Latency-Based Inter-Operator Scheduling for CNN Inference Acceleration on GPU

Yukai Ping, He Jiang, *Member, IEEE*, Xingxiang Liu, Zhenyang Zhao, Zhide Zhou, and Xin Chen, *Associate Member, IEEE*

***Abstract*—Convolutional Neural Networks (CNNs) are widely deployed on the Graphics Processing Unit (GPU) to support Deep Learning (DL) based services. Popular DL frameworks usually ignore the inter-operator parallelism when executing the inference of CNNs, which results in high inference latency. Although some inter-operator scheduling methods have been proposed, there remains a critical trade-off issue between inference latency (effectiveness) and scheduling time (efficiency). In this article, we propose LIOS, a novel latency-based heuristic inter-operator scheduling method to balance inference latency and scheduling time. In LIOS, a CNN latency model is built based on the given CNN and GPU. Then every operator is assigned a priority value to represent its importance. During each iteration of the scheduling process, LIOS identifies the current data-independent operators, selects the operator with the highest priority value, and assigns it to the GPU stream with the smallest finish time. Extensive experimental results have demonstrated the effectiveness and efficiency of LIOS. For the effectiveness, LIOS can speed up the inference of normal-size and large-size CNNs by 1.13$\sim$ 1.59$\times$ compared to sequential scheduling. This result is comparable to IOS, the latest state-of-the-art scheduling method. For the efficiency, LIOS can speed up the scheduling process by 7$\sim$ 9210$\times$ compared to IOS.**

***Index Terms*—Convoluitonal neural network, deep learning, inference acceleration, inter-operator scheduling.**

I. Introduction

AS ONE of the most widely used DL models, CNNs are the foundation of many service applications [1], [2], [3] and have achieved great success in many fields, including computer vision, natural language processing, and speech synthesis [4], [5], [6]. A typical CNN is organized as a data flow graph where each node represents an operator (the basic computing unit that performs tensor operations) and each edge represents a tensor that flows from one operator to another. In modern service systems, CNNs are often deployed on the GPU since they can provide a better parallel computing ability than the CPU when processing tensors [7]. Before deploying a CNN, the execution order of each operator should be determined. This process is known as inter-operator scheduling and the time spent on it is called the scheduling time. After deployment, the inference of a CNN can be executed and its execution time is known as the inference latency. However, with the development of CNN structures, the increasing computation cost results in the high latency of CNN inference in real-world environments [8], [9]. Another severe problem is the limited utilization of parallel computing resources in the GPU since the parallelism inside a CNN is not fully explored [10]. This problem leads to even higher inference latency of CNNs. Therefore, methods are proposed to reduce the CNN inference latency in the GPU environment by making use of CNN parallelism. In the literature, two kinds of parallelism inside a CNN are commonly used [10], [11].

Intra-Operator Parallelism: Intra-operator parallelism means that when executing the inference of a CNN, each operator is separated into small tasks that can be executed in parallel. AutoTVM [11], Ansor [12], and Chameleon [13] explore this parallelism by operator tuning. Specifically, they optimize a CNN by refining the implementation code of each operator. These methods can provide speedup for CNN inference, however, because of the huge search space, the time spent on operator tuning is usually unacceptable. Besides, only focusing on the intra-operator parallelism is not enough for getting a high GPU utilization [10]. Thus, methods based on inter-operator parallelism are proposed.

Inter-Operator Parallelism: Different from intra-operator parallelism where each operator is executed one by one, inter-operator parallelism aims to reduce the inference latency of CNNs by concurrently executing the operators of a CNN, i.e., inter-operator scheduling, which can support CNN-based service applications by improving their efficiency. Nimble [7] is a DL framework that supports concurrent execution of operators. However, the main contribution of Nimble is to avoid redundant scheduling by an ahead-of-time strategy and Nimble only uses a scheduling method that ignores the latency information of CNNs. Another commonly used scheduling strategy similar to

Manuscript received 19 January 2023; revised 3 November 2023; accepted 27 November 2023. Date of publication 22 December 2023; date of current version 6 February 2024. This work was supported in part by the National Natural Science Foundation of China under Grants 62032004, 61902096, and 62302077, in part by CCF-SANGFOR OF 2022003, in part by the China Postdoctoral Science Foundation under Grant 2023M730472, in part by the Fundamental Research Funds for the Central Universities under Grant 2342022DUT22ZD101, and in part by the Xinghai Scholar Awards of DUT. *(Corresponding author: He Jiang.)*

Yukai Ping and Zhide Zhou are with the School of Software, Dalian University of Technology, Dalian 116024, China, and also with the Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, Dalian 116024, China (e-mail: pingyukai@mail.dlut.edu.cn; cszide@gmail.com).

He Jiang is with the School of Software, Dalian University of Technology, Dalian 116024, China, also with the Key Laboratory for Artificial Intelligence of Dalian, Dalian 116024, China, and also with the DUT Artificial Intelligence Insitute, Dalian 116024, China (e-mail: jianghe@dlut.edu.cn).

Xingxiang Liu and Zhenyang Zhao are with the Sangfor Technologies Inc., Shenzhen 518000, China (e-mail: liuxingxiang@sangfor.com.cn; zhaozhenyang@sangfor.com.cn).

Xin Chen is with the School of Computer Science, Hangzhou Dianzi University, Hang Zhou 310018, China (e-mail: chenxin4391@hdu.edu.cn).

Digital Object Identifier 10.1109/TSC.2023.3345952

Nimble is simple greedy scheduling [10], which tries to execute as many operators as possible at a time. Different from the above methods, Ding et al. [10] find out that the scheduling problem is comprised of overlapping sub-problems, each of which has its optimal structure. Thus, IOS, a scheduling method based on dynamic programming, is designed to reduce the inference latency of CNNs. By using IOS, the authors try to find the optimal schedule of a CNN. All the above inter-operator scheduling methods can provide speedup for CNN inference, however, there is a critical trade-off issue between inference latency (effectiveness) and scheduling time (efficiency).

On the one hand, IOS greatly reduces the inference latency of CNNs but suffers from the huge scheduling time. The ability of IOS to reduce inference latency is better than all the existing scheduling methods. However, due to its exponential time complexity and the huge scheduling space of computation graphs, the scheduling time of IOS is very long. Take NASNet-A [14], a widely used CNN, for example, IOS can provide $1.37\times$ inference speedup compared to sequential scheduling, but it takes IOS 46 minutes to execute the scheduling process. In contrast, for the same CNN, the scheduling time of simple greedy method is only less than 1 s. The huge scheduling time would be an obstacle to the deployment of IOS, which is the reason why a more efficient scheduling method is required. Besides, it is difficult for IOS to achieve the optimal schedule in the real-world environment since the execution latency of each operator varies whenever being measured.

On the other hand, some inter-operator scheduling methods (e.g., simple greedy scheduling) do not take the latency of operators into consideration. Though having short scheduling time, these latency-insensitive methods only provide limited inference speedup. For example, on NASNet-A, simple greedy scheduling spends less than 1 s on the scheduling process but only provides $1.2\times$ inference speedup compared to sequential scheduling. It is obvious that the effectiveness of simple greedy scheduling is much worse than IOS. Since latency-insensitive scheduling methods are not good enough at reducing inference latency, a more effective scheduling method is required.

To balance the inference latency and scheduling time, we propose a novel heuristic inter-operator scheduling method named **L**atency-based **I**nter-**O**perator **S**cheduling (**LIOS**). For each operator in a CNN, LIOS decides the execution order and which GPU stream to be used for its execution, based on the proposed two heuristic strategies. For a given CNN and GPU, LIOS works as follows. First, a CNN latency model is built and each operator is assigned with a priority value. Then the loop of scheduling starts. In each iteration, the data-independent operator with the highest priority is selected. After that, LIOS determines the start time, the finish time, and the used GPU stream for the chosen operator by considering which plan can result in the earliest finish time. Finally, the data dependency of operators is updated at the end of this iteration. The process is repeated until all the operators are scheduled and then LIOS produces a schedule.

To evaluate the effectiveness and efficiency of LIOS, we conduct extensive experiments on CNNs. In the experiments, we use the same benchmark as IOS, including NasNet-A [14], SqueezeNet [15], Inception-V3 [16], and RandWire [17]. For effectiveness, the experimental result shows that LIOS can achieve results comparable with IOS, the latest state-of-the-art scheduling method, on normal-size and large-size CNNs. Specifically, LIOS can speed up the inference by $1.13\sim 1.59\times$ for these CNNs, compared to sequential scheduling. In some cases, LIOS can even provide higher inference speedup than IOS. For efficiency, LIOS achieves $7\sim 9210\times$ speedup compared to IOS. These results show that LIOS can achieve a balance between inference latency and scheduling time for normal-size and large-size CNNs.

The main contributions of the paper are summarized as follows:

- To our knowledge, this study is the first to leverage latency-based heuristics to solve the inter-operator scheduling problem.
- We propose LIOS, a latency-based heuristic inter-operator scheduling method, to tackle the critical trade-off issue between inference latency and scheduling time among the existing scheduling methods.
- We conduct extensive experiments on CNNs to show the effectiveness and efficiency of LIOS. The result shows that LIOS can provide inference speedup comparable with IOS for most CNNs in the benchmark while keeping much shorter scheduling time than IOS.

The rest of the paper is structured as follows. In Section II, we introduce the background and present the problem definition of inter-operator scheduling. Motivation is discussed in Sections III and IV explains LIOS in detail. The experimental setup and results are illustrated in Sections V and VI, respectively. We discuss the application scope and potential usage of LIOS in Section VII. In Section VIII, we review the related work. Finally, we summarize the paper in Section IX.

II. Background and Problem Formulation

In this section, we begin with the basic concepts of the CNN. Then, the parallelism of a GPU and its hardware foundation are introduced. Finally, definitions are made to formulate the inter-operator scheduling problem.

A. Convolutional Neural Network and Representation

CNNs are widely used in classification and regression tasks [18], [19]. If we treat a CNN as a black box, for a given input (e.g., a three-dimensional tensor), it outputs a result tensor.

To clearly understand the inside dataflow, a CNN is usually represented by a Directed Acyclic Graph (DAG), which is also known as the computation graph. Fig. 1(a) shows a toy CNN and all the operators are listed, e.g., $conv1\times 1$ represents a convolution layer of which the kernel size is 1×1. The computation graph of the CNN is shown in Fig. 1(b). We stipulate that there is one entry node and one exit node in the DAG. Moreover, $data\ input$ and $data\ output$ is not regarded as a node.[1] In the DAG, each vertex represents an operator (e.g., a convolution layer), which performs tensor operations on the input data and

[1] Operator, node, and vertex are used interchangeably in this paper.

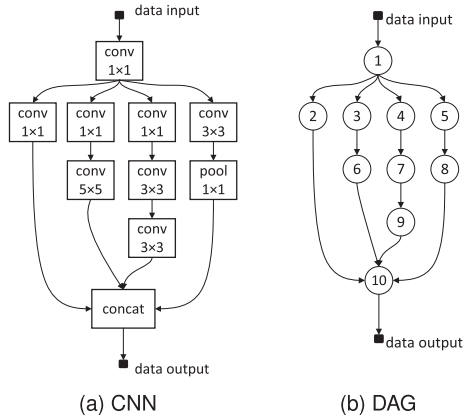


Fig. 1. CNN and its computation graph (DAG).

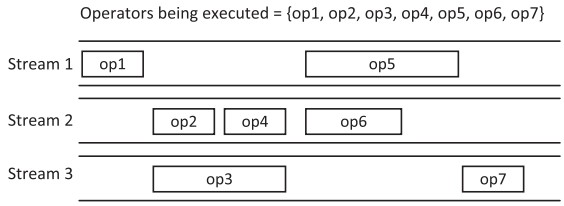


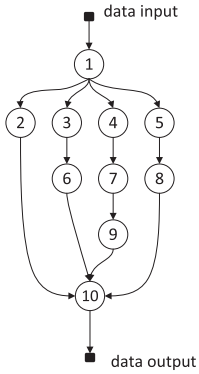
Fig. 2. Example of concurrently executing operators on GPU streams.

outputs the computation result. Each directed edge from vertex A to vertex B means that B uses the output of A as input and is data-dependent on A. For each operator, the execution cannot be interrupted and it can be executed only when all of its predecessors have been executed. The inference process of a CNN can be explained through the DAG. First, the input data is fed to the entry node. Then, the data flows through the DAG according to the edges, and tensor operations are performed. Finally, the result comes out from the exit node.

B. GPU Parallelism

A GPU provides parallel computing ability in two ways. The first kind is intra-operator parallelism. A typical NVIDIA GPU consists of many Streaming Multi-processors (SMs) [20] and each SM has its own computation resources (e.g., memory). When an operator is launched, it is separated into independent tasks that are sent to several SMs for concurrent execution. However, since only a single GPU stream is used in this parallelism, the computation power of a GPU cannot be fully utilized when one operator can not saturate the whole GPU and some SMs are idle [10].

The second kind is inter-operator parallelism, which is the foundation of inter-operator scheduling methods. Fig. 2 shows an example of the concurrent execution of operators. In the figure, seven operators are assigned to three GPU streams. Each GPU stream can be seen as a task queue [7] where operators are executed following the first-in-first-out principle. It can be seen that operators on different streams run in parallel (e.g., $op2$ and $op3$) while operators on the same stream are executed sequentially (e.g., $op2$ and $op4$). Note that the latency of one



Latency Table

Operator	Latency
op1	3
op2	5
op3	5
op4	5
op5	8
op6	15
op7	10
op8	7
op9	13
op10	2

Fig. 3. Weighted DAG and the latency table that stores the weight of each node.

operator, executed alongside other operators, is usually longer than the latency when it is executed solely on the GPU. The reason stems from the additional cost of using multiple GPU streams and the potential contention of GPU resources when multiple operators run in parallel. Still, compared to sequential execution, concurrent execution can greatly reduce the inference latency of a CNN. Different from the former parallelism, inter-operator parallelism uses multiple GPU streams for the concurrent execution of operators, which makes more SMs active and improves the utilization of a GPU.

C. CNN Latency Model

In the inter-operator scheduling problem, for the given CNN and GPU, we build a CNN latency model to represent their information, which consists of two parts: a weighted DAG and a GPU stream set.

Weighted DAG: First, the computation graph of the given CNN is represented by a DAG $G = (V, E)$. Then, each operator in the DAG is executed on the given GPU and its latency is recorded. Finally, we set the weight of each node to its execution latency. As shown in Fig. 3, a weighted DAG is constructed based on the CNN in Fig. 1(a) and the weight of each node is shown in the latency table.

GPU Streams: Since the parallel computing abilities of different GPUs vary in a large range, the number of usable streams in a GPU is not fixed and usually left for user to decide. We use a set $P = \{p_1, p_2, \ldots, p_x\}$ to represent the set of GPU streams, where x determines the number of usable streams.

D. Scheduling Problem Formulation

With the CNN latency model in Section II-C, we can define the inter-operator scheduling problem. For a given CNN and GPU, the key idea of scheduling is to properly arrange the start time, the finish time, and which stream to be used, for the execution of each operator. We begin with some basic definitions.

Definition 1: $EST(v_i, p_j)$ is the Earliest Start Time of vertex v_i on stream p_j. It is defined by

$$EST(v_i, p_j) = \max\left\{free(p_j), \max_{v_k \in pred(v_i)}\{FT(v_k)\}\right\}, \quad (1)$$

where $free(p_j)$ is the earliest time when stream p_j is available for executing operator v_i, $pred(v_i)$ is the set of predecessors of v_i in the DAG, and $FT(v_k)$ is the finish time of operator v_k. The equation shows that $EST(v_i, p_j)$ is constrained by two factors. First, an operator cannot run on an occupied stream. Second, an operator must wait until all of its predecessors are executed.

Definition 2: $ST(v_i, p_j)$ is the Start Time of vertex v_i on stream p_j. It must satisfies $ST(v_i, p_j) \geq EST(v_i, p_j)$.

Definition 3: $EFT(v_i, p_j)$ is the Earliest Finish Time of vertex v_i on stream p_j. It is defined by

$$EFT(v_i, p_j) = EST(v_i, p_j) + w_i, \tag{2}$$

where w_i is the weight of v_i, i.e., the execution latency of operator v_i on the GPU.

Definition 4: $FT(v_i, p_j)$ is the Finish Time of vertex v_i on stream p_j. It must satisfies $FT(v_i, p_j) \geq EFT(v_i, p_j)$.

Definition 5: $makespan$ denotes the inference latency of the CNN, i.e., the time interval between the start and end of the CNN. It is defined by

$$makespan = FT(v_{\text{exit}}) - ST(v_{\text{entry}}), \tag{3}$$

where $FT(v_{\text{exit}})$ is the finish time of the exit node in the DAG and $ST(v_{\text{entry}})$ is the start time of the entry node. In the scheduling problem, $ST(v_{\text{entry}})$ is set to zero and $makespan = FT(v_{\text{exit}})$.

Definition 6: Based on the above definitions, a schedule S is defined by

$$\begin{aligned} S = \{&(v_{\text{entry}}, p_{\text{entry}}, ST(v_{\text{entry}}, p_{\text{entry}}), FT(v_{\text{entry}}, p_{\text{entry}})), \\ &\ldots, \\ &(v_i, p_j, ST(v_i, p_j), FT(v_i, p_j)), \\ &\ldots, \\ &(v_{\text{exit}}, p_{\text{exit}}, ST(v_{\text{exit}}, p_{\text{exit}}), FT(v_{\text{exit}}, p_{\text{exit}}))\}, \end{aligned} \tag{4}$$

where $v_i \in V$ is a node in the DAG, $p_j \in P$ is a stream in the GPU stream set. A schedule S starts with v_{entry}, the entry node of the DAG, and ends with v_{exit}, the exit node. Each tuple $(v_i, p_j, ST(v_i, p_j), FT(v_i, p_j))$ in a schedule S denotes that operator v_i is executed on stream p_j with the start time of $ST(v_i, p_j)$ and the finish time of $FT(v_i, p_j)$.

The purpose of latency-oriented inter-operator scheduling is to find a schedule S that minimizes the $makespan$ of the CNN.

III. Motivation

In this section, we first discuss the importance of inter-operator scheduling in GPU-based deep learning systems and the obstacles faced by state-of-the-art scheduling methods. Then, we explain why latency-based heuristics are leveraged to solve the scheduling problem.

A. Significance and Obstacles of Scheduling

It has become a critical problem to reduce the inference latency of CNNs [21]. A common practice is to make use of the parallelism inside a GPU [10]. Usually, a GPU with more SMs can provide stronger parallel computing power. As shown in Table I, a typical modern GPU usually provides tens of SMs, indicating strong parallelism. However, intra-operator parallelism based methods and popular DL frameworks (e.g., TenorFlow [22] and PyTorch [23]) only use a single GPU stream, which means that the powerful inter-operator parallelism in modern GPUs is ignored [7]. Thus, some inter-operator scheduling methods are proposed to fill the gap and achieve a higher GPU utilization.

TABLE I
The Number of Streaming Multi-Processors in Typical Modern GPUs

	RTX 3060	RTX 3070	RTX 3080	RTX 3090	P100
# SMs	38	46	68	68	108

TABLE II
The Percentage of Latency Change of Each Operator in NASNet-A

Average Percentage of Change	>5%	1 ∼ 5%
# operators	9	93

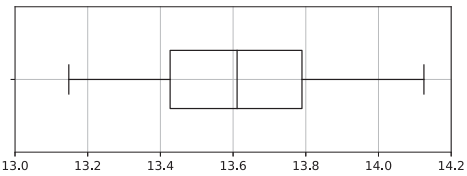


Fig. 4. The distribution of inference latency (ms) of NASNet-A after being scheduled by IOS .

However, there is still room for improvement in inter-operator scheduling methods. Except for the trade-off issue between inference latency and scheduling time, IOS faces another obstacle. Though trying to find the optimal schedule, it is difficult for IOS to achieve this goal in the real-world GPU environment due to the changing execution latency of operators. To illustrate that, we execute NASNet-A 50 times and record the latency of each operator. Then, the percentage of latency change of each operator is calculated and shown in Table II. The result shows that the change rate of 102 out of 139 operators is larger than 1%, which indicates that the execution of an operator on the GPU is not stable and the latency of most operators differs at each execution. Moreover, we use IOS to produce a schedule for NASNet-A and execute the scheduled CNN 50 times. As shown in Fig. 4, the scheduled CNN also has changing inference latency. The reason may be that IOS ignores the above latency instability and has a huge scheduling time, which means that during or after the scheduling process, slight variation in the operator latency will make the schedule non-optimal.

In conclusion, inter-operator scheduling is crucial to reduce the inference latency of CNNs on the GPU environment. Considering the obstacles faced by the existing scheduling methods, we would prefer a new method that can both keep a short scheduling time and produce near-optimal schedules to provide good inference latency reduction.

B. Advantages of Heuristic Methods

To address the inter-operator scheduling problem, we turn to heuristic methods. Heuristic methods are the general name for

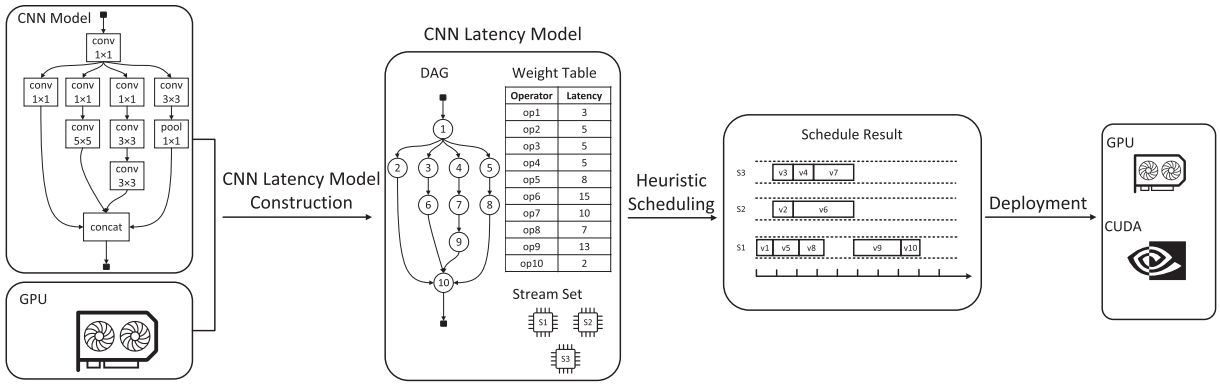


Fig. 5. Main framework of LIOS.

methods that use empirical knowledge to get solutions. The most valuable characteristic of heuristic methods is finding pretty good solutions with polynomial time complexity, instead of exponential time complexity. This kind of method is often used in classic NP-Hard problems since the time spent on finding the optimal solution is unacceptable [24], [25].

Heuristic methods fit very well in solving the trade-off issue among the existing inter-operator scheduling methods. By leveraging a heuristic method, we can get a near-optimal schedule, while keeping a much lower scheduling time. To achieve a better result in terms of reducing the inference latency, we design heuristic strategies based on the latency information of a CNN, which is ignored by Nimble and simple greedy scheduling. Specifically, in LIOS, we take both the data dependency in the CNN computation graph and the latency of each operator into consideration. Then we heuristically assign these operators to different GPU streams for concurrent execution. Another benefit of the short scheduling time is that our heuristic method can quickly adapt to the changing GPU environment.

IV. Method

In this section, we present the Latency-based Inter-Operator Scheduling (LIOS) method. First, we give an overview of LIOS. Then, two core phases in the heuristic scheduling procedure, operator prioritizing and stream selection are introduced with examples. At last, we illustrate the complete workflow and deployment of LIOS.

A. Overview of LIOS

The framework of LIOS is shown in Fig. 5, which has three procedures.

- *CNN Latency Model Construction:* For a given CNN and GPU, we build a CNN latency model as described in Section II-C. First, we construct a weighted DAG based on the computation graph of the CNN, where the weight of each node equals its execution latency on the GPU. Then, a GPU stream set is created to represent the usable streams in the GPU. The CNN latency model will be used as the input of our scheduling method.
- *Heuristic Scheduling:* Heuristic scheduling is the main procedure of LIOS. In this procedure, we first associate each vertex with a priority value to distinguish the importance of each operator. Then, we continually pick an operator and select a stream for its execution. When there is no operator left, the scheduling result is obtained. Two core phases and the complete workflow of this procedure are explained in Sections IV-B, IV-C, and IV-D, respectively.
- *Deployment:* After obtaining a schedule, it is deployed on the GPU. Then we can execute the CNN in a parallel form.

A running example is used to illustrate how IOS works.

Running Example: The CNN latency model in Fig. 3 is used as an example. In the weighted DAG, there are 10 nodes and 12 edges. We set the GPU stream set to $P = \{p_1, p_2, p_3\}$. The scheduling process of each node is shown in Table III.

B. Operator Prioritizing Phase

The basic idea of operator prioritizing is that we can achieve a smaller makespan by executing the important operator first. Intuitively, we believe that an operator is more important if it has longer execution latency. Thus, when facing the condition that several data-independent operators (i.e., operators of which all the predecessors are already scheduled) are ready, we first assign a priority value $Rank(v_i)$ to each operator, which is defined by

$$Rank(v_i) = node\_weight(v_i), \tag{5}$$

where $node\_weight(v_i)$ is the weight of vertex v_i in the weighted DAG, which equals the execution latency of v_i. Then, in the operator prioritizing phase, the data-independent operator with the highest priority value will be scheduled first. The result of our ablation experiment shows that this heuristic strategy works very well.

Example: We focus on step 2 in Table III. After scheduling v_1, four data-independent nodes are appended to $wait\_list$, including v_2, v_3, v_4, and v_5. Then, in the operator prioritizing phase, a node should be selected. According to the execution latency of each operator in the latency table from Fig. 3, the priority values of the four nodes are 5, 5, 5, and 8, respectively. Then, node v_5 is picked for further scheduling since it has the highest priority value among the candidate operators.

TABLE III
THE SCHEDULING PROCESS OF THE CNN LATENCY MODEL IN FIG. 3

Step	Wait List	Operator Selected	EST			EFT			Stream Selected	Schedule Tuple
			p1	p2	p3	p1	p2	p3		
1	v_1	v_1	0	0	0	**3**	3	3	p1	$(v_1, p_1, 0, 3)$
2	v_2, v_3, v_4, v_5	v_5	3	3	3	**11**	11	11	p1	$(v_5, p_1, 3, 11)$
3	v_2, v_3, v_4, v_8	v_8	11	11	11	**18**	18	18	p1	$(v_8, p_1, 11, 18)$
4	v_2, v_3, v_4	v_2	18	3	3	23	**8**	8	p2	$(v_2, p_2, 3, 8)$
5	v_3, v_4	v_3	18	8	3	23	13	**8**	p3	$(v_3, p_3, 3, 8)$
6	v_4, v_6	v_6	18	8	8	33	**23**	23	p2	$(v_6, p_2, 8, 13)$
7	v_4	v_4	18	23	8	23	28	**13**	p3	$(v_4, p_3, 8, 13)$
8	v_7	v_7	18	23	13	28	33	**23**	p3	$(v_7, p_3, 13, 23)$
9	v_9	v_9	23	23	23	**36**	36	36	p1	$(v_9, p_1, 23, 36)$
10	v_{10}	v_{10}	36	36	36	**38**	38	38	p1	$(v_{10}, p_1, 36, 38)$

C. Stream Selection Phase

The stream selection phase is the next step of the operator prioritizing phase. The core idea of this phase is that we hope making the current operator finish earlier will finally result in a smaller global makespan. Thus, for a selected operator in the former phase, LIOS chooses a GPU stream for its execution based on its EFT value. In detail, LIOS first calculates the EFT value of each stream. Then the stream with the smallest EFT is selected. For a vertex v_i, the selected stream p^* is defined by

$$p^* = p_{j^*}$$
$$j^* = \underset{j}{argmin}\, EFT(v_i, p_j). \tag{6}$$

Where v_i is the operator currently being scheduled and $EFT(v_i, p_j)$ is the earliest finish time of operator v_i on stream p_j. This equation means that LIOS selects the stream that makes the current operator finish as early as possible. Combining the two phases, it can be seen that LIOS tries to reduce the makespan by finishing the execution of long-latency operators first.

Example: We continue with the example in Section IV-B. Now that node v_5 is the selected operator, in the stream selection phase, a stream should be selected for its execution. Note that node v_1 is placed on stream p_1 for execution in step 1. Thus, node v_5 cannot be executed on $S2$ or $S3$ at timestamp 0 because it is data-dependent on node v_1. Since the EST value of each stream is the same, all the 3 streams have the same smallest EFT value. Finally, LIOS selects the first stream that achieves the smallest EFT, which is $S1$ in this case.

D. LIOS and Deployment

In this section, LIOS is explained in detail. First, the pseudo-code of LIOS is shown in Algorithm 1 and we go through it step by step. Second, a complete example is presented for a better understanding of LIOS. Third, we explain how to deploy a schedule to the hardware (e.g., a GPU) for the concurrent execution of CNN operators.

As shown in Algorithm 1, LIOS begins with the CNN latency model construction procedure at line 1, which is described in Section II-C. The heuristic scheduling procedure starts at line 2. From line 2 to line 3, LIOS initializes the $wait\_list$ and result list S. The former is used to store operators waiting for scheduling and the latter keeps the scheduling result. Each element in S is a schedule tuple defined in Definition 6. Then, LIOS computes the priority value $Rank$ for each operator, according to the node weight of the operator. In the $while$ loop from line 5 to line 15, LIOS iteratively schedules every operator in the weighted DAG. At line 6, the operator with the highest $Rank$ in $wait\_list$ is picked for scheduling. From line 7 to line 8, the EFT value of v_i on each stream is calculated. Then LIOS selects the stream with the smallest EFT at line 10. At line 11, LIOS determines the schedule tuple of operator v_i and stores it in S. After finishing the scheduling of an operator, LIOS first removes the operator from $wait\_list$ and then updates the data-dependency among operators, which is described from line 12 to line 13. The reason for updating the data dependency is that some new operators will be ready for execution after the execution of their predecessors. The final statement inside the $while$ loop is at line 14, where those newly ready operators are appended to $wait\_list$. When there is no operator in the $wait\_list$, the loop ends, and the schedule S is obtained.

Algorithm 1: Pseudocode of LIOS.

Input: A CNN model and GPU
Output: A schedule S
1: Construct CNN latency model: a weighted DAG $G(V, E)$ and a set P which has x usable GPU streams
2: Create an empty scheduling result list S
3: Create an empty list $wait\_list$ and put v_{entry} as the first element
4: Compute the priority value $Rank(v_i)$ for each operator
5: **while** $wait\_list$ is not empty **do**
6: $v_i \leftarrow$ operator with the highest $Rank$ value in $wait\_list$
7: **for** each stream p_j in P **do**
8: Compute $EFT(v_i, p_j)$
9: **end for**
10: $p^* \leftarrow$ stream p_j which minimizes the $EFT(v_i, p_j)$
11: Append $(v_i, p^*, EFT(v_i, p^*), EST(v_i, p^*))$ to S
12: Remove v_i from $wait\_list$
13: Update data dependency among operators
14: Put data-independent operators into $wait\_list$
15: **end while**
16: **return** S

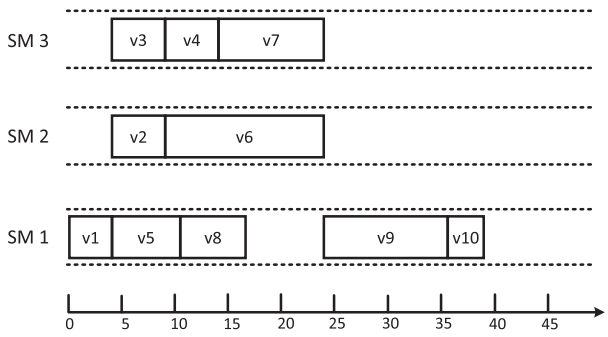


Fig. 6. Visualized scheduling result in Table III (makespan=38).

For time complexity, it can be seen from the pseudocode that the time complexity of LIOS is $O(xn^2)$ for a GPU stream set with x usable streams and a weighted DAG $G = (V, E)$ with n vertices.

Example: We illustrate the working process of LIOS in detail and start from step 1 in Table III. First, v_1 is appended to $wait\_list$ because the entry node of DAG has no predecessor. Then, after operator prioritizing and stream selection, v_1 is assigned to p_1 with a start time of 0 and a finish time of 3. In step 2, data-dependency is updated and the newly data-independent nodes (v_2, v_3, v_4, and v_5) are appended into $wait\_list$ for operator selection. Then v_5 is selected and assigned to p_1 with a start time of 3 and a finish time of 11. The above process repeats until all the nodes are scheduled, which is step 10 in the table. The final scheduling result in Table III is visualized and shown in Fig. 6. Compared to the unscheduled CNN, of which the operators are sequentially executed, LIOS reduces the theoretical makespan from 73 to 38.

In the end, we describe the deployment of LIOS. After obtaining a schedule for a CNN and GPU, CUDA [20], the application programming interfaces provided by NVIDIA, is used to create GPU streams, control the synchronization among streams, and perform parallel execution of the CNN on the GPU. For each operator in the CNN, cuDNN [26], the NVIDIA CUDA deep neural network library, is used to create the corresponding CUDA kernel. Though LIOS is designed for GPU, the scheduling method can be applied in various hardware that supports concurrent execution of CNN operators (e.g., multi-core CPUs). The reason is that LIOS treats each GPU stream as a task queue and this abstract conception can be adapted to other hardware.

V. Experimental Setup

In this section, we first describe the benchmark, baselines, implementation detail, and implementation environment. Then, we present the three Research Questions (RQs) which will be investigated in the experiment.

A. Benchmark, Baselines and Implementation Detail

In this section, we introduce the CNNs selected for the experiment, the scheduling methods used for comparison, and the details of implementation.

TABLE IV
Information of the Computation Graphs of CNNs in the Benchmark

CNNs	# vertices	# edges
SqueezeNet	50	65
InceptionNet-V3	119	153
RandWire	120	260
NASNet-A	374	576

Benchmark: We follow the setup of IOS to build our benchmark. Popular and representative CNNs are used in the experiment, including NasNet-A [14], SqueezeNet [15], Inception-V3 [16], and RandWire [17]. The number of vertices and edges in the computation graph of each selected CNN is shown in Table IV. According to the number of vertices in the computation graph, we divide the four CNNs into three categories. First, we regard SqueezeNet as a typical small-size CNN which has a small number of operators. By using small convolution layers (e.g., $conv1 \times 1$), the number of parameters of SqueezeNet is extremely small. Second, InceptionNet-V3 and Randwire are regarded as normal-size CNNs commonly used in computer vision tasks [27], [28]. The former is well-designed by people with expert knowledge while the latter is generated by some random methods. Third, we use NASNet-A to represent the large-size CNNs designed by neural architecture search methods [29].

Baselines: We show the effectiveness and efficiency of LIOS by comparing it with the baseline methods. Among the inter-operator scheduling methods, two kinds of scheduling methods are selected for comparison. First, sequential scheduling and simple greedy scheduling are used since they are classic methods. Second, IOS and Nimble, two state-of-the-art methods, are selected. The detailed description of each method is as follows.

- Sequential scheduling is the most classic method that is widely used in popular DL frameworks [7]. In this method, no inter-operator parallelism is used and all the operators are executed one by one. Thus, the ability of an inter-operator scheduling method to provide inference latency reduction can be shown by calculating its speedup compared to sequential scheduling.
- Simple greedy scheduling is another commonly used method [10]. It only takes the data-dependency among operators into consideration and tries to execute as many operators as possible at each time.
- Nimble [7] is a state-of-the-art method. It first transforms the computation graph of a CNN to its Minimum Equivalent Graph (MEG). Then, a bipartite graph is constructed based on the MEG. Finally, a schedule is generated from the maximum matching of the bipartite graph.
- IOS [10] is the latest state-of-the-art scheduling method that leverages dynamic programming to find a schedule. Besides, IOS is not a pure inter-operator method, since it employs operator merging, an optimization method for computation graph, to provide extra inference latency reduction in some conditions.

Implementation: The open-source version of IOS is used while we implement LIOS and the other baseline methods by using Python 3.7. For the hyperparameter setting, we follow the

experiment setting of IOS, and set the number of usable streams to 8 and the batch size to 1 in all the used methods to make fair comparisons.

Environment: All the experiments are executed on an NVIDIA RTX 3080 with NVIDIA Driver 470.129.06, CUDA 11.0, and cuDNN 8.0.5. Specifically, NVIDIA RTX 3080 is a GPU with 68 SMs, which means that it has a strong parallel computing ability. For CPU and memory, two Intel Xeon Gold 6226R @2.90 GHz processors and 256 G RAM are installed on the server.

B. Research Questions

We propose and study the following three research questions. RQ1 compares LIOS and other inter-operator scheduling methods to evaluate the effectiveness of LIOS. RQ2 compares the time cost of LIOS and the baselines to evaluate the efficiency of LIOS. RQ3 is an ablation experiment to study the contribution of the two heuristic strategies of LIOS. RQ4 compares LIOS with a typical operator fusion method to give a more comprehensive evaluation of the effectiveness of LIOS.

RQ1: Can LIOS produce schedules that reduce the inference latency of CNNs?

Motivation: RQ1 is proposed to show the effectiveness of LIOS. Since reducing the inference latency of CNNs is the main purpose of inter-operator scheduling methods, the inference latency of CNNs after being scheduled by different methods need to be compared.

Method: RQ1 is answered by comparing the inference latency reduction provided by LIOS and the baseline methods. First, we apply LIOS and the baseline methods to produce schedules for each CNN in our benchmark. Then, we run each scheduled CNN 50 times and use box plots to record the distribution of inference latency. After comparing the inference latency produced by each scheduling method, we calculate the speedup provided by LIOS compared to sequential scheduling.

RQ2: Can LIOS keep a short scheduling time?

Motivation: In addition to reducing the inference latency, another important consideration of inter-operator scheduling methods is to keep a short scheduling time, which is very important for real-world deployment. As described in Section III, IOS has an exponential time complexity and spends huge scheduling time on complex CNNs. LIOS is expected to solve this problem. Thus, the scheduling time of LIOS and IOS needs to be compared.

Method: We answer RQ2 by comparing the scheduling time of LIOS and IOS. Also, we provide the scheduling time of Nimble, sequential scheduling, simple greedy scheduling, and the variations of LIOS though they cannot provide inference latency reduction comparable with IOS or LIOS. For each CNN in the benchmark, we run the above scheduling methods 10 times to produce schedules and record their average scheduling time.

RQ3: What are the contributions of the two heuristic strategies to LIOS?

Motivation: Two heuristic strategies are used in our method. First, in the operator prioritizing phase, we believe that the operator with longer execution latency is more important and should be executed first. Second, in the stream selection phase, we choose the stream with the smallest EFT value for the operator being scheduled. By investigating RQ3, we try to prove that both heuristic strategies have positive contributions to LIOS.

Method: An ablation experiment is designed for RQ3. First, we design three variation methods based on LIOS, including LIOS-Rand1, LIOS-Rand2, and LIOS-Rand. In LIOS-Rand1, we randomly pick an operator from the $wait\_list$ in the operator prioritizing phase, instead of using a heuristic strategy. In LIOS-Rand2, we replace the heuristic in the stream selection phase by randomly selecting a stream for the execution of the current operator. IOS-Rand is a purely random scheduling method in which we use random selection in both phases. To investigate RQ3, we compare the inference latency reduction provided by LIOS, LIOS-Rand1, LIOS-Rand2, and LIOS-Rand. For comparison, we apply the four scheduling methods to produce schedules for each CNN in our benchmark. Then, each scheduled CNN is run 50 times and box plots are used to record the distribution of the inference latency.

RQ4: How is the effectiveness of LIOS compared to operator fusion methods?

Motivation: There are two kinds of typical methods to explore inter-operator parallelism, inter-operator scheduling and operator fusion [10]. Operator fusion is a technique that merges multiple operators into a single operator to improve GPU utilization and reduce the execution latency [21]. By fusing multiple operators, the time cost of launching operators is reduced and the efficiency of memory access is improved. We compare LIOS with operator fusion methods to give a more comprehensive evaluation of its scheduling effectiveness. Apollo [30] (integrated in Mindspore [31]) is an operator fusion method that integrates graph-level node grouping and operator-level loop fusion closely, widening the fusion search space. TensorRT [32] is a popular inference framework provided by NVIDIA, which provides typical operator fusion techniques. HFuse [33] provides horizontal fusion strategies, which is different from the standard.

Method: In RQ4, we compare the inference latency reduction provided by LIOS, TensorRT, and Apollo. First, we apply LIOS and the operator fusion methods to optimize each CNN in the benchmark. Then, we run each optimized CNN 50 times and use box plots to record the distribution of inference latency. According to the experimental result, we give a more comprehensive evaluation of LIOS.

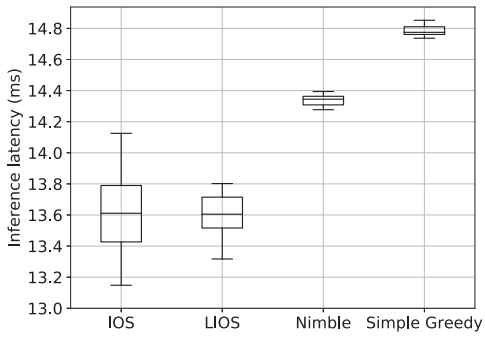
HFuse is not compared in RQ4 since it is not an end-to-end optimization for DL networks and cannot be directly applied to CNNs, while LIOS mainly focuses on optimizing the inference of the whole network.

VI. Experimental Results

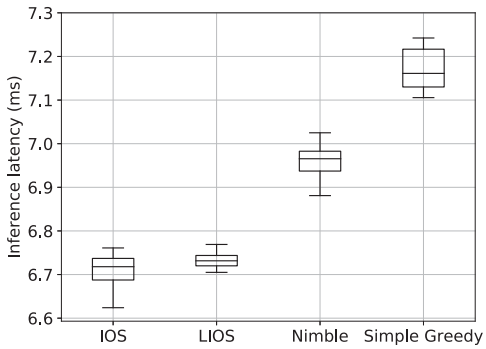
A. Investigation of RQ1

In this RQ, we investigate the effectiveness of LIOS by comparing its ability to reduce the inference latency of CNNs with the baseline methods.

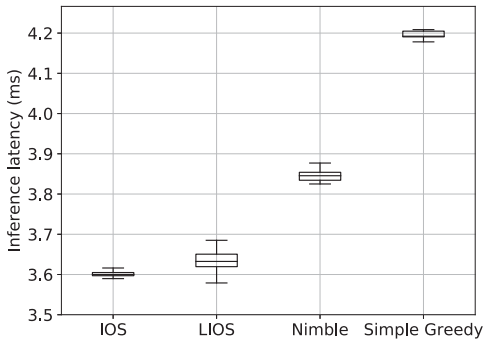
Result: After being scheduled by LIOS and baseline methods, the inference latency of RandWire, NASNet-A, InceptionNet-V3, and SqueezeNet is shown in Fig. 7(a), (b), (c), and (d), respectively. Note that the median latency of each CNN after being scheduled by sequential scheduling is shown under the caption of



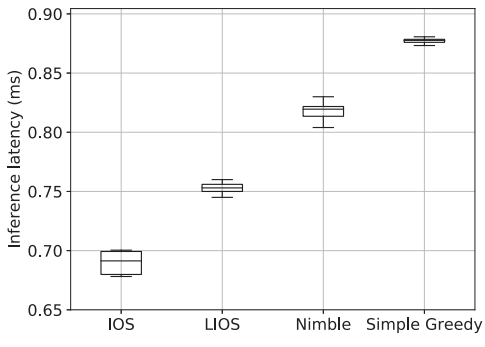
(a) NASNet-A
$(seq\_latency = 18.100\ ms)$



(b) RandWire
$(seq\_latency = 7.631\ ms)$



(c) InceptionNet-V3
$(seq\_latency = 5.735\ ms)$



(d) SqueezeNet
$(seq\_latency = 0.769\ ms)$

Fig. 7. Box plots of the inference latency of each CNN after being scheduled by IOS, LIOS, and simple greedy.

each subgraph and defined as $seq\_latency$. The reason is that the inference latency produced by sequential scheduling are usually much longer than the other methods, except for SqueezeNet. A box plot is used to show the distribution of inference latency of each scheduled CNN after being executed 50 times. The median of inference latency is shown as a line in the center of a box plot, which represents the typical inference latency of a scheduled CNN. The height of each box plot shows the latency variation of a scheduled CNN. A scheduling method is more effective and stable if it produces shorter median latency and smaller latency variation for a CNN.

From the result, it can be seen that both IOS and LIOS outperform the other scheduling methods on the benchmark. On SqueezeNet, although LIOS produces shorter latency than Nimble and simple greedy, it produces latency longer than IOS. Since IOS and LIOS are superior among the methods, we compare them in detail.

First, we analyze the results on RandWire, NASNet-A, and InceptionNet-V3. By comparing the inference latency of CNNs scheduled by IOS and LIOS, we can see that the latency reduction produced by them is very close. On NASNet-A, LIOS produces slightly shorter median latency and achieves much smaller variation than IOS. On Randwire, the median latency produced by LIOS is a little longer than IOS, but LIOS still has smaller variation. On InceptionNet-V3, though IOS is better than LIOS in both the above two indexes, the gap between them is still small. An interesting result on InceptionNet-V3 is that LIOS can produce shorter latency than IOS in some cases.

Second, we compare LIOS and IOS on SqueezeNet, a small-size CNN of which the $seq\_latency$ is only 0.769 ms. On this CNN, LIOS can produce slightly shorter latency than sequential scheduling, but the latency can still be longer than $seq\_latency$ sometimes. The reason comes from the synchronization overhead when operators are concurrently executed on GPU streams, which is obvious when the inference latency of the CNN is extremely short like SqueezeNet. IOS produces shorter median latency than LIOS in this situation since it can list all the schedules and pick the one that minimizes the synchronization overhead.

Synchronization overhead is not quantitatively analyzed due to the following two reasons. First, the number of synchronizations cannot be determined since some synchronizations are triggered within a CuDNN kernel, which is not open-sourced. Second, it is hard to analyze the accurate time cost of synchronizations inside a CuDNN kernel with the current profiling tools.

Conclusion: LIOS can generate schedules to provide inference speedup for RandWire, NASNet-A, and InceptionNet-V3, of which the speedup can achieve 1.13×, 1.36×, and 1.59×, respectively, compared to sequential scheduling. This result is comparable to or even better than IOS in some cases. Considering the huge scheduling time of IOS, LIOS is a better choice when dealing with normal-size and large-size CNNs. While for small-size CNNs like SqueezeNet, LIOS suffers from the synchronization overhead. In this case, IOS is more suitable since it can provide latency reduction and its scheduling time is acceptable on small-size CNNs.

TABLE V
SCHEDULING TIME(S) COMPARISON FOR IOS AND LIOS ON THE BENCHMARK

	IOS	LIOS	Speedup
SqueezeNet	0.34	0.05	×7
InceptionNet-V3	38.28	0.27	×142
RandWire	3775.94	0.41	×9210
NASNet-A	2782.66	1.79	×1554

B. Investigation of RQ2

In this RQ, we investigate the efficiency of LIOS by comparing the scheduling time between IOS and LIOS. Other baseline methods are not compared and the reason is described in RQ2 in Section V-B.

Result: Table V shows the scheduling time of LIOS and IOS on each CNN of the benchmark. It can be seen that LIOS finishes scheduling within 2 seconds on all the CNNs, while the scheduling time of IOS is much longer.

We analyze the result from two perspectives. First, for Randwire, NASNet-A, and Inception-V3, LIOS shows its superiority by providing huge speedup for the scheduling process, which is 9210×, 1554×, and 142×, respectively, compared to IOS. More significantly, LIOS reduces the scheduling time from hour-level to second-level, which makes it more possible for the deployment of LIOS in the real-world environment. Second, for SqueezeNet, a small-size CNN, though the scheduling time of IOS is not so large and acceptable in time-insensitive situations, LIOS still provides 7× speedup compared to IOS.

Besides, the time cost of the rest scheduling methods is shown in Table VI. From Tables V and VI, it can be seen that sequential and simple greedy scheduling yield the smallest scheduling time cost (even can be ignored) among all the methods. LIOS, variations of LIOS, and Nimbe have a similar second-level time cost. As the complexity of CNN grows, the time cost of IOS increases substantially.

Conclusion: LIOS greatly reduces the scheduling time in all cases compared to IOS. The polynomial complexity of LIOS ensures reasonable scheduling time in all situations, while IOS suffers from its exponential complexity.

C. Investigation of RQ3

In this RQ, we investigate the contributions of the two heuristics used in LIOS and analyze the result of LIOS-Rand, a purely random scheduling method.

Result: After being scheduled by LIOS, LIOS-Rand1, LIOS-Rand2, and LIOS-Rand, the inference latency of each CNN in the benchmark is shown in Fig. 8(a), (b), (c), and (d), respectively. From the result, it can be seen that LIOS outperforms all the variant methods. Among the three variant methods, LIOS-Rand1 produces the smallest median latency in most cases except for SqueezeNet. In SqueezeNet, all the variant methods produce similar median latency and LIOS-Rand2 has a slightly better result. Besides, on every CNN, LIOS produces the smallest latency variation compared to all the variant methods.

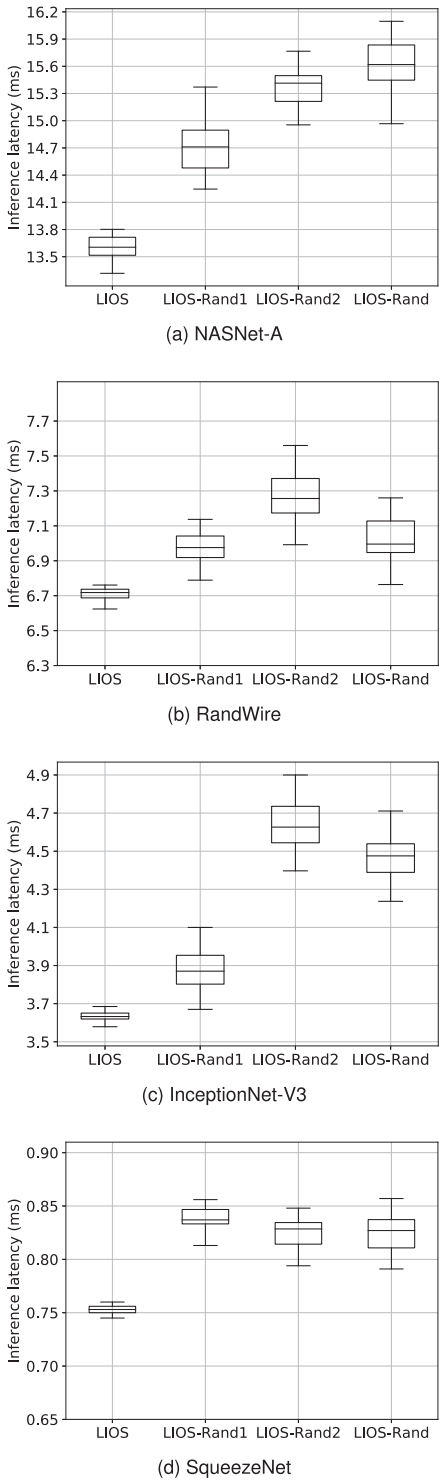


Fig. 8. Box plots of the inference latency of CNNs after being scheduled by LIOS, LIOS-Rand1, LIOS-Rand2, and LIOS-Rand.

We analyze the result from two perspectives. First, both the two heuristic strategies in LIOS are proved having positive contributions since LIOS produces shorter inference latency in all the CNNs compared to LIOS-Rand1 and LIOS-Rand2. Second, the heuristic strategy used in the stream selection phase has a bigger contribution than the strategy in the operator prioritizing

TABLE VI
SCHEDULING TIME (S) RESULT OF VARIATIONS OF LIOS, SEQUENTIAL SCHEDULING, SIMPLE GREEDY SCHEDULING, AND NIMBLE

	LIOS-Rand	LIOS-Rand1	LIOS-Rand2	Sequential	Simple Greedy	Nimble
SqueezeNet	0.10	0.09	0.09	2.46×10^{-5}	1.26×10^{-4}	0.08
InceptionNet-V3	0.48	0.46	0.46	1.21×10^{-4}	3.08×10^{-4}	0.51
RandWire	0.80	0.77	0.77	5.70×10^{-5}	2.93×10^{-4}	0.69
NASNet-A	2.54	2.56	2.42	2.27×10^{-4}	7.60×10^{-4}	1.99

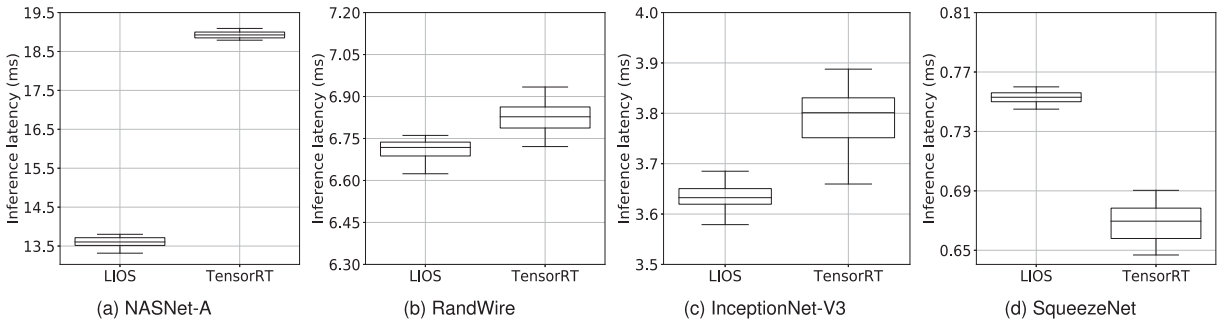


Fig. 9. Box plots of the inference latency of each CNN after being scheduled by LIOS and TensorRT

TABLE VII
AVERAGE INFERENCE LATENCY (MS) OF CNNS AFTER BEING OPTIMIZED BY LIOS, TENSORRT, AND APOLLO

	LIOS	TensorRT	Apollo
SqueezeNet	0.75	0.67	3.16
InceptionNet-V3	3.64	3.79	10.85
RandWire	6.71	6.83	51.11
NASNet-A	13.62	18.93	52.00

phase since LIOS-Rand1 produces shorter median latency than LIOS-Rand2 in most cases.

In addition, there are two findings. First, though having large latency variation, random scheduling (LIOS-Rand) can be a pretty good method sometimes. For example, in the best condition, the latency produced by LIOS-Rand can be close to LIOS on RandWire. Second, by observing the latency variation of each scheduling method, we can see that LIOS is more stable than all the variant methods.

Conclusion: Both the two heuristic strategies in LIOS have positive contributions. The heuristic in the stream selection phase has a bigger contribution in most cases. By combining the two heuristics, LIOS achieves higher inference latency reduction and stability.

D. Investigation of RQ4

In this RQ, we compare the inference latency provided by LIOS, TensorRT, and Apollo to give a better analysis of the effectiveness of LIOS.

Result: After being optimized by LIOS, TensorRT, and Apollo, the average inference latency of NASNet-A, RandWire, InceptionNet-V3, and SqueezeNet is shown in Table VII. The inference latency distribution of the benchmark CNNs after being optimized by LIOS and TensorRT is shown in Fig. 9(a), (b), (c), and (d). Overall, it can be seen that LIOS provides more latency reduction than TensorRT and Apollo.

First, from Table VII, it can be seen that Apollo has the worst performance and provides much longer latency than LIOS and TensorRT for all the CNNs. The huge performance gap may result from the difference in the fusion strategies (e.g., Apollo can only merge two reductions with the same tile size [34]) and runtime engines.

Second, we analyze the results of LIOS and TensorRT on NASNet-A, RandWire, and InceptionNet-V3. From Fig. 9(b), we can see that LIOS consistently outperforms TensorRT by providing shorter latency. For example, on NASNet-A, LIOS provides up to $1.59\times$ speedup compared to sequential scheduling. However, TensorRT even produces longer latency than sequential scheduling, which may result from the fact that certain operator types of NASNet-A are not supported by TensorRT. In contrast, the scheduling strategy of LIOS is not limited by the types of operators.

However, on SqueezeNet, LIOS shows its limitation that the extra time cost of stream synchronizations results in longer latency than sequential scheduling, just as we state in the result of RQ1. TensorRT can still provide latency reduction ($1.11 \sim 1.19\times$ speedup compared to sequential scheduling) in this situation because as an operator fusion technique, it is free from the extra time cost of executing operators in parallel.

Conclusion: In general, LIOS is more effective than TensorRT and Apollo by providing shorter inference latency for NASNet-A, RandWire, and InceptionNet-V3. While for small CNNs like SqueezeNet, TensorRT can be a better choice.

VII. DISCUSSION

From the experimental result in Section VI, it can be seen that LIOS can significantly reduce the inference latency of CNNs while maintaining short scheduling time. However, LIOS shows

its limitations in small-size CNNs, or CNNs without multiple branches such as VGG [35]. In this section, we provide a detailed discussion of the application scope for LIOS.

Generally, LIOS is suitable for DL models with multiple branches and certain-size operators. As the computation ability of hardware grows rapidly, this kind of network has been preferred since it can make better use of the hardware resources and provide better performance in certain scenarios. Additionally, rigid sequential structures designed by humans may constrain the search space and potential of CNNs [17], which makes CNNs with multiple branches important for various research fields. Therefore, though limited, the application scope of LIOS (inter-operator scheduling) is still broad.

For non-CNN workloads, such as transformers, inter-operator parallelism exists while the fixed computation graph makes automatic scheduler unnecessary. However, LIOS is effective for situations where CNN and transformer are combined [36], or potentially more complex transformer-based models in the future. For non-inference workloads, the training procedure of DL models, including forward propagation and back propagation, can be accelerated by LIOS. Forward propagation is the same as inference and back propagation can be seen as a reverse version of forward propagation.

Furthermore, inter-operator scheduling relies on the concurrent execution of operators to saturate the GPU resources, including threads. Therefore, if the GPU is already highly utilized, such as when employing a large operator or large batch size, inter-operator scheduling will not bring latency reduction and even increase it due to the additional cost of using multiple streams.

VIII. Related Work

In this section, we present the related work of accelerating the inference of deep learning models. First, we introduce the study of inter-operator scheduling, the research area to which our paper belongs. Then, we give an overview of deep learning compilers, which use computation graph optimization and operator tuning to provide speedup for the inference of DL models. Finally, we summarize the research in the model placement area, which focuses on the distributed system.

A. Inter-Operator Scheduling

In short, inter-operator scheduling is a kind of method that manipulates the execution order of operators in a DL model for different optimization purposes. There are two main categories. The first is latency-oriented and focuses on reducing the inference latency of CNNs by exploring the inter-operator parallelism. The second arises from the memory-constrained situation at the edge and tries to reduce the peak memory footprint of a DL model. LIOS belongs to the first category.

Latency-oriented inter-operator scheduling methods are usually designed for GPUs at cloud servers or PCs since the utilization of these GPUs is usually limited when executing the inference of a DL model. To solve this problem, Nimble [7] employs ahead-of-time (AOT) scheduling and parallel execution of operators for inference acceleration. However, the main contribution of Nimble is its AOT strategy and it uses a latency-insensitive scheduling method. Rammer [37] explores the parallelism in a different way. In Rammer, operators are separated into $rTask$, which is the basic scheduling unit. Thus, Rammer is not a pure operator-level method. Based on dynamic programming, IOS [10] is a scheduling method trying to find the optimal schedule. Different from the above methods, LIOS focuses on leveraging latency-based heuristics to find effective sub-optimal schedules and keep the scheduling time short.

Memory-oriented inter-operator scheduling methods are used to reduce the peak memory footprint for deep learning models deployed in edge devices, where the limited memory resource hinders the execution of large DL models. Serenity [9] leverages a dynamic programming algorithm to optimize the memory footprint of CNNs designed by the Neural Architecture Search (NAS) technique. HMCOS [38] improves Serenity by proposing a hierarchical method to greatly reduce the time cost of Serenity. However, instead of lowering memory usage, the main purpose of LIOS is to reduce the inference latency.

B. Deep Learning Compiler

Deep learning compilers aim at generating more efficient implementation code for DL models on different target hardware [39], [40]. In the process of transforming a high-level DL model into low-level hardware implementation code, lots of optimizations can be applied to reduce the inference latency of DL models. In general, these optimizations can be divided into two categories: operator tuning and computation graph transformation.

Operator tuning focuses on optimizing the implementation code of each operator in the DL model. Usually, a DL operator is composed of nested loops. For example, $Conv2D$, one of the most commonly used operators in CNNs, is made of loops since the essence of convolution is several matrix multiplication. Auto-TVM [11] proposes an automatic method based on manual templates to tune the parameters (e.g., tiling and reordering) in the execution of loops. In this way, Auto-TVM can achieve inference speedup better than vendor-provided libraries such as TensorRT [32]. To get rid of the dependency on manually written program templates, Ansor [12] uses a hierarchical representation and automatically builds the search space for loop optimizations. In addition, TC [41] is a fully automatic DL compiler based on the polyhedral model and can create a much bigger search space than TVM. Tiramisu [42] is also a polyhedral-based method, but it gives users more control by providing four kinds of scheduling commands.

In contrast to operator tuning, computation graph transformation focuses on optimizing the computation graph. The purpose of graph transformation is to generate functionally identical but more efficient substitutions for the original computation graph. TVM [21] divides operators into four categories and then applies operator fusion strategies to build a new graph. TASO [43] and MetaFlow [44] are DL compilers focusing on graph optimizations and can automatically generate graph substitutions. In TASO, instead of manually designed graph optimization strategies, a cost-based search algorithm is used to decide the used optimizations, e.g., operator fusion, operator splitting, and layout transformation.

Different from the inter-operator parallelism leveraged in LIOS, the above optimizations mainly explore the intra-operator parallelism. After applying these optimizations, the execution speed of a single operator is improved but operators in a DL model are still executed sequentially.

C. Model Placement

For DL services deployed at the edge, multiple computing units (e.g., GPUs and CPUs) are provided to meet the need for computing ability required by modern DL models, since a single computing resource is not powerful enough [45]. Thus, how to divide a DL model into subgraphs and place them into different computing resources for execution becomes a significant problem. The model placement problem consists of two sub-problems: graph partition and subgraph assignment.

We introduce two kinds of model placement methods. First, reinforcement learning (RL) based methods are proposed to solve the graph partition and subgraph assignment problem [46], [47], [48]. These methods can achieve good performance but lack versatility and require a huge time cost. To improve the versatility, new methods are proposed by combining RL and heuristics [49], [50]. Except for RL, methods based on integer programming [51] or heuristics [45] are also proposed.

The above methods focus on accelerating a DL model for the multi-processor system at the edge. In contrast, LIOS is designed for the single-GPU system at a cloud server or PC. LIOS can be used together with model placement methods since the inter-operator parallelism in each subgraph is still ignored by these methods. The co-using is possible because modern edge devices have strong computing ability. For example, NVIDIA Jetson AGX Xavier possesses 8 SMs, which is capable of handling the concurrent execution of a small number of operators.

IX. Conclusions and Future Work

Inter-operator scheduling is crucial to reduce the inference latency of CNNs in the GPU environment. However, existing inter-operator scheduling methods face a trade-off issue between the inference latency and scheduling time. In this paper, we propose LIOS, a latency-based inter-operator scheduling method to solve the conflict. In the experiment, LIOS can provide $1.13\sim 1.59\times$ inference speedup for normal-size and large-size CNNs, compared to sequential scheduling. This result is comparable to or even better than IOS. For the scheduling time, LIOS can speed up the scheduling process by $7\sim 9210\times$ compared with IOS. The experimental results show that LIOS can achieve a balance between the inference latency and scheduling time for normal-size and large-size CNNs.

In the future, we have the following directions. First, we plan to expand the scope of application from CNNs to all DL models. Second, we intend to study the possibility of combining LIOS with DL compilers to achieve a better inference acceleration. Third, we will try to leverage reinforcement learning methods to improve our heuristic method.

References

[1] C. Wang, X. Li, Q. Yu, A. Wang, P. Hung, and X. Zhou, "SOLAR: Services-oriented learning architectures," in *Proc. IEEE Int. Conf. Web Serv.*, 2016, pp. 662–665.

[2] B. Wang, L. Xu, M. Yan, C. Liu, and L. Liu, "Multi-dimension convolutional neural network for bug localization," *IEEE Trans. Serv. Comput.*, vol. 15, no. 3, pp. 1649–1663, May/Jun. 2022.

[3] J. Tian, J. Zhou, and J. Duan, "Hierarchical services of convolutional neural networks via probabilistic selective encryption," *IEEE Trans. Serv. Comput.*, vol. 16, no. 1, pp. 343–355, Jan./Feb. 2023.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[5] H. Zhuang, C. Wang, C. Li, Q. Wang, and X. Zhou, "Natural language processing service based on stroke-level convolutional networks for chinese text classification," in *Proc. IEEE Int. Conf. Web Serv.*, 2017, pp. 404–411.

[6] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 12, pp. 6999–7019, Dec. 2022.

[7] W. Kwon, G.-I. Yu, E. Jeong, and B.-G. Chun, "Nimble: Lightweight and parallel GPU task scheduling for deep learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 8343–8354.

[8] Y. Wu et al., "A comparative measurement study of deep learning as a service framework," *IEEE Trans. Serv. Comput.*, vol. 15, no. 1, pp. 551–566, Jan./Feb. 2022.

[9] B. H. Ahn, J. Lee, J. M. Lin, H.-P. Cheng, J. Hou, and H. Esmaeilzadeh, "Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices," *Proc. Mach. Learn. Syst.*, vol. 2, pp. 44–57, 2020.

[10] Y. Ding, L. Zhu, Z. Jia, G. Pekhimenko, and S. Han, "IOS: Inter-operator scheduler for CNN acceleration," *Proc. Mach. Learn. Syst.*, vol. 3, pp. 167–180, 2021.

[11] T. Chen et al., "Learning to optimize tensor programs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 3393–3404.

[12] L. Zheng et al., "Ansor: Generating High-Performance tensor programs for deep learning," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 863–879.

[13] B. H. Ahn, P. Pilligundla, A. Yazdanbakhsh, and H. Esmaeilzadeh, "Chameleon: Adaptive code optimization for expedited deep neural network compilation," in *Proc. Int. Conf. Learn. Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=rygG4AVFvH

[14] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 8697–8710.

[15] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," 2016, *arXiv:1602.07360*.

[16] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2818–2826.

[17] S. Xie, A. Kirillov, R. Girshick, and K. He, "Exploring randomly wired neural networks for image recognition," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2019, pp. 1284–1293.

[18] Y. Yang, W. Ke, W. Wang, and Y. Zhao, "Deep learning for web services classification," in *Proc. IEEE Int. Conf. Web Serv.*, 2019, pp. 440–442.

[19] Z.-Q. Zhao, P. Zheng, S.-T. Xu, and X. Wu, "Object detection with deep learning: A review," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 11, pp. 3212–3232, Nov. 2019.

[20] NVIDIA, "Cuda c programming guide," 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[21] T. Chen et al., "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 578–594.

[22] M. Abadi et al., "Tensorflow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[23] A. Paszke et al., "Automatic differentiation in pytorch," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst. Autodiff Workshop*, 2017. [Online]. Available: https://openreview.net/forum?id=BJJsrmfCZ

[24] C.-F. Tsai, C.-W. Tsai, and C.-C. Tseng, "A new hybrid heuristic approach for solving large traveling salesman problem," *Inf. Sci.*, vol. 166, no. 1/4, pp. 67–81, 2004.

[25] N. Agatz, P. Bouman, and M. Schmidt, "Optimization approaches for the traveling salesman problem with drone," *Transp. Sci.*, vol. 52, no. 4, pp. 965–981, 2018.

[26] S. Chetlur et al., "cuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.

[27] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 4278–4284.

[28] A. Vahdat, A. Mallya, M.-Y. Liu, and J. Kautz, "UNAS: Differentiable architecture search meets reinforcement learning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 11263–11272.

[29] P. Ren et al., "A comprehensive survey of neural architecture search: Challenges and solutions," *ACM Comput. Surv.*, vol. 54, pp. 1–34, 2021.
[30] J. Zhao et al., "Apollo: Automatic partition-based operator fusion through layer by layer optimization," *Proc. Mach. Learn. Syst.*, vol. 4, pp. 1–19, 2022.
[31] Huawei, "Mindspore," 2020. [Online]. Available: https://www.mindspore.cn/en
[32] NVIDIA, "Nvidia tensorrt: Programmable inference accelerator," 2018. [Online]. Available: https://developer.nvidia.com/tensorrt
[33] A. Li, B. Zheng, G. Pekhimenko, and F. Long, "Automatic horizontal fusion for GPU kernels," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2022, pp. 14–27.
[34] C. Xia et al., "Optimizing deep learning inference via global analysis and tensor expressions," in *Proc. ACM Int. Conf. Architectural Support Program. Lang. Operating Syst*, 2023. [Online]. Available: https://eprints.whiterose.ac.uk/203681/
[35] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[36] C. Si, W. Yu, P. Zhou, Y. Zhou, X. Wang, and S. Yan, "Inception transformer," in *Proc. Adv. Neural Inf. Process. Syst.*, 2022, pp. 23495–23509.
[37] L. Ma et al., "Rammer: Enabling holistic deep learning compiler optimizations with rTasks," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 881–897.
[38] Z. Wang, C. Wan, Y. Chen, Z. Lin, H. Jiang, and L. Qiao, "Hierarchical memory-constrained operator scheduling of neural architecture search networks," in *Proc. 59th ACM/IEEE Des. Automat. Conf.*, 2022, pp. 493–498.
[39] M. Li et al., "The deep learning compiler: A comprehensive survey," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 708–727, Mar. 2021.
[40] Y. Ding, C. H. Yu, B. Zheng, Y. Liu, Y. Wang, and G. Pekhimenko, "Hidet: Task-mapping programming paradigm for deep learning tensor programs," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 370–384.
[41] N. Vasilache et al., "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," 2018, *arXiv: 1802.04730*.
[42] R. Baghdadi et al., "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2019, pp. 193–205.
[43] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 47–62.
[44] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken, "Optimizing DNN computation with relaxed graph substitutions," *Proc. Mach. Learn. Syst.*, vol. 1, pp. 27–39, 2019.
[45] P. Lin, Z. Shi, Z. Xiao, C. Chen, and K. Li, "Latency-driven model placement for efficient edge intelligence service," *IEEE Trans. Serv. Comput.*, vol. 15, no. 2, pp. 591–601, Mar./Apr. 2022.
[46] A. Mirhoseini et al., "Device placement optimization with reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 2430–2439.
[47] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *Proc. Int. Conf. Learn. Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=Hkc-TeZ0W
[48] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1676–1684.
[49] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in parallelizing convolutional neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 2279–2288.
[50] S. Bojja Venkatakrishnan et al., "Learning generalizable device placement algorithms for distributed machine learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/71560ce98c8250ce57a6a970c9991a5f-Abstract.html
[51] N. D. Lane et al., "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in *Proc. 15th ACM/IEEE Int. Conf. Inf. Process. Sensor Netw.*, 2016, pp. 1–12.

Yukai Ping received the bachelor's degree in computer science from Zhejiang Sci-Tech University, Hangzhou, China. He is currently working toward the master's degree in software engineering with the School of Software, Dalian University of Technology, Dalian, China. His current interests include machine learning and deep learning system.

He Jiang (Member, IEEE) received the PhD degree in computer science from the University of Science and Technology of China, Hefei, China. He is currently a professor with the Dalian University of Technology, Dalian, China. He is also a member of the ACM and the CCF (China Computer Federation). He is one of the ten supervisors for the Outstanding Doctoral Dissertation of the CCF, in 2014. His current research interests include intelligent software engineering, software testing with focus on system software, and search-based software engineering (SBSE). His work has been published at premier venues like ICSE, FSE, and ASE, as well as in major IEEE Transactions like the *IEEE Transactions on Software Engineering*, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Transactions on Cybernetics*, and *IEEE Transactions on Services Computing*.

Xingxiang Liu received the bachelor's degree from Yangtze University, Jingzhou, China. He is currently a senior engineer with Sangfor Technologies Inc., Shenzhen, China. His current interests include optimizing the training and inference of AI models.

Zhenyang Zhao received the bachelor's degree from Xi'an University of Posts and Telecommunications, Xi'an, China. He is currently a senior engineer with Sangfor Technologies Inc., Shenzhen, China. His current interests include cloud computing, edge computing, and edge intelligence.

Zhide Zhou received the PhD degree in software engineering from Dalian University of Technology, Dalian, China. He is currently a research associate with OSCAR (Optimizing Software by Computation from ARtificial intelligence) research group at Dalian University of Technology, headed by Prof. He Jiang. His current research interests include intelligent software engineering, program analysis techniques, deep learning compiler, and autonomous driving systems testing.

Xin Chen (Associate Member, IEEE) received the PhD degree in software engineering from the School of Software, Dalian University of Technology, Dalian, China. He is currently an Associate Professor with Hangzhou Dianzi University, Hangzhou, China. He is a member of the CCF and the ACM. His research interests include smart software engineering, mining software repositories, and evolutionary computation.