the cache for the threads to store the sampling results. The pseudo-code of the optimized kernel is presented in Algorithm 1. Before a warp samples a vertex, we use all the threads in a warp (thread cooperation) to write the initial sample results (lines 9-11 in Algorithm 1). This process is efficient since the threads in a warp read consecutive addresses in the global memory and write the results to the consecutive addresses in shared memory. The coalesced global access can combine multiple global memory accesses into a single memory transaction, while the consecutive writes in share memory also avoid the bank conflict. Then, a warp synchronization instruction (instead of a block synchronization instruction), $syncwarp()$, is used to ensure the data dependency of the writes from the whole warp with the minimal synchronization overhead (line 12). Next, the warp travels all the edges for the sampled vertex iteratively and uses the $AtomicMax$ function to concurrently write the results to the shared memory (lines 16-18). Note that an atomic operator in SM is much cheaper than that in global memory [28]. Last, the sampling results are written back to the consecutive addresses in global memory by the threads in a warp, which is once again the coalesced access to global memory (lines 23-28).

Degree-guided Adaptive Shared Memory Optimizations. Usually, exploiting the shared memory in GPU can improve the data access efficiency, especially when the data have to be frequently reused. However, the shared memory is very limited (e.g., 3.6MB in NVIDIA P100 while the average memory demand of training the ogbn-products dataset is as high as 48MB when batchsize=200000, fanout=30). Moreover, allocating large shared memory space to a thread block will affect the concurrency degree of the thread blocks, which has been demonstrated by the experiments in 4. Therefore, simply transferring the atomic operations from the global memory to the shared memory may not bring significant performance gain if the fanout is large (the shared memory demand is proportional to the value of the fanout). As mentioned before, the number of atomic operators is related to the degree of the sampled vertex. This means that the benefit brought by caching the sampling results in the shared memory is determined by the degree of a vertex. Since a real graph usually follows the power-law distribution, i.e., only a small number of vertices in the graph have very high degrees. Therefore, we only need to cache a few high-degree vertices, which not only reduces the global atomic operations effectively but also reduces the consumption of the scarce shared memory resource.

Inspired by this idea, we propose a degree-guided, adaptive method to optimize the usage of shared memory. In this method, we first sort the sampling vertices by their degree and then divide the vertices into two partitions based on a threshold (denoted by $D_{threshold}$, the value of which is determined by the GPU hardware resource, the graph structure, and the number of samples). One partition (called high degree partition) holds the vertices whose degree is equal to or greater than $D_{threshold}$, while the other partition holds the rest vertices. The sampling results obtained for the vertices in the high-degree partition are cached in the shared memory, while the results for other vertices are written using the native algorithm (written into the global memory). Finally, the sampled results from the two partitions are merged. Both partitions use a thread block scheduling strategy, which is to be presented next. As illustrated in Fig. 2, the sampling process for vertex 9 from the high-degree partition is conducted in the shared memory, while it is performed in the global memory for vertices 1 and 5 from the low-degree partition.

Degree-guided Thread Block Scheduling. In addition to the aforementioned optimization for memory access, we design an efficient task assignment policy dedicated to the hardware scheduling mechanism of GPU. Because real graphs usually obey the power-law distribution, the sampling vertices produced by the random shuffling of training sets will also have such a distribution. The latest sampling algorithm in DGL (V0.8.1) assigns the consecutive sampled vertices to the consecutive warps without considering the degree of the vertices. Since the complexity of sampling each vertex is proportional to its degree, we found that this algorithm caused a severe load imbalance among the warps and the thread blocks. In view of this, we propose a task reassignment strategy to balance the workload for each warp in a thread block. It can also prioritize the execution of thread blocks with high workloads and overlap the execution of high-workload thread blocks with low-workload ones.

Specifically, the proposed task reassignment strategy first sorts the sampled vertices in decreasing order by their degree. If the number of sampled vertices is small (e.g., for low layers in a GNN model), the number of thread blocks required is also small. The sorted vertices are then assigned to each thread block in an interleaved way. For example, assuming that there are m thread blocks and $m \times k$ vertices, Vertices 1 to m are assigned to thread blocks 1 to m; then vertices $m+1$ to $2m$ are assigned to thread blocks m to 1. The assignment goes on in an interleaved manner until all vertices are assigned to the thread blocks. This interleaved assignment can help balance the workload among thread blocks.

When the number of the sampled vertices is large (for the higher layers in GNN), a block of consecutive vertices is assigned to a thread block in a round-robin manner. In a vertex block, the vertices are assigned to the warps in a thread block in the interleaved manner described above. In this way, the workload within the same thread block is balanced among multiple warps (within a vertex block, the degree of vertices changes more evenly, and the interleaved vertices assignment makes it easier to achieve load balance among warps). Although the workload may not be balanced among different thread blocks, this assignment is still effective for two reasons. First, the workload of each warp in a thread block is balanced due to the interleaved assignment. Thus, the warps can complete the task at almost the same time. Consequently, the entire block can retire timely, releasing the occupied resources to other thread blocks. Second, Our experiments show that (and also noted by [36]) the hardware scheduling unit [1] appears to schedule the thread blocks in the order of the thread block ID (the thread blocks with lower IDs are scheduled first). Our task assignment policy sorts the vertices in the descending order of their degree and assigns the processing of vertices to each thread block in the descending order of vertex degree. The thread blocks with low IDs get the heavy-load task (high-degree vertexes). This means that the priority is given to the heavy-loaded thread blocks inexplicitly and they will be scheduled to run first. Moreover, since the number of the sampled vertices is large, a large number of thread blocks are used to run the above vertex blocks. Consequently, even if a heavy-loaded thread block has not completed the task, GPU can always find other thread blocks to run. As the result, other more lightly-loaded thread blocks are more likely to run concurrently with the heavy-load thread blocks, which can improve GPU resource utilization.

The pseudo-code for the entire sampling scheduling

1. The schedule of the thread blocks can hardly be changed via software, but is controlled by hardware, and little information is released to the public.

procedure is presented in Algorithm 2. The sampled vertices are first sorted in decreasing order by their degrees (line 1). The vertices are then divided into two partitions: the high-degree and the low-degree partitions (line 2). For each partition, if the number of thread blocks needed for processing the partition is small, the interleaved vertices assignment is applied for each thread block (lines 4-6). Otherwise, the blocks of consecutive vertices are assigned to the thread blocks while within each vertices block, the vertices are assigned to the warps in the interleaved manner (lines 7-14). Next, the high-degree partition uses the shared memory-based sampling kernel (line 16) while the low-degree partition uses a native sampling kernel (line 17). Finally, the results obtained from the two partitions are merged (line 18).

algorithm 1 Atomic operator optimized sampling algorithm

```
Input: $G$: the graph CSR, $N$: the number of vertices to
    be sampled, $T$: the number of tasks per thread block
    $fanout$: the number of neighbors to be sampled for each
    vertex.
Ouput: $result$: the sampled graph.
 1: extern __shared__$int$sm_sample_result[]
 2: startV← blockIdx.x*T+threadIdx.y
 3: endV← min((blockIdx.x+1)*T, num_rows)
 4: index←threadIdx.y*fanout
 5: while $startV < endV$ do
 6:     $degree$ ← compute degree of $startV$.
 7:     if $degree > fanout$ then
 8:         idx ← threadIdx.x
 9:         for $idx = threadIdx.x$; idx < fanout; idx+=32 do
10:             sm_sample_result[index+idx] = idx
11:         end for
12:         syncwarp()
13:         idx ← fanout+threadIdx.x
14:         while $idx < deg$ do
15:             generate rand number $number$
16:             if $num < fanout$ then
17:                 AtomicMax(sm_sample_result+index+num,
    idx)
18:             end if
19:             idx+=32
20:         end while
21:         syncwarp()
22:         idx← threadIdx.x
23:         while $idx < fanout$ do
24:             position=sm_sample_result[idx+index]+G.inptr[startV]
25:             result.rows[index+idx] = row
26:             result.cols[index+idx] = G.index[position];
27:             idx += 32
28:         end while
29:     end if
30:     startV += WarpPerBlock
31: end while
```

3.3 Asynchronous pipeline-based scheduling

The GPU-based sampling training can be divided into three stages. We first analyze the resource requirements of different stages, which motivate us to design the asynchronous pipeline-based scheduling technique.

algorithm 2 The whole sampling process

```
Input: $G$: the graph to be sampled; $fanout$: the number of
    neighbor to be sampled; $vertex$: the array of sampled
    vertices; $N$: the number of sampled vertices; $D_{thread}$:
    degree threshold;
Ouput: $result\_graph$: the sampled subgraph.
 1: sorting the vertex by degree decreasing
 2: divide vertex sorted array int two array $P1$ and $P2$ with
    $D_{thread}$.
 3: for (p∈ p1,p2) do
 4:     if (size(p)<$N_{threshold}$) then
 5:         Map interleaved vertex to different thread blocks
 6:     end if
 7:     if (size(P)>=$N_{threshold}$) then
 8:         Map consecutive vertex to different thread blocks
 9:         for $block \in threadblocks$ do
10:             for $warp \in block$ do
11:                 map interleaved vertex for $warp$
12:             end for
13:         end for
14:     end if
15: end for
16: $sample1$←apply memory-optimized kernel to $p1$
17: $sample2$←apply original kernel to $p2$
18: result_graph←merge($sample1$, $sample2$)
```

GPU sampling stage. The sampling stage mainly traverses the edges. Like traditional graph computation tasks (e.g., PageRank, BFS), the sampling stage has a low computation/memory ratio. Therefore the memory bandwidth is the major performance bottleneck.

Feature extraction and transfer stage. This stage is conducted on the CPU. In this stage, the feature rows missing in the GPU cache are collected from the CPU's main memory. Those features are scattered over different locations in the feature array. The collected features are then transferred to the GPU memory through PCIe. Usually, this stage is memory- and communication-hungry [13]. The workloads (i.e., the features that are missing in the cache and have to be collected) depend on the size of the GPU cache, cache policy [12], and the structural properties (e.g., degree distribution) of the graph. If all the feature data can be cached in the GPU or the cache mechanism works well for the graph structure and the access pattern, this stage incurs low cost or even no cost at all if all the feature data can be cached in GPU.

Model training stages. Unlike full-graph-based training, whose computations in the training are dominated by sparse matrix multiplication, the core kernels for sampling-based training are dense matrix multiplication.These kernels have been actively optimized in the GPU libraries implemented by the hardware manufacturers (such as CuBlas [23], CuDNN [24]) and the academic community (such as [25]). The core mechanisms are data tiling, share memory reuse, register optimization, and instruction scheduling [24], [25]. Based on these optimizations, the dense matrix multiplication is computation hungry due to its high computational intensity, very friendly data access pattern and large data reuse in the cache (such as share memory and registers) [25]. Also, it can be effectively accelerated by the Tensor

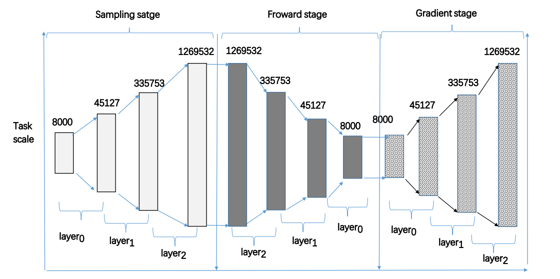


Fig. 3: Workload distribution between different stages and inside a stage in an iteration (model: GCN, dataset: ogbn-products, batch size: 8000). The number above the bar stands for the number of source or destination vertices in a subgraph in a layer.

cores from the new generation GPU.

Based on the above analysis, we made the following observations. On the one hand, different stages have markedly different resource requirements, and the resource demands between different stages are often complementary. On the other hand, the computation inside a stage is not balanced between different layers (steps), which is illustrated in Fig. 3. As mentioned in Subsection 2.2, K subgraphs are sampled for a K-layer GNN over iterations. The number of vertices in the subgraphs sampled in each iteration (hence the workload) increases exponentially. The increase rate is determined by the fanout. Namely, the number of the sampled vertices for the i-th layer in a mini-batch is nearly *fanout* times that of the $(i-1)$-th layer. These K subgraphs are the input of the next forward propagation stage. But the K subgraphs are processed in the forward stage in the reverse order. Namely, the subgraph sampled in the last iteration of the sampling stage will be processed first by the first layer of the GNN model. Therefore, the workload between different layers in the forward stage decreases exponentially. Moreover, the workloads of different layers in a stage have to be run strictly in sequence due to data dependency. Therefore, GPU utilization tends to increase or decrease greatly in a stage.

Therefore, we propose an asynchronous pipeline-based scheduling technique to accelerate the execution process. We decouple different stages and run them in different processes. These processes communicate via the queues, and the processes and the queues form a pipeline. This way, different processes can execute different stages concurrently in the same GPU. More specifically, the *sampling process* fetches a mini-batch of vertices as the seed vertices, conducts the sampling in the GPU, and pushes the sampled subgraphs into the *sampling queue*. At the same time, the *feature extraction process* fetches the subgraphs from the sampling queue, performs the feature extraction, and pushes the extracted feature into the GPU. It then puts the subgraphs that are ready to be trained in the *training queue*. The *training process* obtains the input from the training queue and executes the forward and backward computation of the GNN model.

The rationale behind this asynchronous pipeline-based scheduling is explained as follows. First, GPUs are now equipped with a large number of computing units and various on-chip memory units, even heterogeneous computing units (such as tensor cores). Since different stages have different resource requirements (such as cores, shared memory, and memory bandwidth) and are often complementary, the mechanism of asynchronous pipeline-based scheduling can improve the degree of parallelism and increase overall resource utilization. Second, due to the workload imbalance between different layers in a stage and that the layers in a stage must be executed serially, the low-workload layers from either the sampling or training stage can hardly utilize the GPU resource fully and resulting in low utilization. With our asynchronous pipeline-based scheduling, the layers from different stages can be executed concurrently, which reduces the load imbalance and consequently increases GPU utilization.

It is worth mentioning that in asynchronous pipeline-based scheduling, different stages are run in different processes. The traditional way of implementing concurrent executions on GPU is to use multi-threading with multiple streams and place each stage in an individual stream. However, Python multi-threading is not efficient. This is because that Python implements the global interpreter lock (GIL) and the GPU events such as dynamic memory allocation, which is very common when the subgraphs are generated in the sampling stage, can cause implicit synchronization among multiple streams. Therefore, we chose to use the multi-process approach.

However, the consequent challenge of adopting the multi-process approach is the high data serialization cost in inter-process communication. For example, the sending process (such as the sampling process) needs to serialize the information in the class instances of the generated subgraph, which includes the metadata of vertices and edges and the topology data of the subgraphs (e.g., the edge list of a subgraph), to the byte sequence and sends it to the receiving process (such as the feature extraction process). The receiving process then needs to deserialize the received byte sequence into the classes. We adopt the following method to reduce the data serialization cost. Specifically, we find that a large proportion of the data for a subgraph comes from the edge-list tensors. The edge list tensors are read-only and have a very simple format. Therefore, we decouple this part of the data and exploit the *direct tensor sharing* in Pytorch, which calls $torch.multiple.simpleQueue$, to share the edge list directly among the sampling and training processes. The $simpleQueue$ can share data in the GPU global memory so that different processes can access it directly and avoid repeated data copying. The rest of the subgraph data (such as the metadata of vertices) is still communicated through the queue mechanism, which involves data serialization (such as the sampling queue when the sampling process communicates with the feature extraction process). Since the volume of the vertices metadata of a subgraph is much less than that of its edge list, most data serialization cost is avoided. Note that $torch.multiple.simplQueue$ only supports simple tensors, so we cannot use it to share the entire subgraph class among processes.

Usually, the pipeline mechanism is most efficient when the progress of each pipeline stage is relatively balanced. Therefore, another challenge in pipeline-based execution is to maintain such balance and avoid resource contention where one process occupies too much GPU resource, which

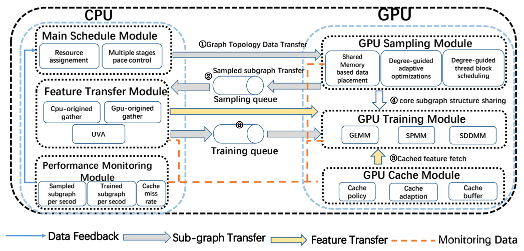


Fig. 4: The Architecture of TurboGNN

may stall other processes' progress. We address this issue by introducing a Runtime Monitoring and Dynamic Scheduling (RMDS) mechanism, which aims to balance the number of tasks run by different processes. The RMDS mechanism exploits the CUDA MPS (Multi-Process Service) [27] to set the best MPS configuration based on the historical data or the results of several initial iterations of running. At runtime, it dynamically controls the pace of different stages. For example, if the number of subgraphs in the sampling queue keeps increasing for more than a certain period (e.g., five iterations), it indicates that the training process cannot consume in time the subgraphs generated by the sampling process. The sampling process is then deemed to be running too fast. When this happens, the RMDS mechanism sets a limit to the speed at which the subgraphs are generated. The GPU sampling process will check this limit before starting a new sampling iteration. If the limit has been reached, the sampling process sleeps for a period of time. With the RMDS mechanism, all processes can maintain a steady state.

3.4 Architecture and Implementation of TurboGNN

We have implemented the proposed optimization techniques for the GPU sampling and training to DGL-SOTA and developed a highly efficient GNN training framework called TurboGNN. There are six core modules in TurboGNN, as shown in Fig.4: main scheduling module, performance monitoring module, GPU sampling module, feature transfer module, GPU training module, and GPU caching module. In the following, we present some implementation details of the six modules.

Main Schedule Module is in the master process, which spawns multiple other processes responsible for other modules such as the GPU sampling module and the GPU training module. Next, the multiple queues in the asynchronous pipeline-based scheduling are created by Python multiprocessing. Queue with the custom serializer and deserializer to parse the subgraph data between processes. Then, the main schedule module starts all the processes and queries the Performance Monitoring module to control the running pace of different processes using the RMDS mechanism proposed in the asynchronous pipeline-based scheduling.

Performance Monitoring Module. This module is run by a process and collects the values of the performance metrics such as the number of subgraphs generated by the sampling process per second, the number of subgraphs trained by the training process per second, the lengths of the queues between the processes in the asynchronous pipeline-based scheduling and the cache miss rate of the GPU caching module. Then, these metrics values are passed to the main schedule module through the message queue, which in turn controls the pace of different processes.

GPU Sampling Module. The GPU sampling module integrates our share memory-based data placement, degree-guided adaptive optimizations, and degree-guided thread block scheduling technique into the sampling kernel and performs the GPU sampling iteratively. The sampled vertices are sorted using $cub :: DeviceRadixSort$ from the NVIDIA CUB, and are divided into two partitions: high-degree partition and low-degree partition. Each partition is run in an individual kernel and stream to exploit the concurrent execution based on the degree-guided thread block scheduling. We run the two partitions in different kernels because the data about the high-degree partition need to be placed in shared memory while the data about the low-degree partition is in the global memory. We encapsulate the subgraph sampling module, which includes sorting of vertex degree, partitioning of high- and low-degree vertices, graph sampling, and merging of the sampling results from the two partitions, as a function with the same interface and parameters as in the default sampling function (i.e., $CSRRowWiseSamplingUniform()$) in DGL. This newly implemented sampling function replaces the default sampling function in DGL while other parts of the implementation in DGL remain unchanged.

Feature Transfer Module. The feature extraction module is responsible for feature collection and transferring to GPU memory. This implementation of this module is the same as that in PaGraph [12].

GPU Training Module. This module queries the training queue and conducts the training using the user-defined model. We reused the common graph aggregation operations and the optimized kernel implementation of SPMM and SDDMM from DGL [7].

GPU Caching Module. This module caches frequently accessed vertex features in the GPU memory. It first collects the memory footprints for several iterations, which include the graph data size and the workspace size in sampling and training. By subtracting the size of the collected memory footprint from the total GPU device memory, the cache size for storing the graph features is determined and the cache space is then allocated accordingly. Next, according to the cache policy, the frequently accessed features are identified and copied to the allocated cache space. The reason why we implement the GPU caching as a separate module is because this way the caching policy can be decoupled from the main execution control. The users can customize their own caching policies if they want to.

4 EVALUATION

We developed TurboGNN with all the proposed optimization methods. In this section, we compare it with DGL(V0.8.1), a popular GNN training system with the message passing model to facilitate usurers' programming. Since DGL does not support the GPU cache yet and the feature extraction dominates the time of training large graphs [12], we integrated the GPU cache optimization [12] into DGL (called DGL-SOTA) and uses it as the performance baseline for a fair comparison in this section.

4.1 Experiment Setup

Cluster configuration: We conducted our experiments on a platform equipped with two eight-core Xeon-2670 2.60GHz CPUs with 264GB memory. The platform is installed with Ubuntu 16.04, and GCC 6.5.0. We used CUDA 10.1 together with CuDNN V7.4.2 in all the experiments. All the experiments were conducted on a single NVIDIA P100 except for the experiments in Figure 10, which were performed on P100 and V100.

Workload configuration. Our experiments used six datasets (four homogeneous datasets and two heterogeneous datasets) which are popular for GNN training. (1) Reddit [19]. Reddit is a post-to-post graph and each vertex stands for a post while an edge connects two posts commented by the same users. It has 232,965 vertexes, 114,615,892 edges with an average degree of 492, and a feature dimension of 602. (2) Ogbn-products [10]. The Ogbn-products dataset represents an Amazon product co-purchasing network with 2,449,029 vertices and 61,859,140 edges. (3) Ogbn-arxiv [10]. Ogbn-arxiv is a directed citation network graph for CS papers with 169,343 vertices 1,166,243 edges, and a feature dimension of 128. (4) Ogbn-papers100M [10]. Ogbnn-papers100M is a directed citation graph with 111,059,956 vertices 1,615,685,872 edges and a feature dimension of 128. (5) BGS [29]. BGS is a geological measurements graph in Great Britain with 94,806 vertices, 672,884 edges, and 96 edge types. (6) AM [29]. AM describes the information about the artifacts in the Amsterdam Museum. It has 881,680 vertices, 5,668,682 edges, and 96 relations. We tested TurboGNN with four representative GNN models. (1) GCN [18]. It is a basic model for the GNN network with three-layer neighbor aggregation. (2) GraphSage [11]. It is an inductive GNN learning model to learn the general aggregation functions (such as sum, mean, pool, and LSTM) for different layers. (3) GAT [30]. It is the first work that introduces the attention mechanism to the aggregation process of graph convolutional networks. (4) R-GCN [31]. R-GCN is the first GCN modeCl for modeling the relational data.

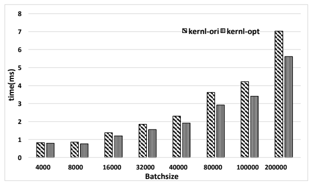
4.2 Evaluation of the Sampling Kernel

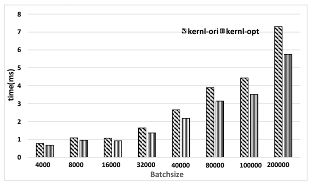
We first evaluate the optimization techniques developed for the sampling kernel (the adaptive shared-memory-based data placement policy with the thread block scheduling method; the corresponding kernel is called kernel-opt), and compare the sampling time with that of the original GPU implementation based on DGL-SOTA (called kernel-ori). The time of kernel-opt includes the extra overhead such as vertex sorting. We run the experiments with two datasets (Reddit and ogbn-products). The number of sampled vertices (called batch size) tested in the experiments is 4K, 8K, 16K, 32K, 40K, 80K, 100K and 200K, while the number of neighbors to be sampled for a vertex (i.e., *fanout*) is 10 15 and 30. These are common configurations in the default GNN networks and training process [11], [18]. All the results reported in the figures or the tables are the averages over 500 iterations. The experimental results are shown in Fig. 5. Our method (kernel-opt) outperforms kernel-ori in all the cases and achieves up to 1.35x speedup while the average usage of the memory bandwidth increases to 87% of the peak value. When the number of sampled vertices (i.e., batch size) is small, our method achieves less performance gain (e.g., 1.15X for Reddit dataset with batchsize=4000, fanout=30). This is because the small number of thread blocks cannot fully utilize the massively parallel power of the GPU while other costs such as kernel launch still exist. Since the sampling time for the higher layers (which contain more vertices) dominates the first stage execution (e.g., the batch size for the third layer on ogbn-products is 335753 while the batch size for the first layer is only 8000), performance optimization is especially needed for large batch sizes. When the batch size increases, the performance gain increases compared with kernel-ori (e.g., increases from 1.15X to 1.31X when the batch size increases from 4000 to 200000 on the Reddit dataset and fanout=30). This is because when the number of sampled vertices increases, more thread blocks are needed, and consequently kernel-ori experiences a more severe load imbalance among the thread blocks and the warps in a block. Moreover, more atomic operations are performed in kernel-ori as the number of atomic operations is positively related to the number of the sampled vertices and the degree of vertices. In this situation, our method optimizes the kernel's efficiency in accessing memory and scheduling thread blocks. Therefore, kernel-opt shows a more prominent advantage.

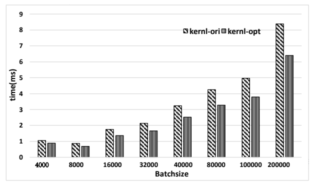
We also find that the speedup increases when fanout increases. For example, we achieve 1.28X, 1.30X, and 1.35X speedup for fanout 10, 15, and 30 respectively on ogbn-products dataset with batchsize=200000. This is because when the fanout increases, more atomic operations are required. Our optimization method uses the shared memory in GPU to cache the intermediate results in training, which effectively reduces the number of global atomic operations and therefore improves the performance.

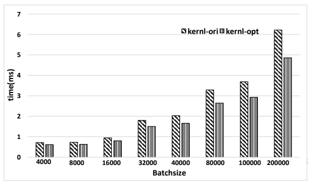
To evaluate how much each optimization policy contributes to the performance gain, we conducted experiments with different combinations of individual optimization policies. The breakdown of the contribution is shown in Fig.6. Note that the extra overheads such as vertex sorting are included in the final result. When only the policy of the degree-guided block scheduling is applied, kernel-opt achieves the 1.20X speedup compared with kernel-ori (on the Reddit dataset, batchsize=100K, fanout=30). When the naive shared memory-based caching policy, which caches everything in the shared memory without incorporating the degree-guided adaptive policy, is added, the speedup further increases to 1.22X. This further speedup is small, which suggests that simply applying the share memory-based caching policy cannot improve the performance significantly due to other factors such as block resource contention. If we further add the degree-guided adaptive policy, the speedup increases to 1.31X, which suggests the effectiveness of our adaptive policy.

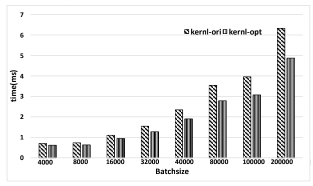
Moreover, it can be observed from the figures that different optimization policies bring different performance improvement for different configurations and graphs. For example, for ogbn-products, which exhibits more power-law distribution, the degree-guided block scheduling policy achieves more improvement than Reddit (1.27X vs. 1.18X when batchsize=80000, fanout=30). For large values of fanout, the adaptive share memory optimization in con-


(a) Dataset:Reddit, F:10


(b) Dataset:Reddit, F:15


(c) Dataset:Reddit, F:30


(d) Dataset:ogbn-products, F:10


(e) Dataset:ogbn-products, F:15

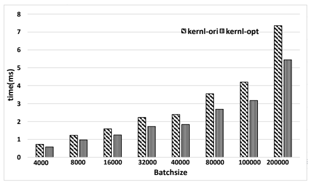
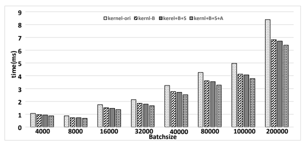

(f) Dataset:ogbn-products, F:30

Fig. 5: One sampling iteration time for different datasets, batch size, fanout (F). The X-axis is the different kernel and batch size, and the Y-axis is the time of one sampling iteration in milliseconds.


(a) Dataset=Reddit, F=30

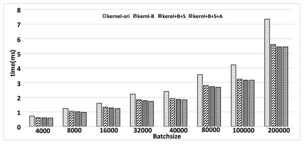
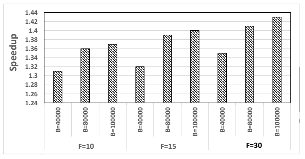

(b) Dataset=ogbn-Products, F=30

Fig. 6: Running time of one sampling iteration for different combinations of individual optimization policies on (a) the Reddit dataset and (b) ogbn-products dataset; fanout=30. ”B” stands for threads block scheduling policy, ”S” for naive shared memory optimization, and ”A” for adaptive shared memory optimization.


(a) Dataset=Reddit

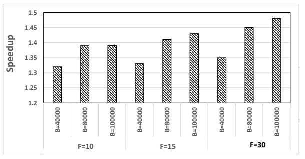

(b) Dataset=ogbn-Products

Fig. 7: The sampling kernel speedup compared to NextDoor [32] on (a) the Reddit dataset and (b) ogbn-products dataset with different batchsize and fanout

junction with thread block scheduling brings more benefits (1.28X and 1.35X speedups for fanout 10 and 30 respectively with ogbn-products and the batch size of 200000). This is because with a larger fanout, a large shared memory footprint is needed for each thread block, which may decrease the degree of concurrency. Adaptive shared memory optimization only assigns the shared memory for the set of ”the most valuable” vertices, which not only reduces the number of global atomic operations but also reduces the contention for the limited shared memory.

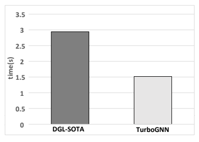
We also compared our kernel optimization methods with the latest work, NextDoor [32], in the multiple-hop sampling. The results are presented in Fig.7. Our method outperforms NextDoor [32] and achieves up to 1.48X speedup. It should be noted that NextDoor [32] targets the complicated sampling algorithm with multiple sampling paths and utilizes the transit-parallel paradigm, which is not suitable for the bipartite graph-based computation paradigm in DGL.

4.3 Evaluation with Asynchronous Pipeline-based Scheduling

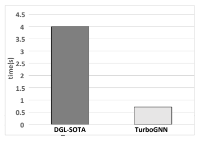
To evaluate the efficiency of asynchronous pipeline-based scheduling, we conducted experiments to compare TurboGNN with DGL-SOTA in terms of the end-to-end time of one training epoch. The experimental results are shown in Fig.8. Our proposed methods outperform DGL-SOTA in all cases and can achieve up to 5.6X speedup. The speedup on ogbn-paper100M [10] is less than that on ogbn-arxiv because all feature data of ogbn-arxiv can be cached in the GPU memory, which eliminates the bottleneck caused by the feature transfer. On ogbn-paper100M, which has a large feature transfer cost (occupying about an average of 37.1% of the entire training time of an epoch), our methods can hide the graph transfer cost with the GPU sampling, which utilizes the idle GPU resources to sample more subgraphs for the next iteration, or use the training stage to hide the graph transfer needed for next iteration.

Note that the speedup of GCN is more than that of GraphSage. For example, we obtain 2.5X and 1.9X speedup for GCN and GraphSage respectively on ogbn-arxiv with batch size 4000. This is because GCN has more layers (3 vs. 2 in GraphSage) and consequently more workload imbalance inside a processing stage. In this case, our asynchronous pipeline-based scheduling method can improve the GPU resource utilization when a layer in a stage cannot fully utilize the parallel resources in GPU. Also, the speedup on R-GCN and GAT is more than that on GraphSage models (e.g., 2.6X, 3X and 1.9X respectively for the three models). Since R-GCN has heterogeneous edges and multi-level aggregation modes, while GAT have more complicated interleaved computation from vertices and edges, the load in the sampling and training process is more diverse and unbalanced. Our pipeline scheduling can help tackle such imbalance and improve resource utilization.

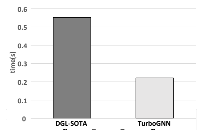
Fig.9 shows the speedup when the batch size increases on the four datasets. TurboGNN has different performances on different datasets and models. For GCN on ogbn-arxiv, the speedup is 2.3, 2,5, 2,6 and 2.9 respectively when the batch size is 2000, 4000, 8000, and 10000. ogbn-arxiv is a very sparse graph and so each stage has very small kernels that can hardly fully utilize the GPU resources. When the batch size increases, there is a more severe load imbalance between the layers within a stage. Therefore, there is more room for our asynchronous pipeline-based scheduling method to run the computations of the layers from different



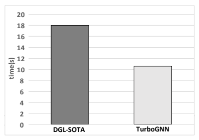
(a) GCN Reddit



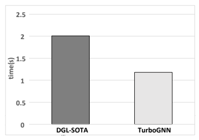
(b) GCN ogbn-prodects



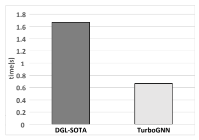
(c) GCN ogbn-arxiv



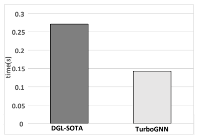
(d) GCN ogbn-paper100M



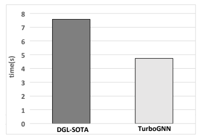
(e) GraphSage Reddit



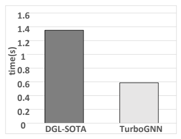
(f) GraphSage ogbn-prodects



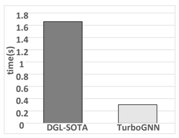
(g) GraphSage ogbn-arxiv



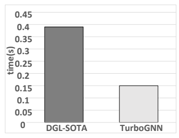
(h) GraphSage ogbn-paper100M



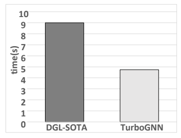
(i) GAT Reddit



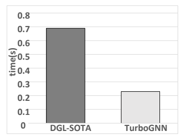
(j) GAT ogbn-prodects



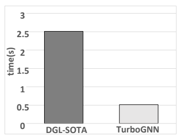
(k) GAT ogbn-arxiv



(l) GAT ogbn-paper100M



(m) R-GCN BGS

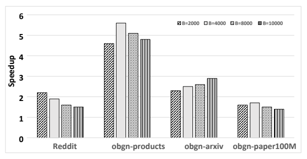


(n) R-GCN AM

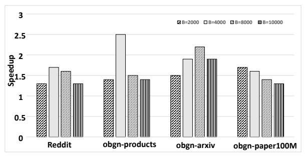
Fig. 8: Running time of one epoch of GCN, GraphSage, GAT, and R-GCN on different datasets

stages concurrently (e.g., the sampling and the training stage). For ogbn-products and ogbn-paper100M with the GCN model, the speedup increases first and then decreases. For example, the speedup is 4.6, 5.6, 5.1 and 4.8 respectively when the batch size is 2000, 4000, 8000 and 10000 on ogbn-products dataset. There is such a trend because when the batch size increases, our method can bring more benefits with an increasingly unbalanced workload (such as in ogbn-arxiv). When the batch size increases further, the workload of all the kernels in every stage increases, and there are fewer chances to run two (bigger) kernels concurrently. More resource demand and contention caused by multiple processes may even harm performance, which causes a decrease in speedup.

We also tested the effectiveness of the proposed method in different GPUs. Figure 10 shows the speedup of TurboGNN with GCN and GraphSage on two GPUs (NVIDIA P100 and V100 GPU). The two GPUs are from different generations of architecture (Pascal and Volta) and have different computation capacities. The batch size is 4000. TurboGNN achieves better performance on V100 (an average of 29% increase) since V100 is equipped with higher computation and memory capacities. It suggests our sampling



(a) GCN



(b) GraphSage

Fig. 9: Speedup of different batchsize for different datasets and models compared to DGL-SOTA

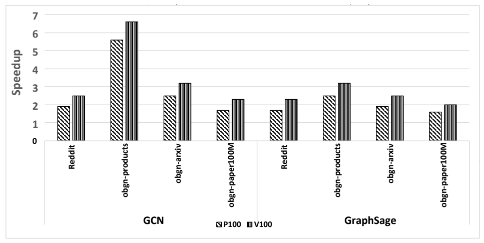


Fig. 10: Speedup of TurboGNN on different GPUs

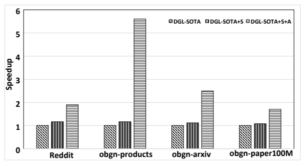
and pipeline-based scheduling optimization can better use the parallelization opportunity provided by more powerful GPU devices.

Though our methods mainly target the medium-scale graphs, on which the sampling and training can run concurrently in a single GPU and the cache works effectively, we compared our method with GNNLab [41], which mainly optimizes the cache invalidity caused by big graphs in the multi-GPU scenario. We simply extend our method to multiple GPUs with data parallelism. The results are presented in Fig.12. In a single GPU, our method achieves better performance thanks to the optimized sampling kernel and the pipeline-based scheduling. In the scenario of multiple GPUs, when the vertex feature can be fully or mostly cached by the GPU (e.g., the cases of Reddit, ogbn-product, ogbn-arxiv), our method still achieves better performance because there is less inter-GPU communication. For large graphs (e.g., ogbn-paper100M), the working space of sampling will compromise the cache efficiency due to memory shortage, and the feature transfer will impose more impact on the training time. Because of the decoupled design in GNNLab for better cache efficiency, our method is worse than GNNlab (e.g., the speedup is 0.7 for GCN). It is worth noting that the GPU memory is becoming increasely bigger in new generations of GPU (e.g., A100 is equipped with 80GB memory), our method will work for larger graphs in those GPUs. We also plan it as our future work to develop special optimization techniques for multi-GPU scenarios and larger graphs.

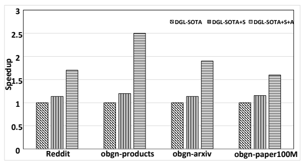
Table. 1 shows the average GPU resource utilization for training one epoch on different datasets (the table shows the results with the batch size of 4000; other batch sizes have similar results and are not presented due to space limitation). It can be found from the table that the average GPU utilization is low (e.g., 53% for ogbn-paper100M with GraphSage), which is due to the large feature extraction cost. Even when the feature extraction cost is eliminated when all the features for ogbn-products can be cached in GPU memory, the utilization is still only 63% on average. Our meth-

TABLE 1: The average GPU utilization (%) comparison

dataset	utilization (%)		dataset	utilization (%)	
	DGL-SOTA-GCN	TurboGNN-GCN		DGL-SOTA-Graphsage	TurboGNN-GraphSage
Reddit	63	95	Reddit	61	94
ogbn-products	67	97	ogbn-products	63	95
ogbn-arxiv	66	95	ogbn-arxiv	66	94
ogbn-paper100M	50	93	ogbn-paper100M	53	91



(a) GCN



(b) GraphSage

Fig. 11: (a). The speedup in the running time of one epoch for different combinations of individual optimization methods with (a) GCN and (b) GraphSage. The batch size is 4000. "S" stands for sampling optimization and "A" for asynchronous pipeline-based scheduling.

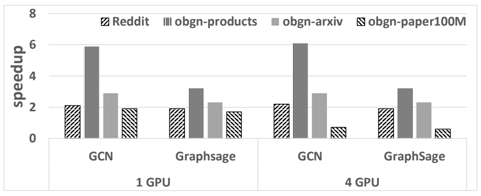


Fig. 12: The end-to-end speedup compared to GNNlab [41]

ods achieve much higher utilization (e.g., average 95% on ogbn-products for GraphSage), thanks to our asynchronous pipeline-based scheduling policy. Our scheduling policy can fully overlap imbalanced workloads from different stages. Note that since the new generation of GPUs are equipped with more cores and higher bandwidth memory (e.g., A100 has 80GB memory and 6912 FP32 cores), more features can be cached and therefore our methods are expected to show a more prominent advantage.

In order to show the contribution breakdown made by the sampling optimization and the asynchronous pipeline-based scheduling to the end-to-end performance improvement, we conducted the experiments with different optimization combinations and compared with DGL-SOTA. The results are shown in Fig. 11 (the performance of DGL-SOTA is plotted as the value of 1 in the figure). As seen from the figure, both methods contribute to performance improvement for different models. The performance improvement brought by the sampling optimization is less than that by asynchronous pipeline-based scheduling. This is because the sampling optimization only works for the sampling stage, which accounts for a part of the entire processing (e.g., the time spent by the sampling stage occupies about 54% of the time in one iteration for the Reddit dataset with the batchsize of 4000). In contrast, the asynchronous pipeline-based scheduling can have impact on the execution of multiple stages, given that the resource utilization in GPU is typically not high with the existing GNN learning frameworks to date. Although graph sampling only happens in one stage in GNN processing, the optimization is worthwhile given that sampling is an important and indispensable stage for graph learning.

4.4 Cost Evaluation

We also evaluated the cost of our methods. First, for the sampling optimization, the extra costs lie in the sorting process, which can be implemented efficiently by the NVIDIA CUB library. The sorting time is less than 2% of the kernel execution time. For example, when the node to be sampled increases from 8,000, 80,000 to 1,200,000 on the ogbn-product dataset (with 2,449,029 vertices), the percentage of the sorting time is 1.9%, 0.7%, and 0.3% respectively. The sorting overhead decreases in percentage because although the workload of both the sampling kernel and the sorting kernel increase with the batch size, the sampling kernel is more complicated than the sorting kernel and increases much faster. For bigger graphs, such as Twitter with 41.7M vertices, the sorting overhead is less than 0.1%.

Second, for the asynchronous pipeline-based scheduling, the extra costs come from the data serialization and data copying between the processes. Without our tensor-sharing methods, the percentage of the overhead is 11% on average. By efficiently sharing the topology tensor of sampled subgraphs between different processes, the percentage of the overhead decreases to only 0.7%. Even on the Twitter graph (41.7M vertices), this overhead is less than 1%. Also, our methods promote the performance not by changing the batch size or the semantics of the parameter synchronization. Therefore the model accuracy is not affected.

5 RELATED WORK

Full graph-based GNN training. In full graph-based training, the core computation kernels are sparse matrix multiplication (SpMM) for vertex computation and the sampled dense-dense matrix multiplication (SDDMM) for edge computation [33], [34]. Many works have been proposed to improve the two kernels. Hong et al. [33] propose intra-row reordering and adaptive tiling to increase cache utilization. [34] further proposes two-phase sorting to increase data reuse. GE-SPMM [35] proposes the Coalesced Row Caching and the Coarse-grained Warp Merging to reduce redundant data loading and improve instruction parallelism. Seastar [36] proposes a vertex-centric programming model and applies the automatic kernel fusing. NeuGraph [37] proposes a general programming model named SAGA and an efficient graph-aware data flow. Huang et al. [38] propose several optimizations for GNN inference, including edge grouping, vertex renaming, sparse fetching, and redundant computation reduction. GNNAdvisor [39] proposes the neighbor grouping and dimension scheduling for accelerating the

GNN training kernels. To scale to large graphs, DGCL [40] proposes an efficient communication library that can use heterogeneous links effectively. However, those works that target full graph-based training have scalability issues due to their large memory demand and the cost of distributed communication.

Sampling-based training. By using mini-batches, sampling-based training can scale to large graphs and has shown accuracy competitive with the full graph-based training [14], [15], [16]. GraphSage [11] first introduces the fixed-number vertex sampling and proposes the general message aggregation methods in GNN, such as sum, max pool, average, and LSTM. PinSage [1] uses the map-reduce style of CPU-GPU cooperation to scale the training to large recommend systems. Since feature transfer becomes a performance bottleneck in sampling-based training, many works have been proposed to solve this problem. PaGraph [12] proposes the vertex degree-based GPU cache policy to reduce the feature transfer cost and has shown the effectiveness for power-law graphs. However, its performance is affected by the GPU memory capacity, which may not work for very big graphs. Pytorch-direct [13] utilizes a GPU-oriented communication kernel to increase the transfer efficiency by automatic address alignment. However. when the feature transfer time is reduced, another time-consuming stage, CPU-based sampling, becomes the new performance bottleneck. DGL [7] introduces the GPU-based sampling when the graph data can be placed in the GPU. GNNLab [41] proposes the factored system to divide different stages into different GPUs, which may introduce the communication cost and load imbalance problem. It also proposes a new pre-sampling-based caching policy to adapt to different graph datasets and sampling algorithms. However, decoupled design may introduce large inter-GPU communication overhead. Our work mainly focuses on single-GPU training since a GPU is becoming more powerful and is equipped with more high-bandwidth memory capacity. Its methods are orthogonal to our work when our method can be easily scaled to multiple GPU settings.

6 CONCLUSION AND FUTURE WORK

In this paper, we proposed a combination of methods for optimizing the GNN sampling and training on GPUs, aiming to improve the end-to-end performance of GNN training. In particular, to optimize the GPU sampling, we proposed i) a shared memory-based data placement policy to reduce the number of global atomic operations. ii) a degree-guided adaptive caching policy to make the best use of shared memory but not compromise the concurrency degree of thread blocks, and iii) a degree-guided thread block scheduling method to achieve workload balance and optimize thread block execution. As for end-to-end GNN training, we proposed an asynchronous pipeline-based scheduling method to improve GPU resource utilization. Even though the proposed optimization methods mainly target the GNN training on a single GPU, it can be further extended to the training on multiple GPUs through data or model parallelism and multi-GPU unified memory management, which is the plan of our future work. GNN is an import application in next-generation computing [42]. The large-scale dynamic graphs bring huge challenges to the parallelization strategy, memory management, computing schedule, and partitioning algorithm of GNN systems. Therefore, we plan to continue to carry out the research in the following directions: 1) hybrid parallelization policy for GNN training on the dynamic graphs, 2) GPU-supported snapshot and incremental GNN training system for dynamic graphs, 3) partitioning algorithms for Large-scale graphs in GNN training and 4) efficient GPU memory management in the training and inference of large-scale dynamic graph.

ACKNOWLEDGMENT

This work was supported in part by the National Key R&D Program of China under Grant 2020AAA0108501, and the Key R&D Program of Hubei under Grant 2020BAA020.

REFERENCES

[1] Rex Ying, Ruining He, Kaifeng Chen, et al. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18). 2018, pp. 974–983.

[2] Wei Wu, Bin Li, Chuan Luo, et al. Hashing-Accelerated Graph Neural Networks for Link Prediction. In Proceedings of the Web Conference 2021 (WWW '21). 2021, pp. 2910–2920.

[3] Tian Xia and Wei-Shinn Ku. Geometric Graph Representation Learning on Protein Structure Prediction. In Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD '21). 2021, pp. 1873–1883.

[4] Namyong Park, Andrey Kan, Xin Luna Dong, et al. Estimating Node Importance in Knowledge Graphs Using Graph Neural Networks. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '19). 2019, pp. 596–606.

[5] Wenming Cao, Canta Zheng, Zhiyue Yan, et al. Geometric deep learning: progress, applications and challenges. Science China Information Sciences, 65(2), 2022, pp. 126101:1–126101:3.

[6] Shengwen Liang, Ying Wang, Cheng Liu, et al. EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks. in IEEE Transactions on Computers (TC '21), 70(9), 2021, pp. 1511-1525.

[7] DGL, 2016, Access: JAN. 5, 2022. [Online]. Available:https://www.dgl.ai/.

[8] Euler 2.0: A Distributed Graph Deep Learning Framework, 2020, Access: JAN. 5, 2022. [Online]. Available:https://github.com/alibaba/euler.

[9] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. arXiv preprint arXiv:1903.02428. 2019, pp. 1-9.

[10] Node Property Prediction, 2022. Accessed: Sep. 5, 2021. [Online]. Available: https://ogb.stanford.edu/docs/nodeprop/

[11] William L. Hamilton, Rex Ying, et al. Inductive representation learning on large graphs. In Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS '17). 2017, pp.1025–1035.

[12] Zhiqi Lin, Cheng Li, Youshan Miao, et al. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20). 2020, pp. 401–415.

[13] Seung Won Min, Kun Wu, Sitao Huang, et al. Large graph convolutional network training with GPU-oriented data communication architecture. In Proceedings of the VLDB Endowment (PVLDB '21), 14(11), 2021, pp. 2087–2100.

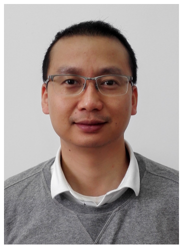
[14] Jianfei Chen, Jun Zhu, and Le Song. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In Proceedings of the 35th International Conference on Machine Learning (ICML '18). 2018, pp. 941–949.

[15] Xin Liu, Mingyu Yan, Lei Deng, et al. Sampling methods for efficient training of graph convolutional networks: A survey. arXiv preprint arXiv:2103.05872. 2021, pp.1-32.

[16] Marco Serafini and Hui Guan. Scalable Graph Neural Network Training: The Case for Sampling. ACM SIGOPS Operating Systems Review (2021), 55(1), 2021, pp. 68–76.
[17] Xue Li, Mingxing Zhang, Kang Chen, et al. 3-D Partitioning for Large-Scale Graph Processing. in IEEE Transactions on Computers (TC '21), 70(1), 2021, pp. 111-127.
[18] Thomas N. Kipf, and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. arXiv preprint arXiv:1609.02907. 2016, pp. 1-14.
[19] Reddit Datasets, 2022. Accessed: Sep. 5, 2021. [Online]. Available: https://www.reddit.com/r/datasets/
[20] Jeffrey S Vitter. Random sampling with a reservoir. ACM Transactions on Mathematical Software (TOMS '85), 11(1), 1985, pp. 37–57.
[21] NVIDIA Visual Profiler, 2008, Accessed: Sep. 5, 2021. [Online]. Available: https://developer.nvidia.com/nvidia-visual-profiler
[22] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, et al. Crossbow: scaling deep learning with small batch sizes on multi-GPU servers. In Proceedings of the VLDB Endowment (PVLDB '19). 12(11), 2019, pp. 1399–1412.
[23] NVIDIA cuBLAS, 2013, Accessed: Sep. 5, 2021. [Online]. Available: https://developer.nvidia.com/cublas
[24] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, et al. cuDNN: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759. 2014, pp. 1-9
[25] Guangming Tan, Linchuan Li, Sean Triechle, et al. Fast implementation of DGEMM on Fermi GPU. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11). 2011, pp. 35:1–11.
[26] MULTIPROCESSING BEST PRACTICES, 2022. Accessed: Sep. 5, 2021. [Online]. Available: https://pytorch.org/docs/stable/notes/multiprocessing.html
[27] NVIDIA Multi-Process Service, 2022. Accessed: Sep. 5, 2021. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html
[28] NVIDIA GPU Programming Guide. 2022. Accessed: Sep. 5, 2021. [Online]. Available: https://developer.nvidia.com/nvidia-gpu-programming-guide
[29] Petar Ristoski, Gerben Klaas Dirk de Vries, and Heiko Paulheim. A collection of benchmark datasets for systematic evaluations of machine learning on the semantic web. In Proceedings of International Semantic Web Conference, 2016, pp. 186–194.
[30] Petar Veličković, Guillem Cucurull, Arantxa Casanova, et al. Graph Attention Networks. arXiv preprint arXiv:1710.10903. 2017, pp. 1-12.
[31] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, et al. Modeling Relational Data with Graph Convolutional Networks. arXiv preprint arXiv:1703.06103. 2017, pp. 1-9.
[32] bhinav Jangda, Sandeep Polisetty, Arjun Guha, et al. Accelerating graph sampling for graph machine learning using GPUs. In Proceedings of the 16th European Conference on Computer Systems (EuroSys '21). 2021, pp. 311–326.
[33] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, et al. Adaptive sparse tiling for sparse matrix multiplication. In Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19). 2019, pp. 300–314.
[34] Peng Jiang, Changwan Hong, and Gagan Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20). 2020, pp. 376–388.
[35] Guyue Huang, Guohao Dai, Yu Wang, et al. GE-SpMM: general-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20). 2020, pp. 72:1–12.
[36] Yidi Wu, Kaihao Ma, Zhenkun Cai, et al. Seastar: vertex-centric programming for graph neural networks. In Proceedings of the 16th European Conference on Computer Systems (EuroSys '21). 2021, pp. 359–375.
[37] Lingxiao Ma, Zhi Yang, Youshan Miao, et al. Neugraph: parallel deep neural network computation on large graphs. In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19). 2019, pp. 443–457.
[38] Kezhao Huang, Jidong Zhai, Zhen Zheng, et al. Understanding and bridging the gaps in current GNN performance optimizations. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21). 2021, pp. 119–132.
[39] Yuke Wang, Boyuan Feng, Gushu Li, et al. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21). 2021, pp. 515–531.
[40] Zhenkun Cai, Xiao Yan, Yidi Wu, et al. DGCL: an efficient communication library for distributed GNN training. In Proceedings of the 16th European Conference on Computer Systems (EuroSys '21). 2021, pp. 130–144.
[41] Jianbang Yang, Dahai Tang, Xiaoniu Song, et al. GNNLab: a factored system for sample-based GNN training over GPUs. In Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22). 2022, pp. 417–434.
[42] Sukhpal Singh Gill, Minxian Xu, Carlo Ottaviani, et al. AI for next generation computing: Emerging trends and future directions. Internet of Things, 19, 2022, pp. 100514-100547.

Wenchao Wu is a PhD student in the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China. He is now doing research on system for deep learning.

Xuanhua Shi is a professor in School of Computer Science and Technology, Huazhong University of Science and Technology, China. He is the Deputy Director of the National Engineering Research Center for Big Data Technology and System (NERC-BDTS). He published more than 100 peer-reviewed publications (such as ASPLOS, VLDB, TOCS, TPDS). Shi is a senior member of the IEEE.



Ligang He is now a Reader in the Department of Computer Science at the University of Warwick, UK. His research areas are mainly in parallel and distributed computing, high performance computing, and distributed machine/deep learning. He has published more than 140 papers in these research areas.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. Jin received his PhD in computer engineering from HUST. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He was awarded Excellent Youth Award from the National Science Foundation of China. Jin is the fellow of IEEE, fellow of CCF, and a member of ACM.