Computation Offloading Scheduling for Deep Neural Network Inference in Mobile Computing

Yubin Duan and Jie Wu
Center for Networked Computing, Temple University, Philadelphia, USA
Email:{yubin.duan, jiewu}@temple.edu

***Abstract*—The quality of service (QoS) of intelligent applications on mobile devices heavily depends on the inference speed of Deep Neural Network (DNN) models. Cooperative DNN inference has become an efficient way to reduce inference latency. In cooperative inference, a mobile device offloads a part of its inference task to cloud servers. The large communication volume usually is the bottleneck of such systems. Priory research focuses on reducing the communication volume by finding optimal partition points. We notice that the computation and communication resources on mobile devices can work in pipeline, which can hide the communication time behind computation and further reduce the inference latency. Based on the observation, we formulate the offloading pipeline scheduling problem. We aim to find the optimal sequence of DNN execution and offloading for mobile devices such that the inference latency is minimized. If we use a directed acyclic graph (DAG) to model a DNN, the complex precedence constraints in DAGs bring challenges to our problem. Notice that most DNN models have independent paths or tree structures, we present an optimal path-wise DAG scheduler and an optimal layer-wise scheduler for tree-structure DAGs. Then, we proposed a heuristic based on topological sort to schedule general-structure DAGs. The prototype of our offloading scheme is implemented on a real-world testbed, where we use Raspberry Pi as the mobile device and lab PCs as the cloud. Various DNN models are tested and our scheme can reduce their inference latencies in different network environments.**

***Index Terms*—computation offloading, mobile cloud computing, pipeline scheduling, QoS-aware scheduling.**

I. Introduction

With the wide deployment of deep neural networks (DNNs) in mobile applications, the quality of service (QoS) of those applications heavily depends on the DNN inference *response time* or *latency*. For example, augmented reality applications running on mobile devices usually uses DNNs for image segmentation. A high inference latency may harm the user experience. Besides augmented reality, there is a large amount of mobile applications that utilizes DNNs for natural language processing [1] or computer vision [2], [3]. Some of those applications may require real-time response. Therefore, it is necessary to reduce the DNN inference time, especially for resource-constraint mobile devices.

Cooperative DNN inference over mobile devices and cloud servers has become an efficient way to reduce the response time of DNN inference tasks [4], [5]. In cooperative inference, mobile devices will *offload* the computation workload of some DNN layers to cloud servers. Essentially, DNN inference is a round of forward propagation. In the forward propagation, each layer in the network accepts data from its predecessor, processes it using its specific activation function, and passes the result to its successors. If mobile devices send outputs of some intermediate layers to cloud servers, the computation workload of successive layers is offloaded to cloud servers. In cooperative DNN inference, those intermediate layers or *cut-points* should partition the DNN into two parts. A partition example of GoogLeNet is shown in Fig. 1. Layers marked with shadow are cut-points. Mobile devices are responsible for computing layers prior the cut-points. The other part is executed by the cloud. This approach can utilize the strong computation power of cloud servers with the cost of enlarging the network communication burden. In addition, unlike DNN model compression approaches, cooperative inference can accelerate DNN inference without changing DNN structures or harming inference accuracies. On account of the rapid development of wireless communication technologies, cooperative inference becomes more and more attractive.

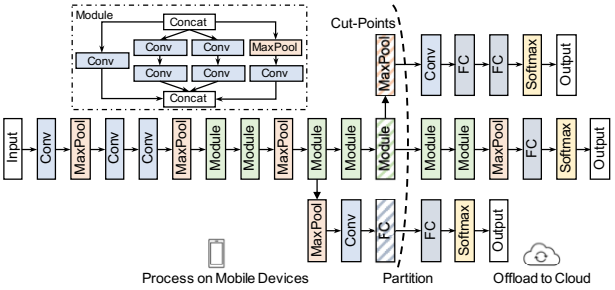


Fig. 1. A DNN partition example of GoogLeNet.

Prior research has discussed the optimal partition strategy for cooperative DNN inference [5], [6]. They apply min-cut algorithms to find the best subset of DNN layers that should be offloaded to cloud servers in different network environments. The best partition is calculated in an effort to minimize the inference response time. Their partition algorithms can obtain "sweet" cut-points which minimizes the overall computation and communication time. However, they ignore that mobile devices can perform the computation and communication *in pipeline*. By using the pipeline, the communication time can be hidden behind the computation, which can further reduce the inference latency. Start from the observation, we investigate other factors that help reduce the completion time of cooperative DNN inference besides DNN partition.

This research was supported in part by NSF grants CNS 1824440, CNS 1828363, CNS 1757533, CNS 1629746, CNS 1651947, and CNS 1564128.

Although the pipeline can hide the computation time, we need to carefully schedule the offloading sequence to minimize the inference latency. Before scheduling, we assume the DNN partition is given. A set of cut-points can identify a partition. If a DNN layer belongs to the cut-point set, its output should be sent from mobile devices to servers. It is not trivial to determine the communication sequence of those outputs. Besides, the output of a cut-point is not available until all its predecessors are processed. The precedence relationship among layers in a DNN can be complex. It is challenging to find the optimal layer-wise processing schedule. For example, if we process all layers that are not cut-points first, data communication cannot start until the computation of those layers is finished. It may cause unnecessary idle of communication resources, which enlarges the inference completion time. In contrast, a good schedule should overlap the computation time with data communication as much as possible. These observations motivate us to formulate the offloading scheduling problem with the objective of minimizing the DNN inference latency.

Generally, if we treat each layer in DNN as a vertex, a DNN can be modeled as a directed acyclic graph (DAG). After partition, vertices left on mobile devices may contain computation and/or communication operations. Optimizing the layer-wise processing sequence to minimize the completion time for those layers is a scenario of the NP-hard DAG shop scheduling problem. We utilize DAG structures to optimize the scheduling problem for cooperative DNN inference.

We notice that most DNN models have tree structures, especially after partition. Some DNN models, such as Alexnet-parallel [7], even consist of multiple independent paths. For those path-independent DNNs, we investigate an optimal path-wise scheduling algorithm. Then, we extend the path-wise scheduling to fit arbitrary tree-structure DAGs by introducing a graph conversion algorithm that can convert any tree-structure DAGs into path-independent graphs. Note that the optimal path-wise schedule of the converted graph might be suboptimal for the original graph. To find the optimal schedule for tree-structure DAGs, we further propose a recursive scheduling algorithm. It recursively merges optimal schedules of subtrees along branches from leaves to the root. Besides, a vertex grouping trick is proposed to speed up the recursive scheduling. Finally, we propose a heuristic that is inspired by the topological sort to schedule general-structure DAGs.

We implement the prototype of offloading pipeline scheme in Python. Our testbed uses Raspberry Pi as mobile devices, and a lab PC as the cloud server. The cooperative inference is implemented using PyTorch. Varieties of DNN models are tested. Specifically, we use Alexnet-parallel as a path-independent graph, use GoogLeNet, ResNet, and MobileNet as tree-structure graphs, and use RandWire as general-structure DAG. Compared with partition-only schemes, our approach can significantly reduce the inference latency.

Our major contributions are summarized as follows:

- We propose an offloading pipeline scheme that can speed up the cooperative DNN inference, which improves the QoS of varieties of learning-based mobile applications.
- We formulate the offloading scheduling problem to minimize the overall inference latency and investigate optimization strategies for different DAG structures.
- We propose an optimal path-wise scheduling algorithm for path-independent graphs, an optimal layer-wise scheduling for tree-structure DAGs, and a heuristic for general-structure DAGs.
- We implement the prototype of our offloading scheme and test its performance with various DNN models. Our scheme can significantly reduce their inference latencies.

II. Related Works

The research efforts for DNN inference acceleration can be categorized into three approaches: model compression [8]–[11], system design optimization [12]–[15], and cloud/edge offloading [16]–[19].

One way to accelerate the inference speed on mobile devices is to compress or simplify the existing DNN models. In simplified models, the computation workload is reduced by removing some layers with the sacrifice of the inference accuracy. For example, Iandola *et al.* [8] shown SqueezeNet, which significantly reduce the size of DNNs for image classification. Huang *et al.* [11] presented YOLO-lite, which can achieve real-time object detection on mobile devices without GPUs, as a simplified version of YOLO [20]. Those efforts make it possible to deployment DNN-based applications on mobile devices. However, the inference performance, such as the classification accuracy, decreases as well. In contrast, we propose to speedup DNN inference on mobile devices by fully utilizing the computation and communication resources with pipeline. Our approach will not affect the inference performance of DNN model.

Another approach for DNN inference acceleration from the system/architecture design aspect. Niu *et al.* [12] proposed PatDNN, which is an efficient framework to process DNN on mobile devices with the help of architecture-aware compiler optimizations. Besides, hardware architecture can be specialized to efficiently process DNNs. For example, DeepBurning [21] helps to implement or design an FPGA-based neural network accelerator. However, they did not fully investigate the scheduling problem for DAG-style DNN execution. Plus, our scheduler can be easily integrated into a mobile cloud computing system.

Cloud/edge offloading becomes more and more popular with the rapid improvement of wireless communication bandwidth. This approach utilizes the collaboration between mobile and cloud computation resources. The idea of offloading DNN computation workload is first proposed in Neurosurgeon [4]. However, Neurosurgeon does not consider that computation and communication resources can work in pipeline, which can bring additional speedup for DNN inference. Our paper considers the scheduling problem for DNN offloading. Besides, DDNN [22] presented a distributed DNN computing systems over mobile, edge, and cloud devices with the objective of minimizing the communication data size. Our paper has a different objective that is to reduce the inference makespan.

The makespan consists of computation and communication latencies. More recently, Zhang *et al.* [6] further improved the partition algorithm by proposing a more accurate latency estimation model. Those works mainly investigated the DNN partition problem. Different from them, we consider the scheduling problem when offloading multiple paths in partitioned DNNs. Our scheduling algorithm supports any partition scheme. Finding the optimal offloading schedule can further accelerate the DNN inference compared with merely considering the partition problem.

III. Model

A. Preliminaries

We first explain the procedures of DNN offloading before formulating our scheduling problem. A DNN usually contains multiple *layers* to progressively extract features from the input data. Layers can be categorized into different types, such as convolutions layers or pooling layers. Each type of layer contains multiple neurons that are layer-specific functions. Neurons in a layer are connected in a layer-specific way. In our paper, we treat a layer, instead of a neuron, as a scheduling atomic. It is not wise to partition neurons within a layer since the connection of the neurons is dense. The computation of each layer could be modeled by a tensor or matrix operation which has been well optimized in major machine learning frameworks such as PyTorch or Tensorflow. Partition neurons within a layer would lose the speedup brought by those optimizations and may also lead to a large communication cost. Therefore, we consider the *layer-wise* partition and scheduling in this paper.

We focus on reducing the response time of *DNN inference tasks* for mobile devices. DNN inference tasks involve a single round of forward propagation on well-trained DNN models. Specifically, after the input data is fed into a DNN model, each layer accepts the intermediate result from its previous layer, processes it according to the layer-specific function, and passes to the successive layer. The response time or latency is used to measure the duration from when the input data is loaded to the DNN until a result is generated.

Offloading part of the DNN computation workload from mobile devices to cloud servers can speed up the DNN inference. Cloud servers usually have much stronger computation power than local mobile devices. Also, the bandwidth of wireless communication has been greatly increased. [4]–[6] have demonstrated the acceleration of the DNN inference brought by the offloading. Those papers mainly focus on the *DNN partition problem*, which aims to find the optimal set of layers whose computation would be offloaded to cloud servers such that the overall response time of DNN inference is minimized. However, they ignore the useful fact that computation and communication resources can work *in pipeline* on mobile devices, which can help to further reduce the inference latency.

Mobile devices can transfer the results of some layers to cloud servers while computing the output of other layers. Assuming a DNN partition is given, the cooperation between mobile devices and cloud servers for a DNN inference task contains three steps: 1) computation on mobile devices, 2) upload intermediate results from mobile devices to cloud servers, 3) computation on the could and send back the inference result. The first two steps are performed on mobile devices. Among them, the computation is performed on the CPU/GPU and the communication is performed by the network interface. Therefore, they can work in a pipelined manner. The third part is executed on cloud servers and it would not start until all required intermediate results are received. Considering there are multiple layers of partitioned DNN that would be executed on the mobile devices, their processing sequence inevitably affects the inference latency. The *DNN scheduling problem* is to find the optimal processing sequence to further minimize the DNN inference latency after a partition is given.

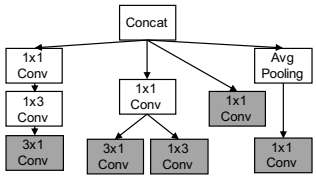


Fig. 2. Partitioned Inception module is in tree-structure.

B. Notations

We model DNNs as directed acyclic graphs (DAGs) and treat a layer in DNN as a vertex in the DAG. The connections between layers are represented by edges between vertices. Formally, let $G = (V, E)$ denote a DAG, where V is the vertex set and E is the edge set. Each node $v \in V$ represents a layer in the DNN instead of a neuron since our partition granularity is layer-wise. An edge $e \in E$ represents the data communication between two vertices that are incident to e. The direction of the edge shows the data flow direction. Specifically, an edge (v_i, v_j) means the output of v_i should be sent to v_j. This precedence relation is denoted as $v_i \prec v_j$. The vertex that has no predecessors (or successors) is the source (or destination) vertex of the DAG. The edge weight represents the communication volume. The forward propagation of the DNN inference can be viewed as a flow from the source to the destination in the DAG.

Before scheduling, we assume the partition of DNN is known. Let $P \subset V$ denote the partition of the DNN model. P contains a set of cut-points. Cut-points partition the DAG G into two sub-graphs. Specifically, all layers $v \in P$ and their predecessors would be executed on mobile devices. The output of all layers $v \in P$ should be sent from mobile devices to cloud servers via wireless communication channels. The cloud server cannot start computation until it receives all output of $v \in P$. We focus on optimizing the scheduling for mobile devices. Let G' denote the sub-graph that describes the data flow on mobile devices. Formally, $G' = (S, (S \times S) \cap E)$, where $S = \{v \in V | v \prec u, \forall u \in P\} \cup P$. The set S represents DNN layers that are executed on mobile devices after partition. The computation of layers that are successors of $v \in P$ would be offloaded to cloud servers. The edge set $(S \times S) \cap E$ keeps the precedence relations among vertices in S. We need to

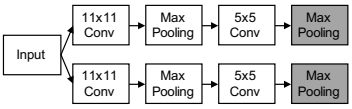


Fig. 3. Partitioned Alexnet-Parallel is in path-independent.

carefully schedule the processing order of layers in S to reduce the DNN inference latency.

On mobile devices, we focus on finding a layer-wise processing sequence for the partitioned DNN. Formally, let $\sigma = (x_1, x_2, \ldots, x_{|S|})$ denote the schedule, which is a permutation of S. x_i represents a vertex $v \in S$. The subscript of x_i shows its processing priority. x_i is processed before x_j if $i < j$. According to the order in σ, the mobile device would process one layer at a time. Our model does not consider the parallel processing of multiple layers since mobile devices usually have one CPU. Splitting the computation resource may enlarge the processing time [23]. For DNN layers $v \in P$, their output should be sent to the cloud. The communication starts immediately after the output of v is generated if the network resource is idle. Otherwise, the output is queued for sending. If there is another layer u which is scheduled after v, the computation of u also starts without waiting for the completion of network communication. In this way, the computation and communication resources work in the pipeline, which helps to reduce the response time of DNN inference. In our scheme, the cloud will not start computation until it receives all required data. Therefore, the cloud computation is not considered as a part of the pipeline. We mainly investigate the scheduling problem for mobile devices.

The purpose of the scheduling is to reduce the response time or latency of the DNN inference. To formulate the response time, we introduce two more notations to denote the communication and computation time of each layer. Specifically, Let functions $f\colon V \to \mathbb{R}$ and $g\colon V \to \mathbb{R}$ denote the computation and communication time of each layer, respectively. The computation time $f(v)$ of layer v can be accurately predicted [6]. For example, on mobile devices, the computation time of different types of layers can be predicted with regression models in a short time. The regression model can be pre-trained and installed on mobile devices. The communication time is mainly related to the data volume and the network environment. Note that a layer v has non-zero communication time only if $v \in P$, i.e., it is a cut-point. For other layers $u \notin P$, $g(u) = 0$. We use a linear regression model to calculate the communication time. Let $w(v)$ denote the output volume of layer v. Then, the communication time $g(v) = c + w(v)/b$, where b is the bandwidth of the communication channel and c is the end-to-end delay time that includes the propagation delay and the time for establishing the channel.

The inference latency is measured by the length of duration from when the input data is available until the inference result is calculated. In our mobile computing scenario, it consists of three parts. They are the computation time on mobile devices, the communication time, and the computation time on cloud devices. The cloud computation time is usually much smaller than others. Also, given a DNN partition, its processing time on the cloud is usually constant. Therefore, we omit the cloud computation time when formulating the inference latency. Similarly, the time consumption of sending inference results from the cloud to the mobile is negligible.

Let τ denote the latency of DNN inference. Without using the pipeline, $\tau = \sum_{k=1}^{|S|}(f(x_i) + g(x_i))$. We propose to reduce the latency by hiding the communication time behind computation using the pipeline. In this case, the latency can be recursively calculated based on the completion time of each layer x_i in the schedule σ. Let t_i denote the wall-clock time at which the execution (including both computation and communication) of x_i is completed. We use t_0 to denote the arrival time of the inference task. Then, the value of t_i and τ can be calculated with the following proposition.

***Proposition** 1:* Given a schedule σ for a partitioned DNN, its inference latency on mobile devices is $\tau = t_{|S|} - t_0$, where $t_i = \max\{t_0 + \sum_{k=1}^{i} f(x_i), t_{i-1}\} + g(x_i), i = 1, 2, \ldots, |S|$.

Proof: The term $t_0 + \sum_{k=1}^{i} f(x_i)$ represents the completion time of x_i's computation. t_{i-1} is the completion time of x_i's communication. When the network environment is bad, data communication can take longer time than computation. It is possible that network resources are busy when the computation of x_i is completed. The communication of x_i cannot start until its computation and the previous layer's communication are finished. Those conditions can be formulated as $\max\{t_0 + \sum_{k=1}^{i} f(x_i), t_{i-1}\}$. Therefore, the completion time of x_i is $t_i = \max\{t_0 + \sum_{k=1}^{i} f(x_i), t_{i-1}\} + g(x_i)$. The DNN inference latency is the duration between t_0 when the task arrives and $t_{|S|}$ when the execution of the last layer is finished. Therefore, $\tau = t_{|S|} - t_0$. ■

C. Problem Formulation

This paper investigates the layer-wise pipeline scheduling problem for cooperative DNN inference. Our objective is to minimize DNN inference latency by finding the best schedule σ. The scheduling problem can be formulated as following.

$$\min_{\sigma} \quad \tau = t_{|S|} - t_0 \tag{1}$$

$$\text{s.t.} \quad t_i = \max\{t_0 + \sum_{k=1}^{i} f(x_i), t_{i-1}\} + g(x_i), \forall x_i \in \sigma \tag{2}$$

$$i \leq j, \forall x_i \prec x_j, \forall x_j \in \sigma \tag{3}$$

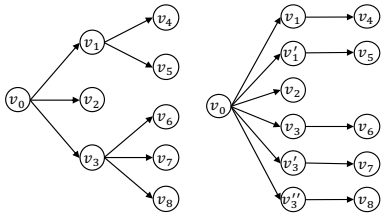
$$\cup_{x_i \in \sigma} x_i = S, |\sigma| = |S| \tag{4}$$

Eq. (1) shows our objective of minimizing the latency of DNN inference. Eq. (2) shows the recursive calculation of the completion time for a specific schedule σ. This equation explains the correlation between the latency τ and a schedule σ. Eq. (3) is the precedence constraint. A feasible schedule σ cannot violate the precedence constraints indicated by the DAG. For all layers x_j in the schedule, if a layer x_i is a predecessor of x_j in the DAG representation of the DNN, then x_i should be processed before x_j, i.e. $i < j$. Eq. (4) is the permutation constraint. A feasible schedule σ should also be a permutation of S. It means all layers in the partitioned DNN should be executed exactly once on mobile devices.

Algorithm 1 Johnson's rule [24]

Input: Set of paths $H = \{h_1, h_2, \ldots, h_n\}$
Output: The optimal path-wise schedule

1: Communication-heavy set $S_1 \leftarrow \{h_i \in H | f(h_i) < g(h_i)\}$. Computation-heavy set $S_2 \leftarrow \{h_i \in H | f(h_i) \geq g(h_i)\}$
2: $\sigma_1 \leftarrow$ Sort S_1 for ascending order of $f(h_i)$.
3: $\sigma_2 \leftarrow$ Sort S_2 for descending order of $g(h_i)$.
4: $\sigma \leftarrow \sigma_1 || \sigma_2$.
5: **return** σ



(a) Original DAG (b) After conversion

Fig. 4. An illustration of the tree structure DAG.

D. Problem Analysis

It is challenging to find the optimal solution for our scheduling problem. Our scheduling problem can be viewed as a spatial version of the DAG shop scheduling problem. In DAG shop scheduling, each vertex in the DAG has multiple operations which need to be processed on different types of resources. Even when the number of operations and resources are 2, the DAG shop scheduling problem is NP-hard [23]. The precedence constraints in the DAG make it challenging to find the optimal schedule.

There are some useful properties that can be used when solving our scheduling problem. The first observation is that for many DNN models, the partitioned DNN left on mobile devices has tree structures. For example, after partition, the GoogLeNet [25] using the partition algorithm proposed in [6], the partitioned DNN left on mobile devices is a tree as shown in Fig. 1. Similarly, a partitioned Inception module is also a tree as shown in Fig. 2. Scheduling a tree-structure DAG is easier than scheduling a general DAG since some tricky precedence relations (such as diamond structures) are removed. Another useful property is that $g(v) = 0$ for vP. Only if a layer is a cut-point, its communication time is non-zero. For a tree-structure DAG, this property means that only leaf nodes have network communication. We only need to care about the computation time for non-leaf nodes in this case. Based on those observations, we first investigate the scheduling problem for tree-structure DAGs in Section IV. Then, we extend our solution to general-structure DAGs.

IV. Scheduling for Tree-Structured DAGs

According to our observation, many partitioned DNNs on mobile devices have tree structures. Among those DAGs, many DNN models have parallel paths, such as Alexnet-Parallel [7]. For those models, we propose an optimal path-wise scheduling algorithm. For other tree-structure DAGs, we investigate a method that converts them into trees with independent paths. Besides, we propose a better approach that optimally schedules them without conversion.

Algorithm 2 DAG Conversion Algorithm

Input: Tree-structure graph $G' = (S, (S \times S) \cap E))$
Output: A path-independent DAG

Initialize a queue $Q = [v_r]$, where v_r is the root of G'.
while Q is not empty **do**
  $v \leftarrow$ poll from Q. Push direct children of v into Q.
  if $v \neq v_r$ and out-degree(v) > 1 **then**
    $U \leftarrow$ children of v; $m \leftarrow$ out-degree of v.
    Duplicate v and its incoming edge $m - 1$ times.
    For each replicate, link a unique $u_i \in U$.
return G' as the path-independent graph.

A. Path-wise Scheduling

The precedence constraint among vertices is a major challenge for DAG scheduling. If a DAG only contains independent paths, we can perform path-wise scheduling, which is much easier to solve. Fig. 3 shows a DAG that has independent paths. If a DAG does not consist of independent paths, it can be easily converted into a graph that has multiple independent paths as illustrated by Fig. 4(b). Therefore, we first investigate the path-wise scheduling.

In path-wise scheduling, DNN layers on the same path are treated as a block. For example, vertices v_1 and v_4 in Fig. 4(b) is viewed as a virtual vertex in the path-wise scheduling. We use h_i to denote a path instead of a DNN layer or a vertex when scheduling the converted DNN. With a slight misuse of notations, $f(h_i)$ represents the computation time of the path and its value is the summation of computation time of layers clustered in h_i. Besides, $g(h_i)$ refers to the communication time of the path and it equals to the communication time of the last layer in path h_i. The optimal schedule of those independent paths can be easily found.

The path-wise scheduling can be optimally solved by Johnson's rule [24]. The scheduling problem can be categorized as a flow shop scheduling problem [26], [27]. The flow shop problem can be optimally solved by using Johnson's rule, when there are only two types of operations (computation and communication). The procedures of the Johnson's rule is illustrated in Alg. 1. In line 1, we split all paths into two groups based on their computation and communication time. The communication-heavy set S_1 contains all paths whose communication time is longer than its computation time. Formally, $S_1 \leftarrow \{h_i \in H | f(h_i) < g(h_i)\}$. The computation-heavy set S_2 contains the other paths. Formally, $S_2 \leftarrow \{h_i \in H | f(h_i) \geq g(h_i)\}$. Then, we sort the paths in the communication-heavy set S_1 based on their computation time in ascending order. Line 3 sorts the paths in S_2 based on their communication time in descending order. Finally, we concatenate two sorted order set σ_1 and σ_2. Paths from the computation-heavy set S_2

Algorithm 3 Tree-structure DNN Scheduling

Input: Tree-structure graph $G' = (S, (S \times S) \cap E))$
Output: The schedule σ for all $v \in S$

1: Initialize a mapping function φ.
2: **for** each leaf node v of G' **do**
3: Add a key-value pair $\langle v, [(f(v), g(v))] \rangle$ to φ.
4: **while** $|S| > 1$ **do**
5: $v \leftarrow$ pick a leaf node with the largest depth in current G'. $U \leftarrow$ all sibling nodes of v, including v. $v_p \leftarrow$ the parent node of v.
6: $L \leftarrow \{\varphi(u) | \forall u \in U\}$
7: $l \leftarrow$ merge scheduling lists in L using Alg. 4
8: Get the head element $(f(v_l), g(v_l))$ of l. Retrieve the corresponding DAG vertex as v_l.
9: Group v_p and v_l, and use v_p to represent the group.
10: Update the head element of l into $(f(v_p)+f(v_l), g(v_p)+g(v_l))$. $S \leftarrow S \setminus U$. Update G'.
11: Add a key-value pair $\langle v_p, l \rangle$ to φ
12: **return** $\varphi(v_r)$ as the final schedule, where v_r is the element left in S and it is also the root of the tree.

Algorithm 4 Merging Scheduling List

Input: A set L of scheduling lists
Output: A new list l that merges all lists in L

1: $l \leftarrow [], T \leftarrow \{\}$.
2: **while** L is not empty **do**
3: **for** $l_i \in L$ **do**
4: Peek the first tuple m_i in l_i. $T \leftarrow T \cup m_i$
5: $r \leftarrow \arg\min T$ (Find r such that tuple m_r is the smallest in T). Johnson's rule is used for tuple comparisons.
6: Append m_r to l. Remove m_r from corresponding l_r.
7: Remove l_r from L if $l_r = []$.
8: **return** l as the merged list.

are placed after paths from S_1. After concatenation, the order set σ contains the optimal path-wise scheduling.

The insight behind Johnson's rule is greedy. For paths with larger communication time, their communication should start as early as possible. Their communication cannot begin until their computation is over. Therefore, communication-heavy paths with shorter computation time should be executed first. Computation-heavy paths are sorted symmetrically.

If a DAG does not originally have independent paths, it can be easily converted into a path-independent one [28]. The procedures are shown in Alg. 2. Intuitively, we perform a breadth-first search (BFS) on G'. Duplicate each internal node based on its out-degree and reconnect its successors such that each replicated node has one direct child. Specifically, line 1 initializes a queue Q using the root of G'. Then, in each iteration, a vertex v is polled from Q. Direct children of v are pushed into Q for BFS. In lines 4-7, if the out-degree m of v is greater than 1, then we duplicate the vertex $m-1$ times and reconnect $m-1$ outgoing edges such that each replicate has a unique successor. Note that the root node is skipped since there is no need to duplicate input layers. Eventually, the out-degree of every internal node in G' becomes 1. After conversion, G' only contains independent paths.

We need to pay additional attention when actually executing the schedule of a converted graph. To make sure the computation result is not changed after the conversion, duplicate layers should be executed only once. When executing a path-wise schedule, we use a table to record the execution status and result of each layer. If a layer has been executed in other paths, the computation of the layer shall be skipped. The following layer should use the output recorded in the table. In this way, we can guarantee that the converted DNN generates the same result as the original DNN.

Although we can find the optimal solution for the path-wise scheduling, it is not optimal for the original layer-wise scheduling problem since path-wise scheduling may miss potential optimization chances. Especially for scheduling with graph conversion, redundant layers inserted for conversion are not actually executed when running the DNN inference, while our path-wise scheduling algorithm assumes they are. Therefore, the optimal layer-wise schedule might be missed.

B. Layer-wise scheduling

The optimal path-wise scheduling may be suboptimal for the layer-wise scheduling problem. Besides, inserting duplicate vertices and maintaining the execution status table bring additional memory costs. Therefore, we try to extend the Johnson's rule and present another scheduling algorithm that does not require the DAG conversion.

For layer-wise scheduling of tree-structure DNNs, we propose to recursively merge the scheduling of subtrees. Specifically, if there are no precedence constraints among all children nodes in a subtree, i.e., all children nodes are leaves, then we apply Johnson's rule to sort the children nodes. For other cases, we recursively schedule subtrees rooted at internal nodes and merge the scheduling results of sibling nodes. In this way, we convert each subtree into a list. The lists are recursively merged into the final schedule. This idea is inspired by merge sort. The novelty is that we use Johnson's rule to determine the relative order of nodes when merging the scheduling results of sibling nodes. Besides, for each subtree, we group its root with the first element of the merged scheduling list to reduce the solution space. The grouping will not lose the optimal schedule. More details are explained in Theorem 1 after showing the procedures of our scheduling algorithm.

Before diving into algorithm details, we need to explain addition notations used in the scheduling algorithm. Our scheduling algorithm will convert subtrees into lists. We use l to denote a list. Each element in list l is a tuple $(f(v), g(v))$, where $v \in S$. The tuple record the computation and communication time of a vertex $v \in S$. The order of elements in the tuple represents the processing sequence of the corresponding DNN layers. Elements with smaller indexes should be executed first. We use L to denote the set of lists, considering there are multiple lists that are converted from different subtrees. Let $\varphi \colon S \to L$ denote the mapping

function. It maps a vertex in tree-structure DAG into a list. $\varphi(v)$ represents the scheduling list of the subtree rooted at v.

To explicitly illustrate our scheduling algorithm, we show its iterative implementation in Alg. 3. In lines 1 to 3, we initialize the mapping function $\varphi(v)$ that maps a DAG vertex v into a scheduling list. The scheduling list of a leaf vertex only contains itself. Hence, we first add the mapping $\langle v, [(f(v), g(v))]\rangle$ into φ for each leaf node v of G'. The loop in lines 4 to 11 updates the mapping φ and graph G'. In each iteration, a vertex v is randomly picked from the deepest leaf nodes of G'. All sibling vertices of v, including v, are stored in set U. Line 5 also finds the parent vertex v_p of v. Then, line 6 constructs a set L that contains the scheduling list of every element $u \in U$ using the mapping function φ. Scheduling lists in L are merged into a single list l by calling Alg. 4 in line 7. List l contains the schedule of vertices in the subtree rooted at v_p, except v_p itself. Instead of simply inserting v_p to the head of the schedule, we propose to group v_p with the first element of the schedule. The grouping can efficiently reduce the time complexity of scheduling without missing the optimal schedule. Theorem 1 shows that our scheduling algorithm with grouping is optimal. The complexity analysis is investigated in Theorem 2. Line 8 retrieves v_l which corresponds to the first element $(f(v_l), g(v_l))$ of list l. Note that v_l can be either a single DAG vertex or a representative of a vertex group. In both cases, line 9 creates a new group that collects v_p and v_l. The new group is represented by v_p. The computation and communication time of the group are updated accordingly in line 10, i.e., $f(v_p) \coloneqq f(v_p)+f(v_l)$ and $g(v_p) \coloneqq g(v_p)+g(v_l)$. Besides, G' is updated by removing vertices in U from the graph. Line 11 adds $\langle v_p, l\rangle$ to φ. The key v_p represents the root of the subtree and the value l is the schedule of vertices in the subtree. When there is only one vertex left in the graph, the scheduling is finished. We use v_r to denote this vertex, considering it is the root of the original tree. $\varphi(v_r)$ is returned as the final schedule in line 12.

The list merging algorithm is shown in Alg. 4. Intuitively, we iteratively compare the head of every list and append the smallest element to the final merged list. When comparing two elements in the list, we use Johnson's rule shown in Alg. 1 to determine their relative order. Specifically, line 1 initializes a list l to store the final result and a set T to keep the head elements of lists in L. In lines 2 to 8, the smallest head element is iteratively chosen and added to l, until L becomes empty. Lines 3 and 4 peek head elements and put them in T. Line 5 finds the smallest element based on Johnson's rule. Each element is a tuple $(f(v), g(v))$ representing the computation and communication time of a DAG vertex. For two tuples $(f(a), g(a))$ and $(f(b), g(b))$, we first determine whether they are communication-heavy or computation-heavy. If both of them are communication-heavy, then $(f(a), g(a)) < (f(b), g(b))$ if $f(a) < f(b)$. If they are computation-heavy, $(f(a), g(a)) < (f(b), g(b))$ if $g(a) > g(b)$. If they are different types, then the communication-heavy one is smaller. Let m_r denote the smallest element selected in line 5. Then, it is appended to the merged list l in line 6. Also, m_r is removed from the list l_r in which it originally is. If l_r becomes empty, the list is removed from L in line 7. Finally, l is returned as the final merged list in line 8.

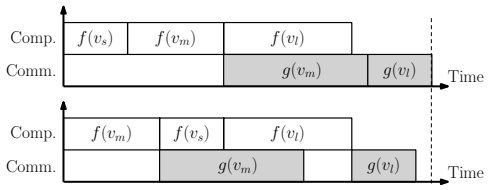


Fig. 5. Switching v_s and v_m will not enlarge the completion time.

***Theorem** 1:* The processing schedule generated by Alg. 3 is optimal for tree-structure DAGs.

Proof: We use mathematical induction to prove this theorem. The base case is a subtree that only contains one vertex. Obviously, our algorithm can generate the optimal schedule for the base case. For other cases, we use v_s to denote the root of a subtree. Let U denote the set of direct children of v_s. Then, we need to prove that if optimal schedules of every subtree rooted at $u_i \in U$ are known, our algorithm can generate the optimal schedule for the subtree rooted at v_s.

Let l_i denote the optimal schedule of the subtree rooted at $u_i \in U$. Our scheduling algorithm merges $l_i \forall u_i \in U$ into a single list. When extending the result list in the merging process, our algorithm iteratively compares and selects the head element of every $l_i \forall u_i \in U$. Hence, the relative order of each l_i is maintained in the final result list. Besides, the order of element selection is based on Johnson's rule which is optimal [24]. Therefore, the result list is optimal.

The result list contains all vertex in the subtree rooted at v_s, except v_s itself. Hence, we also need to prove the list is still optimal after grouping v_r with the first element in the list. Let v_l denote the element corresponding to the first component in the result list. We will show that v_s and v_l are adjacent in one of the optimal schedules of G'. Assuming v_s and v_l are not adjacent in any optimal schedule, then there is at least one vertex v_m that stands between v_s and v_l. Note that v_m is a vertex in graph G' but does not belong to the subtree rooted at v_s since v_l is the first element in the subtree schedule. Switching the position of v_s and v_m will not enlarge the overall completion time. Specifically, $g(v_s) = 0$ since v_s is a internal vertex that cannot be a cut-point. By the definition in Section III-B, only if a vertex is a cut-point, it has nonzero communication time. As illustrated in Fig. 5, the completion time of v_l before switching is $t_a = f(v_s)+f(v_m)+\max\{f(v_l), g(v_m)\}+g(l)$. After switching, it becomes $t_b = f(v_m)+\max\{f(v_s)+f(v_l), g(v_m)\}+g(v_l)$. The difference $t_a - t_b = \max\{f(v_s)+f(v_l), g(v_m)+f(v_s)\} - \max\{f(v_s)+f(v_l), g(v_m)\} \geq 0$. The insight is that the computation completion time would not change because of scheduling, but the communication may start earlier. Thus, switching v_s and v_m will not enlarge the completion time. After switching, v_s and v_l are adjacent eventually. Because switching cannot delay the completion time, the schedule after switching is optimal. It contradicts to the assumption that v_s

Algorithm 5 General-structure DNN Scheduling

Input: General-structure graph $G' = (S, (S \times S) \cap E))$
Output: The schedule σ for all $v \in S$

1: Initialize the schedule list $\sigma \leftarrow []$
2: **while** S is not empty **do**
3: $T \leftarrow \{v_i \in S | (v_i, v_j) \notin (S \times S) \cap E, \forall v_j \in S\}$
4: Communication-heavy set $T_1 \leftarrow \{v \in T | f(v) < g(v)\}$. Computation-heavy set $T_2 \leftarrow \{v \in T | f(v) \geq g(v)\}$
5: $\sigma_1 \leftarrow$ Sort $v \in T_1$ for ascending order of $f(v)$. $\sigma_2 \leftarrow$ Sort $v \in T_2$ for descending order of $g(v)$. $\sigma \leftarrow \sigma_1 || \sigma_2 || \sigma$
6: Update $S \leftarrow S \backslash T$. Remove corresponding edges in G'.
7: **return** σ as the schedule list

and v_l are not adjacent in optimal schedules. Therefore, we can group them together without losing the optimal schedule.

Above all, the schedule found by Alg. 3 is optimal. ■

***Theorem** 2:* The time complexity of scheduling a k-ary tree using Alg. 3 is $O(n^2)$, where n is the number of vertex in G'.

Proof: We first investigate the time complexity of the merging algorithm shown in Alg. 4. Assume that the total number of elements in the final merged list is m. Then, the time complexity of Alg. 4 is $O(|L|m)$, where $|L|$ is the number of lists. Specifically, each selection of element in the final list costs $O(|L|)$ since it needs $|L| - 1$ times comparison to determine the smallest head of $|L|$ lists. If there are m elements in the merged list, the time consumption is $O(|L|m)$.

For a k-ary tree, Alg. 3 needs to merge k scheduling lists at every internal node. For each time of merging, there are $O(n)$ nodes in the merged list. Hence, each merging operation costs $O(kn)$ time. There are at most $O(n)$ internal nodes. Therefore, the time complexity of Alg. 3 is $O(n) \times O(kn) = O(n^2)$. ■

Actually, the real execution time of Alg. 3 can be significantly reduced by the grouping trick. After grouping, the number of elements in the merged group is at most m instead of n, where m is the number of leaf nodes. For example, for a full binary tree, $m = (n + 1)/2$. The grouping trick can roughly reduce the execution time by half, which is helpful for efficient scheduling.

V. Scheduling for General DAGs

For some special DNN models, such as RandWire [29], the partitioned network G' left on mobile devices is not a tree. We propose to combine ideas of topological sort and Johnson's rule to schedule those general DAGs. The topological sort is used to find feasible processing sequences for a general DAG. However, there are lots of feasible topological sequences. We use Johnson's rule to pick one among those sequences. Briefly, we iteratively apply Johnson's rule on a set of DAG vertices with no outgoing edges until all vertices are scheduled. Different from the standard topological sort algorithm, we iteratively work on vertices with no outgoing edges instead of incoming edges.

Detailed procedures of the general-structure DAG scheduling algorithm are illustrated in Alg. 5. We first initialize an empty list σ. In the loop of lines 2-6, we iteratively schedule vertices with no outgoing edges. Set T in line 4 contains those vertices. We apply Johnson's rule on T in lines 5 and 6. Vertices in T are divided into two groups T_1 and T_2 according to their computation and communication time cost. T_1 and T_2 are sorted into lists σ_1 and σ_2, respectively. The sorted lists are concatenated before σ. The concatenation is denoted by $||$. We still can group vertices with no outgoing edges with their predecessors. A vertex may have multiple predecessors in general DAGs. We can arbitrarily choose one. However, we cannot guarantee the result is optimal. Line 6 removes scheduled vertices from S and updates the graph G' for the next iteration. Finally, σ is returned as the final schedule.

Because of the complex precedence constraints in general DAGs, we cannot guarantee the schedule generated by Alg. 5 is optimal. This limitation does not significantly affect the real-world performance of our offloading pipeline. For most widely used DNN models, there are repeated modules in their structures, such as residual blocks. If we treat each repeated module as a block, most of DNNs have tree-structures. Even if a block is partitioned, the partitioned DNN left on mobile devices usually has a tree structure. To the best of our knowledge, only some experimental DNN models, such as RandWire [29], have general DAG structures.

VI. Experiment

A. System Setup

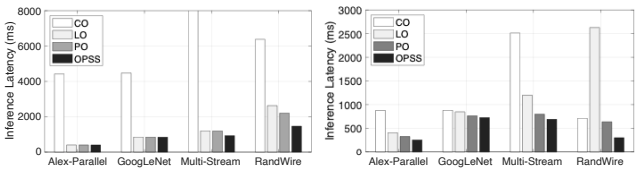
Our cooperative DNN offloading system contains a mobile device and a cloud server. The mobile device would initialize a DNN inference task and offload a part of the task to the cloud server. In our testbed, a Raspberry Pi model 4B is used as the mobile device and a PC is used as the cloud server. The Raspberry Pi is a tiny single-board computer that equips a quad-core CPU (Cortex-A72) and 4GB RAM. It runs the Raspberry Pi OS which is based on Debian Linux. The PC in our lab equips a six-core CPU (Intel i7-8700), a GTX 1080 GPU, and 32GB RAM. Its operating system is Ubuntu 20.04. The cloud inference is performed on the GPU. The mobile device and the cloud server are placed within the same Wi-Fi network to simulate the wireless communication. Besides, the bandwidth of the communication channel is controlled by using the `wondershaper` package to simulate different wireless environments.

Both client and server sides are implemented in Python. The client-side refers to the mobile device and the server refers to the cloud. On both sides, we use `PyTorch` as the DNN inference engine. Although `PyTorch` has its distributed computing package, it does not provide APIs for layer-wise or path-wise DNN offloading to the best of our knowledge. We use `gRPC` to implement our DNN offloading scheme. To achieve the path or layer level offloading, we create the submodels for each path according to the partition and scheduling outcomes. In our experiment, DNN models are pre-cut at all possible points and are pre-allocated at both client and server sides. Each `gRPC` message would contain a string to indicate the submodel and a byte array to store its output.

Specifically, for a DNN inference task, the client first performs DNN partition and scheduling. After that, it loads input images and transforms them into tensors. Then, the client performs the forward propagation by calling the corresponding submodels according to the schedule. For the output tensor of each submodel, the client needs to encode it for serialization before the network transmission. We use the `tensor.save()` API to perform the serialization. The encoded tensor is flushed into a `BytesIO` which is a virtual interface in memory. The client then assembles a `gRPC` request with the serialized tensor and enqueue it for communication. On the server-side, the request would be loaded from the `BytesIO` and decoded with the `tensor.load()` API. Then, the server PC performs a forward propagation using the decoded tensor. To reduce the response time, the server runs all inference tasks on its GPU with `CUDA`. The final output of the DNN inference is sent back to the client with a `gRPC` reply message. Our testbed supports any clients that running on Linux kernel based operating systems. Also, it can be easily extended to support other systems since `gRPC` works across a variety of platforms.

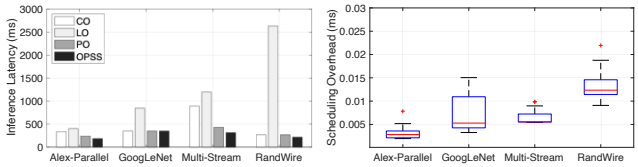
Before partition and scheduling, it is important to accurately estimate the computation and communication time. To reduce the estimation overhead for computation time, each device uses a time table to store the inference time of each layer. Considering the DNN inference time on a specific device is much more stable than the communication time, using a time table is sufficient for the estimation. Besides, the lookup table would only take a small portion of memory space since the number of commonly used DNN types is limited. Mobile devices use `PyTorch Profiler` to measure the computation time and fulfill the table before scheduling. Each device only needs to run the profiler once when it starts. The communication time varies with the network environment. As explained in Section III-B, we use a linear regressing model to estimate the communication time. The regression model needs to be pre-trained before scheduling. Specifically, each mobile device needs to sample the communication delay by sending spatial `gRPC` requests. Sizes of those requests are pre-defined. The client records the response time of those requests and then trains the regression model based on the sampled information. During scheduling, the communication time $g(v)$ of layer v can be quickly calculated using this regression model. It is worth noting the regression model should be updated regularly to detect the changes of the network environment.

We use widely deployed DNN models to validate the proposed algorithms. Specifically, we use Alex-Parallel [7] as the path-independent graph. For tree-structure DAGs, we use GoogLeNet [25] and Multi-Stream network [30]. GoogLeNet contains several Inception modules. If the module is not partitioned, the entire module is treated as a vertex. If the module is cut, each layer left in the module is viewed as a vertex. For general-structure DAGs, we use RandWire [29]. There are no specific modules in the RandWire, and each layer is modeled as a vertex in DAG.



(a) 3G (1.1 Mbps) (b) 4G (5.85 Mbps)

Fig. 6. Comparison on different network environment.



(a) Inference latency (b) Scheduling Overhead

Fig. 7. Comparison on Wi-Fi network.

B. Experiment Results

We first introduce the comparison algorithms. In the experiment, our Offloading Pipeline Scheduling Scheme is label as OPSS. We use LO to denote the local-only scheme. It means that all DNN inference computations are completed on mobile devices. Besides, we also compare OPSS with CO which means the cloud-only scheme. In CO, mobile devices directly update the input tensor to the cloud, and all computations are done by the cloud. Another comparison algorithm is PO which represents the state-of-the-art partition-only scheme [6]. When comparing PO with OPSS, the DNN are partitioned at the same place. The difference is that OPSS uses pipeline to hide its communication time.

Fig. 6 shows the performance comparison on different network environments. Same as [6], the typical bandwidth of 3G and 4G networks are set to 1.1 Mbps and 5.85 Mbps, respectively. From 6(a), we can find that OPSS significantly reduces the inference latency for Multi-Stream and RandWire. However, the speedups on Alex-Parallel and GoogLeNet are not obvious. It is because the communication time in offloading is longer than the local processing time when the network bandwidth is relatively low. In this case, processing the entire model on the mobile device is the best strategy. After we increase the network bandwidth to 5.85 Mbps, OPSS can efficiently reduce the inference time for Alex-Parallel and GoogLeNet as shown in Fig. 6(b). From the figure, we also notice that OPSS can further reduce the inference latency, comparing with the state-of-the-art DNN partition algorithm. It shows that our scheduling algorithm can efficiently hide the communication time behind computation. In addition, comparing the speedup of different DNN models, we notice that the benefits of scheduling are more obvious when the DNN model is complex. There are more optimization opportunities in complex DAGs. Both our tree-structure DAG scheduling algorithm and the general DAG scheduling algorithm can adaptively reduce the inference latency in different network environments.

Fig. 7 shows the comparison on Wi-Fi network. The typical bandwidth of Wi-Fi is 18.88 Mbps which is much faster than 3G and 4G. In this case, offloading more computation load to the cloud is a better choice unless the volume of intermediate results is extremely large. In Fig. 7(a), although OPSS still outperforms PO, the gap between them becomes smaller compared with the 4G case. This result is reasonable. When the bandwidth is large, the communication time is small. Hiding the small communication time by using the pipeline would not significantly reduce the overall inference latency. This inspires us to investigate the feasible region in which OPSS can benefit from pipeline scheduling. Besides the inference latency, we also investigate the scheduling overhead of OPSS on different DAG structures. Fig. 7(b) shows the scheduling overhead of OPSS. We repeatedly run our scheduling algorithm on different DNN models and show the overhead distribution on each model. From the figure, we notice that the overhead is negligible compared with the inference time. After partition, the number of cut-points is usually less than 10. Therefore, our scheduling algorithm for tree-structure DAGs can efficiently find the optimal scheduling with the grouping trick. More importantly, we use a lookup table to store the local inference time. It saves the time cost of profile estimation. In addition, the communication time is estimated by using a simple linear regression model which is also time-efficient.

VII. Conclusion

This paper investigates the pipeline scheduling problem for cooperative DNN inference. The complex precedence constraints in general DAGs bring challenges for the scheduling problem. We utilize DAG structures to optimize the scheduling problem. For path-independent DAGs, we apply Johnson's rule to generate optimal path-wise scheduling. A graph conversion algorithm is proposed to find path-wise scheduling for arbitrary tree-structure DAGs. However, the optimal path-wise schedule may be suboptimal after conversion. We propose another layer-wise scheduler that generates an optimal schedule for any tree-structure DAGs. For general-structure DAGs, we propose a heuristic scheduling algorithm based on the topological sort. Experiment results on a real-world testbed show that our pipeline scheduling scheme can significantly reduce the inference latency.

References

[1] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *ICML*, 2008, pp. 160–167.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012, pp. 1097–1105.

[3] J. Wang, Z. Feng, Z. Chen, S. George, M. Bala, P. Pillai, S.-W. Yang, and M. Satyanarayanan, "Bandwidth-efficient live video analytics for drones via edge computing," in *IEEE/ACM SEC*, 2018, pp. 159–173.

[4] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[5] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive dnn surgery for inference acceleration on the edge," in *IEEE INFOCOM*, 2019, pp. 1423–1431.

[6] S. Zhang, Y. Li, X. Liu, S. Guo, W. Wang, J. Wang, B. Ding, and D. Wu, "Towards real-time cooperative deep inference over the cloud and edge end devices," *ACM Ubicomp*, vol. 4, no. 2, pp. 1–24, 2020.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE CVPR*, 2016, pp. 770–778.

[8] 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," *arXiv preprint arXiv:1602.07360*, 2016.

[9] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[10] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *IEEE CVPR*, 2017, pp. 1492–1500.

[11] R. Huang, J. Pedoeem, and C. Chen, "Yolo-lite: a real-time object detection algorithm optimized for non-gpu computers," in *IEEE Big Data*, 2018, pp. 2503–2510.

[12] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren, "Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning," in *ACM ASPLOS*, 2020, pp. 907–922.

[13] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *ACM MobiSys*, 2016, pp. 123–136.

[14] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. of the IEEE*, 2017.

[15] P. Xu, X. Zhang, C. Hao, Y. Zhao, Y. Zhang, Y. Wang, C. Li, Z. Guan, D. Chen, and Y. Lin, "Autodnnchip: An automated dnn chip predictor and builder for both fpgas and asics," in *ACM FPGA*, 2020, pp. 40–50.

[16] X. Liu, Q. Yang, J. Luo, B. Ding, and S. Zhang, "An energy-aware offloading framework for edge-augmented mobile rfid systems," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 3994–4004, 2018.

[17] S. Dey, A. Mukherjee, A. Pal, and P. Balamuralidhar, "Partitioning of cnn models for execution on fog devices," in *Proc. of ACM International Workshop on Smart Cities and Fog Computing*, 2018, pp. 19–24.

[18] X. Wang, Y. Han, C. Wang, Q. Zhao, X. Chen, and M. Chen, "In-edge ai: Intelligentizing mobile edge computing, caching and communication by federated learning," *IEEE Network*, vol. 33, no. 5, pp. 156–165, 2019.

[19] S. Zhang, Y. Li, B. Liu, S. Fu, and X. Liu, "Enabling adaptive intelligence in cloud-augmented multiple robots systems," in *IEEE SOSE*, 2019, pp. 338–3385.

[20] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *IEEE CVPR*, 2017, pp. 7263–7271.

[21] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "Deepburning: automatic generation of fpga-based learning accelerators for the neural network family," in *ACM/EDAC/IEEE DAC*, 2016, pp. 1–6.

[22] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *IEEE ICDCS*, 2017, pp. 328–339.

[23] Y. Duan, N. Wang, and W. Jie, "Reducing makespans of dag scheduling through interleaving overlapping resource utilization," in *IEEE MASS*, 2020, pp. 392–400.

[24] S. M. Johnson, "Optimal two-and three-stage production schedules with setup times included," *Naval research logistics quarterly*, vol. 1, no. 1, pp. 61–68, 1954.

[25] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE CVPR*, 2015, pp. 1–9.

[26] P. Brucker and P. Brucker, *Scheduling algorithms*. Springer, 2007.

[27] J. M. Framinan, J. N. Gupta, and R. Leisten, "A review and classification of heuristics for permutation flow-shop scheduling with makespan objective," *Journal of the Operational Research Society*, vol. 55, no. 12, pp. 1243–1255, 2004.

[28] N. Wang, Y. Duan, and J. Wu, "Accelerate cooperative deep inference via layer-wise processing schedule optimization," in *IEEE ICCCN*, 2021, pp. 1–9.

[29] S. Xie, A. Kirillov, R. Girshick, and K. He, "Exploring randomly wired neural networks for image recognition," in *IEEE ICCV*, 2019, pp. 1284–1293.

[30] Y.-W. Chao, Y. Liu, X. Liu, H. Zeng, and J. Deng, "Learning to detect human-object interactions," in *IEEE WACV*, 2018, pp. 381–389.