Optimizing Job Offloading Schedule for Collaborative DNN Inference

Yubin Duan, *Student Member, IEEE,* and Jie Wu, *Fellow, IEEE*

Abstract—Deep Neural Networks (DNNs) have been widely deployed in mobile applications. DNN inference latency is a critical metric to measure the service quality of those applications. Collaborative inference is a promising approach for latency optimization, where partial inference workloads are offloaded from mobile devices to cloud servers. Model partition problems for collaborative inference have been well studied. However, little attention has been paid to optimizing offloading pipeline for multiple DNN inference jobs. In practice, mobile devices usually need to process multiple DNN inference jobs simultaneously. We propose to jointly optimize the DNN partitioning and pipeline scheduling for multiple inference jobs. We theoretically analyze the optimal scheduling conditions for homogeneous chain-structure DNNs. Based on the analysis, we proposed near-optimal partitioning and scheduling methods for chain-structure DNNs. We also extend those methods for general-structure DNNs. In addition, we extend our problem scenario to handle heterogeneous DNN inference jobs. A layer-level scheduling algorithm is proposed. Theoretical analyses show that our proposed method is optimal when computation graphs are tree-structure. Our joint optimization methods are evaluated in a real-world testbed. Experiment results show that our methods can significantly reduce the overall inference latency of multiple inference jobs compared to partition-only or schedule-only approaches.

Index Terms—collaborative DNN inference, job offloading, makespan minimization, mobile cloud computing, pipeline scheduling

1 Introduction

Machine learning technologies have been applied to a wide range of mobile applications. Among abundant leaning methods, deep neural networks (DNNs) are attracting more attention with the rapid growth of available training data. Considering the constrained computational power on mobile devices, most mobile applications rely on pre-trained DNN models instead of training DNN models on device. Performing inference on pre-trained DNN models requires much less computational power compared to training itself. There are lots of frameworks that are proposed with the intent of accelerating DNN training for GPU clusters [2] or IoT devices [3]. Nevertheless, it is also critical to reduce the DNN inference latency for mobile devices.

Collaborative inference is a promising approach to reduce the DNN inference latency for mobile devices. In particular, [4] observed that the first several layers in a DNN model usually extract features from input data and notably reduce the data volume with slight computational costs. [4] further proposed that properly partitioning a chain-structure DNN model and offloading partial inference workloads from mobile devices to cloud servers can significantly reduce the DNN inference latency compared to performing inference solely on mobile devices. The general DNN structure can be modeled by directed acyclic graphs (DAGs). [5], [6] proposed partitioning schemes for DNN models that have complex DAG structures. With partitioned DNN models, mobile devices would perform inference or forward propagation on the partial models assigned to them and offload intermediate results to cloud servers. Once cloud servers receive intermediate results, they would complete the remaining inference workload and send inference results back to mobile devices. However, existing collaborative inference methods mainly focus on partitioning a single DNN model and pay no attention to scheduling multiple DNNs simultaneously.

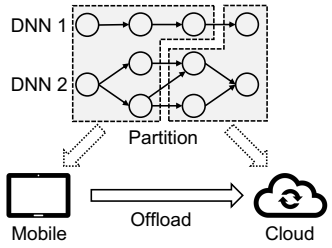
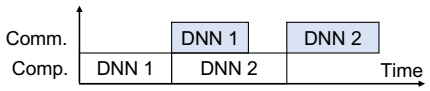


Fig. 1. An overview of the DNN partitioning and offloading.

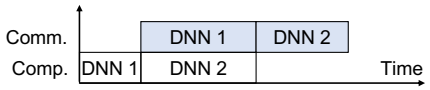
In real-world applications, mobile devices usually need to process multiple DNN inference tasks at the same time. For example, there are usually multiple cameras or sensors installed in IoT devices, such as virtual reality devices or autonomous vehicles [7]. Those sensors collect many data samples simultaneously. To analyze these data samples, multiple DNN inference tasks must be initialized. Those tasks can be homogeneous or heterogeneous, depending on the application scenarios. When the data is collected from the same type of IoT devices, homogeneous DNN inference tasks (i.e., inference tasks on the same DNN model with different input samples) are launched to process the data. For more general cases, if the data samples are collected from multiple types of IoT devices, the corresponding DNN inference tasks are heterogeneous. A typical application scenario is autopilot, in which the road condition data is

- *Yubin Duan and Jie Wu are with Temple University, Philadelphia, USA. E-mail: yubin.duan@temple.edu, jiewu@temple.edu*
- *This research was supported in part by NSF grants CNS 2128378, CNS 2107014, CNS 1824440, CNS 1828363, and CNS 1757533.*
- *This paper is an extended version of the conference paper [1] published in ICPP 2021.*

Manuscript received May 15, 2023.



(a) Optimal partition for DNN 1



(b) Optimal partitions for DNNs 1 and 2

Fig. 2. Illustrations of offloading pipelines.

collected by ultrasonic sensors, LiDAR sensors, and others. Multiple heterogeneous DNN inference jobs can be triggered simultaneously to process the data. For both homogeneous and heterogeneous cases, reducing their processing makespan is critical for these time-sensitive applications.

Existing work for accelerating the cooperative DNN inference process on mobile devices mainly focuses on optimizing the DNN partition strategy [4], [6], [8], [9]. In particular, those frameworks would decide whether a DNN layer should be processed locally or remotely given the computation graph of a DNN model. To the best of our knowledge, existing work pays little attention to the offloading pipeline. We notice that carefully scheduling the offloading sequence can significantly reduce the inference latency, which motivates us to investigate the offload pipeline scheduling problem for cooperative inference. Moreover, although existing DNN partition methods can be directly applied to each job when there are multiple inference jobs, the generated processing schedule may be sub-optimal. This is because the DNN partitioning and pipeline scheduling strategies interact with each other. This motivates us to jointly consider the DNN partitioning and pipeline scheduling problem. A motivation example is illustrated in Fig. 2.

Fig. 2 illustrates the offloading pipeline for collaborative DNN inference. In the computation phase, mobile devices perform DNN inference locally. In the communication phase, intermediate inference results are sent from mobile devices to cloud servers. We ignore the time consumption of sending inference results from cloud servers back to mobile devices, since sizes of inference results are usually much smaller than sizes of intermediate results. Fig. 2(a) shows that organizing computation and communication phases in pipelines can hide communication time behind computation time. In addition, Fig. 2(b) shows that optimally partitioning every DNN in the pipeline may lead to a suboptimal latency of processing all DNNs. Specifically, DNNs 1 and 2 are optimally partitioned in Fig. 2(a). The individual latency of each DNN is minimized. However, the latency of processing those two DNNs in the example can be further reduced, as shown in Fig. 2(b). The latency of DNN 1 is enlarged, but the completion time of both tasks is minimized. Therefore, jointly considering DNN partitioning and offload pipeline scheduling can further reduce the overall latency for multiple DNN inference tasks. This example illustrates the motivation for organizing the computation and communication operations in a pipeline to reduce the end-to-end latency. Existing research [10] has shown that the pipeline approach can hide the communication time behind the computation time and improves the utilization of computational and communication resources.

It is not trivial to jointly optimize DNN partitioning and offloading pipeline scheduling. Merely optimizing the schedule of offloading pipelines is challenging. In particular, DNN layers may have complex precedence constraints among each other. Scheduling DAG-style DNN layers can be categorized as a DAG shop scheduling problem, which is NP-hard [11]. In addition, the optimal offloading schedule depends on the network environment. The offloading schedule needs to be updated when the network environment changes. Jointly considering partitioning and pipeline scheduling brings additional challenges. Scheduling and partitioning strategies are correlated. Considering those challenges, we first integrate our observations to analyze optimal conditions for chain-structure DNNs and then extend our solution to more general cases.

Some useful observations of chain-structure DNNs can be used to simplify the optimal condition analysis. In particular, the first several layers in chain-structure DNNs are usually used to extract features from input data. The output tensor sizes of those layers become smaller with the layer depth. For convolutional neural networks, the tensor size usually decreases exponentially. The observation shows that the offloading traffic volume is usually decreasing as the partition layer moves down the DNN layers. Even if the tensor sizes increase or remain after passing some intermediate layers, we can group those layers as a virtual block without losing the optimal partition. This is because partitioning after layers in the virtual block cannot reduce the communication cost but increases the computation cost for processing more layers on mobile devices. In contrast, the local computation time increases if more layers are assigned to mobile devices. According to the observation, we restrict the trends of computation and communication time in our analysis.

With our observation, we theoretically analyze the optimal conditions of joint DNN scheduling and partitioning for homogeneous chain-structure DNNs. In addition, a joint optimization scheme is proposed to minimize the makespan of cooperative DNN inference for homogeneous chain-structure DNN inference jobs. We also extend the scheme to process general-structure DNNs. Moreover, we extend our problem scenario to deal with heterogeneous DNN inference jobs. Different from the homogeneous case, we can no longer find a common partition point for heterogeneous DNNs since they may have different structures and sizes. Therefore, we cannot directly apply the algorithms proposed for the homogeneous case to schedule heterogeneous DNNs. A layer-level DNN scheduling method is proposed. We show that our proposed method can optimally schedule tree-structure DNN graphs. Our proposed methods are evaluated in real-world testbeds. Evaluation results show that our joint optimization methods can significantly reduce the inference makespan, compared to merely considering DNN partitioning or scheduling.

Our contributions are summarized as follows:

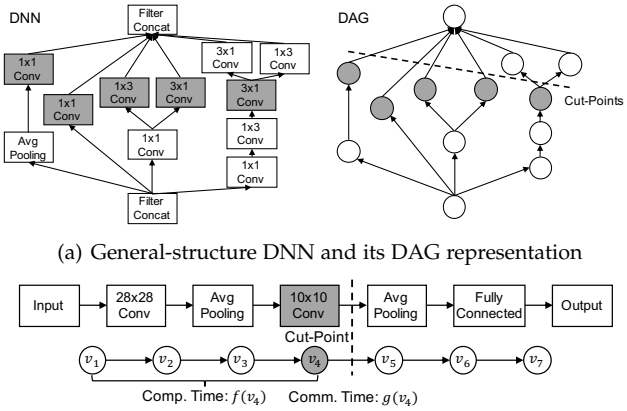
- We propose a joint optimization problem that aims to minimize the makespan of multiple cooperative DNN inference jobs. Our problem jointly considers DNN partitioning and pipeline scheduling.
- We show the optimal conditions of our joint op-

timization problem when DNN inference jobs are homogeneous and DNN models are chain-structure. We present a joint optimization method for homogeneous chain-structure DNNs and extend it for homogeneous general-structure DNNs.
- We have extended the problem scenario discussed in [1] to deal with heterogeneous DNNs. Different from the homogeneous case discussed [1], heterogeneous DNNs have no common partition points. A large number of combinations of partition points brings additional challenges for the heterogeneous case.
- The scheduling algorithms proposed for the homogeneous case cannot be applied to the heterogeneous case. In this paper, a novel layer-level scheduling method is proposed for the heterogeneous case.
- The optimal conditions of the heterogeneous scheduling problems are different from the homogeneous case. We have analyzed in which condition our heterogeneous DNN scheduling method is optimal.
- We evaluate our methods in real-world testbeds. Experiment results on AlexNet, MobileNet, ResNet, and GoogLeNet, show that our proposed method significantly reduces the inference makespan, compared to partition-only or schedule-only schemes.

2 Related Work

Cloud/edge offloading approaches investigate the collaboration between local and remote computation resources. Neurosurgeon [4] proposed the idea of computation offloading for DNNs. However, Neurosurgeon only considers the partition of a single DNN. Our paper jointly considers the partition and scheduling of multiple DNNs. Teerapittayanon *et. al.* [8] proposed DDNN to reduce the communication data size in a distributed computing system containing mobile devices and the cloud. Different from their objective, we aim to reduce the makespan that contains both communication and computation latencies. Wang *et. al.* [6] presented an adaptive DNN partition scheme for inference acceleration. [12] further proposed an optimal partition algorithm for tree-structure DNNs. Although the authors considered multiple DNNs, the scheduling of multiple jobs is not discussed. In our paper, we allow DNN offloading stages to work in a pipeline. Carefully scheduling those jobs could further reduce their completion time.

Other inference acceleration approaches include DNN model compression [13], [14], [15], [16], [17], [18] and hardware acceleration [19], [20], [21], [22]. Our methods are compatible with those approaches. In addition, DNN model compression reduces the inference latency by simplifying DNN models and reducing the number of model parameters. This approach investigates the trade-off between computation workload and model performance. The DNN inference latency can be reduced with a slight sacrifice of model performance. The model performance is usually measured by inference accuracy in image classification jobs. Compressing DNN models can efficiently accelerate the DNN inference process for some application scenarios where the slight decrease in model accuracy is acceptable. However, for some other applications, such as autopilots, it is critical to maintaining a high inference accuracy. Our proposed method can reduce the inference latency without affecting the model performance or inference accuracy.



(a) General-structure DNN and its DAG representation

(b) Chain-structure DNN and its DAG representation

Fig. 3. DNNs and its DAG representations.

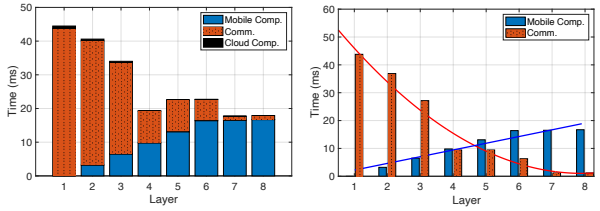
The DAG scheduling problem has been widely studied. Scheduling a DAG-structure graph with multiple phases is NP-hard [23]. Theoretical analyses [24], [25] provide approximation bounds for simple cases. Topocuuglu *et. al.* [26] present an efficient heuristic solution for scheduling DAG-structure jobs over heterogeneous devices. Our problem scenario is different from the existing work. In particular, [26] focuses on finding the optimal task processing sequence among multiple processors without worrying about the execution sequence of communication operations. This is feasible for scheduling tasks among processors, where the communication bandwidth is not a limited resource. However, for task offloading in the mobile computing environment, optimizing the communication operation sequence is critical in our DAG job scheduling problem.

3 Model

3.1 Problem Formulation

We use a Directed Acyclic Graph (DAG) to model a DNN. Formally, let $G = (V, E)$ denote a DAG, where V is the node 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. A weighted edge $e \in E$ represents the data communication between two vertices that are incident to e. The edge weight shows the communication volume. Fig. 3(a) illustrates the structure of inception-v4 [27] and its DAG representation. Although a DNN could have a complex DAG representation, many widely-used DNNs are simple and have chain structures as shown in Fig. 3(b). For example, VGG16 [28], Tiny YOLOv2 [29], and NiN [30] are commonly used in computer vision applications, and all have chain-structure representations with different sizes.

Let J denote the set of DNN jobs, where j is used to index a job and n is the number of jobs. Each job j is a DNN inference task whose computation graph is G. Merely processing the job on mobile devices could be time consuming because of the weak computational power. A better approach is to offload a part of G to a cloud server [4]. The collaboration between mobile devices and the cloud contains three steps: 1) computing parts of G on mobile devices, 2) offloading intermediate results to the cloud server, and 3) computing the remaining parts of G on the cloud. The cloud server needs to send inference results



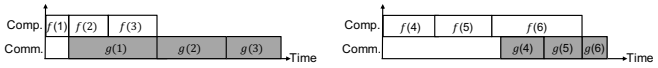
(a) Cloud comp. is negligible (b) Trend of time consumption

Fig. 4. Time consumption of each layer of AlexNet.

back to mobile devices, but this communication volume is small and negligible. In this approach, the computation graph G of each job j is partitioned into two sections, which introduces the model partition problem. After partition, the mobile device needs to determine the processing sequence of partitioned graphs, which is the scheduling problem.

Although jobs have the same DAG structure, they could have different specific partitions. Let $P_j \subset V$ denote the partition of job j and be a set of cut-points. For chain-structure DNNs, cutting one edge is sufficient to split it into two parts, one for local processing and the other for offloading. By definition, the partition set P_j of a chain-structure DNN only contains one element and $|P_j| = 1$. We can choose a partition layer and cut the DNN after the layer. For general-structure DNNs, the set P_j can include multiple cut-points. Specifically, all computation nodes $v \in P_j$ and their predecessors are processed on mobile devices. We use $f(P_j)$ to denote the time consumption of processing those nodes on the mobile device. The output of cut-points $v \in P_j$ needs to be offloaded to the cloud. The time consumption of sending the output is denoted as $g(P_j)$. The value of $f(P_j)$ and $g(P_j)$ could be predicted using regression models [4]. The successors of cut-points $v \in P_j$ are computed on the cloud. The computation power of cloud servers is usually much larger than that of mobile devices. Therefore, the processing time in the cloud is negligible. Fig. 4(a) shows the time consumption of each step when partitioning AlexNet on different layers. The figure shows the cloud processing time is negligible. The notations are illustrated in Fig. 3(b), where v_4 is the cut-point. Nodes v_1, v_2, v_3, v_4 are processed on the mobile device and their time consumption is denoted as $f(v_4)$. The intermediate results generated by v_4 are sent to the cloud, and the communication time is denoted as $g(v_4)$.

After inference jobs are partitioned, the mobile device needs to schedule their processing sequence. For job j, the mobile device need to process the cut-points $v \in P_j$ and their predecessors, which is referred to as the *computation stage* of j. After the computation stage is done, intermediate results are sent to cloud servers, this step is denoted as the *communication stage* of j. When the computation (communication) stage of job j starts, it acquires all computation (network) resources. Otherwise, the makespan may be enlarged [31]. That being said, the computation and network resources could be used in a *pipelined* manner. The computation stage of a job can overlap with the communication stage of another job. The lengths of computation and communication stages of job j are $f(P_j)$ and $g(P_j)$, respectively. The communication stage cannot start until the corresponding computation stage is done. The completion time of job j is denoted as τ_j. All jobs in J are available at the time 0.



(a) Communication-heavy job set (b) Computation-heavy job set

Fig. 5. Illustrations of sorted order sets in scheduling.

3.2 Problem Analysis

The complexity of the problem mainly comes from the correlation between partitioning and scheduling. Specifically, the lengths of the communication and computation stages of job j are functions of partition methods P_j. This shows that the partitioning and scheduling strategies interact with each other, and we need to jointly consider them when optimizing the inference latency. It is difficult to determine what is a good partition of a job, especially when the computation and network resources are used in a pipelined manner. The communication stage of a job could be completely hidden behind the computation stage of the next job, which helps to reduce the overall makespan. Therefore, it is necessary to jointly consider the partition of n jobs. However, examining all possible partitions of n DAGs cannot be done in polynomial time. Formally, assume each DAG has c ways of partitioning. The number of all possible partitions for n DAGs is $O(c^n)$. This leads to the question: do we need to investigate all possible partitions for n DAGs? What is the best partition strategy of each DAG when considering the potential pipelined speedup among multiple jobs?

After investigating some typical DNNs, we notice that f and g functions have useful monotonicity and convexity properties when the DAG has a chain structure, which could help us answer those questions. Many widely used DNNs in computer vision applications have chain structures, such as VGGNet [28] and YOLO [29]. The chain structure makes partitioning easier, since the partition P_j only contains one vertex, and both f and g become unary functions.

More importantly, if we index vertices in the DAG by their depths as shown in Fig. 3(b), then f is monotonically increasing and g is non-increasing. The computation time f increases as the partition layer moves forward because more layers need to be processed on the local mobile device. Admittedly, the communication time may increase as the partition layer goes deep. However, we can cluster the layers, after which the offloading volume increases, as a virtual block without ruining the optimal partition point. Partition after any layer in the virtual block would enlarge the offloading communication volume and the local computation time. Therefore, it must not be the optimal partition point. Our assumptions are feasible after the clustering. We use the DAG in Fig. 3(b) to illustrate the monotone property. If the cut-point changes to v_5 instead of v_4, then the mobile device needs to process an additional average pooling layer that corresponds to v_5. At the same time, the additional average pooling layer could reduce the volume of the intermediate results, and the communication time could be reduced. Other types of layers such as convolutional layers or normalization layers would maintain the intermediate result's size. Therefore, the communication workload is non-increasing. Based on this property, we could simplify the formulation of the makespan.

Algorithm 1 DNN Scheduling Algorithm

Input: Set of partitions $P = \{P_1, P_2, \ldots, P_n\}$.
Output: The optimal schedule for the mobile device.
1: Evaluate $f(P_j), g(P_j)$ with regression models for each $j \in J$.
2: Communication-heavy set $S_1 \leftarrow \{j \in J | f(P_j) < g(P_j)\}$. Computation-heavy set $S_2 \leftarrow \{j \in J | f(P_j) \geq g(P_j)\}$.
3: $S_1 \leftarrow$Sort S_1 in ascending order of $f(P_j)$.
4: $S_2 \leftarrow$Sort S_2 in descending order of $g(P_j)$.
5: $S \leftarrow S_1 || S_2$.
6: **return** S as the optimal schedule.

4 DNN SCHEDULING

4.1 Scheduling for Arbitrary Partitions

Scheduling for partitioned DAGs can be viewed as flow shop problems [23] and can be optimally solved by Johnson's rule [32]. Given the partition P_j for each job $j \in J$, we obtain the value of $f(P_j)$ and $g(P_j)$ by applying regression techniques [4]. With those known values, the scheduling problem can be categorized as a 2-stage flow shop problem. With an objective of minimizing the makespan of all jobs, Johnson's rule [32] can be applied to optimally solve the scheduling problem.

The procedures of the scheduling algorithm based on Johnson's rule are illustrated in Alg. 1. Specifically, at line 1, we evaluate the values of $f(P_j)$ and $g(P_j)$ with regression models. Lines 2-5 show the procedures of Johnson's rule. Jobs in J are first split into two groups. The communication-heavy set S_1 contains all jobs whose communication stage is longer than the computation stage. The computation-heavy set S_2 contains the other jobs. Then, the jobs in the communication-heavy set are sorted in ascending order of their computation stage lengths. The jobs in S_1 are stored in increasing order of $f(P_j)$. Jobs in S_2 are sorted in descending order of $g(P_j)$. Illustrations of the sorted S_1 and S_2 are shown in Fig. 5. Finally, the sorted jobs in S_2 are concatenated after jobs in S_1, and S stores the optimal solution.

4.2 Makespan of Chain-structure DAGs

For chain-structure DAGs, the partition P only contains one element, after the chain-structure DAG is partitioned. Therefore, the functions f and g become unary functions in discrete domains. Let x_j denote the index of the cut-point for DAG j. Then, the makespan $\max_j \tau_j$ has a closed-form formulation. Concatenating the S_2 shown in Fig. 5(b) after the S_1 shown in Fig. 5(a) would cause idle time slots for either computation or communication resource but not both. Therefore, we have the following proposition.

Proposition 1. For chain-structure DAGs, if mobile devices schedule partitioned DAGs based on Johnson's rule, then the makespan of n jobs is $\max_j \tau_j = f(x_1) + \max\{\sum_{i=2}^{n} f(x_i), \sum_{i=1}^{n-1} g(x_i)\} + g(x_n)$.

An illustration of the proposition is shown in Fig. 6. Notably, $l_1, ..., l_n$ in Fig. 6 represents partitions in discrete domain. They are sorted based on Johnson's rule. To ease the formulation of our problem in both discrete and continuous domains, we use $x_1, ..., x_n$ instead of $l_1, ..., l_n$ in the problem formulation. $x_1, ..., x_n$ are also sorted according to Johnson's rule. When n is large or $n \to \infty$, the makespan becomes large or $\max_j \tau_j \to \infty$. Therefore, it is meaningful to investigate the average makespan $(\max_j \tau_j)/n$. To simplify the following analyses, we first rewrite the formulation of the average makespan:

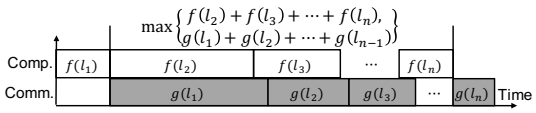


Fig. 6. An illustration of makespan calculation.

$$\lim_{n\to\infty}\frac{\max_j \tau_j}{n} = \lim_{n\to\infty}\frac{\max\{g(x_n)+\sum_{i=1}^{n} f(x_i), f(x_1)+\sum_{i=1}^{n} g(x_i)\}}{n}$$
$$= \lim_{n\to\infty} \max\{\sum\nolimits_{i=1}^{n} f(x_i)/n, \sum\nolimits_{i=1}^{n} g(x_i)/n\}$$

Then, the objective of our optimization problem is equivalent to $\min \max\{\sum_{i=1}^{n} f(x_i)/n, \sum_{i=1}^{n} g(x_i)/n\}$. Notice that n is a finite number in real-world applications, so it is treated as a finite number in the following analysis. Nevertheless, the formulation of the average makespan is still a good approximation. Let k denote the length of the chain-structure DAG, i.e., $k = |V|$. Use $l \in \{1, \ldots, k\}$ to index vertices in V from left to right (source to termination node). Our optimization problem becomes:

$$\text{P1: } \min \quad \max\{\sum\nolimits_{i=1}^{n} f(x_i)/n, \sum\nolimits_{i=1}^{n} g(x_i)/n\}$$
$$\text{s.t.} \quad \prod\nolimits_{l=1}^{k}(x_j - l) = 0, \forall j \in J$$

To simplify the problem, we relax the domain of $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ to real numbers, i.e. $\mathbf{x} \in \mathbb{R}^n$. The relaxed problem becomes:

$$\text{P2: } \min \quad \max\{\sum\nolimits_{i=1}^{n} f(x_i)/n, \sum\nolimits_{i=1}^{n} g(x_i)/n\}$$
$$\text{s.t.} \quad x_j > 0, \forall j \in J$$

5 DNN PARTITION

5.1 Partition for Chain-structure DNNs

The objective of the partition is to minimize the makespan. We first analyze the relaxed problem P2 in continuous domains, then extend results to discrete domains.

In the continuous domain, we use convexity and monotonicity properties of functions f and g to analyze the optimal conditions of the problem. In particular, DNNs are mainly constructed by repeatedly placing blocks of convolution and pooling layers. The computation time of each block is similar. It makes the function $f(x)$ almost increase linearly with x. After each block, the sizes of intermediate results decrease exponentially because of the pooling layers. Even if the pooling layer is not inserted between two blocks, the size would not increase. Hence, $g(x)$ can be fit by a convex function. Based on this observation, we assume f is an increasing linear (also convex) function and g is a decreasing convex function in the following analysis. The functions in the continuous domain are shown in Fig. 7.

When both f and g are convex, the problem P2 becomes a convex optimization problem. More interestingly, the problem holds a strong duality as shown in Lemma 1.

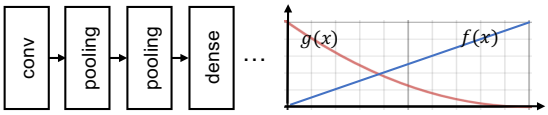
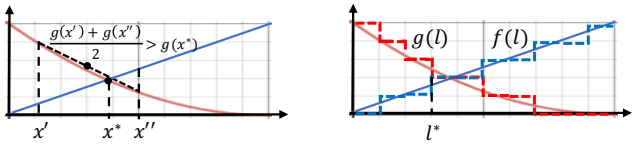


Fig. 7. Continuous time consumption functions.



(a) Continuous domain (b) Discrete domain

Fig. 8. Graph explanation of the optimal partition.

This is mainly because the summation and the maximum of convex functions are still convex.

Lemma 1. Our optimization problem P2 holds a strong duality if both $f(x)$ and $g(x)$ are convex.

Proof: The objective function of P2 is convex when $f(x)$ and $g(x)$ are convex since the summation and the maximum of convex functions are still convex. The constraints are also convex. Therefore, P2 is a convex optimization problem.

The strong duality holds since our convex optimization problem satisfies Slater's condition. Formally, we need to find a point $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ in the feasible solution domain such that $x_i > 0, \forall i = 1, 2, \ldots, n$. By definition, x_i represents the partition for job i and x_i is strictly greater than zero. Such point exists since the partition of each job is independent and can be placed at any layer in middle of the DAG. ■

Because of the strong duality, we can find the optimal solution to the problem according to KKT conditions. As shown in Theorem 1, our analysis reveals an interesting property that all of n identical DNN inference jobs should be cut at the same point when we investigate the problem in the continuous domain.

Theorem 1. After relaxing the partition point into a continuous space, partitioning all homogeneous chain-structure DAGs at the same point could reach the optimal makespan.

Proof. Before proceeding to further analysis, we smooth the max function using the LogSumExp (LSE) function.

$$\max\{\sum_{i=1}^{n} f(x_i)/n, \sum_{i=1}^{n} g(x_i)/n\} = \lim_{\alpha\to\infty} \frac{1}{\alpha} \ln\left(\exp(\alpha \sum_{i=1}^{n} f(x_i)/n), \exp(\alpha \sum_{i=1}^{n} g(x_i)/n)\right).$$

According to Lemma 1, the strong duality holds. Hence, the KKT conditions hold at the optimal point. Specifically, $\mathbf{x}^*$ is the optimal solution to the primal problem if and only if $\nabla_{\mathbf{x}} \lim_{\alpha\to\infty} \frac{1}{\alpha} \ln\left(\exp(\frac{\alpha \sum_{i=1}^{n} f(x_i)}{n}), \exp(\frac{\alpha \sum_{i=1}^{n} g(x_i)}{n})\right) = 0$ and $x_i > 0, \forall i = 1, 2, \ldots, n$.

The gradient of the objective function is formed by a vector of partial orders of x_i. Formally, the partial order of each x_i is $f'(x_i) \exp\left(\frac{\alpha}{n} \sum_{i=1}^{n} f(x_i)\right) + g'(x_i) \exp\left(\frac{\alpha}{n} \sum_{i=1}^{n} g(x_i)\right)$.

For each x_i, at the optimal point, it satisfies

$$f'(x_i)\exp\left(\frac{\alpha}{n}\sum_{i=1}^{n} f(x_i)\right) = -g'(x_i)\exp\left(\frac{\alpha}{n}\sum_{i=1}^{n} g(x_i)\right) \quad (1)$$

In the continuous domain, the computational workload increases along with x, while the communication volume decreases. Therefore, $f'(x) > 0$ and $g'(x) < 0$. Then, $f'(x_i) \exp\left(\frac{\alpha}{n} \sum_{i=1}^{n} f(x_i)\right) > 0$ and $-g'(x_i) \exp\left(\frac{\alpha}{n} \sum_{i=1}^{n} g(x_i)\right) > 0$.

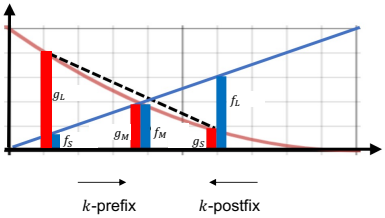


Fig. 9. Comp. and comm. costs of different partitions.

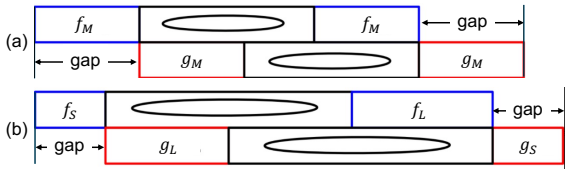


Fig. 10. The uniform partition and our pairing approach.

Take the logarithm of both sides of Eq. (1), we have:

$$\ln(f'(x)) + \frac{\alpha}{n}\sum_{i=1}^{n} f(x_i) = \ln(-g'(x)) + \frac{\alpha}{n}\sum_{i=1}^{n} g(x_i).$$

Rearrange terms in the previous equation, we have:

$$\sum_{i=1}^{n}(f(x_i) - g(x_i)) = (n/\alpha)\ln(-g'(x_i)/f'(x_i)) \quad (2)$$

When $\alpha \to +\infty$, the previous equation becomes $\sum_{i=1}^{n}(f(x_i) - g(x_i)) = 0$, since $-g(x_i)/f(x_i)$ is finite for $x_i > 0$. Let x^* denote the point such that $f(x^*) = g(x^*)$. If we set $x_i = x^*, \forall i = 1, 2, \ldots, n$, Eq. (2) holds for all $i = 1, 2, \ldots, n$. According to the KKT condition, $x_i = x^*, \forall i = 1, 2, \ldots, n$ is an optimal solution to our optimization problem. This shows that when both $f(x)$ and $g(x)$ are convex functions, partitioning multiple homogeneous DNNs at the same location achieves the optimal result. ■

A graph explanation of Theorem 1 is shown in Fig. 8(a). For any partition layer other than x^*, it enlarges either the communication or computation time. Besides, the increment of the communication or computation time of a job cannot be averaged out by pairing it with another job that has a different cut-point. In the example, the average communication time of partitioning at x' and x'' is still larger than optimal.

However, in real-world applications, the partition point is not continuous. The optimal solution we formulated previously may not be able to be reached. In this case, partitioning all DAGs at the same point may no longer be optimal. We investigate the sufficient conditions in which partitioning all DAGs at the same point is still optimal. In addition, we show the conditions in which performing two types of partitions is sufficient to reach the optimal solution.

We further investigate the sufficient condition where performing partition at x^* always outperforms performing two types of partitions at x' and x'' for the discretized version as shown in Fig. 9. In the discretized setting, with g (for communication) being a monotonic non-increasing convex function and f (for computation) being a monotonic non-decreasing convex function, we assume that the set of partitions has a virtual intersection at M, such that $g_M = f_M$ as shown in Fig. 9. In the left most partition, it has the longest comm. bar and the shortest comp. bar, which is denoted as g_L and f_S, respectively. Likewise, the

rightmost partition for comm. and comp. are denoted as g_S for the shortest comm. and f_L for the longest comp. As both g and f are convex, we can show that the intersection point M has the minimum of $\max\{g, f\}$ for any partition. Therefore, $\max\{\sum g_M, \sum f_M\} \leq \max\{\sum g_i, \sum f_i\}$. (Note that M is a fixed value and i is an index.) However, unlike the continuous space shown in Theorem 1, the parts for communication (for g) and computation (for f) are not perfectly aligned. There is a "gap" in front and another "gap" at the back as shown in Fig. 10. They are the initial and lagging phases of the offloading pipeline. The following theorem shows a sufficient condition for the intersection M to be optimal. The gaps shown in Fig. 10 have been taken into consideration when comparing different partition strategies during the proof.

Theorem 2. The uniform partition at virtual intersection point M beats any other types of partitions when $3f_M < f_S + f_L + g_S$ and $3g_M < f_S + g_L + g_S$.

Proof. The proof can be shown by comparing the makespan of two cases shown in Fig.10: the upper one (a) for uniform partition at virtual intersection M and the lower one (b) for any other partition. The oval shaped regions in (a) represent $\sum g_M$ and $\sum f_M$, excluding the leftmost and rightmost partitions. Likewise, the two oval shape regions in (b) are for $\sum g_i$ (summation of communication) and $\sum f_i$ (summation of computation), excluding the leftmost and rightmost partitions. Since $\max\{\sum g_M, \sum f_M\} \leq \max\{\sum g_i, \sum f_i\}$, together with the two conditions of this theorem: $3f_M < f_S + f_L + g_S$ and $3g_M < f_S + g_L + g_S$, we can derive that the makespan of (a) is no more than the makespan of (b). ■

We use the DNN time distribution shown in Fig. 4(a) as an example. Let the uniform partition cut the DNN at layer 4. Then, $f_M = f(4) = 9.8\text{ms}$ and $g_M = g(4) = 9.5\text{ms}$. For simplicity, we assume the other types of partitions are only allowed between layers 3 and 6. Then, $f_S = f(3) = 6.7\text{ms}$, $f_L = f(6) = 16.5\text{ms}$, $g_S = g(6) = 6.4\text{ms}$, and $g_L = g(3) = 27.2\text{ms}$. We can verify that the sufficient condition shown in Theorem 2 holds in this case. Any partition strategies that contain cut-points other than layer 4 would have a larger inference latency compared to the uniform partition. We use a toy-example to verify this point. Assume we have 3 repeated DNN inference tasks. Partitioning them after layers 5 or 6 (layer 3) causes larger communication (communication) time compared to the uniform partition. Partitioning one of them after layer 3 and two of them after layer 6 causes larger communication time and a larger latency of 46.1ms. Partitioning all three DNNs after layer 4, i.e., the uniform partition, is the optimal strategy and achieves a latency of 39.2ms.

We can extend the sufficient condition by considering k left-most partitions (called k-prefix) and k right-most partitions (called k-postfix). k-prefix.g and k-prefix.f correspond to the summation of k left-most communication costs and the summation of k left-most computation costs, respectively, as shown in Fig. 9. k-postfix.g and k-postfix.f are defined in a similar way, but they are applied to the k right-most partitions as shown in the same figure.

Corollary 1. The uniform partition at vitual intersection point M beats any other types of partitions when $(k+2)g_M < k\text{-prefix}.f_S + k\text{-postfix}.f_L + k\text{-postfix}.g_S$ and $(k+2)g_M < k\text{-prefix}.f_S + k\text{-prefix}.g_L + k\text{-postfix}.g_S$.

Proof. This result can be proved using the same idea which was used to prove Theorem 2. ■

Algorithm 2 Chain-structure DNN Partition

Input: Chain-structure DNNs with k layers.
Output: The partition layers of the DNNs and the ratio.

```
Estimate computation and communication time after partitioning of each layer f(l_i), g(l_i).
Initialize the partition points l = 1, r = k.
while l < r do
    mid ← ⌊(l + r)/2⌋.
    if f(mid) < g(mid) then
        l ← mid + 1.
    else
        r ← mid.
Ratio ← ⌊(f(l) − g(l))/(g(l − 1) − f(l − 1))⌋.
return l − 1, l, and Ratio.
```

Inspired by the optimal partition condition of continuous cases, we try to partition the discrete layers such that the difference between f and g values is small. In the discrete domain, f and g have discrete values as shown in Fig. 8(b). Let l denote the partition layer. $f(l)$ is increasing with l and $g(l)$ is non-increasing. Hence, the absolute difference between $f(l)$ and $g(l)$ first decreases along with l, then increases. To find the smallest absolute difference, we only need to find the left-most layer l^* such that $f(l^*) \geq g(l^*)$. If $f(l^*) = g(l^*)$, then cutting n identical DAGs after l^* gives the optimal makespan. To show that this can optimize the makespan, the graph explanation for the continuous case can be directly applied. If $f(l^*) > g(l^*)$, cutting all DAGs at l^* is no longer optimal. We consider using either $l^* - 1$ or l^* as the cut-point for a DAG. Theorem 3 shows that performing those two types of partitions is sufficient to minimize the makespan in certain scenarios.

Theorem 3. When $f(l^* - 1) + f(l^*) = g(l^* - 1) + g(l^*)$ and $g(l^* - 1) = f(l^*)$, performing two types of partitions on different DNNs is sufficient to reach the optimal makespan.

Proof. In this scenario, we partition half of the DNNs after layer $l^* - 1$ and cut the other half after layer l^*. Note that l^* is the left-most layer such at $f(l^*) \geq g(l^*)$. DNNs partitioned after $l^* - 1$ belong to communication-heavy set S_1 in scheduling since $f(l^* - 1) < g(l^* - 1)$. The others belong to computation-heavy set S_2. After concatenating the sorted S_2 after S_1, the communication time is perfectly hidden after computation. When the conditions shown in Theorem 3 are satisfied, swapping a job in S_1 with another job which is partitioned after $l' < (l^* - 1)$ would enlarge the makespan. Although $f(l') < f(l^* - 1)$ after swapping, the communication time increases since $g(l') > g(l^* - 1)$ and it becomes the bottleneck. The increment on the communication time g cannot be hidden behind the computation. Hence, the makespan increases. Similarly, swapping a job in S_2 with another job that is partitioned after $l'' > l^*$ would enlarge the makespan given the conditions shown in Theorem 3. Besides, simultaneously performing the two swapping would not reduce the makespan since

Algorithm 3 General-structure DNN Joint Optimization

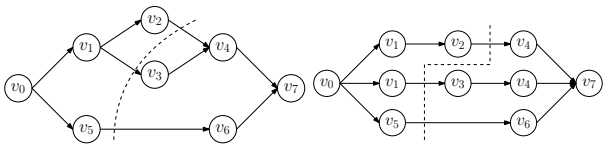
Input: A general-structure DNN.
Output: The partition and scheduling of the DNN.
1: Convert the input into a DAG with independent paths.
2: Initialize the partition set $P \leftarrow \emptyset$.
3: **for** each path i in the DAG **do**
4: $v \leftarrow$ Find the cut-point for path i with Alg. 2.
5: $P = P \cup v$.
6: Schedule the independent paths with modified Alg. 1.

$g(l') + g(l'') \geq g(l^*) + g(l^* - 1)$ when $l' < l^* - 1 < l^* < l''$. If some of the n DNNs are partitioned after layers other than $l^* - 1$ and l^*, its impact on scheduling can be reduced to one of the three cases mentioned above. None of them would reduce the makespan. Therefore, performing two types of partitions is sufficient. ■

The intuition behind Theorem 3 is that performing two types of partitions can still fully fulfilled the offloading pipeline without causing bubble slots if the conditions given in Theorem 3 are satisfied. Given those conditions, combining adjacent operations in the offloading pipeline can still make the pipeline aligned. To satisfy the condition mentioned in Theorem 3, the difference between two adjacent partition layers cannot be drastic. Real-world applications usually do not satisfy the conditions. However, inspired by the Theorem 3, we attempt to reduce the makespan by reducing the accumulated difference between computation and communication time. When performing two types of partitions, we can adjust the ratio between them to reduce the accumulated difference. Specifically, when $f(l^* - 1) - g(l^* - 1) \neq g(l^*) - f(l^*)$, the ratio between the number of DNNs partitioned after $l^* - 1$ with the number of DNNs partitioned after l^* should be $\lfloor (f(l^*) - g(l^*))/(g(l^* - 1) - f(l^* - 1)) \rfloor$.

5.2 Binary-Search-Based Partition Algorithm

When actually partitioning DNN models, we cannot directly apply the solution to Problem P1, i.e., cutting at the virtual intersection of functions f and g. The actual partition is in a discrete domain instead of the continuous domain. Inspired by the analyses for Problem P2, we propose to cut DNN models at the layers that are close to the virtual intersection of f and g. Specifically, we propose to perform two types of partitions at x' and x'', where x' is the closest layer left to x^* and x'' is the closest layer right to x^*. Although the lengths of computation and communication operations are not identical after performing those two types of partitions, their lengths are more likely to be similar compared to other combinations of partition plans. Also, the maximum length of the computation and communication operations are more likely to be minimized since the partition points x^* and x'' are close to the virtual intersection point. Moreover, to reduce the bubble slots caused by the non-identical length of computation and communication operations in the offloading pipeline, we propose to adjust the ratios between the number of those two types of partitions. Considering all those factors, a binary-search-based partition is presented in this subsection as a solution to partition chain-structure DNNs in the discrete domain.



(a) Before conversion (b) After conversion

Fig. 11. An illustration of DAG conversion.

The steps of the partition algorithm are shown in Alg. 2. At line 1, we estimate the communication and computation time of each layer with linear regression models. The time consumption can be accurately estimated according to the layer type and shape, as well as the network bandwidth [4]. After acquiring the values of functions f and g, we initialize two cut-points at line 2. k is the length of the chain-structure DAG. Then, we iteratively update the cut-points by investigating the middle point mid of l and r. If $f(mid) < g(mid)$, the partition layer is located in the right side of mid. We update l into $mid + 1$ accordingly. Otherwise, the partition layer is left of mid and we update r into mid. The while loop terminates when $l = r$. The partition layer could either be l or $l - 1$. The ratio between the number of partitions at $l - 1$ and l is $\lfloor (f(l) - g(l))/(g(l - 1) - f(l - 1)) \rfloor$.

The correctness of the partition algorithm can be guaranteed by the loop invariant. In Alg.2, we can always guarantee that $f(l-1) < g(l-1)$ and $f(r) \geq g(r)$. Within the while loop, if the branch $f(mid) < g(mid)$ shown at line 5 is taken, then we have $f(l - 1) < g(l - 1)$ after l is updated by line 6. On the other hand, if the other branch shown at line 7 is taken, then $f(r) \geq g(r)$ is guaranteed after r is updated. When the while loop terminates, we have $l = r$. Taking those loop invariant properties into consideration, we have $f(l - 1) < g(l - 1)$ and $f(l) \geq g(l)$. The partition layer would be either l or $l - 1$ since any further increments or decrements to l or $l-1$ would enlarge the difference between communication and computation time after partition. The complexity of the search algorithm is $O(\log k)$.

5.3 Partition for General-Structure DNNs

In real-world applications, a DNN model may have more complex structures other than chain-structures. The corresponding DAG contains multiple paths. A path is a subgraph of the DAG which has a chain-structure that starts from the input layer and ends at the output layer. This provides more opportunities to fine-tune the length of the local computation and communication. However, the correlation among paths raises challenges for partition creation.

To decouple the correlation among paths, we convert the general DAG into a multi-path DAG structure without changing the partial order relations, as shown in Fig. 11. We convert each node in their topological orders. For a node, if its out-degree is larger than 1, then we duplicate the node based on its out-degree. Symmetric rules are applied to nodes whose in-degree is greater than 1.

After the conversion, we focus on DAGs with multiple independent paths as shown in Fig. 11(b). Extensively exploring all possible combinations of cut-points in each path is computationally complex. We use a heuristic approach that partitions each path individually. For example, let there be 2 identical DAGs with structure as Fig. 11(a). They are converted into 2×3 individual paths, where 3 is the number

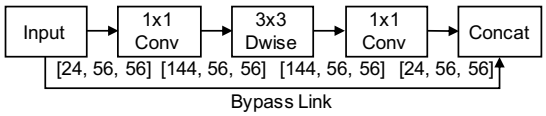


Fig. 12. A bottleneck residual module in MobileNet. (The 3-tuple under each link shows the shape of the tensor transmitted between layers.)

of independent paths in each converted DAG. Procedures of the general-structure DNN partition and scheduling are shown in Alg. 3. Specifically, after converting the input DAG into a DAG with multiple individual paths at line 1, we initialize a cut-point set at line 2. Then, from line 3 to line 5, we find cut-points of different paths individually using the searching algorithm illustrated in Alg. 2. After partition, we use Alg. 1 to schedule the execution of independent paths. Note that a slight modification of Alg. 1 is applied, i.e., duplicated nodes are only counted once when they are executed, although Johnson's rule is applied to all nodes, including duplicated nodes, when scheduling. Specifically, the scheduler would go through the path processing sequence generated by Alg. 2. Following the path processing sequence, the scheduler records the vertices of each path with a lookup table. If a vertex has already been record in the lookup table from previous paths, then the vertex is duplicated for execution. The duplicated nodes would be removed from the path when actually processing the DNN inference job.

6 Scheduling for Heterogeneous DNNs

After introducing the DNN partitioning and scheduling for homogeneous DNNs, we can extend our problem scenario to a more general case where mobile devices need to simultaneously process multiple heterogeneous DNNs. In particular, real-world applications on mobile devices may use multiple DNN models at the same time. For example, some applications need to simultaneously process video and audio inputs [33]. Inference tasks of different DNN models may be called at once, and those DNNs can be chain-structure or general-structure. This is necessary to reduce the overall inference latency of those heterogeneous DNNs. In this section, we present our partitioning and scheduling methods for heterogeneous DNNs. Our major contributions include: 1) introducing an optimal path-level scheduling method, 2) proposing a heuristic layer-level scheduling method for general-structure DAGs, and 3) presenting an optimal scheduling method for tree-structure DAGs.

6.1 Conversion

We propose to convert multiple heterogeneous DNNs into one by introducing some dummy nodes and edges. In particular, we add a dummy source node which is the predecessor of all DNNs' first layers. Notably, inserting the dummy source node would not delay the process of any DNN inference tasks since we assume they arrive at the same time. Similarly, a dummy sink node is added, and it is the successor of all DNNs' last layer. We can view the dummy sink node as a layer that gathers inference results of all DNN inference tasks in J. After inserting the dummy source and sink, we convert the computation graph

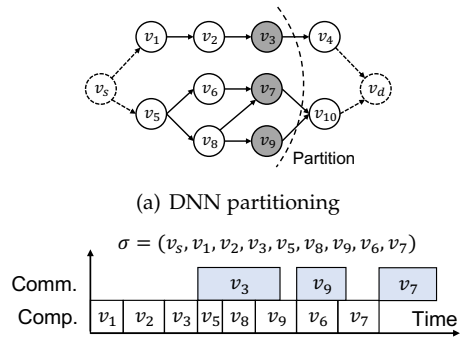


Fig. 13. Partitioning and scheduling heterogeneous DNNs.

of processing multiple heterogeneous DNNs into a DAG. Formally, the converted DAG is denoted as G_J.

Fig. 13(a) illustrates the conversion. In particular, there are two DNNs in the example. One is a chain-structure DNN which consists of layers from v_1 to v_4. The other is a DAG-structure DNN consisting of layers from v_5 to v_{10}. A dummy source v_s and a dummy sink v_d are added to the graph, as well as edges $(v_s, v_1), (v_s, v_5), (v_4, v_d)$, and (v_{10}, v_d). After conversion, two heterogeneous DNNs are merged into a DAG-structure computation graph.

6.2 Partition

With the converted computation graph G_J, we can partition heterogeneous DNNs following the method introduced in Section 5.3. Specifically, we first further convert G_J into a multi-path DAG as illustrated in Fig. 11. Then, we partition each path by following the steps shown in Alg. 2. The insight of the heuristic partition method is that we hope to balance the time consumption of computation and communication phases for each heterogeneous DNN. The balanced partition can reduce the occurrence of bubble or idle time slots in the offloading pipeline. If durations of computation and communication phases vary significantly, the computation phase of a DNN inference task may block the communication phase of the same task, which causes idle slots of communication resources and reduces resource utilization. After partition, let P_G denote the cut-points for G_J. All predecessors of P_G (including nodes in P_G) are processed in local mobile devices. After uploading outputs of P_G to cloud servers, successors of P_G would be processed on the servers. We use G'_J to denote the computation graph that is assigned to local devices after partition. Formally, $G'_J = (V', E')$, where $V' = \{v \in V | v \preceq p, \forall p \in P_G\}$ and $E' = (V' \times V') \cap E$.

However, there is no guarantee that the computation and communication phases have the same length after partition for real-world DNN applications. It is necessary to optimize the processing and offloading schedule for DNN layers in G'_J. We propose to refine the offloading scheduling granularity to shrink the resource idle slots and reduce the overall cooperative inference latency.

6.3 Scheduling for General-Structure Graphs

A simple offloading pipeline scheduling method is treating each path in the converted DAG as an individual task. We

Algorithm 4 Layer-level Heterogeneous DNNs Scheduling

Input: The computation graph $G'_J = (V', E')$ which contains all DNN layers assigned to the local mobile device

Output: A layer-level processing schedule σ for all DNN layers in G'_J

1: Initialize the layer-level schedule σ to an empty list [ ]
2: **while** $V' \neq \emptyset$ **do**
3: Find set of nodes that have no outgoing edges $S \leftarrow \{v_i \in V' | (v_i, v_j) \notin (V' \times V') \cap E', \forall v_j \in V'\}$
4: Communication-heavy set $S_1 \leftarrow \{v \in S | f(v) < g(v)\}$; Computation-heavy set $S_2 \leftarrow \{v \in S | f(v) \geq g(v)\}$
5: $\sigma_1 \leftarrow$Sort $v \in S_1$ in ascending order of $f(v)$; $\sigma_2 \leftarrow$Sort $v \in S_2$ in descending order of $g(v)$.
6: $\sigma \leftarrow \sigma_1 || \sigma_2 || \sigma$
7: Update $V' \leftarrow V' \backslash S$. Remove corresponding edges in G'_J.
8: **return** σ as the schedule list

can follow Johnson's rule explained in Section 4.1 to schedule the processing sequences of those paths in the converted DAG. We denote this approach as path-level scheduling since the scheduling granularity is each path. Path-level scheduling is easy to implement. However, this approach ignores the potential collaboration opportunities among paths. Specifically, it is not necessary to fully complete the forward propagation of a path before processing another path. In the middle of processing a path, mobile devices can switch to process another path if it can reduce the overall latency. Path-level scheduling does not consider this possibility and therefore may be sub-optimal. We propose to fine-tune the offloading schedule by considering layer-level scheduling. Different from path-level scheduling, layer-level scheduling would generate a processing sequence that consists of DNN layers in G'_J instead of paths.

Formally, we use σ to denote the layer-level processing schedule, where σ is a feasible topological sort of nodes in V'. Specifically, σ is a list of size $|V'|$. For $v_i, v_j \in \sigma$, $v_i \prec v_j$ in the list σ if $v_i \prec v_j$ in G'_J. Mobile devices would perform forward propagation of DNN layers following the schedule σ. The computation phases of different layers are not overlapped. Only one forward propagation operation is processed at a time. If a layer does not belong to the cut-point set P_G, its output would not be transmitted to cloud servers. Otherwise, a communication phase would be performed adjacent to the computation phase. Computation and communication phases of different layers can be processed in parallel since they require different types of resources. An illustration of the layer-level processing and offloading pipeline is shown in Fig. 13(b).

Generating the layer-level schedule σ under DAG-style precedence constraints can be categorized as a DAG shop scheduling problem, which is NP-hard [11]. Inspired by topological sort, we propose to iteratively sort DNN layers which have no outgoing edges with Johnson's rule. In this way, we can obtain a feasible topological sort of DNN layers left on mobile devices. In addition, our approach applies Johnson's rule during sorting rather than randomly picking a feasible topological order.

Alg. 4 shows the steps of layer-level scheduling for heterogeneous DNNs. In particular, line 1 initializes the layer-level schedule σ to an empty list. The loop in lines 2-7 iteratively adds DNN layers to σ. In each iteration, we first need to identify DNN layers that have no outgoing edges as illustrated in line 3. Notably, we identify nodes that have no outgoing edges in each iteration, rather than finding vertices with no incoming edges as the standard topological sorting method. Formally, let S denote the set of DNN layers that are contained in the node set V' and have no outgoing edges. Then, line 4 applies Johnson's rule on DNN layers in S. Specifically, DNN layers $v \in S$ are split into communication-heavy set S_1 and computation-heavy set S_2 based on relative lengths of their computation time $f(v)$ and communication time $g(v)$. DNN layers in the communication-heavy set S_1 are further sorted in ascending order of their computation time $f(v)$. DNN layers in S_2 are sorted in descending order of their communication time $g(v)$. The sorting results are stored in lists σ_1 and σ_2, respectively. Line 6 inserts lists σ_1 and σ_2 before the current schedule σ. We use $||$ to denote the concatenation operation. For example, $\sigma_1 || \sigma_2$ means appending the list σ_2 after the list σ_1. In this way, we guarantee that precedence constraints among DNN layers are not violated, and Johnson's rule is applied. DNN layers in S are removed from V' in line 7. This prevents the same layer from being processed repeatedly. After the loop is completed, in line 8, the list σ is returned as the layer-level processing schedule.

Considering the complex precedence constraints in DAG structures, our layer-level heterogeneous DNN scheduling method is sub-optimal. Typically, the DAG scheduling problem is NP-hard [11]. It is difficult to find the optimal solution in polynomial time. Nevertheless, we notice that the computation graph G'_j assigned to mobile devices usually has tree structures. Precedence constraints in tree-structure graphs are less complex and we find an optimal layer-level scheduling method for heterogeneous DNN with tree-structure computation graphs.

6.4 Scheduling for Tree-Structure Graphs

We find that computation graphs assigned to mobile devices usually have tree structures. Typically, fork nodes are close to the input layer in DNN computation graphs and join nodes are close to the output layer. After partition, the computation graph assigned to mobile devices may have no join nodes, and therefore is a tree. For example, we have reviewed the partition results of the DNN models that are pre-defined in the `Torchvision` library. After converting and partitioning, most of the computation graphs G'_J assigned to mobile devices are trees. Compared to the general DAG scheduling problem, scheduling tree-structure graphs is less complex.

We propose to recursively schedule leaf nodes in tree-structure graphs and merge the results. In particular, there are no precedence constraints among leaf nodes. If leaf nodes are siblings (i.e., they have the same parent node), an arbitrary permutation of those nodes would be a feasible processing order. For those sibling nodes, we propose to apply Johnson's rule to determine their processing sequence. We use σ_{v_p} to denote this processing sequence, where v_p is the parent node of those sibling nodes. Then, we let the parent node v_p record the processing sequence σ_{v_p} and remove those sibling nodes from the graph. After removal,

Algorithm 5 Scheduling for Tree-structure Graphs

Input: Computation graph $G'_J = (V', E')$
Output: A layer-level processing schedule σ for G'_J

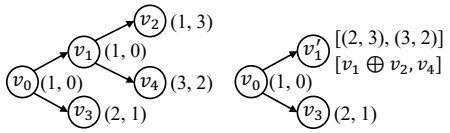
1: $\delta \leftarrow$ an empty dictionary.
2: **for** every leaf node $v \in V'$ **do**
3: Add the key-value pair $\langle v, [(f(v), g(v))]\rangle$ to δ
4: **while** $|V'| > 1$ **do**
5: $v \leftarrow$ the deepest leaf node in G'_J; $v_p \leftarrow$ the parent node of v; $S \leftarrow$ the set of all sibling nodes of v, including v.
6: Retrieve schedule lists stored in sibling nodes, or formally $L \leftarrow \{\delta(v) | \forall v \in S\}$; initialize a schedule list $\theta_{v_p} \leftarrow [\,]$.
7: **while** $L \neq \emptyset$ **do**
8: Peek the first element of every list in L, or formally $F \leftarrow \{\eta_i | \forall \sigma_i \in L\}$, where $\eta_i = \sigma_i.\text{peek\_first}()$.
9: Sort F using Johnson's rule for comparison; $\eta_r \leftarrow$ first element in F after sorting
10: Append η_r to θ_{v_p}; remove η_r from σ_r; remove σ_r from L if σ_r becomes an empty list.
11: Get the first element η_θ of θ_{v_p}.
12: Update the first element $\eta_\theta \leftarrow \eta_\theta + (f(v_p), g(v_p))$ in θ_{v_p}
13: Add the key-value pair $\langle v_p, \theta_{v_p}\rangle$ to dictionary δ
14: Remove nodes in S from G'_J and update $V' \leftarrow V' \setminus S$
15: **return** $\delta(v_r)$ as the layer-level processing schedule for G'_J, where v_r is the root of the tree.

the parent node v_p becomes leaf nodes and it represents the processing schedule of the sub-tree rooted at v_p. We can obtain the final schedule for all nodes in the graph by repeatedly executing this procedure on siblings in leaf nodes. However, sibling nodes may contain processing schedules of their children nodes during the repeated execution. We need to determine how to merge the processing schedules recorded in sibling nodes.

We follow the idea of merge sort to merge processing schedules. Specifically, a schedule list σ_{v_p} is an ordered list. Each element in the list is a tuple $(f(v), g(v))$ representing the computation and communication time of node v, where node v is in the sub-tree rooted at v_p. We use the tuple in the list since we plan to apply Johnson's rule to sort the list. When merging multiple schedules, we build the merged list iteratively. In each iteration, we peek at the first element of every schedule list and pick the best one to append to the result list. We use Johnson's rule to determine element ranks when choosing the best one. When comparing two tuples $(f(v_a), g(v_a))$ and $(f(v_b), g(v_b))$, we first determined whether they are computation-heavy or communication-heavy. If they are both computation-heavy (i.e., $f(v_a) \geq g(v_a)$ and $f(v_b) \geq g(v_b)$), then the tuple with a larger communication cost has a higher priority. If they are both communication-heavy (i.e., $f(v_a) < g(v_a)$ and $f(v_b) < g(v_b)$), then the element with a smaller computation cost has a higher priority. If one tuple is computation-heavy and the other one is communication-heavy, then the communication-heavy element has a higher priority.

The complete steps of our layer-level scheduling method for tree-structure computation graphs are shown in Alg. 5. Specifically, we use a dictionary to maintain the schedule list stored in each node of the computation graph $G'_J = (V', E')$. Let δ denote the dictionary and it is initialized following steps in lines 1-3. Starting from an empty dictionary, the loop of lines 2-3 inserts the key-value pair $\langle v, [(f(v), g(v))]\rangle$ for every node $v \in V'$. Notably, the value $[(f(v), g(v))]$ is a list of tuples. Each tuple $(f(v), g(v))$ contains the computation and communication time of the corresponding node v. We iteratively merge and extend those schedule lists in the loop of lines 4-14. In particular, line 5 finds the deepest leaf node v, its parent node v_p, and a set of its sibling nodes S in the current graph G'_J. Each sibling node in S should contain a schedule list which is stored in the dictionary δ. Line 6 retrieves the set of schedule lists of all sibling nodes in S and denotes the set L. Formally, $L = \{\delta(v) | \forall v \in S\}$. Also, a schedule list θ_{v_p} is initialized in line 6. The inner loop in lines 7-10 would merge schedule lists in L into a single list θ_{v_p}. Inspired by the merge sort algorithm, we iteratively find the best tuple from lists in L and insert it to θ_{v_p}. Line 8 gets candidate tuples from schedule lists in L. In line 8, we peek at the first element of each list in L and store the element in set F. Line 9 sorts the elements in F using Johnson's rule. Recall that each element in F is a tuple that consists of computation and communication time. Johnson's rule can be applied to sort those tuples. The best element (or the first element in the sorted F) is denoted as η_r. Line 10 appends the selected η_r to the end of θ_{v_p} and removed η_r from the schedule list σ_r, where σ_r represents the list that contains η_r. If σ_r becomes empty after removing η_r, then the list σ_r is removed from L. The inner loop continues until L becomes empty. After the inner loop, schedule lists in L is merged into θ_{v_p}. Then, we associate list θ_{v_p} with the node v_p. Line 11 retrieves the first element of θ_{v_p} and denotes it as η_θ. Note that η_θ is also a tuple of computation and communication time. We use $(f_{\eta_\theta}, g_{\eta_\theta})$ to denote the tuple. Line 12 updates η_θ by adding it with the computation and communication time of v_p. Formally, $f_{\eta_\theta} = f_{\eta_\theta} + f(v_p)$ and $g_{\eta_\theta} = g_{\eta_\theta} + g(v_p)$. After updating, line 13 adds the key-value pair $\langle v_p, \theta_{v_p}\rangle$ to the dictionary δ. Line 14 updates the graph G'_J by removing nodes in S and the associated edges. The outer loop terminates when the graph only contains the root node v_r. The schedule list $\delta(v_r)$ is returned in line 15 as the final layer-level processing schedule.

Fig. 14 shows an example of Alg. 5. There are five vertices in the example. The tuple $(f(v_i), g(v_i))$ next to each vertex v_i shows its computation and communication time. Following steps of Alg. 5, we first sort v_2 and v_4. Based on Johnson's rule, their schedule is $[v_2, v_4]$. Then, the first node in the list, i.e., v_2, is grouped with its predecessor v_1 to keep the precedence constraints. Also, nodes in the schedule list $[v_2, v_4]$ is removed from the graph. After this iteration, the updated graph is shown Fig. 14. The vertex v_1 becomes v'_1, and v'_1 represents the schedule list for the sub-tree rooted at v_1. The schedule list $[v_1 \oplus v_2, v_4]$ next to v'_1 shows the processing sequence, where $v_1 \oplus v_2$ indicates the vertices v_1 and v_2 are grouped together during merging. The time tuples of elements in the schedule list are $[(2,3),(3,2)]$, where $(2,3)$ shows the time consumption of the group $v_1 \oplus v_2$ and $(3,2)$ shows the time consumption of v_4. In the next iteration, leaf nodes v'_1 and v_3 that have no outgoing edges are sorted, i.e., we need to merge two lists $[(2,3),(3,2)]$ and $[(2,1)]$. Similar to the procedures of merge sort, the first element $(2,3)$ in



(a) A tree-strcutre graph (b) After scheduling subtree rooted at v_1

Fig. 14. An example of scheduling for tree-structure graphs.

the list $[(2,3),(3,2)]$ is compared with $(2,1)$ and $(2,3)$ is selected to be inserted to the merged list based on Johnson's rule. Two lists become $[(3,2)]$ and $[(2,1)]$. The merged list is $[(2,3)]$. Then, $(3,2)$ is compared with $(2,1)$ and $(3,2)$ is selected. Two lists become [ ] (empty) and $[(2,1)]$. The merged list becomes $[(2,3),(3,2)]$. Finally, $(2,1)$ is selected. The merged list is $[(2,3),(3,2),(2,1)]$. After merging, the first element $(2,3)$ in the list is grouped with its predecessor $(1,0)$. Eventually, the schedule list is $[v_0 \oplus v_1 \oplus v_2, v_4, v_3]$ and the corresponding list of time tuples is $[(3,3),(3,2),(2,1)]$. Vertices in the group $v_0 \oplus v_1 \oplus v_2$ would be processed in sequence during execution. Therefore, the schedule list can be written as $[v_0, v_1, v_2, v_4, v_3]$.

Merging schedule lists recorded in sibling nodes would not break precedence constraints in the computation graph. In addition, applying Johnson's rule guarantee the generated schedule lists are optimal. Theorem 4 shows our layer-level scheduling for tree-structure DAGs is optimal.

Theorem 4. When the computation graph G'_J is a tree, the layer-level schedule list generated by Alg. 5 is optimal.

Proof: This theorem can be proved using mathematical induction. For the base case, where the tree-structure graph G'_J contains one node, Alg. 5 obviously generates the optimal schedule list. For other cases, let v_p denote the root node and S denote the set of children of v_p. Assume the subtree rooted at v_s is optimally scheduled for every $v_s \in S$, then we will show that Alg. 5 optimally merges schedule lists of those subtrees and generates the optimal schedule for the tree rooted at v_p. When merging schedule lists, Alg. 5 iteratively compares the head element of every schedule list for all $v_s \in S$. Therefore, the relative order of elements in every schedule list is not changed after merging. In addition, Alg. 5 compares and selects list elements in the order that follows Johnson's rule. Because Johnson's rule is optimal [32], the schedule list generated by Alg. 5 is optimal. Moreover, line 12 of Alg. 5 groups v_p with the first element in the merged schedule list to reduce the time complexity of the algorithm. This is because v_p has no communication time cost, or formally $g(v_p) = 0$. Recall that only leaf nodes in G'_J need to offload their output to cloud servers and v_p is not a leaf node. Considering $g(v_p) = 0$, inserting a node after v_p in the schedule list has no benefit since it cannot further reduce the overall communication cost but may increase the overall computation time. Therefore, the grouping operation will not lose the optimal schedule. We may concluded that the layer-level schedule list generated by Alg. 5 is optimal. ■

7 Experiment

7.1 System Setup

Our offloading system testbed consists of a mobile device and a cloud server. We use a Raspberry Pi model 4B as the mobile device and a PC in our lab as the cloud server. Specifically, the Raspberry Pi model 4B uses a quad-core Cortex-A72 (ARM v8) SoC as its CPU, and it has 4GB RAM. Our PC has a six-core Intel i7-8700 CPU with 32GB RAM and a GTX1080 GPU. The operating system installed on the PC is Ubuntu 20.04. The communication channel between the mobile and cloud devices is over a wireless network. The Wi-Fi is set up on a NETGEAR Model R6230 router. Our lab PC equips a USB Wi-Fi adapter to access the wireless network. To simulate the communication delay at different bandwidths, we use the `wondershaper` package to limit the upload and download bandwidth of the Raspberry Pi. Notably, the roundtrip delay in the Wi-Fi network we used in our experiment is smaller than the delay in the real-world cellular network. However, the communication latency is still the bottleneck of cooperative inference in our experiment environment, and our proof-of-concept experiment can fairly evaluate the performance of our proposed method. Our experiment results also verify that our proposed method can adaptively adjust the partitioning and scheduling strategies and consistently outperforms comparison methods in various simulated network environments.

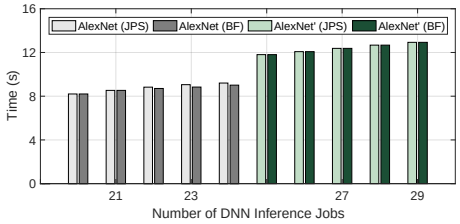
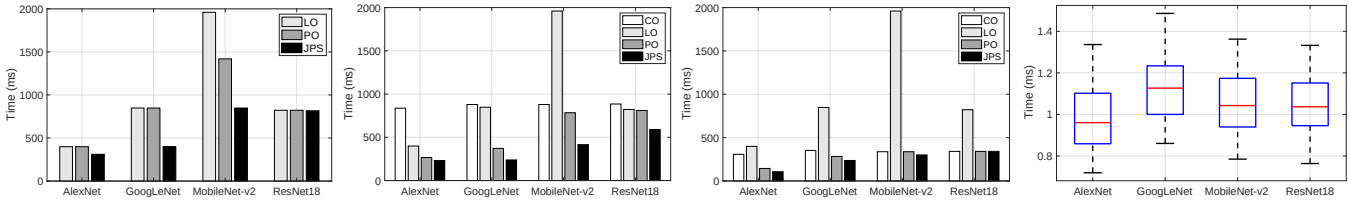


Fig. 15. Compare with brute force search.

The prototype of our joint optimization system is implemented with Python. The client-side is running on the Raspberry Pi and the server-side is running on the PC. Both client and server use `PyTorch` to perform DNN inferences. The server runs all inference tasks on its GPU with `CUDA`. The network communication between the client and server is established with `gRPC`. In a round of a DNN inference task, the client first loads the input image, transforms it to a tensor, and performs the forward propagation on the partitioned DNN. Then, the client collects the output tensor and encodes it as serialization for network transmission. A `gRPC` client is called to collect bytes array through the `BytesIO` and sent it to server as a `gRPC` request message. The server sends the classification result back to the client via a `gRPC` reply message.

Our scheduler is implemented on mobile devices. Before partitioning and scheduling, the scheduler needs to estimate the computation time of local DNN inference and the communication time of offloading. To reduce the estimation overhead, we build a lookup table for computation time considering the local computation time stable. The communication time changes with network bandwidth. Therefore, our scheduling algorithm uses a simple linear model to estimate the communication delay. Specifically, the communication time $t_n = w_0 + s/b$, where w_0 is the round-trip latency and s/b is the ratio of the message size s to the bandwidth b. The value of the model parameter w_0 is estimated based on latency data samples collected in the Raspberry Pi. Although the generated schedules are based on the estimated time cost, we executed the generated



(a) 3G (1.1 Mbps) (b) 4G (5.85 Mbps) (c) Wi-Fi (18.88 Mbps) (d) JPS overhead

Fig. 16. Comparison of total inference latency.

schedule in our testbed to record the real-world latency for evaluation.

We use `PyTorch Profiler` to measure the performance of DNN inference on the client and server when building the lookup table. To measure the communication delay, the `gRPC` reply message contains a field to record the total computation time t_c of the cloud server. The client will start a timer when it sends the `gRPC` request message, and stop the timer when it received the reply message. The duration of timer t_d includes the communication delay and the cloud computation delay. The difference $t_d - t_c$ is the communication delay. Our preliminary experiment result shows that the cloud computation delay is usually much smaller than the communication delay. Therefore, our scheduler only considers a two-stage scheduling problem and uses the time duration of t_d to train the regression model for communication delay.

In the experiments, we validate the proposed algorithms on different types of DNNs which are widely used in CV applications. For the chain architecture, we use AlexNet [34] and MobileNet-v2 [35]. It is important to mentioned that MobileNet contains multiple *bottleneck residual modules* as shown in Fig. 12. There is a bypass link in the module. Considering the bypass link, MobileNet does not have a chain-structure. However, from Fig. 12, we notice that the output sizes of layers within a bottleneck residual module are not decreasing. Partitioning at a layer within the module does not bring benefits for scheduling, and it should be clustered as a virtual block according to our analysis in section 3.2. After clustering and converting, we treat MobileNet as a chain-structure DAG. For the general architecture, we use GoogLeNet [36]. GoogLeNet contains several Inception modules illustrated in Fig. 3(a). The Inception module should not be clustered as a virtual block, because the output tensor sizes of its intermediate layers are smaller than input tensor sizes. We treat the GoogLeNet as a general-structure DAG.

7.2 Comparison Algorithms

We compare our scheme that jointly considers the partition and schedule (which is denoted as JPS), with *partition only* (PO), *cloud only* (CO), and *local only* (LO) schemes. For PO, we implement the state-of-the-art DNN partition algorithm [5], which generates homogeneous cut-points for all jobs. However, this scheme does not consider the collaboration between partitioning and scheduling. For CO, the entire inference workload is done on the cloud server. The local mobile device uploads all input tensors to the server. For LO, the inference jobs are processed solely on mobile devices without offloading. In addition, we implement the brute-force (BF) approach to find the optimal partition and schedule for small size inputs.

TABLE 1
Latency reduction ratio compared with LO (%)

Model	3G		4G		Wi-Fi	
	PO	JPS	PO	JPS	PO	JPS
AlexNet	0	**22.06**	33.33	**42.11**	63.91	**73.43**
MobileNet-v2	27.60	**56.73**	60.00	**78.83**	82.81	**84.69**
GoogLeNet	0	**52.83**	56.13	**71.93**	66.63	**72.17**
ResNet18	0	**0.73**	1.46	**28.22**	58.52	**58.52**

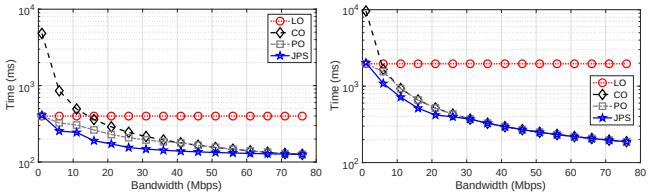
7.3 Experiment Results

We first compare our JPS with the BF approach to show the gap between our schedule with the optimal one. Fig. 15 shows the overall time consumption of multiple DNN inference jobs. In AlexNet, our scheme could generate optimal scheduling when the number of identical jobs is less than 23. On a synthetic DNN AlexNet′, whose communication time is sampled from the fitted curved, our scheme could find the optimal schedule. These experiment results verify that if the conditions stated in Theorem 2 hold, our scheme can find the optimal schedule for multiple identical DNNs.

We then evaluate our algorithms on chain-structure and general-structure DAGs. In this experiment, we generate 100 repeated jobs for each type of DNN, and we record the average completion time over different bandwidths. Specifically, we choose three typical bandwidths to simulate 3G, 4G, Wi-Fi network conditions. According to [5], the typical bandwidths of 3G, 4G, and Wi-Fi are 1.1Mbps, 5.85Mbps, and 18.88Mbps, respectively. The experiment results are shown in Fig. 16. In general, we can see that our joint optimization scheme JPS has the best performance for all types of DNNs in all network environments. Over each bandwidth configuration, our scheme outperforms the other comparison algorithms. The PO scheme ignores the collaboration among multiple jobs in scheduling, while the LO scheme does not make use of the powerful cloud.

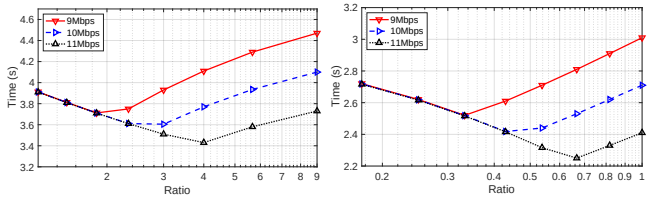
Fig. 16(a) illustrates the performance comparison on the 3G network. The CO time is not shown in the figure because it costs more than 4,000ms to upload the input tensor into the cloud server for all DNNs. It is much larger than other offloading schemes. From the figure, we notice that the JPS would significantly reduce the inference time for AlexNet, GoogLeNet, and MobileNet. Especially for GoogLeNet, the JPS reduces the inference time by 52.5% compared with LO and PO. The improvement of JPS for ResNet is not obvious. This is because the network speed is too slow and offloading the intermediate result of any layer of ResNet would cost more time than computing the model locally.

When the network bandwidth increases to 5.85 Mbps, our JPS scheme achieves significant improvement for all DNNs used in our experiment. Compare Fig. 16(a) and Fig. 16(b), we can notice that the state-of-the-art PO algorithm



(a) AlexNet (b) MobileNet-v2

Fig. 17. Inference latency under different bandwidths.



(a) ResNet (b) GoogLeNet

Fig. 18. The impact of the ratio between two types of jobs.

can barely reduce the total inference time for ResNet, even when the network condition is improved from 3G to 4G. Without scheduling, the bandwidth improvement is wasted. In contrast, our JPS would make full use of the bandwidth increase and reduce the overall inference time by 27.2% compared to PO. The reduction ratio of the inference time compared with LO is summarized in Table 1. Fig. 16(c) shows the performance comparison on the Wi-Fi network. The bandwidth of Wi-Fi is large, and simply offloading all computation workload to the cloud server is a good strategy. In this situation, our JPS still could reduce the inference time for AlexNet, GoogLeNet, and MobileNet. Fig. 16(d) shows the overhead of our JPS scheme. From the figure, we notice that the overhead is negligible compared with the inference time. It is because both binary search and scheduling algorithms are fast. More importantly, we use a lookup table to store the local inference time, which saves the time cost of profile estimation. In addition, the communication time is estimated using a simple linear regression model which is also time-efficient.

Fig. 16 shows that there is a range in which our JPS scheme can reduce the overall inference time. When the network conditions are too poor, offloading brings no benefits. Similarly, when the bandwidth is large enough, the mobile device should simply upload all computation workload to the cloud server, which is much faster. Fig. 17 shows the DNN inference time under different bandwidths. From the figure, we find that our JPS scheme can speed up both AlexNet and MobileNet in bandwidth range of $[1, 20]$ Mbps, which covers the bandwidth from 3G network to Wi-Fi network. Comparing Fig. 17(a) and Fig. 17(b), we notice that AlexNet has a wider benefit range in which JPS can reduce the inference time. This shows that even when wireless upload bandwidth exceeds 50Mbps, our JPS scheme is useful.

We also investigate the impact of the ratio between computation- and communication-heavy jobs. The results are shown in Fig. 18. Fig. 18(a) shows that the optimal ratio between two types of jobs is not 1, and it varies with the bandwidth configurations. Comparing Fig. 18(a) and Fig. 18(b), we notice that if the communication-heavy jobs have larger differences between computation and communication stages, then the optimal ratio between computation-heavy and communication-heavy jobs is low. Otherwise, there should be more communication-heavy jobs. The optimal ratio shifts with bandwidth changes.

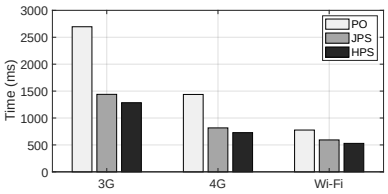


Fig. 19. Inference latency of processing heterogeneous jobs.

Fig. 19 shows the evaluation results for scheduling heterogeneous DNNs. In this experiment, we equally mix inference jobs of AlexNet, MobileNet-v2, and GoogLeNet. We directly apply PO and JPS on each individual job and record the overall inference latency. HPS refers to our proposed heterogeneous partitioning and scheduling method. From the experiment result, we can find that our proposed HPS method can significantly reduce the inference latency for heterogeneous jobs.

8 Conclusion

In this paper, we investigate the model partitioning and offloading pipeline scheduling problems for collaborative DNN inference. Our objective is to minimize the overall inference makespan of multiple DNN inference jobs. We explore the homogeneous case where DNN models have the same structure. Particularly for homogeneous chain-structure DNNs, if we relax the problem to the continuous domain, then partitioning all DNNs at the same point is sufficient for makespan optimization. On the discretized domain, two types of partitions are sufficient when the time difference between the two adjacent partition points is not drastic. A binary-search-based partitioning and scheduling algorithm is proposed. Moreover, we extend the problem scenario to process heterogeneous DNNs. A layer-level scheduling algorithm that integrates Johnson's rule and ideas of topological sort is proposed. The proposed method is optimal when the computation graph is tree-structure. In a tree-structure computation graph, there are only fork nodes. The prototype of our joint optimization methods is implemented and tested in a real-world testbed. Evaluation results on AlexNet, MobileNet, ResNet, and GoogLeNet reveal that our methods outperform partition-only and schedule-only schemes in all network circumstances.

In future work, we plan to investigate the collaborative inference framework that covers mobile devices, edge servers, and cloud servers. Adding edge servers to the framework can further reduce the inference latency since edge servers provide much lower communication latencies. Moreover, we plan to investigate the scheduling problem for online arrival jobs, which is more practical in real-world applications.

REFERENCES

[1] Y. Duan and J. Wu, "Joint optimization of dnn partition and scheduling for mobile cloud computing," in *50th International Conference on Parallel Processing*, 2021, pp. 1–10.

[2] W. Xu, Y. Zhang, and X. Tang, "Parallelizing dnn training on gpus: Challenges and opportunities," in *Companion Proceedings of the Web Conference 2021*, 2021, pp. 174–178.

[3] H. Wang, Z. Qu, Q. Zhou, H. Zhang, B. Luo, W. Xu, S. Guo, and R. Li, "A comprehensive survey on training acceleration for large machine learning models in iots," *IEEE Internet of Things Journal*, 2021.

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

[7] C. Liu, K. Kim, J. Gu, Y. Furukawa, and J. Kautz, "Planercnn: 3d plane detection and reconstruction from a single image," in *Proceedings of the IEEE CVPR*, June 2019.

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

[9] T. Mohammed, C. Joe-Wong, R. Babbar, and M. Di Francesco, "Distributed inference acceleration with adaptive dnn partitioning and offloading," in *Proceedings of the IEEE INFOCOM*, 2020, pp. 854–863.

[10] C. Kai, H. Zhou, Y. Yi, and W. Huang, "Collaborative cloud-edge-end task offloading in mobile-edge computing networks with limited communication capability," *IEEE Transactions on Cognitive Communications and Networking*, vol. 7, no. 2, pp. 624–634, 2020.

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

[12] Y. Duan and J. Wu, "Computation offloading scheduling for deep neural network inference in mobile computing," in *IEEE/ACM 29th International Symposium on Quality of Service (IWQoS'21)*, Virtual Conference, Jun. 2021.

[13] E. Variani, X. Lei, E. McDermott, I. L. Moreno, and J. Gonzalez-Dominguez, "Deep neural networks for small footprint text-dependent speaker verification," in *Proceedings of the 45th IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2014, pp. 4052–4056.

[14] 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 *Proceedings of the ACM MobiSys*, 2016, pp. 123–136.

[15] 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.

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

[17] B. Taylor, V. S. Marco, W. Wolff, Y. Elkhatib, and Z. Wang, "Adaptive selection of deep learning models on embedded systems," *arXiv preprint arXiv:1805.04252*, 2018.

[18] C. Wang, S. Zhang, Y. Chen, Z. Qian, J. Wu, and M. Xiao, "Joint configuration adaptation and bandwidth allocation for edge-based real-time video analytics," in *Proceedings of the IEEE INFOCOM*, 2020, pp. 1–10.

[19] 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 *Proceedings of the ACM/EDAC/IEEE DAC*, 2016, pp. 1–6.

[20] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," 2011.

[21] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *Proceedings of ACM/IEEE IPSN*, 2016, pp. 1–12.

[22] 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 *Proceedings of ACM/SIGDA FPGA*, 2020, pp. 40–50.

[23] P. Brucker and P. Brucker, *Scheduling algorithms*. Springer, 2007, vol. 3.

[24] K. Agrawal, J. Li, K. Lu, and B. Moseley, "Scheduling parallel dag jobs online to minimize average flow time," in *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2016, pp. 176–189.

[25] C. Chekuri, A. Goel, S. Khanna, and A. Kumar, "Multi-processor scheduling to minimize flow time with ε resource augmentation," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. ACM, 2004, pp. 363–372.

[26] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[27] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proceedings of AAAI*, 2017.

[28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[29] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," *arXiv preprint arXiv:1612.08242*, 2016.

[30] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[31] W. Shao, F. Xu, L. Chen, H. Zheng, and F. Liu, "Stage delay scheduling: Speeding up dag-style data analytics jobs with resource interleaving," in *Proceedings of ICPP*, 2019, pp. 1–11.

[32] 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.

[33] R. Arandjelovic and A. Zisserman, "Objects that sound," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 435–451.

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

[35] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE CVPR*, 2018, pp. 4510–4520.

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

Yubin Duan received his B.S. degree in Mathematics and Physics from University of Electronic Science and Technology of China, Chengdu, China, in 2017. He is currently a Ph.D. student in the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania, USA. His current research focuses on scheduling algorithms for distributed systems and parallel computing.

Jie Wu is the Director of the Center for Networked Computing and Laura H. Carnell professor at Temple University. His current research interests include mobile computing and wireless networks, routing protocols, network trust and security, distributed algorithms, applied machine learning, and cloud computing. Dr. Wu regularly published in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including IEEE Transactions on Mobile Computing and IEEE Transactions on Service Computing. Dr. Wu is a Fellow of the AAAS and a Fellow of the IEEE.