TurboMGNN: Improving Concurrent GNN Training Tasks on GPU With Fine-Grained Kernel Fusion

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

***Abstract*— *Graph Neural Networks* (GNN) have evolved as powerful models for graph representation learning. Many works have been proposed to support GNN training efficiently on GPU. However, these works only focus on a single GNN training task such as operator optimization, task scheduling, and programming model. Concurrent GNN training, which is needed in the applications such as neural network structure search, has not been explored yet. This work aims to improve the training efficiency of the concurrent GNN training tasks on GPU by developing fine-grained methods to fuse the kernels from different tasks. Specifically, we propose a fine-grained *Sparse Matrix Multiplication* (SpMM) based kernel fusion method to eliminate redundant accesses to graph data. In order to increase the fusion opportunity and reduce the synchronization cost, we further propose a novel technique to enable the fusion of the kernels in forward and backward propagation. Finally, in order to reduce the resource contention caused by the increased number of concurrent, heterogeneous GNN training tasks, we propose an adaptive strategy to group the tasks and match their operators according to resource contention. We have conducted extensive experiments, including kernel- and model-level benchmarks. The results show that the proposed methods can achieve up to 2.6X performance speedup.**

***Index Terms*—GNN training, concurrent multi-tasks, GPU, kernel fusion.**

I. Introduction

As AN efficient representation learning tool, *Graph Neural Networks* (GNN) learn the structure and properties of graphs and provide high-dimensional feature representation for downstream tasks such as node classification and link prediction, which is widely used in many applications such as recommendation systems [1], social networks [2], and knowledge graphs [3]. Since training GNNs is a very time- and resource-consuming task [4], [5], general-purpose *graphics processing units* (GPUs) are often used to accelerate the GNN training and many general GNN learning frameworks have been developed [6], [7], [8].

Manuscript received 2 November 2022; revised 12 March 2023; accepted 25 March 2023. Date of publication 17 April 2023; date of current version 8 May 2023. This work was supported in part by the National Key R&D Program of China under Grant 2020AAA0108501, and in part by the Key R&D Program of Hubei under Grant 2020BAA020. Recommended for acceptance by M. Si. *(Corresponding author: Hai Jin.)*

Wenchao Wu, Xuanhua Shi, and Hai Jin are with 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@hust.edu.cn; xhshi@hust.edu.cn; hjin@mail.hust.edu.cn).

Ligang He is with the Department of Computer Science, University of Warwick, CV4 7AL Coventry, U.K. (e-mail: ligang.he@warwick.ac.uk).

Digital Object Identifier 10.1109/TPDS.2023.3267943

Concurrent GNN Training (CGT), as a new scenario for GNN training, is becoming more and more common and critical. On the one hand, data science researchers may need to explore multiple models simultaneously to fit the targeted graphs, which is called *Network Architecture Search* (NAS). These models usually have the same network structure with different hyper-parameters or even have different types of networks. On the other hand, the GPU is equipped with more computation resources such as thousands of cores (e.g., 6912 FP32 cores in A100) and memory with large capacity and high bandwidth (e.g., up to 80 GB HBM2 memory in A100). This provides the vast capacity to run multiple GNN training tasks concurrently on a GPU, especially in the multi-tenant virtualized cloud. As a consequence, the need is increasing for a system that can efficiently support CGTs.

However, previous GNN training systems [6], [7], [8] only focus on improving the running of a single training task such as operators optimization [9], [10], [11], task scheduling [12], [13], programming model [14], [15], and communication reduction [16], [17]. In a concurrent training setting, it is likely that different GNN tasks need to visit the same graph multiple times and exhibit a similar computation pattern. Our studies show that ignoring these underlying relations between the concurrent tasks may result in redundant data accesses and severe resource contention, and ultimately hurts the end-to-end performance of a training task.

The works from other fields such as graph computation have attempted to improve the efficiency of concurrent graph tasks. Seraph [18] proposed graph-level sharing for different tasks to share common graph data in one process or multiple processes. GraphM [19] proposed partition-level sharing which reorganizes the scheduling of the graph partitions into higher-level memory and executes the corresponding concurrent tasks to maximize data locality and cache efficiency. However, these methods cannot be directly applied to Concurrent Graph Training. On the one hand, all the graph vertexes and edges should be accessed when each task is run. However, since different tasks are run in separate kernels in GPU, it is difficult to maintain good data locality by scheduling common thread blocks or to coordinate task scheduling in a unified way due to the limitations of the CUDA programming model. On the other hand, even using these methods, each task still needs to load the graph structure data repetitively, incurring a large number of redundant data accesses. Moreover, a CGT task is more complicated than an ordinary graph processing task because the GNN training model in a CGT task is formed by stacking multiple layers and there

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

are dependencies between forward and backward propagation stages. Unlike an ordinary graph processing task, a CGT task cannot be expressed as simple vertex functions [20], [21] to coordinate the task scheduling in an efficient way.

To overcome these limitations, we propose a novel fine-grained kernel fusion method, which can completely eliminate the redundant loading of graph data and improve GPU resource utilization when running CGT tasks. First, we find that although a GNN model is very complex and different models may have different aggregation and updating methods, they all have a common operator, *Sparse Matrix Multiplication* (SpMM), which is the most time-consuming operator in a GNN computation graph. Moreover, different SpMMs share the same pattern when accessing vertexes and edges. In this paper, a fine-grained kernel fusion method is proposed to fuse the SpMMs from different tasks to a single kernel. By doing so, vertex- and edge-level sharing is enabled and redundant graph data accesses by GNN tasks are eliminated.

Second, when the concurrently running GNN tasks have different numbers of model layers, it becomes difficult to efficiently fuse the operators from different tasks. Simply aligning the layers in different tasks may delay the execution of the task with fewer layers. In order to address this issue, we propose a novel pull-and-push-based method to enable the fusion of the operators from different propagation stages (i.e., the forward and the backward stages). With this method, there will be more fusion opportunities and the kernel synchronization overhead can be reduced.

Third, as the number of concurrent GNN tasks increases, the kernel that contains the fused operators will demand more resources (such as the registers), which will limit the degree of concurrency in GPU. Moreover, the GNN tasks to be fused may have different model topologies. Simply fusing them may introduce unnecessary dependencies between operators. To solve this problem, we propose a strategy to adaptively group the tasks and match their operators according to resource contention. This strategy can effectively reduce the average turnaround time of the tasks.

In summary, there are the following contributions in this paper.

1) We propose a fine-grained SpMM-based kernel fusion method that can completely eliminate redundant graph data accesses and improve kernel efficiency for concurrent GNN training jobs.
2) We propose a novel pull-and-push-based method to fuse the operators in different forward and backward stages, aiming to create more fusion opportunities and reduce the kernel synchronization cost when the concurrent GNN tasks have different numbers of model layers.
3) We propose an adaptive task grouping and operator matching strategy to fuse the operators from different heterogeneous network models, which further improves resource utilization while reducing resource contention.
4) We implement the fine-grained SpMM-based kernel fusion method on top of DGL and develop an efficient concurrent GNN training framework called TurboMGNN.

We conducted extensive experiments. The results show that TurboMGNN can improve the end-to-end training performance by up to 2.6x.

The rest of this paper is organized as follows. The background information is presented in Section II. The detail of TurboMGNN and the kernel fusion methods are presented in Section III. The system design and implementation are described in Section IV. The experiments with TurboMGNN are presented and the results are analyzed in Section V. Related work is presented in Section VI. Finally, the conclusions and future work are presented in Section VII.

II. Background

Given a graph $G = (V, E)$, each $v \in V$ represents a vertex in the graph with a feature vector f_v. Each edge represents a relationship between connected vertices. In order to obtain high-dimensional feature representations for vertices and edges, a GNN model iteratively aggregates the information from the neighbors of a center vertex and updates the feature vectors of the center vertex. There are two main computations for a model layer: message aggregation and feature transformation, which are formulated in (1), where h_v^k represents the feature of the vertex v at the k_{th} layer. Different GNN models may use different aggregation and transformation methods, as well as different combinations of these computations to extract the features with different scales. GNN models usually stack multiple layers (2-3 layers) to improve the model accuracy.

Usually, the aggregation can be expressed as *Sparse Matrix Multiplication* (SpMM), whose execution can be accelerated by the existing CUDA libraries such as [22], [23] and the optimization methods such as [9], [10], [11]. In contrast, the transformation computation can be expressed as the dense matrix multiplication. The GNN training can be classified into full-graph and sampling-based training. The former trains the whole graph at each iteration while the latter trains the whole training sets in mini-batches with each iteration sampling a sub-graph from the original graph for processing. The methods proposed in this paper mainly focus on the full-graph based training.

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

A. Scenarios of Concurrent GNN Training

As an efficient representation learning tool, the GNN model learns the structure and properties of graphs and provides high-dimensional feature representation for downstream tasks such as node classification, and link prediction. However, determining a model suitable for specific graph data is an exploratory and iterative process, which requires testing different network models and hyper-parameters. During this process, multiple models are trained simultaneously on the same graph. Models with poor performance will be gradually discarded and the best performer will eventually win. Another scenario in which multiple GNN

training models are run concurrently is that although these training models target different applications they refer to the same graph. Each model may consist of different node and edge properties features that are more relevant to the targeting task. These models need to visit the same graph data. If the scheduling of these tasks cannot be coordinated well, redundant data loading and resource competition will lead to resource waste and performance degradation, eventually increasing the time of model development and the cost of hardware investment.

B. GPU

With a large number of processing cores and high-speed memory, GPU has become a popular accelerator for many computation-intensive tasks such as deep learning [24], [25], [26] and even memory-intensive tasks such as graph processing [27], [28], [29], [30], [31]. For example, the NVIDIA A100 GPU with the latest generation of Ampere architecture has 6912 FP32 cores. Moreover, the GPU memory capacity is also increasing rapidly. NVIDIA A100 GPU can have up to 80 GB of memory with a bandwidth of 1,134 GB/s. These abundant hardware resources together with the parallelization support provided by CUDA (e.g., multiple streams and hyper-Q technology) enable GPUs to run multiple tasks simultaneously and improve resource utilization, which is especially important in data centers. How to coordinate and schedule these parallel tasks efficiently has become a major challenge for GPU resource management.

III. Methodology

In this section, we first explain our motivation for this work and then propose our methods for optimizing the execution of concurrent GNN training jobs, which include the fine-grained kernel fusion method to eliminate redundant graph structure access, a pull-and-push-based kernel fusion technique to fuse the kernels from different forward and backward stages, and an adaptive task grouping and operator matching strategy for a batch of the GNN training tasks with the heterogeneous model networks.

A. Motivation and Challenges

When multiple GNN training tasks are run concurrently on a GPU, these tasks usually have common access to the graph data and manifest similar computation patterns. Coordinating the scheduling and execution of these tasks can improve performance, such as reducing the storage space of the underlying graph data [18] and increasing data locality [19]. However, different from traditional graph processing tasks, it poses challenges to coordinate concurrently running GNN tasks.

First, unlike ordinary concurrent graph processing tasks, all the graph vertexes and edges should be accessed when each GNN task is processed. This feature leaves limited optimization space for improving data locality by scheduling common blocks.[1] This is because each task is now in an individual GPU kernel. Given the limitation of the current CUDA programming model, different kernels are isolated from each other and can only communicate through global memory. Therefore it is hard for users to control the schedule of thread blocks from different kernels and uniformly coordinate them to improve high-level cache hit rate and achieve data reuse. Moreover, even with the existing optimization methods for storage scheduling [19], each task still needs to access the graph data (edge and vertex data) repeatedly, which incurs redundant memory accesses. It poses big challenges to coordinate the running of different GNN training tasks and reduce redundant data accesses.

Second, we argue that although computation fusion is an efficient way to reduce redundant computations and memory accesses, it cannot be directly applied to concurrent GNN training since a GNN task has a complicated DAG due to its multi-layer structure. Also, each layer in a GNN is composed of various types of operators, including graph propagation operators, DNN operators, etc. These operators are often interleaved to form a complicated model. They cannot be expressed with a simple vertex-centric function like PageRank, BFS, or SSSP in ordinary graph processing tasks. Therefore, although computation fusion is a promising approach, it is very difficult to fuse the concurrent GNN tasks at the task level and it is challenging to design a fine-grained computation fusion method that is both generic and efficient for supporting the GNN tasks with heterogeneous operators. Moreover, just like a DNN model, a GNN training task needs the backward stages to compute the gradients. It is common that different tasks may have different layer sizes. As a training iteration progresses, the layers from the forward and backward stages may mingle with each other. It is also a very challenging task to efficiently fuse the forward and backward computation stages from multiple tasks.

Last, the concurrent GNN tasks may have heterogeneous training models. It is another challenge to fuse these heterogeneous tasks while avoiding the synchronization cost and resource competition.

Based on the above challenges, we propose an efficient kernel fusion technique to combine the core computations from different tasks into a single GPU kernel, which addresses the first challenge. Unlike the previous graph- and partition-level sharing, this method adopts vertex- and edge-level sharing among the GPU kernels, which can completely eliminate the problem of repetitive graph data loading and reduce the kernel launching cost. For the second challenge, we propose a fine-grained SpMM-based kernel fusion method, which not only is generic for handling different GNN tasks but also greatly improves the efficiency of kernel executions. As for the third challenge, a hybrid pull-and-push-based kernel fusion method is proposed to combine the operations in the forward and the backward propagation, which maximizes the kernel fusion opportunity and reduces the kernel waiting/synchronization cost for the task with different numbers of layers. To address the fourth challenge, we propose an adaptive, resource-contention-aware grouping and operator-matching technique for heterogeneous GNN training models. This technique not only further improves fusion efficiency but also mitigates resource contention.

[1] It means scheduling those thread blocks which all need to access the same loaded graph partition to increase data reuse and cache hit rate.

B. The Fine-Grained Kernel Fusion

As discussed in the previous section, the existing concurrent GNN training tasks on a GNN processing systems such as DGL [6] use separate processes and kernels for each training task without any coordination among them. This solo execution i) incurs redundant accesses to the graph structure data although many operators require the same graph traversal, and ii) brings uncertain resource contention. These may compromise the overall performance. The question arises whether one can develop a task coordination strategy to fully exploit the common computations among concurrent tasks and eliminate redundant data accesses.

This paper aims to develop such a technique. First, although a GNN model is more complicated than a traditional graph processing task in the sense that it contains both graph message aggregation and DNN computations, we find that the most time-consuming operators in different training networks are usually the *Sparse Matrix Multiplication* (SpMM). SpMM takes 31%-86% of the overall time in different training models such as GCN, GIN, and GAT when being trained on commonly used datasets such as Pubmed, ogbn-arxiv, Reddit, and ogbn-products. Also, as the core function for message pass in GNN, the SpMM is commonly used in both forward and backward propagation. Second, although different SpMM operators have different aggregation methods (e.g., average, sum, and max-pooling) and different feature inputs, they share the common computation pattern and graph accessing path. Specifically, all the SpMM kernels will aggregate the features from the incoming edges of a center vertex and update its feature vector. Third, the vertexes assignment policy is usually the same for different SpMM kernels on the same graph.

Based on the above observations, we propose an efficient and generic SpMM-based kernel fusion method to fuse the workload of the SpMM kernels from different GNN tasks into a single kernel. Unlike the graph- and partition-level sharing methods in the literature, the fine-grained SpMM-based kernel fusion exploits vertex- and edges-level sharing among multiple SpMM kernels, which completely eliminates redundant graph data accesses. We use the following example to illustrate the principles of the proposed method.

Assume there are two GNN training tasks and each has an SpMM kernel in its forward propagation. Algorithm 1 outlines the main steps to fuse these two SpMM kernels. The algorithm efficiently integrates the traversal loops over vertexes, feature dimensions, and edges in different SpMM kernels, and reuses the data of the accessed vertexes and edges to produce multiple SpMM results. Specifically, the outermost loop (the loop over vertices) traverses the center vertexes allocated to each warp (line 4). The second level loop (the loop over each feature dimension) traverses the feature dimensions assigned to each thread as each thread in a warp may compute multiple feature dimensions (line 6). It should be noted that different SpMM tasks may have different feature dimensions and therefore the algorithm uses the max feature dimension among the SpMM kernels. The innermost loop (the loop over edges, i.e., line 9) traverses the neighboring edges of a center vertex. The reason why the algorithm can combine these three loops in different SpMM kernels is that a specific center node needs to traverse all its incoming edges to update its feature vector and the vertex assignment for each SpMM kernel is the same for all SpMM tasks. In this way, each vertex and edge will be accessed only once and reused by different SpMM tasks to achieve the vertex- and edge-level sharing.

Last, each SpMM task will perform its own aggregation operations. An SpMM kernel will aggregate its feature tensor (e.g., u1 and u2 for these two SpMM kernels in Algorithm 1) (lines 10-14). After the edge traversal loop, an SpMM kernel will finally write its aggregation result tensor into the global memory (e.g., o1 and o2 in lines 16-17 in Algorithm 1).

Algorithm 1: A Simple Example of Fusing Two Froward SpMM Kernels

Input: $rowptr$: row point for CSR, $colindex$: column index for CSR, $u1(2)$: input feature for task1(2), $o1(2)$: output feature for task1(2), $in(out)featureLen1$:the size of feature dimension for u1(o1), $in(out)featureLen2$:the size of feature dimension for u2(o2), $outFeatureLen$: the aligned feature dimension for o1 and o2, $numberV$:the number of vertexes.

```
1: vertex← blockIdx.y*blockDim.y+threadIdx.y
2: stridev ← blockDim.y * gridDim.y
3: strided ← blockDim.x * gridDim.x
4: while vertex < numberV do
5:   featureindex← = blockIdx.x * blockDim.x +
     threadIdx.x;
6:   while featureIndex<outFeatureLen do
7:     localAccum1←0
8:     localAccum2←0
9:     for i =rowptr[vertex]; i < rowptr[vertex+1]; i++ do
10:        cid = colindex[i]
11:        uoff1 = u1 + cid * infeatureLen1
12:        uoff2 = u2 + cid * infeatureLen2
13:        localAccum1←Agg1(uoff1 + featureIndex)
14:        localAccum2←Agg1(uoff2 + featureIndex)
15:     end for
16:     o1[vertex * outFeatureLen1 + featureIndex] +=
        localAccum1;
17:     o2[vertex * outFeatureLen2 + featureIndex] +=
        localAccum2;
18:     featureIndex+= strided
19:   end while
20:   vertex += stridev;
21: end while
```

C. Forward and Backward Kernel Fusion Mechanism

The fine-grained (vertex- and edge-level) SpMM-based kernel fusion can bring several performance benefits. First, kernel fusion can reduce the overhead of kernel launch. Second, it greatly improves the efficiency of kernel execution as the vertex

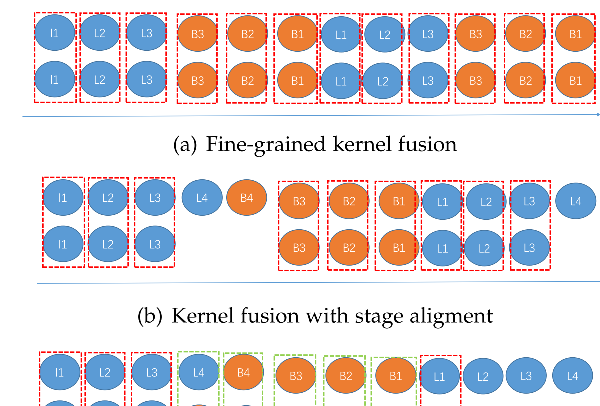
Algorithm 2: The Hybrid Pull-and-Push-Based Kernel Fusion Method for SpMM

Input: $rowptr$: row point for CSR, $colindex$: column index for CSR, $u1$: input feature for task1, $u2$: input gradient for task2, $o1$: output feature for task1, $o2$: output gradient for task2, $in(out)featureLen1$:the size of feature dimension for u1(o1), $in(out)featureLen2$:the size of feature dimension for u2(o2), $outFeatureLen$: the aligned feature dimension for o1 and o2, $numberV$:the number of vertexes,

```
1: vertex ← blockIdx.y∗ blockDim.y+threadIdx.y
2: stridev ← blockDim.y * gridDim.y
3: strided ← blockDim.x * gridDim.x
4: while vertex < numberV do
5:    featureindex← = blockIdx.x * blockDim.x +
      threadIdx.x;
6:    while featureIndex < outFeatureLen do
7:      localAccum1←0
8:      gradient← u2[vertex * inFeatureLen2 +
        featureIndex];
9:      for i =rowptr[vertex]; i < rowptr[vertex+1]; i++ do
10:        cid = colindex[i]
11:        uoff1 = u1 + cid * infeatureLen1
12:        uoff2 = o2 + cid * outfeatureLen2
13:        localAccum1←Agg1(uoff1 + featureIndex)
14:        atomicAdd(uoff2 + featureIndex, gradient)
15:      end for
16:      o1[vertex * outFeatureLen1 + featureIndex] +=
         localAccum1;
17:      featureIndex+= strided
18:    end while
19:    vertex += stridev;
20: end while
```

and edge-level kernel fusion can completely eliminate redundant graph data loading compared to the graph-level or partition-level sharing. When multiple GNN tasks have the same type of networks with the same number of layers, our method can efficiently fuse the SpMM operators from the same layers in different networks, as shown in Fig. 1(a). However, when the networks in different tasks have a different number of layers (illustrated in Fig. 1(b)), the fusion becomes more complicated, which may have an adverse effect on the overall performance without careful consideration. Since the backward computation can also be expressed as performing the SpMM on the transposed graph (by reversing the direction of all the edges), the existing GNN systems such as DGL convert the graph format from CSR to CSC[2] before the backward stage, and use SpMM for gradient propagation on the CSC graph. As the consequence, there are different numbers of SpMM operations in the forward and backward stages for different training tasks.

[2]CSR and CSC are two types of spare sparse matrix representation formats, which are row-major and column-major, respectively.



(a) Fine-grained kernel fusion

(b) Kernel fusion with stage aligment

(c) The Forward and backward kernel fusion

Fig. 1. (a). Kernel fusion for the tasks with isomorphic models and the same number of model layers. (b). Stage-aligned kernel fusion for tasks with the isomorphic model but different layer sizes. (c). The hybrid forward&backward kernel fusion for the tasks with the isomorphic model but different layer sizes. The two models are GCN with three and four layers respectively. Li and Bi stand for the forward and backward computation of layer i. The color stands for different stages (forward and backward). (a) Fine-grained kernel fusion (b) Kernel fusion with stage aligment (c) The Forward and backward kernel fusion.

When fusing the tasks with different numbers of network layers, one simple extension to the fusion method proposed in the previous subsection is to align the stages (forward or backward stages) of different concurrent tasks and only fuse the SpMM operations within a stage, which is illustrated in Fig. 1(b). This extension is reasonable because it is difficult to fuse individual SpMM kernels from different stages. Even if it can be done, it brings little benefit because different stages access different graphs (the graph and its reversed graph respectively) and consequently cannot share the common computation paths.

However, this simple extension limits the opportunity for kernel fusion to the same stage. Furthermore, it incurs high costs of cross-stage synchronization. This is because a fused kernel has to complete all operations in one stage (e.g., the forward stage) before the next fused kernel can start on the next stage (e.g., the backward stage). This will delay the progress of the GNN tasks with shallow networks (i.e., a smaller number of network layers). As shown in Fig. 1(b), The network of task 2 has three layers in the forward stage while that of task 1 has four layers. Although the forward stage of task 2 has been completed after the fused L3 has been completed in this iteration, it has to wait for L4 in task 1 to complete before it can progress to the backward stage. Such delay will be accumulated as the training proceeds through iterations and will have a significant impact on the overall performance of task 2.

To address the issue discussed above, we propose a hybrid Pull-and-Push-based kernel fusion method, which can unify and fuse the SpMM computations in the forward and backward stages. Our method is based on the following observation. Essentially, the forward SpMM aggregates the features of the incoming neighboring vertexes on the original graph, while the backward SpMM for gradient propagation aggregates the gradient of the

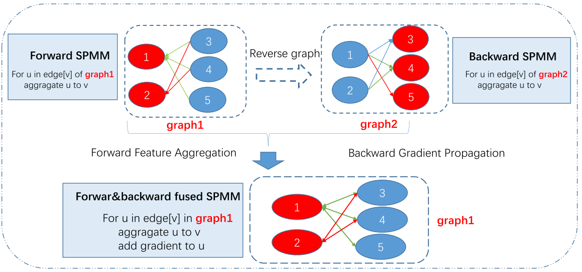


Fig. 2. The forward and backward kernel computation and fusion. The oval is the node and the number stands for node ID.

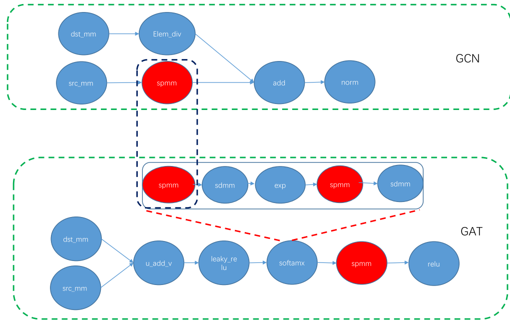


Fig. 3. Operators matching on fine-grained task DAG on two models (GCN and GAT). Circles represent the different operators (kernels).

incoming neighboring vertexes on the transposed graph, which is illustrated in Fig. 2. We can unify the two stages on the same graph and traverse the vertices and edges once to produce the results for both forward and backward stages.

Specifically, the SpMM in the forward stage pulls the features from incoming edges as usual, while the SpMM in the backward stage pushes the gradients to incoming edges, both pull and push operations are performed on the same origin graph. With this method, we can unify the SpMM computations in two stages and fuse them into one kernel. Fig. 2 illustrates the fusion of the operations from the forward and the backward stages. The pseudo-code of the pull-and-push-based kernel fusion method is illustrated in Algorithm 2. The three loops (lines 4-9) are the same as in Algorithm 1, in which the two GNN tasks share the data of vertexes and edges when performing the SpMM operations and once again the redundant graph data loading is completely eliminated. Different from Algorithm 1, the forward SpMM Algorithm 2 aggregates (like pulling) the result from the incoming edges into a local register variable (line 13) and writes the result back after the edge traversal loop (line 16) is completed, while for the backward SpMM, it first reads the gradient from the center vertex before the edge traversal loop (line 8) and directly adds (like pushing) the gradient back to the corresponding gradient tensor of the incoming neighboring vertices (line 14) in the edge traversal loop. Since different vertexes may have the same incoming neighbors, different threads may write the gradients to the same memory address when performing their backward SpMM operations, which causes the data race. The atomic-add instruction from CUDA is used to address this problem.

The benefits of hybrid pull-and-push-based kernel fusion come from two aspects. First, this new fusion method allows cross-stage kernel fusion and therefore provides more kernel fusion opportunities. Second, as illustrated in Fig. 1(c), this hybrid fusion method allows the task with a shallow model to move to the next stage of computations earlier, removing the synchronization cost caused by the stage-alignment-based fusion method. With this method, resource utilization can be improved and the overall turnaround time can be reduced.

D. Operator Matching and Task Grouping

1) Operator Matching: The method proposed in the previous section can be easily applied to concurrent training tasks with homogeneous network models. However, when the tasks have heterogeneous training models, the operators cannot really be aligned according to the model layers. In order to fuse the tasks with heterogeneous models, we further propose the technique to match the operators from different models before performing kernel fusion. In this technique, we identify the same operators from different concurrent tasks and fuse them into one bigger operator. In particular, we decompose each training model into fine-grained operators (such as SpMM) and construct the *Directed Acyclic Graph* (DAG) for the fine-grained operators. Note that the operators in the backward stage should be included in the DAG. In a DAG, each vertex is a fine-grained operator such as matrix multiplication or SpMM, while an edge represents the dependency between the operators.

After constructing the DAG, we perform operator matching, i.e., find the same type of vertices (e.g., the same operators) from different tasks according to the topological order of their DAGs. These matched vertexes will be fused into a bigger vertex and their dependency is also merged. The operator matching is performed in rounds. In each round, any vertex with topological partial order dependency to an already matched vertex cannot participate in the following matching process. This matching process will last multiple rounds. All the matched operators in a single round will be fused into one kernel with the corresponding different feature tensors as inputs. An extra global map table records the fused kernel, the starting position of the feature/gradient tensor of each task, and their local calculation index in the fused kernel. Any data dependency of the original operator from the individual task DAG will be added to the fused operator DAG to ensure correctness. A simple example of operators matching for a GCN and a GAT task is illustrated in Fig. 3. After this process, a new task DAG is constructed and passed to the runtime system.

A runtime system is developed to schedule the fused DAG according to the topological order. The graph path analysis is performed. The kernels without inter-dependency and whose predecessor kernels have been completed will be sent to different streams for parallel execution.

2) Task Grouping: When a relatively large number of tasks (subject to memory constraints) with heterogeneous models are running concurrently, simply performing operator matching and DAG merging as above may result in the following problems.

On the one hand, the fused kernel may be very big. A kernel that fuses too many operators will incur severe resource

contention such as contention of registers files. In lines 7-8 in Algorithm 1, the data are written to the registers. One data item consumes one register. Algorithm 1 fuses two operators and therefore two registers are consumed by the kernel. With too many kernels to fuse, a larger number of registers are needed which will incur registers contention and impair the performance [42].

On the other hand, when multiple DAGs are merged, the dependencies in the original DAGs need to be merged too, which may create dependencies that do not exist in the original DAGs. For example, in Fig. 3, there is no dependency between the sdmm operator in the DAG for GAT and the src_mm operator in the DAG for GCN. However, the dependency is introduced between these two operators after the SpMMs (in the dashed rounded rectangle) in these two DAGs are fused and the two DAGs are merged. When too many heterogeneous DAGs are merged, many unnecessary dependencies may be introduced between the operators, which reduces the degree of concurrency.

Based on the above analysis, we propose a task grouping method to handle the heterogeneous models and limit the size of the fused kernel. We group the tasks according to the similarity of their models (the method of measuring the model similarity is introduced in the next section). The operator matching and DAG merging are only performed within each task group. Since the tasks in each group have similar DAGs, it will not introduce many unnecessary dependencies when the DAGs are merged (like the case where the homogeneous models are fused). A Runtime Execution module is developed (whose implementation is to be presented in the next section) to execute the task groups adaptively according to the resource contention. When the resource contention is low, more task groups will be executed in parallel in different streams, more details of which will be presented in the implementation section. This task grouping method can not only handle the concurrent execution of heterogeneous networks but also mitigate resource contention.

IV. System Design and Implementation

Based on the fine-grained kernel fusion technique, we develop a very efficient framework called TurboMGNN for concurrent GNN training. In this section, we present the architecture of TurboMGNN, including its main modules and implementation details.

A. System Overview

As shown in Fig. 4, there are four core modules in TurboGNN: offline analysis module, online DAG analysis module, runtime execution module, and global storage module.

Offline Analysis Module. This module analyzes the commonly used operators (e.g., SpMM), and generates the kernel fusion templates for different types and different numbers of SpMM operators. In particular, this module first analyzes which operators are commonly used low-level operators in the GNN training, which include SpMM, Dense Matrix Multiplication, SDDMM, EgdeSoftMax, Rele, and so on. The kernel fusion within a single task is optimized in a similar way as in TVM [32]. For example, the matrix multiplication operation is fused with the following

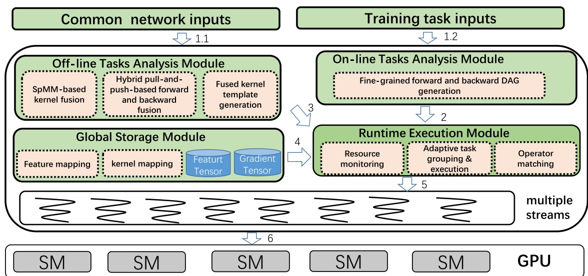


Fig. 4. System architecture.

Algorithm 3. Fused Kernel Template

Input: $Vtask$: vertex task. $Ftaskset$: Forward task in the fused kernel. $Btaskset$: Backward task in the fused kernel.

1: **for** $vertex \in Vtask$ **do**
2: **for** $task \in Btaskset$ **do**
3: Get the gradient of $vertex$
4: **end for**
5: **for** ($task \in Ftaskset$) **do**
6: Initialize the accumulated result variable of $vertex$
7: **end for**
8: **for** ($task \in Ftaskset \cup Btaskset$) **do**
9: Perform feature accumulation for $Ftaskset$.
10: Perform gradient accumulation for $Btaskset$.
11: **end for**
12: **end for**

Relu activation function. After this process, according to the computation semantics and the underlying code implementation, it is checked whether these operators access the common data (such as the graph structure data) and share a similar task execution pattern (such as vertex aggregation pattern) despite the operators may have different input and output tensors. The operators that satisfy the above conditions will be abstracted and their computation loops will be fused into the kernel fusion template.

In addition to the traversal loops in the original kernel, the template of the fused kernel adds a new loop, the task loop, to fuse the computation of different tasks. The new task loop is placed in the most inner level so that the outer loops are shared to enable vertex and edge-level sharing, and eliminate redundant data accesses and computations. We generate the template for fusing the tasks in the forward propagation stage, the template for fusing the tasks in the backward stage, and also the template for fusing hybrid pull&push tasks as discussed in Section III. The pseudo-code of the fused kernel template is shown in Algorithm 3. Because this process is offline and one time, the overhead can be neglected. The templates can be used repeatedly at runtime. Currently, this offline analysis is conducted manually. In the future, we plan to extend it to automatic analysis.

Online Task Graph Analysis Module. This module is responsible for decomposing the training tasks and constructing the DAG of fine-grained operators online. The model definition and hyper-parameter of each task are provided by users in

their model files. The core computation APIs in the models are then analyzed layer by layer for each task. Specifically, for each computation API such as *apply_edges()*, *update_all()*, and *edge_softmax* in each layer, we replace them with basic tensor operators such as *Dense Matrix Multiplications* or *Sparse Matrix Multiplications* (SpMM), and use tensor operators as nodes and data dependencies as edges to construct a fine-grained DAG. Then, TurboMGNN performs task grouping according to the graph similarity. After grouping, the DAG for a single iteration for a task in a group is simply duplicated by 500 (assuming the fusion period is 500; the fusion period is the number of iterations for which the tasks are fused, which is defined in the last paragraph of Section IV-A) times since the DAG for different iterations is the exactly same. After DAG duplication, all the nodes in the final layer of a DAG are connected to the first node of the next DAG. Then, TurboMGNN performs kernel matching, kernel fusion, and template replacement as discussed in Section III-D. At runtime, TurboMGNN adaptively schedules the task groups according to the resource contention and outputs the performance of each task when a fusion period completes.

The decomposition process is recursive since some operators provided by DGL can be further decomposed into more fine-grained operators. For example, the *EdgeSoftMax* is an operator provided by DGL for programmers to write attention modules such as GAT. It can be further decomposed into SpMM and SDDMM operators. Some operators can be directly mapped to low-level operators, for example, *u_add_e_sum* can be mapped into SpMM with the vertex and edge tensors as the input. The example of the fine-grained DAG graph is illustrated in Fig. 3. After this process, the fine-grained DAG for each task is fed into the runtime module (to be introduced next) for performing online task grouping, operator matching, kernel fusion, and inter-group and inner-group scheduling.

Runtime Execution Module. This module performs task grouping, operator matching, kernel fusion, and group scheduling. Based on the task similarity, the number of GNN training tasks, and the hardware resource limitation, similar tasks are put into the same group. The task similarity is determined by the graph similarity. We use the model type and the DAG structure of a GNN task as the features. These features are fed into a graph-to-sequence network (such as [33]) to get a graph representation vector. The dot product of the two tasks' vectors is used to measure the similarity of their DAG graphs.

The tasks placed in the same group share a similar DAG graph. Especially, the number of candidate operators to be fused is similar. We then try to match the operators (i.e., the same operators) from the DAGs of all tasks in the group. The matched operators are fused by using the templates generated in the offline analysis module.

After task grouping, there exist multiple task groups. The runtime execution module monitors the usage and contention of GPU resources. If resource utilization and contention are low, the module will schedule more groups to run in parallel. Otherwise, it schedules a new group in the scheduling queue to run only when a group completes its execution. We obtain the resource utilization of GPU through the NVIDIA GPU performance monitoring library such as *NVIDIA Management Library* (NVML) and *Nsight Compute* (NCU), which collect the resource performance metrics including overall GPU utilization, sm_cycles_active.avg.pct_of_peak_sustained_elapsed, sm_warps_active.avg.pct_of_peak_sustained_elapsed etc, which can be used to infer GPU utilization and resource contention.

The tasks in the same group are scheduled as follows. The operations in a group DAG are scheduled in their topological order. The kernels without dependency are scheduled to run in parallel in different streams. When there exists the data dependency between two kernels (i.e., there is an edge between them in the group DAG) in different streams, we invoke $cudaEventRecord$ (i.e., record a CUDA event) in the source stream and wait for this event to be completed before the kernel in the target steam is launched with $cudaStreamWaitEvent$. Since typically a GNN model is shallow and the number of layers is small (e.g., usually no more than 4), the generated DAG is not big (usually consisting of dozens to hundreds of vertices). Therefore, the overhead incurred by the above analysis process is low (less than 0.5% of the overall time in our experiments).

Global Storage Module. This module stores the graph's original features, intermediate execution results, and gradient tensors. It also records the mapping relation between the operators and the positions of the tensors required by the operators as well as the mapping between the original kernel and the fused kernel. The mapping relations are recorded in a hash table. This information will be passed to the runtime execution module so that the kernels (fused and unfused) can read and write the features or gradient tensors correctly. Due to the stacked structure of the GNN model, a huge amount of intermediate results have to be saved for the backward gradient computation. Different tensors have different life cycles and reuse distances. The global storage module provides the opportunity for potential memory optimization such as CPU-GPU data migration, data reuse, and balance between data reuse and re-computation, which will be our future research work.

Other Implementation Details. We have designed our TurboMGNN to make the best possible effort to fuse kernels from different concurrent training tasks and iterations. However, it is possible for concurrent tasks to require different numbers of iterations to converge. When the numbers of iterations for the models are known in advance or can be predicted, we fuse the models with the least number of iterations among them. Once the model with the least number of iterations has converged, we apply the same fusion strategy to fuse the remaining iterations for the remaining models. If the number of iterations is unknown, we fuse the models for a fixed number of iterations, such as 500 iterations, which is called a *fusion period*, and observe the model convergence. TurboMGNN does not need to synchronize for each iteration, but synchronize once every *fusion period* (e.g., 500 iterations). As a result, the synchronization overhead is small.

B. Discussion

TurboMGNN takes advantage of the commonality among concurrent tasks and uses an inter-tasks operator fusion mechanism to eliminate redundant memory access overhead

and improve instruction parallelism. Even though in this paper, we discuss SpMM-based kernel fusion methods, the proposed method can be applied to SDDMM too since in essence, the calculation process of SDDMM is very similar to that of SpMM. They both get assigned centric vertices and visit the neighbor's edge for each vertex but SpMM aggregates the neighbor's feature vectors while SDDMM performs the vector computation for the two vertices' features. Moreover, the vertex assignment for each warp is the same for different SDDMM on the same training graph. Therefore, we can merge three loops (vertex, feature, neighbor) to fuse the computation and edges visiting from different SDDMM kernels just like we do for SpMM.

V. Evaluation

The concurrent GNN training system called TurboMGNN is presented in previous sections. As there are no previous works on concurrent GNN training to the best of our knowledge, we build a system based on DGL (V0.8.1), which provides the baseline performance that TurboMGNN is compared against. DGL is an efficient GNN system with an easy-to-use interface, rich models, and various optimized low-level graph operators. We first conduct the micro-benchmarking experiments to evaluate the efficiency of the proposed kernel fusion technique and then conduct the experiments to compare the end-to-end performance of TurboMGNN and the DGL-based baseline system.

A. Experiment Setup

Cluster Configuration. The experiments are evaluated on a platform with two eight-core Xeon-2670 2.60 GHz CPUs and 256 GB of memory. The operating system is Ubuntu 16.04. The CUDA version is 10.1 and the CuDNN version is 7.4.2. The GCC version is 6.5.0. Most experiments were performed on a V100 with 32 GB of memory. Due to the out-of-memory problem, Fig. 9 is tested on an A100 with 80 GB of memory.

Workload Configuration. Our experiments use four commonly used datasets. (1) Reddit [34]. The Reddit dataset is a post-to-post graph with 232,965 vertices and 114,615,892 edges. The feature dimension is 602. In this graph, each vertex represents a post, while an edge indicates that the nodes at both ends have been commented on by the same user. (2) Ogbn-product [35]. Ogbn-products is an Amazon product syndication purchase graph where vertices represent the products and an edge represents that the two connected products are purchased together. It has 2,449,029 vertices and 61,859,140 edges with a feature dimension of 100. (3) Ogbn-arxiv [36]. Ogbn-arxiv is a paper citation graph. It has 169,343 vertices and 1,166,243 edges. The feature dimension is 128. (4) Pubmed [37]. Pubmed is a citation network dataset with 19,717 vertices and 88,651 edges with a feature dimension of 500.

We test our system with three representative GNN models. (1) GCN [38]. GCN is the cornerstone of many GNN models and has achieved good performance on many tasks such as node classification. (2) GIN [39]. It is one of the most expressive GNN modes and can be as effective as the Weisfeiler-Lehman graph isomorphism test. (3) GAT [40]. GAT introduces the self-attention mechanism into GNN. It assigns different weights to different neighborhood vertexes or edges when aggregating the features.

B. Micro-Benchmark Evaluation

To evaluate the validity of kernel fusion, we first perform a micro-benchmark evaluation at the kernel level. The test kernel is an SpMM kernel, which, discussed in previous sections, dominates the full-graph-based GNN training time and is common to many different GNN models. The SpMM configurations (e.g., aggregation function, hidden layer dimension, activation function) are extracted from the commonly used GNN model. All the results are averaged over 500 iterations. We conduct experiments to compare the overall execution time of multiple kernels and that of the fused kernel. For a fair comparison, the multiple kernels are run by a single process to enable graph-level sharing. One way to run these multiple kernels is to place them in one stream and execute these kernels sequentially, which is called *baseline1* in this section. Another way is to assign different kernels to run in different streams for concurrent execution, which is called *baseline2*. The fused kernel is executed in one stream, called *fused kernel*.

We also place different SpMM kernels in different processes and exploit the CUDA *Multi-Process Service* (MPS) [41] to set the best-performing MPS configuration which separates resources for each process. We call it *baseline-mps*. We perform the evaluation with different datasets (e.g., Reddit, ogbn-products, ogbn-arxiv, Pubmed), different numbers of kernels to be fused (marked as #F on the x-axis of Fig. 5), different numbers of the forward and backward kernel (marked as #F#B on the x-axis of Fig. 6).

The experimental results with different SpMM forward kernels are present in Fig. 5. It can be seen that our method outperforms *baseline-mps*, *baseline1*, and *baseline2* in all the cases. *baseline-mps* performs worse than *baseline1* for its multiple-process overheads. Moreover, multi-process methods do not share underlying graph storages, which will limit the number of concurrent tasks (kernels) and may incur *out-of-memory* (OOM) problems. *Baseline2* usually outperforms *baseline1* slightly, because *baseline2* uses multiple streams for concurrent execution, which helps improve GPU resource utilization. However, this performance advantage is very limited when the graph is large and dense (e.g., on Reddit the performance advantage is less than 3%). The reason is that when the graph is large and dense, the workload for each kernel is heavy and consequently the GPU is overloaded when running only a kernel. There is little room for improving utilization when multiple kernels are run concurrently. Too heavy workloads may even lead to intense resource contention. The results suggest that it is difficult to improve performance by only leveraging kernel parallelism. Our method achieves a steady performance boost (up to 1.63X). This is because kernel fusion not only eliminates redundant memory accesses but also improves the parallelism of the instructions in the fused kernel.

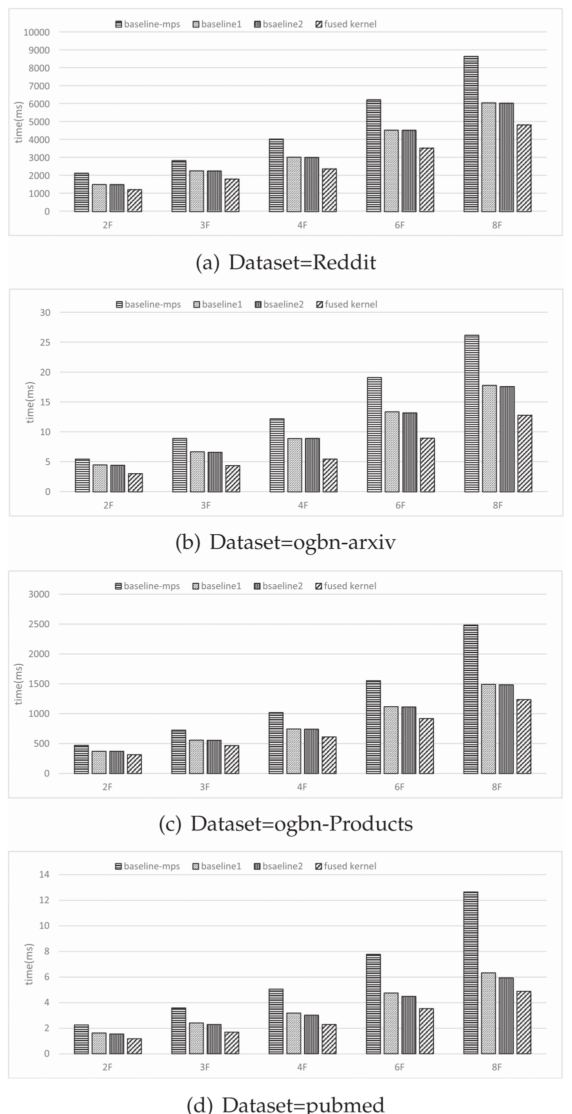


Fig. 5. Run-time of the SpMM kernel on different datasets with different running strategies. The x-axis is the number of SpMM kernels to be fused. (a) Dataset=Reddit (b) Dataset=ogbn-arxiv (c) Dataset=ogbn-Products (d) Dataset=pubmed.

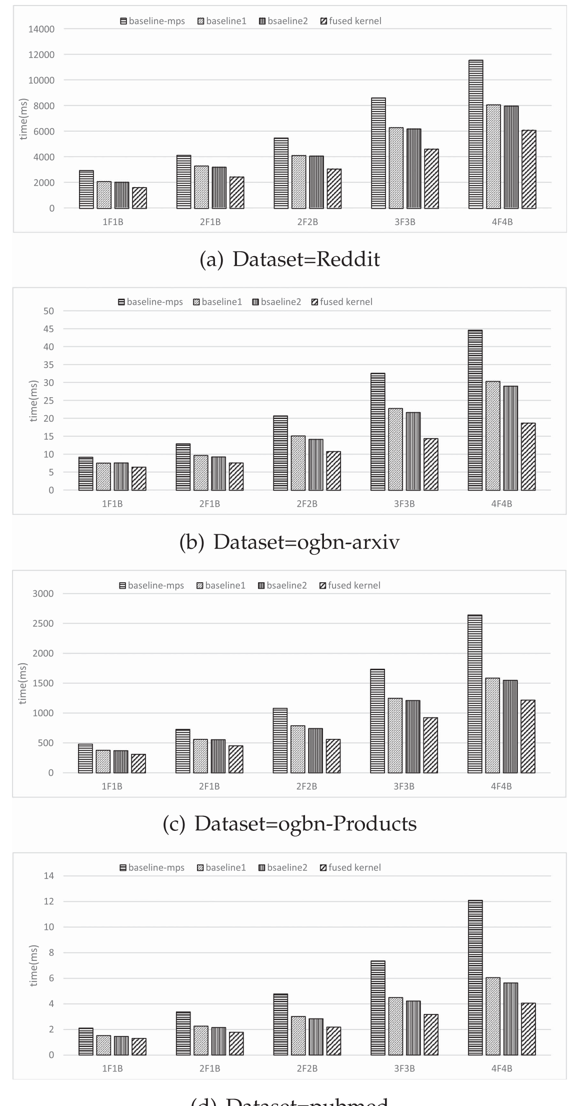


Fig. 6. Run-time of the SpMM forward and backward kernels on different datasets with different execution strategies. the x-axis is the number of forward and backward SpMM kernels to be fused. (a) Dataset=Reddit (b) Dataset=ogbn-arxiv (c) Dataset=ogbn-Products (d) Dataset=pubmed.

We find that when the number of kernels to fuse increases, the speedup (compared with *baseline2*) increases first and then decreases. For example, our kernel fusion technique achieves 1.46X, 1.50X, 1.63X, 1.47X, and 1.38x on ogbn-arxiv when the kernel number is 2, 3, 4, 6, and 8 respectively. These results are expected since when there are more kernels to fuse, more redundant graph memory accesses are eliminated, which results in better performance. When the number of kernels to be fused continues to increase, however, the fused kernel becomes too big, which needs too many on-chip resources (e.g., register files for local aggregation results for each task). These on-chip resources are limited on GPU (e.g, there are only 20,480 KB register files in NVIDIA V100), the excessive demand for these resources will decrease the concurrency of thread blocks, and cause register spills [42]. As the consequence, the decrease in the concurrency of thread blocks will eventually outweigh the benefits brought by reduced memory accesses and increased parallelism of instructions. These results also support our motivation for developing the task grouping strategy in Section III. Namely, task grouping can mitigate the negative impact of too many concurrent tasks.

Fig. 6 presents the results for different numbers of forward and backward SpMM kernels. It can be seen from the figure that the forward and backward kernel fusion is efficient. For example, when there are three forward SpMMs and three backward SpMMs on the ogbn-arxiv dataset, our method can gain a 1.50X speedup, which is comparable to the speedup achieved with the kernel that fused six forward SpMM kernels. Although the hybrid pull-and-push-based kernel fusion may introduce the extra overhead of atomic operations, it completely eliminates the memory accesses in the reverse graphs. Moreover, since the fused kernel has more computation instructions to run by each thread, there are more chances for the warp scheduling

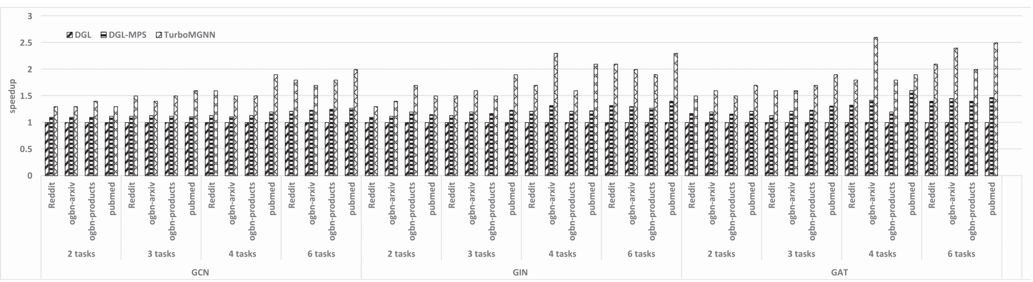


Fig. 7. Speedup for different numbers of homogeneous training tasks on different datasets.

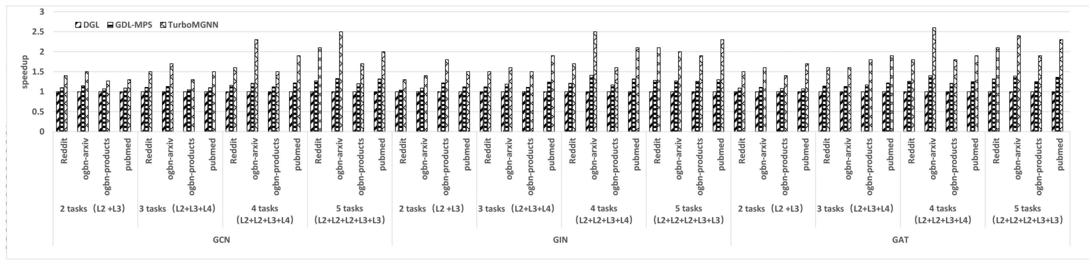


Fig. 8. Speedup for different numbers of homogeneous training jobs on different datasets with different layer sizes for each job. L2 indicates that the layer size of the model is 2.

unit to hide the cost of atomic operators by scheduling other computation instructions.

The fused kernel achieves different speedups for different graphs. For example, with four forward SpMM kernels, our method achieves 1.6X for ogbn-arxiv but a lower speedup of 1.3X for Reddit. This is because the Reddit graph is denser (e.g., the average degree of ogbn-arxiv and Reddit is 7 and 492, respectively). A task in the fused kernel on Reddit has to visit a lot of edges and run many computation instructions, which limits the benefit that can be brought by the concurrent executions of the instructions in the fused kernel. Consequently, the performance boost is not as much as that on more sparse graphs.

C. Macro-Benchmarking Evaluation

We further integrate the proposed kernel fusion methods into DGL and develop an efficient concurrent GNN training system, TurboMGNN. We conducted experiments to compare the end-to-end performance between TurboMGNN and DGL when running multiple GNN training tasks on the same graph. We also compared with other task co-location mechanisms available on GPU such as CUDA MPS and NVIDIA MIG (*Multi-Instance GPU*, introduced since A100) [43]. We choose the best-performing MPS and MIG configuration through model computation complexity estimate and multiple running. We use different numbers of training tasks and different model combinations (GCN, GIN, and GAT) with different numbers of model layers.

The results for homogeneous networks are presented in Fig. 7. The MPS method has an average 1.22x performance speedup compared with default DGL with no resource isolation. TurboMGNN achieves much better performance compared with DGL (an average of 1.74X and up to 2.6X) because it can fuse kernel from different tasks which not only eliminates redundant memory access but also improves kernel execution efficiency. For example, for four GCN tasks, four GIN tasks, and four GAT tasks on the ogbn-arxiv dataset, TurboMGNN achieves the 1.5X, 2.3X, and 2.6X speedup, respectively. The speedup for GAT is higher since the SpMM operator accounts for more run-time (about 86%) in GAT.

Fig. 8 shows the speedup for tasks with the same type of network and different layer sizes. For example, for 4 GCN tasks with two L2, one L3, and one L4 on the Pubmed dataset, TurboMGNN achieves a 1.9X speedup in terms of the average end-to-end run-time. The kernel fusion method with the stage alignment strategy (not shown here due to the space limitation) only achieves the 1.30X speedup. It suggests that simply performing stage alignment cannot make full use of the resources due to the high synchronization cost. These results verify the efficiency of hybrid pull-and-push-based forward and backward kernel fusion on tasks with homogeneous networks.

To investigate how much the operator matching and adaptive task grouping strategies contribute to the decrease in the end-to-end run-time on the tasks with heterogeneous networks, we adopt different combinations of optimization techniques when running these tasks and compare their performance with that of DGL (i.e., *baseline1*). We also compared with MPS and MIG methods to isolate resources for different tasks. The results are presented in Fig. 9. The results show that both MPS and MIG have limited performance improvement (an average speedup is

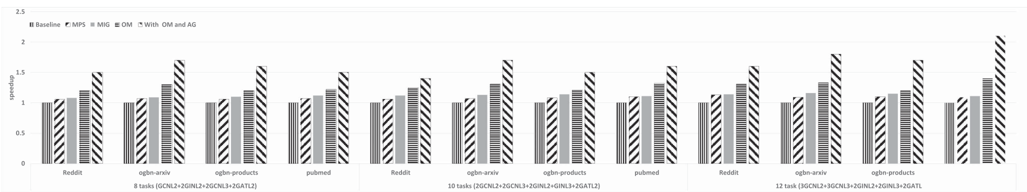


Fig. 9. Speedup for different numbers of heterogeneous training tasks on different datasets with different numbers of layers for each task. 2GINL2 indicates that there are 2 GIN tasks each with 2 layers. The baseline is the default muti-process training. MPS represents using CUDA MPS resource isolation. MIG represents applying MIG resource allocation. OM stands for operator matching and fusion. AG represents adaptive grouping.

1.08 and 1.12 respectively for heterogeneous training tasks in Fig. 9), especially when the number of tasks is large. This is because even though they can isolate resource contention, they cannot eliminate redundant memory accesses among tasks and cause resource shortages for each task when many tasks are concurrently running. MIG's performance is better than MPS for its better hardware isolation mechanism.

TurboMGNN achieves an average of 1.64X and up to 2.1X speedup due to its inter-task kernel fusion, task grouping, and adaptive scheduling mechanism for heterogeneous training tasks, which not only eliminates redundant memory accesses and improves the efficiency of kernel executions, but also avoids resource contention.

As can be seen from Fig. 9, when only adopting operator matching to run 10 tasks on the ogbn-arxiv dataset (4 GCN tasks, 4 GIN tasks, and 2 GAT tasks), TurboMGNN only achieves the 1.3X speedup, which is less than the speedup when running only 4 GCN tasks. This is because, with more heterogeneous tasks, a fused kernel may spread across more graphs. It cannot start execution until all the dependencies within the fused kernel are satisfied, which leads to higher synchronization costs and lower resource utilization. Moreover, fusing too many kernels may cause resource contention such as the contention for registers in the fused kernel, which will eventually reduce the concurrency of thread blocks and affect the end-to-end performance [42].

After adding the adaptive task grouping strategy, the speedup increases to 1.7X. This is because, with adaptive task grouping, the tasks in each group are similar in terms of network structure, which is more friendly for kernel fusion and reduces the synchronization cost. Also, after adopting the task grouping strategy, the number of tasks in a group decreases, which can mitigate the register contention. With the resource-usage-aware scheduling strategy, the runtime execution module can schedule the task groups adaptively according to the utilization of GPU resources, which maximizes resource efficiency.

VI. Related Work

GNN Operators Optimization. In full graph-based GNN training, *Sparse Matrix Multiplication* (SpMM) and *Sampled Dense-Dense Matrix Multiplication* (SDDMM) [10], [11] dominate the execution time. Some work has been proposed to optimize the executions of these two kernels. Hong et al. [10] propose row and column tiling to improve GPU cache efficiency. [11] further proposes row-based sorting to promote data locality. GE-SpMM [9] proposes the shared memory-based caching to reduce the global memory transactions and also warp merging to improve the instruction parallelism in a single SpMM kernel. Huang et al. [12] propose locality-aware task scheduling, neighbor grouping, and data visible range adapter to improve the efficiency of a single GNN inference task. GNNAdvisor [13] proposes several methods to optimize the GNN training kernels including neighbor grouping and dimension scheduling. In order to support GNN training on big graphs, distributed training methods are proposed. DGCL [16] provides a vertex-grained-based optimal link selection algorithm and balanced communication scheduling policy. FlexGraph [15] proposes a compact data structure to support different neighborhood and hierarchical aggregation schemes. However, these optimizations all target the setting of a single GNN task and ignore the correlation among multiple concurrent tasks, which may incur redundant memory accesses and computations. The research on improving the performance of a single kernel is orthogonal to our inter-tasks multi-kernel fusion methods since the underlying training graph is static and fixed and core computation will not change. After adopting the above optimization on a single kernel, our cross-task kernel fusion method can still be applied to them since they use the same optimization for all the kernels.

Compiler Optimization and Kernel Confusion. The DNN compiler automatically optimizes the performance of operators for various hardware accelerators. Halide [44] proposes the domain-specific language and compiler for image processing. TVM [32] is a DNN compiler that utilizes graph-level and operator-level optimization to generate device-efficient codes and a learning-based cost model to guide the search in the optimization space. FlexTensor [45] proposes a scheduling exploration and optimization framework for heterogeneous hardware accelerators without human interference. RAMMER [46] proposes a novel abstract for DNN graph operators and hardware accelerators and utilizes intra- and inter-operator parallelism to generate the optimal execution plans at the compilation time. Seastar [14] proposes a vertex-centric programming model and applies the automatic kernel fusion for GNN training. All of the above works focus on operator optimization and kernel fusion for a single task, which complements our work.

Concurrent Graph Computation. Concurrent graph computations are needed in many realistic scenarios. Several works in the literature target this issue. Quegel [47] and MultiLyra [48] optimize batch tasks from homogeneous queries. Seraph [18] proposes a decoupled data model to share the graph structure data and a copy-on-write semantic to handle concurrent graph modifications. GraphM [19] develops an efficient storage system

for concurrent graph tasks and reorganizes the scheduling order of edge partitions to maximize the data locality. However, these CPU-based methods cannot be directly applied to concurrent GNN tasks on GPU because each GNN task needs to travel the entire graph, leaving limited optimization space for the partition scheduling to exploit the locality, as each task is run in a separate kernel and it is hard to coordinate their memory accesses. Moreover, with those graphs or partition-level sharing, each GNN task with these methods still needs to visit the same edges and vertex data repeatedly, incurring redundant data accesses. Last, concurrent GNN tasks have a more complicated DAG, which has multiple layers-stacked structures, and backward computation. This makes kernel fusion much more difficult to become an effective optimization method, compared with the traditional graph computation tasks (which can be expressed as simple vertex functions). Our work aims to solve these problems.

VII. Conclusion

In this paper, we develop an efficient concurrent GNN training framework on GPUs called TurboMGNN and propose a fine-grained kernel fusion strategy to effectively improve performance. This fine-grained kernel fusion strategy includes 1) an SpMM-based kernel fusion method to eliminate redundant memory accesses and improve the instruction parallelism; 2) a hybrid pull-and-push-based forward and backward fusion method to create more kernel fusion opportunities and reduce synchronization cost; 3) a task grouping and operator matching strategy to fuse kernels for a relatively large number of training tasks with heterogeneous training networks. In this paper, we create the kernel fusion templates offline for the different numbers of kernels and call these templates online when kernel fusion is needed. In the future, we plan to develop the automatic code conversion technique to automatically fuse kernels without the need of creating the templates offline.

References

[1] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, “Graph convolutional neural networks for web-scale recommender systems,” in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 974–983.

[2] W. Wu, B. Li, C. Luo, and W. Nejdl, “Hashing-accelerated graph neural networks for link prediction,” in *Proc. Web Conf.*, 2021, pp. 2910–2920.

[3] W. Cao, C. Zheng, Z. Yan, and W. Xie, “Geometric deep learning: Progress, applications and challenges,” *Sci. China Inf. Sci.*, vol. 65, no. 2, Jan. 2022, Art. no. 126101.

[4] J. Li, H. Zheng, K. Wang, and A. Louri, “SGCNAX: A scalable graph convolutional neural network accelerator with workload balancing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2834–2845, Nov. 2022.

[5] S. Liang et al., “EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks,” *IEEE Trans. Comput.*, vol. 70, no. 9, pp. 1511–1525, Sep. 2021.

[6] DGL, 2016, Accessed: Jan. 05, 2022. [Online]. Available: https://www.dgl.ai/

[7] Euler 2.0: A distributed graph deep learning framework, 2020, Accessed: Jan. 05, 2022. [Online]. Available: https://github.com/alibaba/euler

[8] M. Fey and J. E. Lenssen, “Fast graph representation learning with PyTorch geometric,” 2019, *arXiv:1903.02428*.

[9] G. Huang, G. Dai, Y. Wang, and H. Yang, “GE-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, Art. no. 72, pp. 1–12.

[10] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, “Adaptive sparse tiling for sparse matrix multiplication,” in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, 2019, pp. 300–314.

[11] P. Jiang, C. Hong, and G. Agrawal, “A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs,” in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2020, pp. 376–388.

[12] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen, “Understanding and bridging the gaps in current GNN performance optimizations,” in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 119–132.

[13] Y. Wang et al., “GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs,” in *Proc. 15th USENIX Symp. Operating Syst. Des. Implementation*, 2021, pp. 515–531.

[14] Y. Wu et al., “Seastar: Vertex-centric programming for graph neural networks,” in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 359–375.

[15] L. Wang et al., “FlexGraph: A flexible and efficient distributed framework for GNN training,” in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 67–82.

[16] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, “DGCL: An efficient communication library for distributed GNN training,” in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 130–144.

[17] Y. Bai et al., “Efficient data loader for fast sampling-based GNN training on large graphs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 10, pp. 2541–2556, Oct. 2021.

[18] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, “Seraph: An efficient, low-cost system for concurrent graph processing,” in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 227–238.

[19] J. Zhao et al., “GraphM: An efficient storage system for high throughput of concurrent graph processing,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, Art. no. 3, pp. 1–14.

[20] G. Malewicz et al., “Pregel: A system for large-scale graph processing,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “PowerGraph: Distributed graph-parallel computation on natural graphs,” in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 17–30.

[22] NVIDIA cuBLAS, 2013, Accessed: Sep. 05, 2020. [Online]. Available: https://developer.nvidia.com/cublas

[23] NVIDIA cuSPARSE, 2013, Accessed: Sep. 05, 2020. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html

[24] T. Chen et al., “MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems,” 2015, *arXiv:1512.01274*.

[25] M. Abadi et al., “TensorFlow: A system for large-scale machine learning,” in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[26] A. Paszke et al., “PyTorch: An imperative style, high-performance deep learning library,” in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.

[27] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, “CuSha: Vertex-centric graph processing on GPUs,” in *Proc. 23rd Int. Symp. High- Perform. Parallel Distrib. Comput.*, 2014, pp. 239–252.

[28] F. Khorasani, R. Gupta, and L. N. Bhuyan, “Scalable SIMD-efficient graph processing on GPUs,” in *Proc. Int. Conf. Parallel Architecture Compilation*, 2015, pp. 39–50.

[29] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, “Groute: Asynchronous Multi-GPU programming model with applications to large-scale graph processing,” *ACM Trans. Parallel Comput.*, vol. 7, no. 3, pp. 1–27, Sep. 2020.

[30] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, “Gunrock: A high-performance graph processing library on the GPU,” in *Proc. 21st ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2016, pp. 1–12.

[31] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan, “MultiGraph: Efficient graph processing on GPUs,” in *Proc. 26th Int. Conf. Parallel Architectures Compilation Techn.*, 2017, pp. 27–40.

[32] T. Chen et al., “TVM: An automated end-to-end optimizing compiler for deep learning,” in *Proc. 13th USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 579–594.

[33] K. Xu, L. Wu, Z. Wang, and V. Sheinin, “Graph2Seq: Graph to sequence learning with attention-based neural networks,” 2018, *arXiv:1804.00823v1*.

[34] Reddit datasets, 2022, Accessed: Sep. 05, 2020. [Online]. Available: https://www.reddit.com/r/datasets/

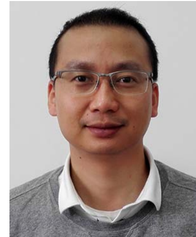
[35] Node property prediction: OGBN-products, 2022, Accessed: Sep. 05, 2020. [Online]. Available: https://ogb.stanford.edu/docs/nodeprop/

[36] Node property prediction: Ogbn-arxiv, 2022, Accessed: Sep. 05, 2020. [Online]. Available: https://ogb.stanford.edu/docs/nodeprop/

[37] PubMed data, 2022, Accessed: Sep. 05, 2020. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/download/
[38] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.
[39] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" 2018, *arXiv:1810.00826*.
[40] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2017, *arXiv:1710.10903*.
[41] NVIDIA multi-process service, 2022, Accessed: Sep. 05, 2021. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html
[42] NVIDIA GPU programming guide, 2022, Accessed: Sep. 05, 2020. [Online]. Available: https://developer.nvidia.com/nvidia-gpu-programming-guide
[43] NVIDIA multi-instance GPU, 2022, Accessed: Sep. 05, 2021. [Online]. Available: https://www.nvidia.com/en-us/technologies/multi-instance-gpu/
[44] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 519–530.
[45] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "FlexTensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 859–873.
[46] L. Ma et al., "RAMMER: Enabling holistic deep learning compiler optimizations with rtasks," in *Proc. 14th USENIX Conf. Operating Syst. Des. Implementation*, 2020, pp. 881–897.
[47] D. Yan et al., "A general-purpose query-centric framework for querying big graphs," *Proc. VLDB Endowment*, vol. 9, no. 7, pp. 564–575, Mar. 2016.
[48] A. Mazloumi, X. Jiang, and R. Gupta, "MultiLyra: Scalable distributed evaluation of batches of iterative graph queries," in *Proc. IEEE Int. Conf. Big Data*, 2019, pp. 349–358.

Wenchao Wu is currently working toward the PhD degree 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, China. He is now doing research work on deep learning system.

Xuanhua Shi (Senior Member, IEEE) is a professor with the 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). His current research interests focus on cloud computing, Big Data processing, and AI systems. He has published more than 100 peer-reviewed publications (such as ASPLOS, *Proceedings of the VLDB Endowment*, *ACM Transactions on Computer Systems*, *IEEE Transactions on Parallel and Distributed Systems*). He received research support from a variety of governmental and industrial organizations, such as the National Science Foundation of China, Ministry of Science and Technology, Ministry of Education, European Union.



Ligang He (Member, IEEE) is a reader with the Department of Computer Science, University of Warwick, U.K. His research is mainly in the area of parallel and distributed computing. He has published more than 180 papers in the area.

Hai Jin (Fellow, IEEE) received the PhD degree in computer engineering from HUST, in 1994. He is a Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is a Fellow of CCF and a life member of the ACM. He has co-authored more than 20 books and published over 900 research papers. His research interests include computer architecture, parallel and distributed computing, Big Data processing, data storage, and system security.