TOP: Task-Based Operator Parallelism for Asynchronous Deep Learning Inference on GPU

Changyao Lin, *Graduate Student Member, IEEE*, Zhenming Chen, Ziyang Zhang, *Graduate Student Member, IEEE*, and Jie Liu, *Fellow, IEEE*

***Abstract*—Current deep learning compilers have made significant strides in optimizing computation graphs for single- and multi-model scenarios. However, they lack specific optimizations for asynchronous multi-task inference systems. In such systems, tasks arrive dynamically, leading to diverse inference progress for each model. This renders traditional optimization strategies based solely on the original computation graph suboptimal or even invalid. Furthermore, existing operator scheduling methods do not account for parallel task pipelines involving the same model. Task pipelines present additional opportunities for optimization. Therefore, we propose Task-based Operator Parallelism (TOP). TOP incorporates an understanding of the impact of task arrival patterns on the inference progress of each model. It leverages the multi-agent reinforcement learning algorithm MADDPG to cooperatively optimize the task launcher and model scheduler, generating an optimal pair of dequeue frequency and computation graph. The objective of TOP is to enhance resource utilization, increase throughput, and allocate resources judiciously to prevent task backlog. To expedite the optimization process in TOP, we introduce a novel stage partition method using the GNN-based Policy Gradient (GPG) algorithm. Through extensive experiments on various devices, we demonstrate the efficacy of TOP. It outperforms the state-of-the-art in operator scheduling for both single- and multi-model task processing scenarios. Benefiting from TOP, we can significantly enhance the throughput of a single model by increasing its concurrency or batch size, thereby achieving self-acceleration.**

***Index Terms*—Multi-task inference, operator scheduling, reinforcement learning.**

I. Introduction

IN CONTEMPORARY AI systems, the concurrent processing of tasks for multiple data sources is a prevalent and challenging requirement. These applications span a wide spectrum, encompassing edge devices within expansive intelligent manufacturing lines, servers hosting multiple AI services, and embedded AI systems in autonomous vehicles. In such scenario, a multitude of terminals continually dispatch AI inference tasks to a central server. These tasks may require different deep neural networks (DNNs, models) for processing, with different requirements for resources and real-time performance. How to optimize from different levels to process the tasks more efficiently is currently a hot research topic.

The development of software and hardware provides a lot of support and opportunities for multi-task inference [1], [2], [3], [4]. The existing efficient deep learning inference frequently combines the characteristics of hardware and software for collaborative optimization [5], and can be categorized into model- [6], [7], layer- [8], and operator-level [9], [10], [11] scheduling. Due to the different layers or operators within DNNs, coarse-grained model-level scheduling cannot fully account for the fluctuated resource consumption. Scheduling at a finer-grained layer- or operator-level can address this shortcoming [12]. The finer the granularity, the more flexible the scheduling, and the more optimization space is introduced, so the optimization cost is also higher.

A DNN with many operators can be abstracted as a directed acyclic graph (DAG), also known as a computation graph, where nodes represent operators, and edges represent data flows and dependencies between operators. The DNN compiler can formulate and optimize the execution strategy for operators on the computation graph. Some work optimizes a single model [10], [13], [14]. However, there are few branches in a single model, so the schedulable space and parallelism are limited, and peak performance cannot be achieved on more and more resource-rich hardware today. In contrast, multi-task deep learning computing with multiple parallel DAGs usually has extensive inter-operator parallelism, which makes the scheduling among models more flexible. This kind of graph scheduling has certain challenges, such as the increased complexity in larger number of operators and scheduling space, and the more complex GPU resource contention, etc [12].

For the operator scheduling of multi-model, it is more suitable to combine the computation graphs of all models for optimization. This kind of work generally assumes that all models share the input, then formulates the strategy and configures the computation graph offline to execute multitask inference on the input at the same time. During online execution, all models need to be synchronized [11], [15], [16]. However, for multi-task inference

Received 16 April 2024; revised 30 November 2024; accepted 2 December 2024. Date of publication 5 December 2024; date of current version 30 December 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62350710797 and in part by the Science and Technology Plan Project of Shenzhen through Project Number JSGG20220831110002004. Recommended for acceptance by B. Ucar. *(Corresponding author: Jie Liu.)*

Changyao Lin and Ziyang Zhang are with the School of Computer Science and Technology, Harbin Institute of Technology, Harbin, Heilongjiang 150001, China (e-mail: lincy@stu.hit.edu.cn; zhangzy@stu.hit.edu.cn).

Zhenming Chen is with the China Construction Steel Structure Engineering Corp., LTD, Shenzhen, Guangdong 518118, China (e-mail: chenzm@cscec.com).

Jie Liu is with the National Key Laboratory of Smart Farm Technologies and Systems, Harbin Institute of Technology, Harbin, Heilongjiang 150001, China (e-mail: jieliu@hit.edu.cn).

Digital Object Identifier 10.1109/TPDS.2024.3511543

systems, sometimes the models need to be executed for different data sources. In this case, each model cannot share the input. The asynchronization of input data leads to offline optimization being difficult to adapt to the dynamically arrived tasks, resulting in invalid scheduling or excessive synchronization overhead. In addition, the existing single- and multi-model optimization methods do not consider the situation of parallel task pipeline for the same model, which will introduce more optimizable space.

Several existing graph- or operator-level scheduling schemes are compiled offline [10], [11], [15], which divide the suggested concurrent operators into the same stage. The generated computation graph is no longer modified at runtime, causing a gap between static compilation and dynamic scenarios in real-world applications, especially when on-device resources are dynamically available. For the typical multi-source task-based inference system, as tasks arrive dynamically, the inference progresses of task instances for each model are different. Therefore, the operator parallel strategy cannot be pre-determined only according to the DAG of each model, which may be suboptimal or even invalid. To fill this gap, a dynamic compilation method is urgently needed to ensure that the on-device resources are fully utilized in an adaptive way at runtime.

One idea is to progressively schedule the current operators and configure the computation graph during inference, but its runtime overhead is too large. Therefore, we comprehensively consider model characteristics, task patterns, and hardware resources to explore how to design parallel strategies that improve the utilization of hardware, thereby increasing the efficiency of processing asynchronous tasks. We innovatively utilize the impact of task pattern on the inference progress of each model, and designs the operator scheduling strategy that conforms to the inference progress according to the task pattern. Compared with existing work, we can schedule more effectively and reduce synchronization overhead.

Specifically, based on multi-agent reinforcement learning algorithm MADDPG [17], this paper proposes Task-based Operator Parallelism (TOP). The TOP framework cooperatively optimizes the task launcher and model scheduler to generate an optimal pair of dequeue frequency and computation graph under the current state. The optimization objective is to enhance resource utilization, increase throughput, and reasonably allocate resources to avoid task backlog. The main contributions of this paper are summarized as follows:

- We are the first to schedule operators according to task pattern, and consider the task pipeline. We achieve efficient collaborative optimization among model-, layer-, and operator-level.
- We propose a task-based operator parallelism framework TOP to effectively schedule operators for asynchronous tasks in inference systems. TOP is applicable to single- and multi-model optimization, and can also handle the case of synchronous multitask.
- In order to reduce the overall optimization time of TOP, we utilize GNN-based Policy Gradient (GPG) algorithm to propose a novel stage partition method.
- Through evaluation on various heterogeneous models and devices, we verify that the proposed method outperforms the state-of-the-art in both asynchronous-single-model and asynchronous-multi-model scenarios. We also verify the feasibility of improving the throughput for a single model by increasing its concurrency or batch size, and TOP can increase the benefit.

II. Related Work

Operator parallelism for single-model acceleration: Some current mainstream deep learning inference frameworks take use of operator-level parallelism to speed up inference. Before executing operators, TensorFlow [18] builds a tensor-based static computation graph through the compilation phase, and then executes the operators sequentially. PyTorch [19] is a deep learning framework that can dynamically configure computation graphs, which means that it does not need to generate computation graphs through the compilation phase. Operators are also executed sequentially in PyTorch. As a machine learning compiler for optimizing inference, TVM [13] abstracts DNN into a unified intermediate representation, and takes use of the learning-based scheduler to generate an optimal schedule to execute operators efficiently on GPU. The above frameworks focus on optimizing the intra-operator parallelism. Due to the limited parallelism within the operator, the hardware utilization is usually low. Therefore, some work began to optimize the inter-operator parallelism to improve the utilization of hardware and further accelerate inference. TensorRT [14] accelerates inference through a series of technologies (such as pruning, quantization, operator fusion, etc.), and constructs computation graphs according to multi-stream processing technology of GPU to execute operators in parallel. IOS [10] takes advantage of dynamic programming (DP) to divide CNN into multiple stages, and each stage utilizes operator merge or concurrent execution to accelerate inference.

Both intra-operator parallelism and inter-operator parallelism can improve the utilization of hardware, thus speeding up inference. However, the above work is aimed at a single model, with few branches and limited schedulable space. On today's more and more resource-rich hardware, peak performance cannot be achieved.

Operator parallelism for multi-model acceleration: Multi-model inference has numerous parallelizable branches, which makes inter-operator scheduling more flexible, but also brings more challenges, such as the increase of scheduling space and resource contention among models. Wang et al. [20] proposed Horizontally Fused Training Array (HFTA) for model training. HFTA fuses operators of the same type and shape in multiple models, then trains the fused model on a shared accelerator to improve hardware utilization and speed up model training. Yu et al. [15] proposed an efficient resource-aware scheduling framework for the multi-tenant DNN inference on GPU. The framework utilizes the unified intermediate representation and learning-based search algorithm to generate the optimal schedule, so that the appropriate operators in various DNNs can be centralized in the same stage for parallel execution. The algorithm can maintain a balanced resource utilization throughout the inference process, and eventually improve the runtime

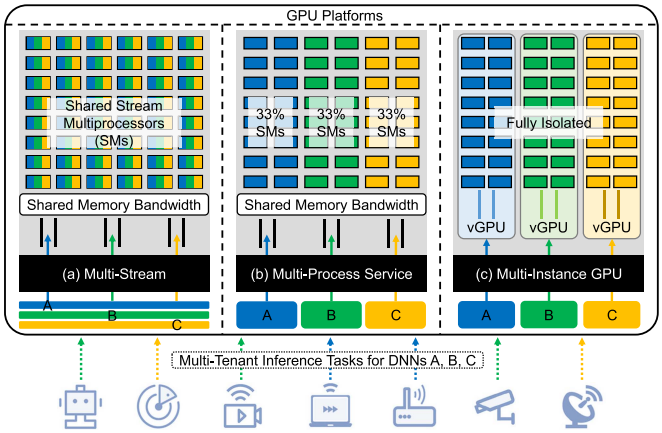


Fig. 1. Multi-task deep learning inference on GPU.

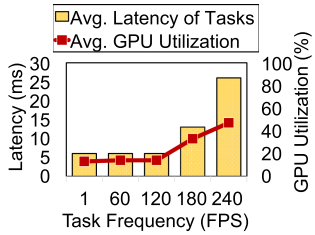


Fig. 2. The test results of ResNet-50 optimized by IOS in the asynchronous-single-model scenario.

efficiency. Zhang et al. [11] proposed a framework POS that addresses the optimization of multi-model inference in edge AI applications. By combining four operator scheduling strategies, POS achieves significant improvements in both inference speed and GPU utilization.

Current multi-model operator-level acceleration techniques are designed for offline static optimization in synchronous tasks. However, when the task frequencies of the models are different (i.e., asynchronous tasks), the inference progresses of task instances are dynamically changing. Consequently, offline optimization is not applicable, and the static computation graphs may result in ineffective scheduling or substantial synchronization overhead.

III. Background and Motivation

A. Multi-Task Inference System

As shown in Fig. 1, today's GPU platforms are increasingly rich in computing and memory resources, with corresponding software providing high concurrency support to users, such as NVIDIA's CUDA Multi-Stream [2], Multi-Process Service [3], and Multi-Instance GPU [4]. With the collaborative development of hardware and software, it is now possible to deploy multiple DNNs on a single machine to process the inference requests for different data sources concurrently [6]. Such devices with certain computility that can be deployed with multiple inference services usually act as the server. The data sources are typically various terminal devices that act as clients, which are usually resource-constrained devices that lack computility and can only collect data (such as sensors). Therefore, the client needs to continuously send data to the server to assign inference tasks, and the server needs to efficiently process the requests from different data sources [12]. These requests (tasks) can be seen as inference instances of different DNNs deployed, so their resource overheads are different.

Many AI+IoT (AIoT) scenarios at the edge or in the cloud can be abstracted into this client-server pattern [21]. For example, edge devices in large intelligent manufacturing lines, servers deployed with multiple AI services in data centers, and embedded devices on autonomous vehicles need to continuously receive data from various sources and invoke the corresponding DNNs for inference.

B. Computation Graph and Operator-Level Inference Acceleration

In order to provide high-quality inference services and mitigate resource contention across different tasks, the server-side devices require coordination of software and hardware for resource scheduling at different levels. Fine-grained-level scheduling can flexibly consider fluctuated resource consumption [12]. Compared with model- and layer-level scheduling, operator-level scheduling performs better for mixed models with different complexities. This is because under coarse-grained scheduling, the execution of large models may inhibit the execution of small models, which can be alleviated by fine-grained scheduling. Therefore, lots of work goes deep into operator-level optimization [10], [11], [15].

The computation graph is an intermediate representation that connects the DNNs with the underlying execution engine. A DNN model can be represented as a computation graph, where vertex set is an abstraction of operator set, and edge set represents the dependency between operators. According to the characteristics of DNN, the computation graph is a directed acyclic graph (DAG). Each operator in the graph can be a convolution calculation or matrix multiplication, etc. The edge (u, v) from vertex u to vertex v is a tensor that is the output of operator u and the input of operator v . Generally, due to the difference in operator types and input shapes, the operators have different computational loads and resource overheads. During the inference process, the operators will be executed sequentially or concurrently according to the strategies formulated on the computation graph, to enhance inference efficiency and resource utilization while mitigating the contention.

C. Acceleration Test for Asynchronous Inference in Single-Model Scenario

In the scenario of multi-task inference, we first test the performance of the advanced single-model operator scheduling method IOS [10]. We preload ResNet-50 [22] on NVIDIA GeForce RTX 3080 [23] and utilize IOS to schedule the operators, then assign inference tasks to it at different frequencies. As shown in Fig. 2, when the task launch frequency is very low (e.g., 1 FPS), the average inference latency is 6ms. Therefore,

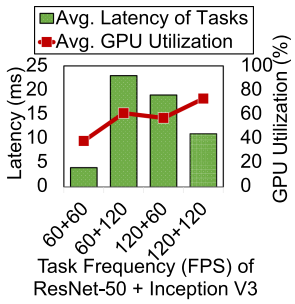


Fig. 3. The test results of ResNet-50 and Inception V3 optimized by POS in the asynchronous-multi-model scenario.

when the task launch frequency is less than $\frac{1}{0.006}(\approx 167)$ FPS, there is no temporal overlap among inference instances, i.e., no pipeline parallelism among tasks, so they do not affect each other, and the average inference latency is the lowest, but due to the low parallelism, the GPU utilization is very low. When the task launch frequency is greater than 167 FPS, pipeline parallelism occurs among instances. However, IOS does not take into account the pipeline parallelism, resulting in resource contention even when the GPU utilization is below 50%, ultimately leading to an increase in average task latency. Therefore, there is still room for operator-level optimization in the asynchronous inference scenario.

D. Acceleration Test for Asynchronous Inference in Multi-Model Scenario

Due to the limited branches, a single model cannot fully utilize resources to achieve the peak performance on resource-rich hardware even after optimization. While multi-model inference can introduce more parallel branches and improve hardware utilization, but it also brings more severe resource contention.

Similarly, we also test the average inference latency and GPU utilization under different task frequencies in the asynchronous-multi-model scenario, after optimizing with the state-of-the-art multi-model operator scheduling method POS [11]. Taking two models, ResNet-50 and Inception V3 [24], as an example, we preload them on NVIDIA GeForce RTX 3080. After optimization with POS, we assign inference tasks to the two models at different frequencies. As shown in Fig. 3, the GPU utilization in the multi-model scenario does increase by around 25% compared to the single-model scenario. However, when the frequency difference between the two models increases (e.g., $60 + 120$ and $120 + 60$), despite the overall task frequency decreasing (compared to $120 + 120$), the average inference latency increases, and the GPU utilization is only around 60%. In addition to not considering the optimization of pipeline parallelism, this is because multi-model optimization methods like POS assume that all models share the input, i.e., they aim at synchronous tasks, which makes it easier to perform offline optimization according to the original computation graph and hardware platform. However, when the task frequencies of the models are different (i.e., asynchronous tasks), the inference progress of task instances is diverse, so the optimization is not applicable. The static computation graph may result in ineffective scheduling or significant synchronization overhead.

E. Summary

For multi-task inference systems, existing operator-level single- and multi-model optimization methods do not consider the situation of pipeline parallelism among the tasks of the same model, which will introduce more optimization potential. Compared to single-model optimization, multi-model optimization can improve hardware utilization, but the existing operator-level optimization assumes that the inputs of all models are synchronous. For some multi-source inference systems, different models need to take use of different data for inference. In such cases, the models cannot share the input. The asynchronization of input data makes it difficult for offline optimization to adapt to the dynamically arriving tasks, leading to invalid scheduling or excessive synchronization overhead. Therefore, for such asynchronous tasks, it is necessary to design operator scheduling strategies that are consistent with the inference progress of each model according to the task pattern, and make reasonable resource allocation to avoid the backlog of high-frequency tasks. Dynamically scheduling at such a fine-grained level is a great challenge, so we have to design a very efficient search framework.

IV. Problem Definition

Model, task, queue, and task launcher: Suppose multiple DNN models $\mathcal{W} = \{w_1, w_2, \ldots, w_W\}$ are preloaded on the device to process various tasks [6]. The device has W task queues, which cache the inference tasks $\mathcal{Q}^{w_i} = \langle q_0^{w_i}, q_1^{w_i}, q_2^{w_i}, \ldots \rangle$ of each model. The task launcher can adjust the dequeue frequency of each queue to control the congestion of tasks under the premise of maintaining the high utilization of GPU, that is, it can control the number of current inference instances according to the resource situation. The dequeued tasks will be distributed to the corresponding models for concurrent inference.

Block: Modern DNNs are typically constructed by stacking multiple blocks, allowing each block to be optimized separately. For example, each residual block of ResNet [22] is a block, each Inception module of Inception V3 [24] is a block, and each Fire module of SqueezeNet [25] is a block.

Pipeline parallelism and inference progress: As shown in Fig. 4, let $\mathcal{B}^{w_i} = \{b_1^{w_i}, b_2^{w_i}, \ldots, b_{B_i}^{w_i}\}$ be the set of blocks of model w_i. When models $\mathcal{W}$ execute tasks concurrently, at a certain time t, there are N inference instances (i.e., the tasks being inferred) $\mathcal{I}_t^{w_i} = \{q_{x_1}^{w_i}, q_{x_2}^{w_i}, \ldots, q_{x_N}^{w_i}\}$ of model w_i. An instance can be a batch of inferences. Suppose these instances infer to blocks $\mathcal{B}_t^{w_i} = \{b_{y_1}^{w_i}, b_{y_2}^{w_i}, \ldots, b_{y_N}^{w_i}\}$ respectively, then the current inference progress of model w_i is $\mathcal{P}_t^{w_i} = \mathcal{B}_t^{w_i}$. In order to reduce the scheduling space, here the block is the pipeline granularity.

Inter-block scheduling for asynchronous tasks: Our objective is to jointly optimize the task launcher and inter-block scheduler, so that they can generate an optimal pair of dequeue frequency and block combination scheme according to the current state. The objective is to enhance resource utilization, increase system

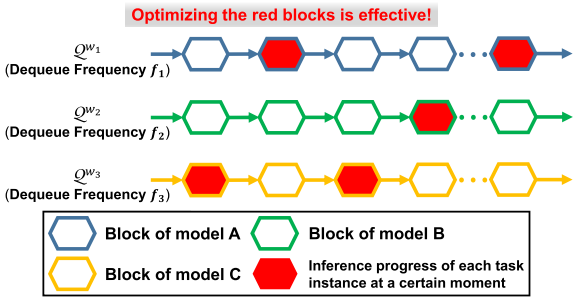
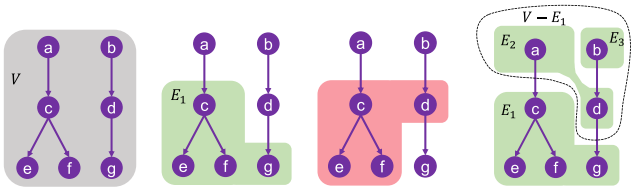


Fig. 4. Under the specific task arrival pattern, each DNN has different processing progresses. We need to design an effective operator parallel strategy according to the progresses.



(a) Operators V (b) E_1 is an ending of V (c) Not an ending of V (d) Partition the graph by endings recursively

Fig. 5. E_i is an ending of V if and only if there is no edge from E_i to $V - E_i$.

throughput, and reasonably allocate resources to avoid task backlog.

Specifically, for inter-block scheduling, we combine some blocks of all models for subsequent intra-block operator scheduling, and finally find the optimal combined block for each block. Each combined block corresponds to an inference progress $\mathcal{P}_t = \{\mathcal{P}_t^{w_1}, \mathcal{P}_t^{w_2}, \ldots, \mathcal{P}_t^{w_W}\}$.

For task launcher, we determine the optimal dequeue frequency of various tasks according to the impact of task pattern on the inference progress of each model, so as to control the inference progress and make the operator scheduling effective. The task launcher also acts as a resource allocator to avoid the backlog of certain tasks by adjusting the dequeue (processing) frequency of various tasks. At the same time, the task launcher should achieve congestion control on the premise of ensuring resource utilization to avoid hardware idle due to low task frequency or resource contention due to high task frequency.

Intra-block scheduling: We continue to optimize fine-grained operator parallelism in the combined block. Here we imitate the idea in IOS [10] (as shown in Fig. 5), and recursively select the endings of vertex (operator) set V in the combined subgraph to partition it into multiple stages. Vertex subset E_i is the ending of V if and only if there is no edge from E_i to $V - E_i$, thereby ensuring that dependent operators are executed sequentially. As shown in Fig. 6, the stages are executed sequentially, from top to bottom. The operators with edges connected in the stage belong to the same group, and the operators in the group are executed sequentially due to dependencies. Multiple groups can be formed in a stage, and the groups are executed in parallel. In this way, we can limit resource allocation and contention within the stage.

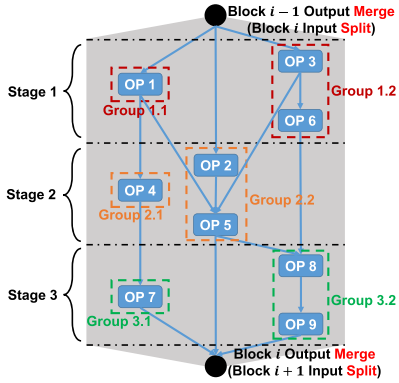


Fig. 6. Block, stage, group diagram. After intra-block operator scheduling, block i is divided into 3 stages, and the 3 stages are executed sequentially. In stage 1, operator 3 and operator 6 belong to the same group, so they are executed sequentially, while concurrently with operator 1in another group. Similarly, in stage 2, operator 2 and operator 5 are executed sequentially, while concurrently with operator 4. In stage 3, operator 8 and operator 9 are executed sequentially, while concurrently with operator 7.

In TOP, tasks of the same model are pipelined in parallel by blocks, so the transitive (dependent) closure of vertices (operators) among different instances of the same model can be cut off, which will lead to more optional endings in the combined block and improve parallelism.

The problem is defined as: how to select ending E_i at each step to minimize the overall execution latency of V. The dynamic programming algorithm used by IOS selects a more effective parallel strategy according to the cost function. The cost function is defined as the latency of running the computation graph. However, in our TOP scenario, the scheduling space of the combined block is much larger than that of the original block, resulting in a long search time for dynamic programming. Therefore, this paper proposes a learning-based stage partitioning method, which pre-trains an intra-block scheduler. In the subsequent stage partitioning, there is no need to traverse all situations.

V. Methodology

Due to the vast scheduling space, it is impractical to directly schedule all operators of the model set. Furthermore, since the structure of DNNs is diverse, manual schedule tuning requires considerable effort and lacks scalability. Therefore, we utilize multi-agent reinforcement learning algorithm MADDPG and GNN-based Policy Gradient (GPG) algorithm to achieve efficient collaborative optimization among model-level (task launcher), layer-level (inter-block scheduling), and operator-level (intra-block scheduling).

MADDPG is used to coordinate the task launcher and model scheduler. The model scheduler includes inter-block scheduling and intra-block scheduling. Inter-block scheduling directly collaborate with the task launcher, while GPG is used to perform more fine-grained intra-block graph encoding and operator scheduling on the results of inter-block scheduling, generating the final computation graph, so as to affect the performance (reward) of MADDPG's decision-making at each step.

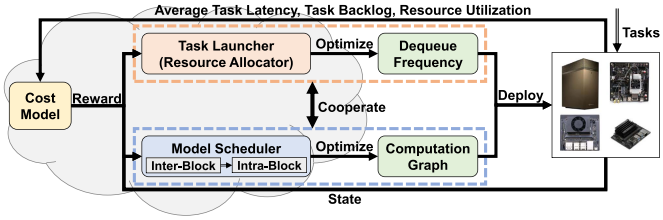


Fig. 7. The offline scheduling framework of TOP.

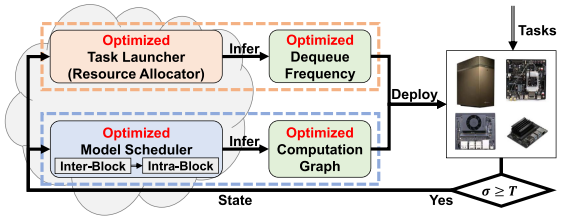


Fig. 8. The online execution framework of TOP.

A. Framework

As shown in Figs. 7 and 8, the TOP framework is divided into two phases: offline scheduling and online execution.

In the offline scheduling phase, the task launcher and model scheduler jointly optimize the dequeue frequency and computation graph, and obtain the specific state and task processing results by deploying the optimization results on a specific device. The cost model calculates the reward through the task processing results and feeds it back to the task launcher and model scheduler for further learning and optimization.

In the online execution phase, the optimal task launcher and model scheduler have been obtained through offline training. They can infer the optimal dequeue frequency and computation graph under the current state. The dequeue frequency and computation graph are re-inferred and deployed only when the variance of the number of tasks in each queue exceeds a certain threshold T. This can not only avoid the task backlog, but also eliminate the runtime overhead as much as possible.

B. MADDPG-Based Frequency Adjustment and Block Combination

MADDPG [17] models the optimization as a multi-agent Markov Decision Process (MDP), selecting an optimal action at each step to transition the state to a state that is conducive to the cumulative future reward. After convergence (Nash equilibrium), the algorithm can determine the optimal dequeue frequency of each queue, and find the optimal combined block for each block according to the current state. The specific components are described as follows:

S (State): during the execution of tasks, we randomly sample n times the inference progress $\mathcal{P}_t$ of all models and the deviation $d_1, d_2, \ldots, d_W$ of the number of tasks in each queue, then stack n samples into state s_t, with dimension $n \times (W + \sum_{i=1}^{W} B_i)$. Equation (1) is a concrete example, where 1 corresponds to the progress.

$$s_t = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ \vdots \\ n \end{array} \begin{array}{c} \begin{array}{cccccccc} b_1^{w_1} & b_2^{w_1} & \cdots & b_{B_W}^{w_W} & d_1 & \cdots & d_W \end{array} \\ \begin{pmatrix} 1 & 0 & \cdots & 1 & d_{1,1} & \cdots & d_{1,W} \\ 0 & 1 & \cdots & 0 & d_{2,1} & \cdots & d_{2,W} \\ 0 & 0 & \cdots & 1 & d_{3,1} & \cdots & d_{3,W} \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ 1 & 0 & \cdots & 0 & d_{n,1} & \cdots & d_{n,W} \end{pmatrix} \end{array} \tag{1}$$

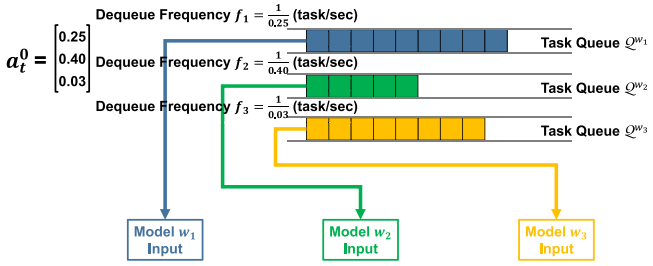


Fig. 9. An example of task launcher action a_t^0 ($W = 3$).

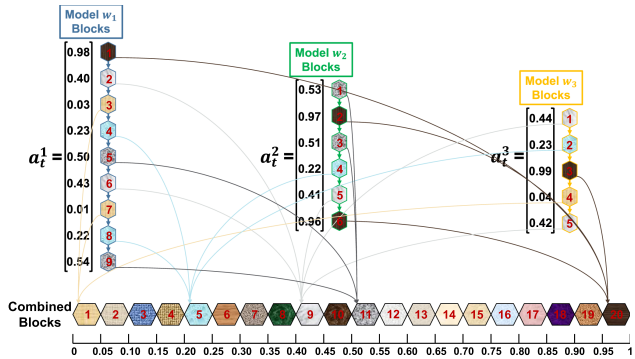


Fig. 10. An example of model scheduler action $\{a_t^1, a_t^2, \ldots, a_t^W\}$ ($W = 3$). The three models are divided into five combined blocks.

A (Action): one agent acts as the task launcher, and the action a_t^0 is the task launch interval (sec/task) corresponding to the W models, so the dimension is W. We limit the action value to (0,1], that is, the minimum dequeue frequency of each queue is 1 task/sec. Fig. 9 is an example of three models (queues).

W agents act as the model scheduler, responsible for W models respectively, and the actions are $a_t^1, a_t^2, \ldots, a_t^W$. Fig. 10 is an example of three models divided into five combined blocks. The action value of each agent is limited to [0,1], and the dimension is the number of blocks in model w_i (usually less than 10). According to the total number of blocks, [0,1] is divided into uniform intervals. When the action value falls in a certain interval, the corresponding block is combined with other blocks falling in the interval. If there is only one block in the interval, the block will not be combined with any other blocks. If there is no block in an interval, the combined block is invalid. By dividing the interval, it also tolerates certain action prediction errors.

Algorithm 1: MADDPG-Based Inter-Block Scheduling.

Input: $s_t, \overline{\xi}_t^*, \overline{l}_t^*, \sigma_t^*$ of each step t, original computation graphs $G_1^{ori}, G_2^{ori}, \cdots, G_W^{ori}$, thresholds $T, H, replay\_size$

1 Initialize system parameters;
2 **Offline scheduling**:
3 **while true do**
4 **for** i←*0 to* W **do**
5 $a_{t-1}^i \leftarrow \rho^i(s_{t-1})$;
6 **end for**
7 Combine blocks according to $a_{t-1}^1, a_{t-1}^2, \cdots, a_{t-1}^W$;
8 **if** *maximum size of the combined blocks* $> H$ **then**
9 $r_{t-1} \leftarrow 0$;
10 **else**
11 Use Algorithm 2 to optimize each combined block, get the $stage\_str$, and store it in $stage\_dict$;
12 Reconstruct combined blocks according to $stage\_str$ to build a new computation graph $G_{combined}^{new}$;
13 Each queue dequeues tasks according to the frequency a_{t-1}^0;
14 Use $G_{combined}^{new}$ to process the tasks concurrently;
15 Sample n groups of progress and deviation to form s_t;
16 Calculate r_{t-1} using Eq. (2);
17 **end if**
18 $a_{t-1} \leftarrow \{a_{t-1}^0, a_{t-1}^1, \cdots, a_{t-1}^W\}$;
19 Store transition $(s_{t-1}, a_{t-1}, r_{t-1}, s_t)$ in replay buffer $\mathcal{D}$;
20 **if** len($\mathcal{D}$)$\geq replay\_size$ **then**
21 Sample batches of transitions from $\mathcal{D}$ to update $Q_\rho^0, Q_\rho^1, \cdots, Q_\rho^W, \rho^0, \rho^1, \cdots, \rho^W$ using Eqs. (3) and (4);
22 **end if**
23 $t \leftarrow t + 1$;
24 **end while**
25 **Online execution**:
26 $t \leftarrow 0$;
27 **while true do**
28 Each queue dequeues tasks according to the optimal frequency;
29 Use the optimal computation graph to process the tasks concurrently;
30 **if** $\sigma_t > T$ **then**
31 Sample n groups of progress and deviation to form s_t;
32 **for** i←*0 to* W **do**
33 $a_t^i \leftarrow \rho^i(s_t)$;
34 **end for**
35 Set dequeue frequency according to a_t^0;
36 Combine blocks according to $a_t^1, a_t^2, \cdots, a_t^W$;
37 Get $stage\_str$ of the combination strategy from $stage\_dict$;
38 Reconstruct combined blocks according to $stage\_str$ to build the optimal computation graph;
39 **end if**
40 $t \leftarrow t + 1$;
41 **end while**

R (Reward): the immediate reward r_t of step t is calculated by the following parts under the dequeue frequency and computation graph corresponding to action a_t: the average GPU utilization $\overline{\xi}_t$, the average inference latency $\overline{l}_t^i$ of the tasks for model w_i, and the variance σ_t of the number of remaining tasks in each queue, as shown in (2), where $\overline{\xi}_t^*$, $\overline{l}_t^{i*}$, σ_t^* are the Z-score normalized value of $\overline{\xi}_t, \overline{l}_t^i, \sigma_t$, respectively. a, b, c are the weights assigned to the three components.

$$r_t^i = \begin{cases} e^{a\overline{\xi}_t^* - b(\sum_{j=1}^W \overline{l}_t^{j*})/W - c\sigma_t^*} & i = 0 \\ e^{a\overline{\xi}_t^* - b\overline{l}_t^{i*} - c\sigma_t^*} & i = 1, 2, \ldots, W \end{cases} \tag{2}$$

The specific process is shown in Algorithm 1. During offline scheduling (training), to avoid mutual interference and inaccurate measurement, the scheduler runs serially with DNNs. During online execution, the scheduler runs in parallel with DNNs to improve task execution efficiency.

Offline scheduling: At each step t, according to state s_{t-1}, use MADDPG to infer the current dequeue frequency a_{t-1}^0 and block combination strategy $a_{t-1}^1, a_{t-1}^2, \ldots, a_{t-1}^W$ (lines 4-6). Lines 8-9 introduce threshold H for pruning (Section V-D). We perform fine-grained operator scheduling in each combined block using the pre-trained GPG algorithm (Section V-C), and store the result $stage\_str$ in $stage\_dict$ (line 11). Then reconstruct the

combined blocks according to the $stage\_str$ to generate a new computation graph (line 12). Each queue dequeues tasks according to its own frequency (line 13), then the computation graph is used to process the tasks concurrently (line 14). At the same time, sample n groups of progress and deviation to form the next state s_t (line 15). Since the performance $(\overline{\xi}^*_{t-1}, \overline{l}^*_{t-1}, \sigma^*_{t-1})$ is obtained after finishing inference, we adopt the experience replay technique [26] to update MADDPG. After processing the tasks at the end of each step, the reward is calculated using (2) (line 16), and the transition $(s_{t-1}, a_{t-1}, r_{t-1}, s_t)$ is stored in the replay buffer $\mathcal{D}$ (lines 18-19). After the number of transitions in $\mathcal{D}$ reaches $replay\_size$, we use (3) to update the parameter $\boldsymbol{\varepsilon}^i$ of value network Q^i_ρ for agent i, where γ is the discount factor, η is the learning rate, $\overline{Q}^i_{\rho'}$ is the target network, and ρ' is the target policy. We also use (4) to update the parameter $\boldsymbol{\varphi}^i$ of policy network ρ^i for agent i [27] (lines 20-22).

$$\boldsymbol{\varepsilon}^i \leftarrow \boldsymbol{\varepsilon}^i - \eta \cdot y \cdot \nabla_{\boldsymbol{\varepsilon}^i} Q^i_\rho(s_{t-1}, a_{t-1}),$$
$$\text{where } y = r^i_{t-1} + \gamma \overline{Q}^i_{\rho'}(s_t, a_t)|_{a_t=\rho'(s_t)} \tag{3}$$

$$\boldsymbol{\varphi}^i \leftarrow \boldsymbol{\varphi}^i + \eta \cdot \nabla_{\boldsymbol{\varphi}^i} \rho^i(a^i_{t-1}|s_{t-1}) \cdot \nabla_{a^i_{t-1}} Q^i_\rho(s_{t-1}, a_{t-1})|_{a^i_{t-1}=\rho^i(s_{t-1})} \tag{4}$$

Online execution: At each step t, each queue dequeues tasks according to the optimal frequency (line 28), then the optimal computation graph is used to process the tasks concurrently (line 29). When the variance of the number of tasks in each queue exceeds threshold T (line 30), sample n groups of progress and deviation to form the state s_t (line 31). Then use the well trained MADDPG to infer the dequeue frequency and block combination strategy (lines 32-36). Get the corresponding $stage\_str$ of the combination strategy from $stage\_dict$ (line 37), and build a new computation graph according to the $stage\_str$ (line 38). By setting the threshold T, the runtime overhead caused by frequent decision-making and graph reconstruction can be significantly reduced.

C. GNN-Based Learning for Intra-Block Operator Scheduling

In our TOP scenario, the scheduling space within the combined block is much larger than that within the original block, which leads to long search time of the existing operator scheduling methods. Therefore, we further propose a learning-based stage partitioning method. The method pre-trains an intra-block scheduler so that in the subsequent stage partitioning, it does not need to traverse all the cases and profile each stage's latency, greatly reducing the scheduling time. In each recursion, we encode the computation graph V into an embedding via GNN, and then input the embedding to a policy network that will output an index of the selected ending.

1) GNN-Based Computation Graph Encoding: There are two disadvantages in directly stacking the vertex and edge information of a computation graph into the state: (i) processing a high-dimensional state requires a complex policy network, which is difficult to train to convergence; (ii) it is impossible to efficiently model the structure and operator information of the computation graph, making TOP hard to generalize to various DNNs. Therefore, we utilize Graph Neural Network (GNN) [28] to encode the computation graph into an embedding.

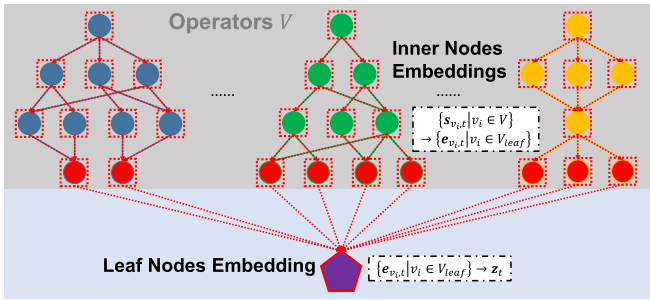


Fig. 11. Two-layer embedding for computation graph.

We define the raw state of each vertex (operator) v_i in the computation graph as $\boldsymbol{s}_{v_i,t}$, including: (i) the operator type, (ii) the shape of the input, (iii) the shape of the output, (iv) the convolution stride for each dimension, (v) the convolution kernel size, (vi) the padding for each dimension, (vii) the number of groups in the convolution, $(viii)$ the activation function (*relu*, *sigmoid*, *tanh*, *identity*), (ix) the pooling type (*max*, *avg*, *global max*, *global avg*).

As shown in Fig. 11, given the state $\boldsymbol{s}_{v_i,t}$ of each vertex/node $v_i \in V$ in the computation graph, we propagate the state information $\{\boldsymbol{s}_{v_i,t}|v_i \in V\}$ from top to bottom along the graph and finally embed it into the leaf nodes $\{\boldsymbol{e}_{v_i,t}|v_i \in V_{leaf}\}$, that is, $\{\boldsymbol{s}_{v_i,t}|v_i \in V\} \rightarrow \{\boldsymbol{e}_{v_i,t}|v_i \in V_{leaf}\}$. Specifically, we traverse each node $v_i \in V$ in the graph from top to bottom. v_i calculates its own embedding $\boldsymbol{e}_{v_i,t}$ by aggregating the embedding information of all its parent nodes and its own state $\boldsymbol{s}_{v_i,t}$, as shown in (5), where $\mathcal{F}(v_i)$ is the set of parent nodes of v_i, $h_1(\cdot)$ and $g_1(\cdot)$ are non-linear transformations implemented by neural networks as aggregation functions. The whole top-down embedding process finally stops at the leaf nodes.

$$\boldsymbol{e}_{v_i,t} = h_1\left[\sum_{v_j \in \mathcal{F}(v_i)} g_1(\boldsymbol{e}_{v_j,t})\right] + \boldsymbol{s}_{v_i,t} \tag{5}$$

Since the node state information propagates along the edges from top to bottom, the propagation also implicitly embeds the edge state information (i.e., the dependencies of operators) of the computation graph.

Finally, in the second-layer embedding, we use (6) to embed all current leaf nodes, that is, $\{\boldsymbol{e}_{v_i,t}|v_i \in V_{leaf}\} \rightarrow \boldsymbol{z}_t$, where $h_2(\cdot)$ and $g_2(\cdot)$ are aggregate functions that are isomorphic to $h_1(\cdot)$ and $g_1(\cdot)$, but have different parameters.

$$\boldsymbol{z}_t = h_2\left[\sum_{v_i \in V_{leaf}} g_2(\boldsymbol{e}_{v_i,t})\right] \tag{6}$$

The first-layer embedding only needs to embed the node set V once from top to bottom. Since the subsequent recursion is from bottom to top, the first-layer embedding can be reused directly, and only the second-layer embedding for the new leaf nodes needs to be recalculated.

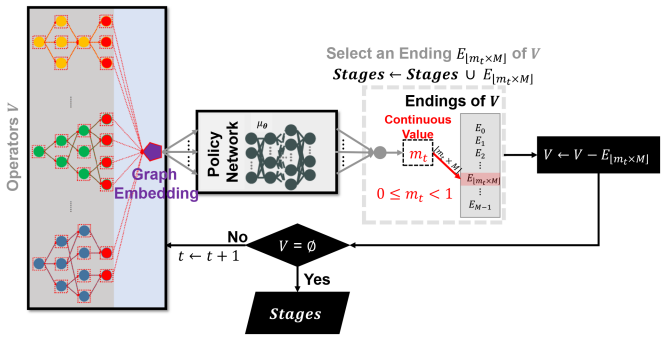


Fig. 12. Intra-block operator scheduling flow chart.

2) GNN-Based Policy Gradient (GPG) Algorithm: As shown in Fig. 12, we use a policy function $\mu(\boldsymbol{z}_t;\boldsymbol{\theta})$ to predict action under state $\boldsymbol{z}_t$, and an action-evaluation function $q(\boldsymbol{z}_t, \mathcal{M}_t;\boldsymbol{\omega})$ (only in the training phase) to predict the value of each action under state $\boldsymbol{z}_t$.μ and q are non-linear functions implemented by neural networks with parameters of $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$. The action space $\mathcal{M}_t$ is the ending set E of the current computation graph V, with a size of M. Due to the large action space, we take use of one-dimensional continuous action. The output action $0 \le m_t < 1$ is a continuous value, and the index of the selected ending is $\lfloor m_t \times M \rfloor$. Then $E_{\lfloor m_t \times M \rfloor}$ is the stage for recursive step t. At the end of each step, we remove $E_{\lfloor m_t \times M \rfloor}$ from V. The recursion ends when $V = \varnothing$.

There is only individual operator difference between index-adjacent endings in E, so although our algorithm may have some prediction errors for the continuous action m_t, which cannot guarantee the optimality, the performance is not much worse, that is, a large amount of traversal time is saved by sacrificing a small amount of performance.

In order to guide the learning of all network parameters, at the end of each recursion, a reward $u_t = e^{-l_t}$ will be generated, l_t is the execution latency (ms) of the ending selected at recursive step t. Since the optimization objective of policy learning is to maximize the expectation of cumulative future reward, the introduction of u_t can minimize the execution latency of the whole computation graph after all the recursive steps.

At each recursion, according to $(\boldsymbol{z}_t, m_t, u_t, \boldsymbol{z}_{t+1})$, we adopt the Temporary Difference (TD) algorithm [29] to update $\boldsymbol{\omega}$, as shown in (7), where δ_t is the TD error. Let $\boldsymbol{\theta}^*$ be the parameters of GPG networks $\{h_1(\cdot), g_1(\cdot), h_2(\cdot), g_2(\cdot), \mu(\cdot)\}$. We adopt the deterministic policy gradient algorithm [27] to update $\boldsymbol{\theta}^*$, as shown in (8).

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \eta \cdot \delta_t \cdot \nabla_{\boldsymbol{\omega}} q(\boldsymbol{z}_t, m_t; \boldsymbol{\omega}) \tag{7}$$

$$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}^* + \eta \cdot \nabla_{\boldsymbol{\theta}^*} \mu(\boldsymbol{z}_t; \boldsymbol{\theta}^*) \cdot \nabla_m q(\boldsymbol{z}_t, \mu(\boldsymbol{z}_t; \boldsymbol{\theta}^*); \boldsymbol{\omega}) \tag{8}$$

The intra-block optimization process is shown in Algorithm 2. First, use (5) and (6) to encode the computation subgraph V into embedding $\boldsymbol{z}_t$ (line 3), and store the intermediate embeddings in dictionary emb_dict for later reuse (line 4). Then start recursion until $V = \varnothing$ (line 5). In each recursion, search for the endings E of V that satisfy the pruning thresholds r and s. The number of groups in E do not exceed s, and the number of operators in each group do not exceed r (line 6). Then input $\boldsymbol{z}_t$ into the pre-trained policy network $\mu_{\boldsymbol{\theta}}$ to output the continuous action m_t, and calculate the $index$ of the target ending. Insert the ending E_{index} as a stage before the head of $Stages$ (lines 7-10). Remove E_{index} from V and get the embeddings of the leaf nodes of new V from emb_dict. Embed the leaf nodes into $\boldsymbol{z}_{t+1}$ using (6) (lines 11-14). If it is in the training phase, test the execution latency of E_{index} to calculate the reward u_t, and use (7) and (8) to update the value network $q_{\boldsymbol{\omega}}$ and the policy network $\mu_{\boldsymbol{\theta}^*}$ according to $(\boldsymbol{z}_t, m_t, u_t, \boldsymbol{z}_{t+1})$ (lines 15-19). Then enter the $t+1$ recursion (line 20). After all the recursive steps, output $Stages$ in string format (lines 22-23).

Algorithm 2: GPG-Based Intra-Block Scheduling.

Input: Computation subgraph V, thresholds r, s
Output: $stage\_str$

1 $t \leftarrow 0$;
2 $Stages \leftarrow \varnothing$;
3 Encode V into an embedding $\boldsymbol{z}_t$ using Eqs. (5) and (6);
4 Store all the intermediate embeddings in emb_dict;
5 **while** $V \neq \varnothing$ **do**
6 Search iteratively for all endings E of V satisfying the thresholds r and s;
7 $M \leftarrow \text{len}(E)$;
8 $m_t \leftarrow \mu_{\boldsymbol{\theta}}(\boldsymbol{z}_t)$;
9 $index \leftarrow \lfloor m_t \times M \rfloor$;
10 Insert E_{index} before the head of $Stages$;
11 $V \leftarrow V - E_{index}$;
12 $leaves \leftarrow \text{LeafNodes}(V)$;
13 Get the embeddings $\{\boldsymbol{e}_{v_i} | v_i \in leaves\}$ from emb_dict;
14 Embed $\{\boldsymbol{e}_{v_i} | v_i \in leaves\}$ into $\boldsymbol{z}_{t+1}$ using Eq. (6);
15 **if** $is\_training$ **then**
16 Measure the latency l_t of stage E_{index};
17 $u_t \leftarrow e^{-l_t}$;
18 Update $q_{\boldsymbol{\omega}}$ and $\mu_{\boldsymbol{\theta}^*}$ according to $(\boldsymbol{z}_t, m_t, u_t, \boldsymbol{z}_{t+1})$ using Eqs. (7) and (8);
19 **end if**
20 $t \leftarrow t+1$;
21 **end while**
22 $stage\_str \leftarrow \text{Stage2String}(Stages)$;
23 **return** $stage\_str$;

We pre-train GPG for different platforms using various block combinations in the model set, and it can generalize well to most of DNNs after convergence. Then the pre-trained Algorithm 2 will be invoked by Algorithm 1 (line 11). This eliminates the need to profile each stage's latency through execution during search, so it is very efficient. Furthermore, in the pre-training and search phases, a large number of intra-block GPG results are stored for subsequent reuse. This will avoid the interaction between intra-block scheduling and task processing during on-line execution.

D. Reduce Optimization Time

Inter-block pruning: Section VI-E verifies that on various devices, due to the limit of hardware resource, the performance

of the combined block (i.e., the number of instances being executed concurrently) to a certain size no longer improves, and as the size increases, the intra-block optimization cost also shows an exponential growth. Therefore, it is necessary to find a suitable threshold H for each device. When the number of sub-blocks contained in the combined block exceeds the threshold H, we call it a *superblock*. The performance of the superblock has reached saturation on the corresponding device, and the intra-block optimization cost is very high. Therefore, if MADDPG generates a superblock, we will no longer perform subsequent intra-block optimization in it, but instead return a penalty ($r_t = 0$), and maintain the previous state, dequeue frequency and computation graph. This can greatly avoid time explosion and memory overflow.

Intra-block pruning: IOS [10] reduces the number of selectable endings per recursion by limiting each ending (stage) to a maximum of s groups and r operators per group, thereby reducing optimization cost. This is a trade-off between optimization cost and effectiveness. We also utilize this mechanism to reduce the action space of GPG to accelerate training.

VI. Evaluation

A. Implementation

We implement MADDPG and GPG based on Python, and implement the underlying execution engine based on C++. The latency of the computation graph is measured in the execution engine to guide the scheduling of the upper level. The execution engine is modified based on the cuDNN library provided by [30] and supports the parallel execution of operators. In order to achieve operator parallelism, we put the operators into different CUDA streams. If there are enough computing resources, the cores in different CUDA streams will execute in parallel. For high-performance GPUs that are equipped with both CUDA Cores and Tensor Cores [31], [32], due to the more significant acceleration effect of Tensor Cores on deep learning inference, the supported operators are preferentially dispatched to Tensor Cores if they are idle; otherwise, the operators will be dispatched to idle CUDA Cores.

We do not instantiate a computation graph for each task, which would result in significant memory overhead. In the underlying execution engine, we only save one graph instance for each model. Therefore, in order to prevent the intermediate results of multiple inference instances of the same model from covering each other, we set that only one inference instance of each model can exist in each sub-block, and cache the intermediate results of each inference instance between blocks until they are used by the next block of the inference.

In order to prevent deadlocks and additional synchronization overhead, we do not set up any other synchronization mechanisms throughout the entire execution process. The effective input ratio $Ratio_{in}$ of a combined block is defined as the ratio of the number of sub-blocks that have obtained input values to the total number of sub-blocks. The combined block with a higher $Ratio_{in}$ will be executed first, and only the sub-blocks that have obtained input values will be executed, without waiting for the inputs of other sub-blocks to arrive. Combined blocks with $Ratio_{in} = 0$ will not be executed.

TABLE I
Devices Information

Device	GPU	Memory	Computility
Jetson Nano	128 CUDA Cores Maxwell	4GB	0.5TFLOPS
Jetson TX2	256 CUDA Cores Pascal	8GB	1.3TFLOPS
Xavier NX	384 CUDA Cores +48 Tensor Cores Volta	8GB	6.8TFLOPS
Server	6912 CUDA Cores +432 Tensor Cores A100	40GB	19.5TFLOPS

B. Experimental Setup

Baselines: We compare TOP with the current mainstream scheduling framework in single-model and multi-model scenarios. In the single-model scenario, TOP will be compared with TensorFlow, PyTorch, TensorRT, TVM and IOS. In the multi-model scenario, TOP will be compared with the state-of-the-art multi-model operator scheduling method POS. These methods are introduced in Section II.

The metrics include the average throughput (FPS), GPU utilization (the number of active warps between two timestamps [10]) and variance of the number of remaining tasks in each queue. In the following results, the *acceleration ratio* is the relative value of the average throughput under each setting, and the *optimization cost ratio* is the relative value of the optimization time under each setting.

Devices: In order to explore the results of operator parallelism under different resource conditions and verify the resource awareness of TOP, we test TOP on a variety of heterogeneous GPU devices, including three resource-constrained edge devices (NVIDIA Jetson Nano [33], NVIDIA Jetson TX2 [34], NVIDIA Xavier NX [31]) and a resource-rich server equipped with NVIDIA A100 [32]. See Table I for specific information about these devices.

Deploying our scheduler on some resource-constrained embedded devices (such as Nano, TX2, NX) may cause insufficient memory and even affect the execution of DNNs. Therefore, for resource-constrained edge devices, we consider separating the scheduler to the cloud, utilizing the distributed scheduling and task execution mode. As shown in the framework of Figs. 7 and 8, the cost model, task launcher, and model scheduler can be deployed in the cloud to accelerate scheduling and separate the scheduling cost at the edge. When optimization results (dequeue frequency and computation graph) are generated in the cloud, we compress and package the results in the form of strings, then transmit them to the edge device for analysis and deployment. The edge device will feed back the execution state and performance to the cloud for subsequent optimization. Here the server also acts as the cloud.

Benchmark DNNs and task arrival patterns: As shown in Table II, we select three DNNs: Inception V3, ResNet-50, and SqueezeNet, which have different operator types as well as the

TABLE II
BENCHMARK DNNS AND TASK ARRIVAL PATTERNS

Model	#Blocks	#Operators	Task Pattern (FPS)	
			Single-	Multi-
Inception V3	13	119	240	30
ResNet-50	5	87	240	60
SqueezeNet	12	50	240	120

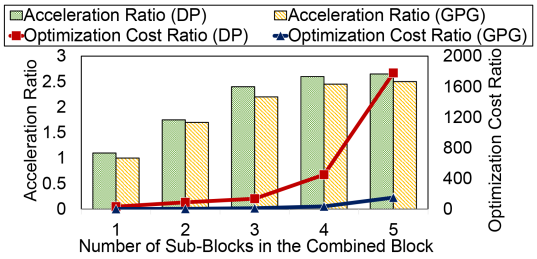


Fig. 13. Comparison of intra-block optimization algorithms DP and GPG on the server in terms of overhead and effectiveness.

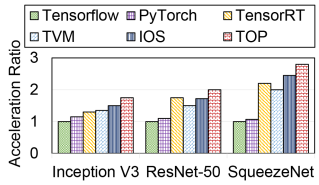


Fig. 14. Comparison between TOP and mainstream frameworks on the server in the single-model scenario.

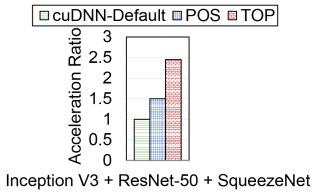
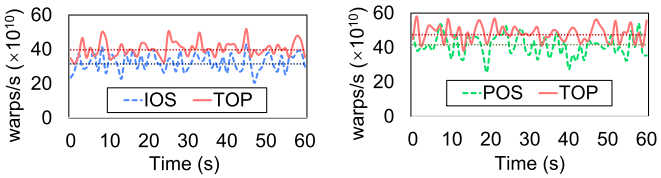


Fig. 15. Comparison between TOP and the SOTA framework on the server in the multi-model scenario.



(a) TOP vs IOS (b) TOP vs POS

Fig. 16. The GPU utilization in the (a) single-model and (b) multi-model scenarios.

number of blocks and operators. In the asynchronous-single-model scenario, only one of the three models is preloaded on the device, and the task arrival rate is set to 240 FPS. In the asynchronous-multi-model scenario, all three models are preloaded on the device simultaneously, and the task arrival rates are set to 30, 60, and 120 FPS respectively to simulate the asynchronous multitask scenario. The input shapes are all $3 \times 224 \times 224$. Initially, there are no tasks in each queue. Since the more powerful the device, the faster it processes tasks, to ensure sufficient tasks, the task arrival rate on the high-performance server is set to double the above.

Parameter settings: For MADDPG and GPG, we use a 3-layer ReLU NN to implement the value network with 1024, 512, and 300 hidden units per layer. Besides, we use a 2-layer ReLU NN to implement the policy network with 500 and 128 hidden units per layer. For GNN, $\{h_i(\cdot), g_i(\cdot)\}_{i=1,2}$ are all implemented using 2-layer ReLU NN, with 64 and 32 hidden units per layer.

Discount factor $\gamma = 0.95$, learning rate $\eta = 0.001$, batch size for MADDPG training is 100, weights $a = 0.4, b = 0.4, c = 0.2$, number of samples $n = 10$, thresholds $T = 2 \times 10^5$, $H_{Nano} = 2, H_{TX2} = 2, H_{NX} = 3, H_{Server} = 4, r = 3, s = 8$, $replay\_size = 2 \times 10^3$. The following results are the average of five tests.

C. GPG versus DP

We compare the proposed GPG-based intra-block optimization method with the SOTA method IOS [10], which also uses pruning thresholds of $r = 3$ and $s = 8$. As shown in Fig. 13, on the server, we perform intra-block optimization in the combined blocks with different number of sub-blocks. The results show that with the increase of the number of sub-blocks, the optimization time of dynamic programming (DP) used by IOS increases exponentially. For well-trained GPG, as it does not need to traverse all cases and directly predict the optimal ending selection strategy for each recursion, its optimization time gradually decreases by one order of magnitude compared to DP as the scale increases, and its performance is only slightly lower than DP. The disadvantage in performance is mainly because our method only takes use of the concurrent execution strategy. On the other hand, although DP takes a long time to optimize, it is stable and the optimality can be guaranteed.

D. Comparison With Mainstream Deep Learning Frameworks

In both single- and multi-model scenarios, TOP performs better than mainstream inference frameworks. We conduct tests on the server, as shown in Figs. 14 and 16(a), in the single-model scenario, TOP can accelerate by 14% to 180% compared to mainstream frameworks, and the GPU utilization is improved by 26% compared to IOS. As shown in Figs. 15 and 16(b), in the multi-model scenario, TOP can accelerate by 63% compared to POS, and the GPU utilization is improved by 14%.

This is mainly because TOP considers the situation of parallel task pipeline for the same model and introduces advanced intra-block scheduling algorithm. Compared to other methods, TOP has an additional task launcher module that can effectively control the progress of the task pipeline and collaborate with the model scheduler for effective scheduling to minimize synchronization overhead.

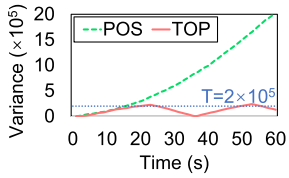


Fig. 17. Task backlogs of POS (SOTA) and TOP in the multi-model scenario.

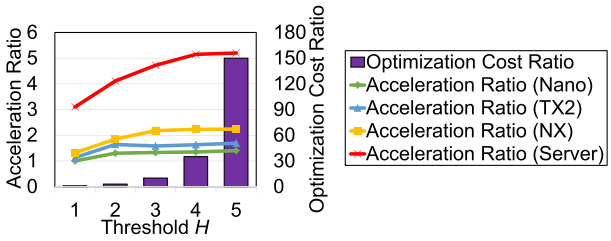


Fig. 18. The relationship between the upper limit threshold H of combined block size (number of sub-blocks) and overall optimization time, and the achievable performance on four devices.

TOP also considers the backlog of tasks in the multi-model scenario, as shown in Fig. 17. During the processing of the three types of tasks, TOP's unique mechanism can maintain the variance of the number of remaining tasks in each queue at a low level (i.e., below the set threshold $T = 2 \times 10^5$). This means that TOP can dynamically adjust the dequeue frequency of various tasks and block combination scheme, so as to prevent task backlog and achieve resource allocation. In contrast, POS does not consider the backlog of tasks, so the variance gradually increases, indicating a backlog of certain types of tasks.

E. Pruning Thresholds and Observations on Different Devices

Theoretically, the more powerful the device, the faster it processes tasks, thus the dequeue frequency on it will be higher, and the combined block (i.e., the number of concurrent tasks) will be larger. We explore the relationship between the upper limit threshold H of combined block size (number of sub-blocks) and overall optimization time on the four devices, and test the corresponding achievable performance. As shown in Fig. 18, the results on the four devices show that as the threshold H increases, the performance gain gradually saturates due to the resource bottleneck of the device, and the optimization overhead increases significantly. This is why we set parameters $H_{Nano} = 2, H_{TX2} = 2, H_{NX} = 3, H_{Server} = 4$. The thresholds s and r are set according to [10].

In addition, as shown in Fig. 20, by observing the optimization results on different devices, we find that the richer the resources, the fewer the number of stages in the computation graph, and correspondingly, the more the number of groups in each stage. This means that the parallelism of the operator is improved to utilize more resource. On the contrary, the computation graph on resource-constrained devices will be more "slender". That is, TOP can achieve heterogeneous resource awareness with strong generalizability.

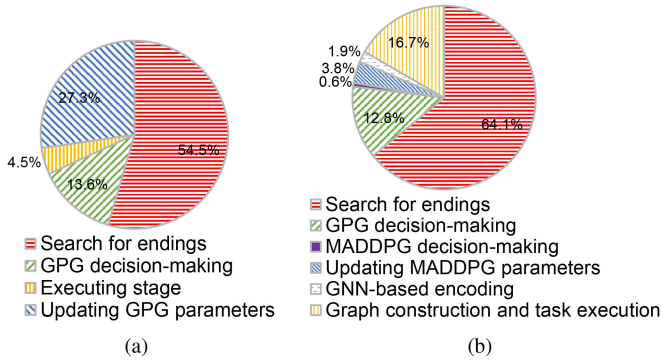


Fig. 19. (a) Percentage of time spent at each step during pre-training GPG and (b) Percentage of time spent at each step during training MADDPG.

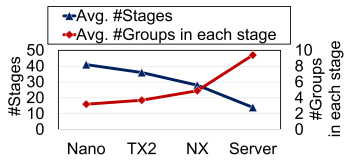


Fig. 20. The graph results of TOP on four devices.

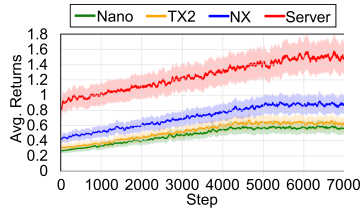


Fig. 21. The training of GPG for the four devices.

F. Scheduling Overhead

We pre-train GPG for the four devices with different block combinations in the model set, and then train the outer MADDPG for the corresponding device.

The pre-training of GPG, as shown in Fig. 19(a), includes: (i) one complete GNN-based computation graph encoding for each combined block; at each step, (ii) the search for endings; (iii) the GPG decision-making; (iv) executing the stage on specific device; and (v) updating GPG parameters. At each step, the search for endings takes the most time, averaging about 6s, and the others take less than 5s in total. As shown in Fig. 21, the GPG can converge within 6000 steps, which takes nearly 20h in total.

Then the outer MADDPG is trained based on the pre-trained GPG, as shown in Fig. 19(b), where each step includes: (i) the MADDPG decision-making to generate multiple combined blocks; (ii) one complete GNN-based computation graph encoding for each combined block; (iii) average 10 times searches for endings and GPG decision-making within each combined block (the (ii) and (iii) can be parallelized across the combined blocks); (iv) updating MADDPG parameters; (v) the computation graph construction and task execution on edge

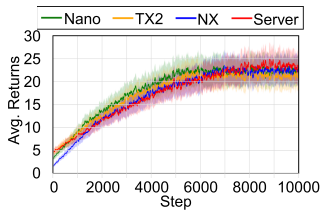


Fig. 22. The training of MADDPG for the four devices.

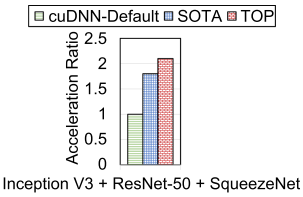


Fig. 23. In the synchronous multitask scenario, TOP still outperforms the SOTA.

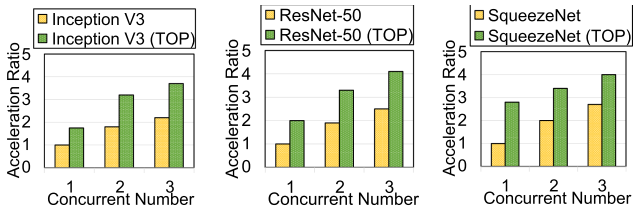


Fig. 24. On the server, we verify the feasibility of improving the throughput for a single model by increasing its concurrency, and TOP can increase this benefit.

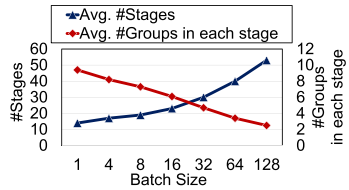


Fig. 25. The graph results for different batch sizes.

device to obtain performance feedback. At each step, the search for endings takes about 6s on average, the computation graph construction on edge device takes less than 3s, and it executes tasks for 10s to obtain performance feedback, while the others take less than 5s in total. As shown in Fig. 22, MADDPG can converge within 8000 steps, taking about one week in total.

Compared to the offline optimization time, the online execution cost is usually more concerning. Our TOP framework can generate near optimal scheduling solution in a very short time. Since a large number of intra-block optimization results are stored during the offline phase, the main runtime overhead arises from inter-block optimization, parsing results, and reconstructing the computation graph.

Under the above experimental settings, the inference time of MADDPG on the server is within 0.05ms. Parsing the results and reconstructing the computation graph on the edge devices can be completed within 3s. Moreover, only when the variance of the number of tasks in each queue exceeds threshold T, the policy is re-made, and such process is parallel with task execution, so the computational overhead of online tuning is highly acceptable.

G. Case Study

1) Synchronous Multitask: Multiple models sharing the same input is a special case for TOP. To simulate the case, we only set one queue with a sufficient number of tasks, and the three models share each task in the queue. Correspondingly, we make the action of TOP task launcher one-dimensional and set the queue variance to 0. Due to the lack of task launcher in other methods, for fairness, we set the same task frequency as TOP.

As shown in Fig. 23, our method can also outperform the SOTA method POS in the case of shared input. TOP not only considers the parallelism between different models, but also considers the situation of parallel task pipeline for the same model. It means that parallelism optimization can be carried out between instances of the same model, which introduces a larger optimization space.

2) Self-Acceleration: We conduct an experiment by increasing the concurrent number of a single model, and then allowing the concurrent models to jointly process tasks (with a sufficient number of tasks in the queue, randomly assigned to each concurrent component for inference, and the task frequency is set to the dequeue frequency after TOP converges under the corresponding concurrent number).

As shown in Fig. 24, we first test the overall throughput of the three models under different concurrent numbers. The results show that increasing the concurrent number of a single model can improve the overall throughput, even under the default scheduling algorithm of NVIDIA cuDNN [35]. Therefore, under our TOP framework, we retest the overall throughput of the three models under different concurrent numbers, and find that TOP can significantly increase the throughput benefit, as model concurrency can introduce more blocks to improve the optimization space of TOP.

3) Batch Inference: Batch inference, which stacks a batch of inputs into a tensor and computes them concurrently, can more efficiently utilize resources [10], [11]. TOP can also take advantage of batch inference to further improve throughput.

To verify this, we test the performance of TOP with different batch sizes in the multi-model scenario. Specifically, we assume that there are sufficient tasks in each queue. During each queue dequeuing tasks according to the frequency, we package the task data into batches and input them into the corresponding model, i.e., one batch as an inference instance. We only test on the server because large-batch inference requires high parallelism and is suitable for resource-rich devices. Real-time applications on edge devices usually use a batch size of 1.

As shown in Fig. 25, we observe that the larger the batch size, the more "slender" the optimized computation graph, which is due to the increased computation load of each operator caused by the large batch size, occupying more computational resources, and reducing the number of parallelizable operators in each stage. In addition, when executing multiple operators in parallel,

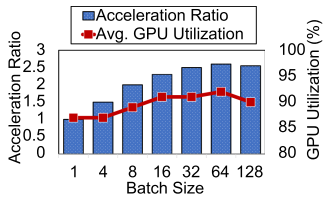
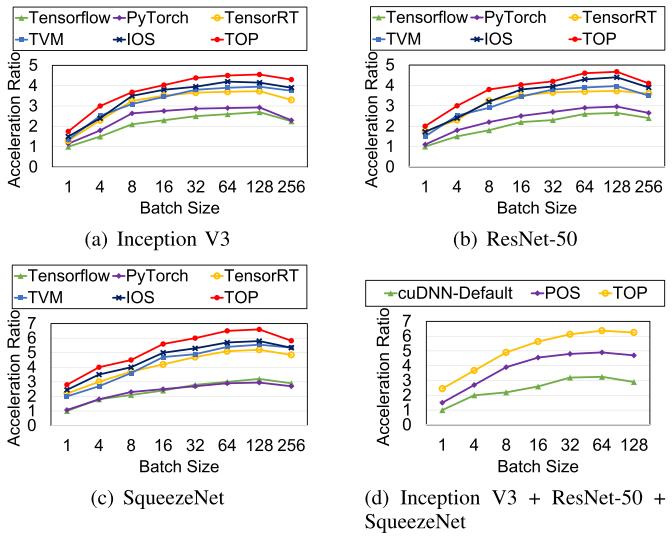


Fig. 26. The performance under different batch sizes.



(a) Inception V3 (b) ResNet-50

(c) SqueezeNet (d) Inception V3 + ResNet-50 + SqueezeNet

Fig. 27. Comparison of throughput for the baselines under different batch sizes in single-model (a)–(c) and multi-model (d) scenarios.

accessing shared resources (such as cache) is prone to conflicts. For large batches, the conflict becomes more severe because the demand for shared resources increases. In such a case, the concurrent execution of too many operators may actually degrade performance. As shown in Fig. 26, as the batch size increases, we observe that the average throughput significantly increases and saturates at size 64, and the average GPU utilization stabilizes at around 90%.

Since the majority of existing frameworks support batch processing, we also compare the performance of TOP and the baselines under different batch sizes. As shown in Fig. 27, the throughput of TOP consistently outperforms all the baselines. Compared to the multi-model scenario, the single-model scenario has fewer parallelizable operators, so it exhibits a larger batch size when the throughput saturates, reaching 128.

In summary, TOP can generate optimal scheduling for different batch sizes, leveraging the advantages of batch processing to achieve higher performance under the same resource overhead.

VII. Discussion

Optimality of solution: This paper can be understood as performing pipelined scheduling for various DL inference tasks at the operator-level. Due to the prohibitively high scheduling overhead at the fine-grained level, this paper proposes time-efficient algorithms and mechanisms through multi-level collaboration to reduce the overhead. However, such time-efficient learning-based algorithms can only find an approximately optimal solution, representing a trade-off. Such multi-granularity collaborative optimization approach is widely used in system design [12].

Applicability of GPG: We do not directly apply the proposed GPG-based efficient stage partitioning algorithm to all operators of the entire model set, as combining all operators results in a large state and action space, leading to significant errors in state encoding and decision-making. After testing, it performs well in the scheduling space of the combined block, highlighting the necessity of multi-level collaborative optimization.

Limitations of TOP: TOP is primarily optimized for Convolutional Neural Network (CNN) models. In reality, the Transformer-based [36] large models require greater consideration for the diversity of inference progress. The idea of task-based inter-block scheduling in this paper can be applied to optimization between Transformer blocks, but the introduction of dynamic tensor shapes [37] makes the computational load of each operator highly uncertain, so more advanced intra-block scheduling methods need to be studied. In the future, parallelism optimization between Transformers and CNNs can also be considered. Additionally, the high concurrency of GPUs provides significant support for this work, TOP may be extended to other heterogeneous hardware.

Uncertain workload: This paper evaluates TOP under periodic inference tasks to simulate real-time applications of sensors that sample at a fixed frequency. In addition to the periodic DNN inference, TOP can also be compatible with uncertain dynamic workloads, such as voice control triggered by keyword spotting [38] and so on. The task launcher of TOP can convert the uncertain workloads (or sampling frequencies) into optimized task patterns.

Integration with existing technologies: TOP can be easily integrated with other inter-operator parallelism strategies, such as operator fusion, to reduce memory access overhead [10], [11], [13]. It is also orthogonal to intra-operator (thread-level) parallelism, such as TVM-AutoTune [13] and Miriam [39]. Since TOP only performs operator-level runtime scheduling on a single machine without modifying the model structure, it can be integrated and collaboratively optimized with model compression [40], computation offloading [41], progressive inference [42], and other technologies in edge-cloud computing [43] to achieve better performance in multi-task inference scenarios. We look forward to TOP being applied in various edge-cloud architectures.

VIII. Conclusion

For deep learning tasks that arrive asynchronously in the inference system, this paper proposes Task-based Operator Parallelism (TOP), which utilizes MADDPG algorithm to collaboratively optimize task launcher and model scheduler. According to the task situation in the queue, TOP can dynamically generate an optimal pair of dequeue frequency and operator parallelism scheme to enhance resource utilization, increase throughput, and avoid task backlog. In order to reduce the intra-block optimization time of TOP, we propose a GPG-based stage

partitioning method. TOP outperforms the state-of-the-art in both single- and multi-model scenarios, and can also handle the case where multiple models share the same input. Benefiting from TOP, we can significantly improve the throughput of a single model by increasing its concurrency or batch size to achieve self-acceleration.

REFERENCES

[1] NVIDIA, "Cuda programming guide," 2020. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] NVIDIA, "Cuda multi-streams," 2015. [Online]. Available: https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

[3] NVIDIA, "Multi-process service," 2020. [Online]. Available: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[4] NVIDIA, "Multi-instance GPU," 2020. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/

[5] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Multi-model machine learning inference serving with GPU spatial partitioning," 2021, *arXiv:2109.01611*.

[6] C. Lin, Z. Zhang, H. Li, and J. Liu, "ECSRL: A learning-based scheduling framework for ai workloads in heterogeneous edge-cloud systems," in *Proc. 19th ACM Conf. Embedded Netw. Sensor Syst.*, 2021, pp. 386–387.

[7] J. Soifer et al., "Deep learning inference service at Microsoft," in *Proc. 2019 USENIX Conf. Oper. Mach. Learn.*, 2019, pp. 15–17.

[8] N. Ling, X. Huang, Z. Zhao, N. Guan, Z. Yan, and G. Xing, "BlastNet: Exploiting duo-blocks for cross-processor real-time DNN inference," in *Proc. 20th ACM Conf. Embedded Netw. Sensor Syst.*, 2022, pp. 91–105.

[9] 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.

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

[11] Z. Zhang, H. Li, Y. Zhao, C. Lin, and J. Liu, "POS: An operator scheduling framework for multi-model inference on edge intelligent computing," in *Proc. 22nd Int. Conf. Inf. Process. Sensor Netw.*, 2023, pp. 40–52.

[12] F. Yu, D. Wang, L. Shangguan, M. Zhang, C. Liu, and X. Chen, "A survey of multi-tenant deep learning inference on GPU," 2022, *arXiv:2203.09040*.

[13] 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.

[14] H. Vanholder, "Efficient inference with TensorRT," in *Proc. GPU Technol. Conf.*, 2016, Art. no. 2.

[15] F. Yu et al., "Automated runtime-aware scheduling for multi-tenant DNN inference on GPU," in *Proc. 2021 IEEE/ACM Int. Conf. Comput. Aided Des.*, 2021, pp. 1–9.

[16] Z. Wang et al., "Stitching weight-shared deep neural networks for efficient multitask inference on GPU," in *Proc. 19th Annu. IEEE Int. Conf. Sens. Commun. Netw.*, 2022, pp. 145–153.

[17] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 6379–6390.

[18] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[19] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, Art. no. 721.

[20] S. Wang, P. Yang, Y. Zheng, X. Li, and G. Pekhimenko, "Horizontally fused training array: An effective hardware utilization squeezer for training novel deep learning models," 2021, *arXiv:2102.02344*.

[21] V. Nigade, P. Bauszat, H. Bal, and L. Wang, "Jellyfish: Timely inference serving for dynamic edge networks," in *Proc. 2022 IEEE Real-Time Syst. Symp.*, 2022, pp. 277–290.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[23] NVIDIA, "Geforce RTX3080 family," 2024. [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080--3080ti/

[24] 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.

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

[26] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2015, *arXiv:1511.05952*.

[27] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 387–395.

[28] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 1, pp. 249–270, Jan. 2022.

[29] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, Cambridge Univ., Cambridge, U.K., 1989.

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

[31] NVIDIA, "NVIDIA Jetson Xavier," 2022. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/

[32] NVIDIA, "NVIDIA A100," 2024. [Online]. Available: https://www.nvidia.com/en-us/data-center/a100/

[33] NVIDIA, "NVIDIA Jetson nano," 2022. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/

[34] NVIDIA, "NVIDIA Jetson TX2," 2022. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/

[35] NVIDIA, "NVIDIA cuDNN documentation," 2020. [Online]. Available: https://docs.nvidia.com/deeplearning/cudnn/developerguide/index.html

[36] A. Vaswani et al., "Attention is all you need," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[37] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[38] G. Chen, C. Parada, and G. Heigold, "Small-footprint keyword spotting using deep neural networks," in *Proc. 2014 IEEE Int. Conf. Acoust. Speech Signal Process.*, 2014, pp. 4087–4091.

[39] Z. Zhao, N. Ling, N. Guan, and G. Xing, "Miriam: Exploiting elastic kernels for real-time multi-DNN inference on edge GPU," 2023, *arXiv:2307.04339*.

[40] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2017, pp. 2736–2744.

[41] Q. Liang, P. Shenoy, and D. Irwin, "AI on the edge: Rethinking AI-based IoT applications using specialized edge architectures," 2020, *arXiv:2003.12488*.

[42] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "SPINN: Synergistic progressive inference of neural networks over device and cloud," in *Proc. 26th Annu. Int. Conf. Mobile Comput. Netw.*, 2020, pp. 1–15.

[43] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

Changyao Lin (Graduate Student Member, IEEE) received the BS and MS degrees from the School of Computer Science and Technology, Harbin Institute of Technology (HIT), Harbin, China, in 2020 and 2022, respectively. He is currently working toward the PhD degree with HIT. His research interests include edge computing, distributed system, and deep learning.

Zhenming Chen is the chief engineer of China Construction Steel Structure Company, Ltd., director of China Construction Intelligent Construction Engineering Research Center, and vice director of the Expert Committee of China Steel Structure Association. He mainly engages in research on intelligent manufacturing of steel structures, complex steel structure construction etc. In the field of intelligent manufacturing of steel structures, he has won the first prize of Guangdong Province Technology Invention Award.

Ziyang Zhang (Graduate Student Member, IEEE) received the MS degree from the School of Electronic Information and Optical Engineering, Nankai University, Tianjin, China, in 2020. He is currently working toward the PhD degree with the School of Computer Science and Technology, Harbin Institute of Technology (HIT), Harbin, China. His research interests include edge computing, machine learning system, and deep learning.

Jie Liu (Fellow, IEEE) received the B.S. and M.S. degrees in automation from Tsinghua University, Beijing, China, in 1993 and 1996, respectively, and the Ph.D. degree in electrical engineering and computer sciences from University of California at Berkeley, Berkeley, CA, USA, in 2001. He is currently the Chair Professor with Harbin Institute of Technology (HIT), Harbin, China, and the Dean of its AI Research Institute. He is the Director of the National Key Laboratory of Smart Farm Technologies and Systems. Before joining HIT, he spent 18 years with Xerox PARC, Palo Alto, CA, USA, Microsoft Research, Redmond, WA, USA, and Microsoft Product Teams. He led the Sensing and Energy Research Group as a Principal Research Manager with MSR, Redmond. In MSR-NexT and Product Groups, he incubated smart retail solutions, which became part of Microsoft Business AI offering. His research interests include AI for IoT systems, sensing, mobile computing, and energy-efficient systems. He has chaired a number of top-tier conferences in sensing and pervasive computing, and was an Associate Editor for several top-tier journals. He has received 6 Best Paper Awards, the Leon O. Chua Award from UC Berkeley in 2001, and the IEEE TCCPS Distinguished Leadership Award in 2021. He is an IEEE Fellow and ACM Distinguished Scientist.