Accelerate Cooperative Deep Inference via Layer-wise Processing Schedule Optimization

Ning Wang*, Yubin Duan†, and Jie Wu†
*Department of Computer Science, Rowan University, Glassboro, USA
†Center for Networked Computing, Temple University, Philadelphia, USA
Email: wangn@rowan.edu and {yubin.duan, jiewu}@temple.edu

***Abstract*—Computation offloading is proposed to solve one obstacle of enabling high-accurate and real-time deep inference in resource-constrained Internet of Things (IoT) devices. Cooperative deep inference is proposed recently to further trade-off the introduced communication latency in computation offloading, which partitions a Deep Neural Network (DNN) model into two parts and utilizes the IoT end device and the server to process the DNN model cooperatively. We observe one important but ignored fact in all previous works: *DNN computation and communication processing can be conducted simultaneously in cooperative deep inference.* As a result, the DNN layer-wise processing schedule has an impact on inference latency and it is non-trivial to find the optimal schedule in State-Of-The-Art (SOTA) DNNs with Directed Acyclic Graph (DAG) computational architectures. The contributions of this paper are as follows. (1) The proposed Deep Inference Optimization with Layer-wise Schedule, DeepInference-L, is a unique pipeline-based DAG schedule problem, which turns out to be NP-hard. (2) We categorize SOTA DNNs into three different categories and discuss the corresponding optimal processing schedule in special cases and efficient heuristic schedules in the general case. (3) The proposed solutions are extensively tested via a proof-of-concept prototype. (4) Results indicate that our algorithms can achieve an 8x speedup compared with local inference in the best case.**

***Index Terms*—computation offloading, mobile computing, edge computing, deep inference, and scheduling.**

I. Introduction

Deep learning has shown success in complex tasks, including computer vision [1], Natural Language Processing (NLP) [2], machine translation [3], and many others. Nowadays, SOTA DNN models, e.g., ResNet-101, VGG-16 [4, 5], can achieve more than 95% top-5 accuracy on ImageNet [1]. The most advanced models, such as FixEfficientNet-L2 [6], can achieve 88.5% top-1 accuracy by using $480M$ parameters. GPT-3, a natural-language deep learning model with $175B$ parameters [7] was released recently. GPT-3 achieves SOTA performance on several NLP benchmarks without fine-tuning.

One obstacle of utilizing deep learning in IoT systems for inference tasks is that IoT devices cannot provide real-time and high-accurate result at the same time due to the limited computation capacity. However, many IoT systems, such as traffic monitoring, require not only high processing speed, but high accuracy as well. To deal with this obstacle, computation offloading is proposed, where an edge/cloud server can assist IoT end devices for deep inference acceleration. However, computation offloading introduces extra communication latency, which may not be negligible due to the high-resolution sensory data and the slow transmission speed.

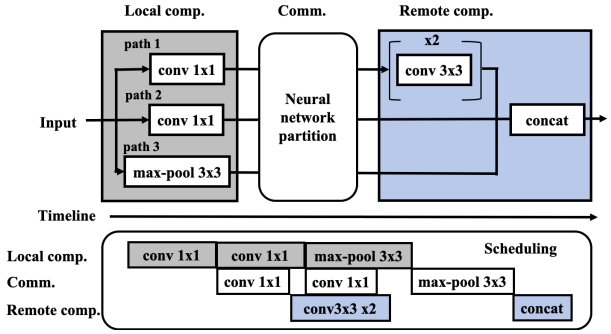


Fig. 1. A motivation example in Inception-v3 module.

To reduce the introduced communication latency in computation offloading, recent research has explored a cooperative deep inference technique [8, 9]. A DNN model is decomposed into a set of small tasks, i.e., *layers*, and process layers in the end device and server cooperatively by following their corresponding dependency relationship. Particularly, cooperative deep inference has three stages. A DNN model is partitioned into two parts for local and remote processing. In stage 1, the local device partially processes the first part of the DNN. In stage 2, the intermediate DNN layer's output is transmitted to the remote server. In stage 3, the remote server continues the DNN processing and get the final inference result. Fig. 1 illustrates these three stages in an Inception-v3 module. The rationale of offloading intermediate data rather than raw data is that the data size of some intermediate DNN layers is significantly smaller than that of raw input data. Therefore, cooperative deep inference can greatly reduce communication latency at the cost of small local processing latency.

In this paper, we focus on the Deep Inference Latency Minimization with Layer-wise schedule, called DeepInference-L, whose goal is to minimize inference completion time in a networked system with a given DNN model partition. We observe that layers in different stages can be processed *in parallel* to further reduce the DNN inference latency. For instance, in Fig. 1, the local processing of the second convolution layer and the communication of the first convolution layer can be conducted simultaneously. Considering the DAG processing constraint and the mismatched computation and communication speeds in three stages, it is challenging to determine the optimal layer-wise processing schedule. The

complex DNN architecture, abstracted as a DAG [8, 10, 11], further complicates the problem.

In this paper, we prove that DeepInference-L is NP-hard in general. Then, we categorize SOTA DNNs into three different network architectures, i.e., line, multi-path, and general DAG. We find the optimal data offloading schedule in line architecture. In multi-path DNN architecture, we adopt Johnson's rule to derive the optimal solution in a special network environment and a performance bounded result in the general case. For general DAG DNN, we propose a graph conversion strategy so that the proposed solutions for multi-path DNN can be adopted. The proposed solutions are tested via a proof-of-concept implementation.

The contributions of this paper are summarized as follows.

- We are the first to consider layer-wise processing schedule optimization in cooperative DNN deep inference.
- We propose an optimal data offloading strategy for line DNNs and prove that DeepInference-L Problem is NP-hard for multi-path and general DAG DNNs.
- For multi-path DNNs, we identify an extended Johnson's algorithm which solves DeepInference-L problem optimally in a special case and has guaranteed performance in the general case. In addition, we present a heuristic schedule algorithm for general DAG DNNs.
- We implement a proof-of-concept prototype and conduct comprehensive testing on a wide range of DNNs, including classification, tracking, scene understanding, to verify the effectiveness of proposed approaches.

II. Related Works

In this section, we briefly summarize research efforts in providing fast and accurate DNN inference in IoT devices via on-device, server-only, and cooperative computation.

On-device Model Optimization: In order to realize inference acceleration, works in this category investigated how to optimize DNN models for IoT devices. For example, Microsoft and Google developed small-scale DNNs for speech recognition on mobile platforms by sacrificing the high prediction [12]. Tensorflow Lite [13] takes existing Tensorflow models and converts them into an optimized and efficient version so that the streamlined model is small enough to be stored on devices and sufficiently accurate to conduct suitable inference. Several popular deep learning models for resource-constrained devices are drawn from Computer Vision. These models include MobileNets [14], Single Shot MultiBox Detector (SSD) [15], YoLo [16], etc. However, with a reduced number of parameters, the inference accuracy decreases as well.

Cloud/Edge-only Offloading: Raw data is offloaded to the remote server in this category. Han et al. proposed generating alternative DNN models to trade off accuracy and performance/energy [17]. Zhou et al. considered a model partition and parallelization so that multiple cloud works could work together to speed up the processing [18]. Canel et al. proposed to use small filters at the edge device to filter out uninterested data and thus reduce the communication cost [19]. In [20, 21], the authors proposed adaptive methods which can dynamically adjust the accuracy requirement based on the wireless link condition. However, they focus more on traffic minimization rather than latency minimization [19, 20]. Approaches in this category are very sensitive to network environments.

IoT-assist Offloading: Kang et al. first considered the large communication latency during the offloading and thus proposed to use IoT devices to conduct partial processing [9]. However, they only discussed the solution in line architecture. Hu et al. further considered the optimal partition by considering the fact the many SOTA DNNs have a DAG architecture and proposed a min-cut formulation to solve latency minimization using a max-flow approach [8]. Zhang et al. further reduce the solution space to increase the running speed to find the optimal cut [22]. Lin et al. considered a case where there are multiple servers and a more general 3-layer network, i.e., mobile, edge, and cloud [23]. A Particle Swarm Optimization (PSO) algorithm was proposed to solve the task allocation problem. This paper belongs to IoT-assist offloading and it further addresses the layer-wise processing schedule issues which does not be covered in [8, 22]. The detailed discussion about the difference between this paper and approaches in [8, 22] can be found in Section III-B.

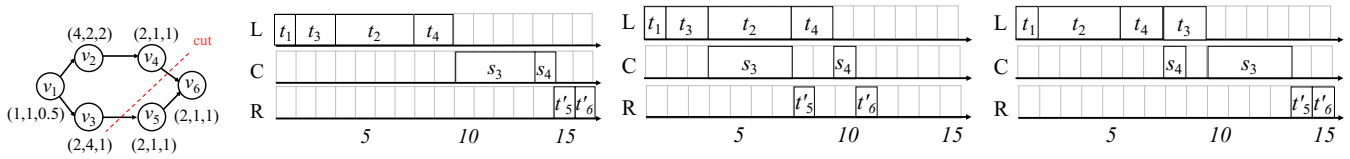
III. Problem Formulation

A. Network Model

We focus on a typical computation offloading environment, where an end device, e.g., a smartphone, and an edge/cloud server are connected via WiFi or cellular network. Without causing any confusion, the end device and server are called local and remote devices respectively in the remainder of this paper. The computational architecture of SOTA DNN models can be graphically abstracted as a DAG, denoted as $G(V, E)$ with n vertices [8, 11, 22, 24][1], where a vertex $v_i \in V$ represents *a computational layer rather a neuron* and a link $e_{ij} = (v_i, v_j) \in E$ represents the processing dependency relationship between two layers. In the remainder of this paper, vertex and layer are used interchangeably. In addition, we use $p(i)$ to denote a path where a set of vertices, $(v_b, v_{b+1}, \cdots, v_e)$, are processed under the partial processing order. The v_b and v_e are the beginning and end vertices, respectively.

We consider a cooperative deep inference mechanism, where the network is partitioned two parts. A network partition decision can be modeled as a cut of a set of edges in $G(V, E)$ and the layer processing assignment can be demoted as a binary vector $X \in \{0, 1\}^n$, where $x_i = 0$ if v_i is assigned to the local device, and $x_i = 1$ if it is assigned to the remote server. Cooperative deep inference has three stages. In stage 1, the local device will process locally assigned layers, and the intermediate results of stage 1 will be transmitted to the remote server, which is stage 2. In stage 3, the remote server will continue the processing and finish the DNN inference. The processing times for vertex v_i in local and remote servers

[1]Recurrent Neural Networks (RNN) models such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are out of the scope of this paper, which is same as references.



(a) a DNN (b) no layer-wise schedule (c) layer-wise schedule 1 (d) layer-wise schedule 2

Fig. 2. A motivation of layer-wise cooperative deep inference (L: local device, C: communication, and R: remote server).

are t_i and t'_i, respectively. Let s_i denote the output data size of v_i. In order to process vertex v_j remotely, vertex v_i's output, simply called vertex v_i, need to be transmitted to the remote server if e_{ij} belongs to the cut.

Fig. 2 illustrates the deep inference mechanism with a DAG DNN with 6 vertices. The red line represents the network partition decision, $X = \{0, 0, 0, 0, 1, 1\}$, where v_1 to v_4 are processed locally and the outputs of v_3 and v_4 are transmitted to the remote server and used as inputs of v_5 and v_6 for remote processing. In Fig. 2, a tuple near a vertex v_i represents the corresponding (t_i, s_i, t'_i).

B. Proposed Idea

The main idea of this paper is that *existing works consider there is no overlap between three stages [8, 22] but we argue that three stages can happen simultaneously.* Therefore, existing works that claimed to achieve optimal inference latency can be further reduced. Fig. 2(b) shows the strategy in [8], which takes 16 time units in this example and there is no overlap between three stages. We argue that v_3 should be transmitted once the local device finishes it, as shown in Fig. 2(c). Similarly, the remote server should process v_5 once it receives v_3. The final completion time is reduced to 11. It is worth noting that even for the same network partition decision, the vertex processing schedule has an influence on the completion time. In Fig. 2(d), the local device processes v_2 first, followed by v_4 and v_3. This processing schedule leads to a completion time of 15.

C. Completion Time Calculation

In Subsection III-B, we discuss that the completion time of a vertex is not only determined by the network partition but also depends on the corresponding schedule strategy, σ, which is the processing order of $(v_1, v_2, \ldots, v_n)$. Specifically, the completion time of vertex v_j, $C_j(X, \sigma)$, can be calculated recursively and formally written as follows:

$$\begin{cases} \max_{e_{ij} \in E} C_i(X, \sigma) + t_{ij}(\sigma) + t'_j, & \{x_i, x_j\} = \{0, 1\}, \\ \max_{e_{ij} \in E} C_i(X, \sigma) + t_j, & \{x_i, x_j\} = \{0, 0\}, \\ \max_{e_{ij} \in E} C_i(X, \sigma) + t'_j, & \{x_i, x_j\} = \{1, 1\}, \end{cases} \quad (1)$$

where $t_{ij}(\sigma)$ is the offloading latency of s_i under the schedule σ. Note that we only consider computation offloading from the end device to the remote server and the assumption is that the remote server is more powerful. In Fig. 2(c), $C_6(X, \sigma) = \max\{C_5(X, \sigma) + t'_6, C_4(X, \sigma) + t_{46}(\sigma) + t'_6\} = \max\{8+1, 9+1+1\} = 11$. It is worth noting that $t_{ij}(\sigma) \geq s_i$, because s_i may be queued during the data transmission stage. For example, if s_3 in Fig. 2(c) becomes 7, $t_{46}(\sigma)$ will change to 2 due to waiting for the s_3's transmission and $C_6(X, \sigma)$ becomes 12.

D. Deep Inference Optimization with Layer-wise Schedule

In this paper, we would like to investigate the impact of layer-wise processing pipeline schedule in cooperative deep inference, called DeepInference-L. The objective of DeepInference-L is to find the best offloading schedule σ to minimize the DNN completion time when the DNN DAG, network environment, local and remote environment, and the deep inference strategy, X, are known. That is,

$$\begin{aligned} P1: \quad & \arg\min_{\sigma} C_n(X, \sigma) \\ \text{s.t.} \quad & C_i(X, \sigma) \leq C_j(X, \sigma), \ \forall e_{ij} \in E \\ & \text{Eq. (1)}, \ \forall i, j \end{aligned} \quad (2)$$

where the objective is equivalent to minimize the last vertex's completion time. The first constraint is processing dependency. A vertex can only be processed if all of its predecessors have been completed. The second constraint is the completion time calculation. Without causing confusion, we use $C_i(\sigma)$ to denote the completion time of a vertex v_i under a given network partition X in the remainder of this paper. We plan to solve the optimal network partition X in the future.

E. Problem Hardness

Theorem 1. *The proposed DeepInference-L is NP-hard.*

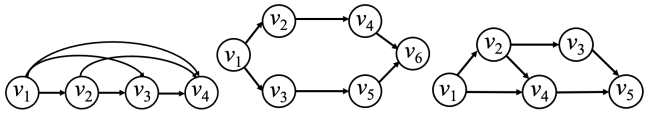
The detailed proof of Theorem 1 can be found on Appendix. The insight is that the DeepInference-L can be reduced to a 3-machine flow shop problem.

IV. Layer-wise Schedule Optimization

In this section, we would like to discuss the optimal schedule algorithms for different network architectures, respectively. We categorize DNNs into three architectures, shown in Fig. 3, and discuss the corresponding schedule solution. The first architecture, shown in Fig. 3(a), is the line architecture. Applications include YoLo, ResNet, DenseNet [25, 26]. The second architecture is the multi-path architecture shown in Fig. 3(b). Inception-v1 [27], ResNeXt [28], novel networks – such as Siamese Networks, Triplet Networks, and Multi-stream [29, 30], also belong to this type. The third architecture is the general DAG architecture shown in Fig. 3(c). Applications include Inception-ResNet [31], NASNet [32], RandWire [10].

A. Line Architecture

Line-architecture DNN has one and only one feasible partial order, i.e., only one vertex is ready for processing at any time. If only one output needs to be transmitted to the server, the schedule is trivial. However, "by-pass" link has recently been used in many models to resolve the vanishing gradient



(a) line (b) multi-path (c) DAG

Fig. 3. State-of-the-art CNN architectures summary.

problem, e.g., ResNet, DenseNet [26, 33]. As a result, multiple outputs may need to be transmitted to the server in the cooperative inference and thus there is an intermediate transmission schedule optimization problem for inference latency minimization. We prove that the optimal schedule can be obtained from Theorem 2.

Theorem 2. *If both $e_{ii'}$ and $e_{jj'}$ are in the cut and $v_{i'} \prec v_{j'}$, $i \neq j$, $i' \neq j'$, v_i's output should be transmitted first in the optimal schedule.*

The detailed proof of Theorem 2 can be found on Appendix. The insight of Theorem 2 is that an earlier vertex's input in the processing sequence should be satisfied first. Therefore, vertices transmission priorities can be determined by successor vertices' processing sequence at the remote server.

B. Multi-Path Architecture

In this subsection, we discuss the schedule strategy for multi-path DNNs. A multi-path DNN has a set of parallel paths which have no overlapping except for the first and the last vertices. Multi-path architecture is a recent technique to improve DNN accuracy with the intuition of multi-scale processing, verification, etc. [27, 29, 30, 34].

1) Path-based Schedule: We find that a cut of a multi-path DNN will separate every path into two parts (possibly including multiple vertices). Based on this observation, we propose a path-based schedule method. Particularly, each path has three stages and each stage of a path can be considered as an entity and always be processed (i.e., computation or communication) consecutively until that stage terminates, i.e., non-preemptive schedule. Without loss of generality, we use p_1, p_2, and p_3 to denote the three stages, i.e., local processing, transmission, and remote processing, of a specific path. By a slight abuse of notation, we also use them to denote the corresponding times of stages.

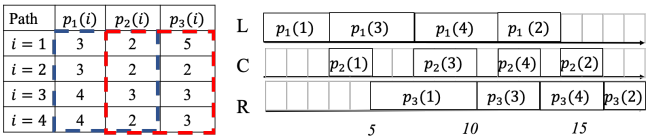
$$p_1 = \sum_{l=b}^{c} t_l, \;\; p_2 = s_c, \;\; p_3 = \sum_{l=c+1}^{e} t'_l, \tag{3}$$

where v_b, v_c, and v_e are the beginning vertex, the last vertex processed locally, and the last vertex of this path, respectively. For a specific path i, we use $p_1(i), p_2(i)$, and $p_3(i)$ to demote the processing time in three stages. It is worth noting that we don't consider *preemptive* schedules, e.g., processing a path partially and resume it later at a stage. The rationale of the proposed path-based schedule, a non-preemptive schedule, can be proved by Theorem 3.

Theorem 3. *In multi-path DNNs, the optimal schedule can be achieved via non-preemptive path-based schedule.*

The detailed proof of Theorem 3 can be found on Appendix. The insight behind Theorem 3 is that a partially processed path in a stage does not reduce its own completion time but may increase the completion time of other paths. As a result, the path-based schedule will not lose schedule optimality.

Path	$p_1(i)$	$p_2(i)$	$p_3(i)$
$i=1$	3	2	5
$i=2$	3	2	2
$i=3$	4	3	3
$i=4$	4	2	3



(a) EJ Algorithm

Path set	Partial order and completion time
$(p(1),p(3))$	$(\boldsymbol{p(1),p(3)})$**: 13**, $(p(3),p(1))$: 15
$(p(1),p(3)) \cup p(4)$	$(p(4),p(1),p(3))$: 17, $(p(1),p(4),p(3))$: 17, $(\boldsymbol{p(1),p(3),p(4)})$**: 16**
$(p(1),p(3),p(4)) \cup p(2)$	$(p(2),p(1),p(3),p(4))$: 19, $(p(1),p(2),p(3),p(4))$: 19 $(p(1),p(3),p(2),p(4))$: 19, $(\boldsymbol{p(1),p(3),p(4),p(2)})$**: 18**

(b) NEH Algorithm

Fig. 4. An illustration of the proposed two algorithms.

2) Problem Formulation: Based on Theorem 3, Problem $P1$ can be re-formulated as a Mixed Integer Program (MIP), whose goal is to determine an optimal path sequence $P = (p(1), p(2), \ldots, p(m))$ to minimize the completion time. We can use a decision variable y_{jk} to denote if $p(j)$ is the kth path in the schedule sequence ($y_{jk} = 1$) or if not, $y_{jk} = 0$. The auxiliary variable I_{ik} denotes the idle time on stage i between the processing of the path in the kth position and $(k+1)$th position, and the auxiliary variable W_{ik} denotes the waiting time of the path in the kth position between stages i and $i+1$. The idle time on the remote server is

$$\sum_{i=1}^{2} p_i(1) + \sum_{j=1}^{m-1} I_{3j} = \sum_{i=1}^{2}\sum_{j=1}^{m} y_{j1} p_i(j) + \sum_{j=1}^{m-1} I_{3j}, \tag{4}$$

where the first part is the waiting time for the remote server to receive the result of the first vertex, and the second part is all the following idle time at the remote server.

Note that minimizing the DNN completion time under an offloading decision is equivalent to ensure that the remote server can finish as soon as possible. Therefore, DeepInference-L problem can be reformulated as follows:

$$\begin{aligned} P2\text{: min} \;\; & \Big(\sum_{i=1}^{2}\sum_{j=1}^{m} y_{j1} p_i(j) + \sum_{j=1}^{m-1} I_{3j}\Big), \\ \text{s.t.} \;\; & \sum_{j=1}^{m} y_{jk} = 1, \forall k, \quad \sum_{k=1}^{m} y_{jk} = 1, \forall j, \\ & I_{ik} + \sum_{j=1}^{m} y_{j(k+1)} p_i(j) + W_{i(k+1)} \\ & = W_{ik} + \sum_{j=1}^{m} y_{jk} p_{i+1}(j) + I_{(i+1)k} \forall i, k, \\ & W_{i1} = 0, \forall i, \quad I_{1k} = 0, \forall k, \end{aligned} \tag{5}$$

where the first constraint ensures that a path has to be assigned to the position k. The second constraint ensures that a path k has to be assigned to one and only one position. The third constraint is the physical constraint between waiting time and idle time. The last two constraints are the initial condition.

3) Optimal Solution in A Special Scenario: Problem $P2$ is NP-hard, as proved in Theorem 1. However, we can derive the optimal solution of $P2$ in a special network environment.

An explanation of the proposed Extended Johnson (EJ) algorithm is shown in Algorithm 1. To meet the $P2$'s objective, we group paths into a high-priority set, H, and a low-priority set, L, based on the processing time summation of the first

Algorithm 1 Extended Johnson Algorithm (EJA)

Input: $G(V, E)$, X, t_i and t'_i, $\forall v_i$
Output: The offloading schedule σ

1: $H \leftarrow L \leftarrow \emptyset$
2: **for** $i = 1$ to m **do**
3: **if** $p_1(i) + p_2(i) \leq p_2(i) + p_3(i)$ **then**
4: $H = H \cup p(i)$
5: **else**
6: $L = L \cup p(i)$
7: Sort H increasingly based on $p_1(i) + p_2(i)$
8: Sort L decreasingly based on $p_2(i) + p_3(i)$
9: Concatenate H and L to obtain σ

Algorithm 2 Nawaz Enscore Ham (NEH) Algorithm

Input: $G(V, E)$, X, t_i and t'_i, $\forall v_i$
Output: The offloading schedule σ

1: Create m paths based on Eq. 3.
2: Generate sequence $P = (p(1), \ldots, p(m))$ in non-increasing order of total processing time $p_1 + p_2 + p_3$, and a partial schedule $\sigma = (p(1))$
3: **for** index $i = 2$ to m in the sorted path set **do**
4: **for** possible insert position $k = 1$ to i **do**
5: Evaluate the new sequence $\sigma = \sigma \cup p(i)$
6: Update the partial schedule σ with $p(i)$ with position causing the minimum completion time

two stages and the processing time summation of the last two stages, i.e., $p_1(i) + p_2(i)$ and $p_2(i) + p_3(i)$ (Lines 1-3). Particularly, if $p_1(i)+p_2(i) \leq p_2(i)+p_3(i)$, the path $p(i)$ will be added into the set H. Otherwise, the path $p(i)$ will be added into the set L (Lines 4-6). To ensure that the remote server can start its processing as soon as possible, the optimal schedule will schedule paths at set H first based on the increasing order of $p_1(i) + p_2(i)$ (Line 7). Ties may be broken arbitrarily. The insight of this step is to ensure the remote server will start its processing as early as possible. To ensure the idle time of the remote is minimized, the optimal schedule will schedule paths at set L based on the decreasing order of $p_2(i) + p_3(i)$ (Line 8). Ties may be broken arbitrarily. The optimality of this algorithm is proved by [35].

Fig. 4(a) shows a running example of Algorithm 1, where there are 4 paths in total $(m = 4)$. The local path processing time, data transmission time, and remote path processing time of all paths are shown on the left table of Fig. 4(a). For instance, the local device needs 3 time units to process the first part of the path $p(1)$, the intermediate result transmission takes another 2 time units, and the remote server needs 5 time units to continue the processing of the path $p(1)$. It is worth noting that in this example, the minimum value of the processing time of four paths on stages 1 and 3 are 3 and 2, respectively, and the maximum communication time for any path in stage 2 is 3. As a result, it satisfies the optimal condition in Theorem 4, i.e., stage 1 dominates stage 2, so that the result generated from Algorithm 1 is optimal.

In Fig. 4, four paths are grouped into two sets, H and L. The processing time sum of the first two stages and the processing time sum of the last two stages are calculated as shown in Fig. 4(a). In this example, $p(1)$ will be inserted into set H and all other three paths will be inserted into set L. As a result, path $p(1)$ will be scheduled first, followed by paths $p(3)$, $p(4)$, and $p(2)$. The overall completion time is 18. This is the optimal solution, since the remote server starts at the earliest time, 5. In the meanwhile, there is no idle time at the remote server and thus it is impossible to find a better schedule.

Theorem 4. *If stage 2 is dominated by either stage 1 or stage 3, i.e.,* $\max\{\min p_1(i), \min p_3(i)\} \geq \max p_2(i), \forall i$, *$P2$ can be optimally solved via Algorithm 1.*

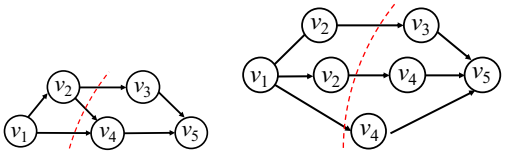
This theorem can be proved via 3-machine flow-shop conversion. Then, we can use the result from [35] to prove it. In [36], the authors further proved that this approach achieves an approximation ratio of $5/3$ in the general case.

4) Heuristic Solution in the General Case: In the general case, we propose to apply Nawaz Enscore Ham (NEH)'s heuristic algorithm, which has good performance in the general case and low time complexity [37]. The key idea of NEH is that it provides time-consuming paths with a high schedule priority. The longest path has the highest schedule priority and is scheduled first. Then, the NEH algorithm iteratively adds one new path into the partial schedule by trying all possible insert positions and select the best one. The detailed explanation for the NEH algorithm can be found in Algorithm 2. First, paths are sorted based on their corresponding processing time summation of three stages (Lines 1-2). Then, the remaining paths will be inserted into the best position in the current schedule one-by-one (Lines 3-7).

The NEH's running procedure for the same toy example is shown in Fig. 4(b). First, the path sorting result is $(p(1), p(3), p(4), p(2))$ since their corresponding processing times are $10, 10, 9$, and 7, respectively. Then, pick the longest path, which is $p(1)$. Ties may be broken arbitrarily. Then, we further add $p(3)$ into the current schedule and there are two possible schedules. For the schedule $(p(1), p(3))$, the overall processing time is 13 time units. However, for the schedule $(p(3), p(1))$, the overall processing is 15 time units. As a result, the best partial scheduling order so far is $(p(1), p(3))$. Then, NEH will further insert the remaining paths, i.e., $p(4)$, $p(2)$, in the best possible positions. Three possible schedules will be generated after the insertion of $p(4)$, which are $(p(4), p(1), p(3))$, $(p(1), p(4), p(3))$, and $(p(1), p(3), p(4))$. The corresponding completion times are 17, 17, and 16, respectively. Then, the best partial scheduling order so far is $(p(1), p(3), p(4))$. Following the same procedure, we can get the final schedule, $(p(1), p(3), p(4), p(2))$, which achieves the best result of 18.

C. General DAG Architecture

In this subsection, we would like to discuss the general DAG DNNs. A general DAG DNN has a set of dependent paths whose vertices may overlap with each other. The dependent



(a) before Conversion, G (b) after Conversion, G'

Fig. 5. Apply to general network architecture.

Algorithm 3 DAG Conversion Algorithm

Input: $G(V, E)$

Output: A multi-path architecture DNN G'

Sort V via the topological order.
while $\exists v_i \in V \setminus \{v_0, v_n\}$ **and** out/in-degree$(v_i) > 1$ **do**
  if out-degree$(v_i) > 1$ **then**
    Replicate v_i and its incoming edge.
    A v_i links to a unique $v_j, e_{ij} \in V$.
  if in-degree$(v_i) > 1$ **then**
    Replicate v_i and its outgoing edge.
    A unique v_j links to v_i, $e_{ji} \in V$.

relationship between paths introduces a new challenge, and we cannot directly use the solution discussed in Section IV-B. An example is shown in Fig. 5(a), where two paths use the layer v_2 to continue their processing.

To decouple the dependency between paths, we propose a network conversion strategy without changing the intermediate results. The detailed conversion is shown in Algorithm 3. For any vertex in the middle of the graph,

- If the out-degree of a vertex is k $(k > 1)$, that vertex replicates k times so that each new vertex connects to one unique successor vertex.
- If the in-degree of a vertex is k $(k > 1)$, that vertex replicates k times so that each new vertex connects to one unique predecessor vertex.

A conversion example of Algorithm 3 is shown in Fig. 5. We first find all vertices whose in-degree or out-degree are larger than 1 and conduct replication. In this example, vertex v_2 is selected and will be replicated once. one v_2 links to v_3 and one links to v_4. In the second round, we find v_4 whose in-degree is 2 and thus it will be replicated once. Then, one links to v_2 and one links to v_1. After the second round, all vertices except v_1 and v_5 have the in-degree and out-degree of one and Algorithm 3 terminates. After the conversion, we can apply the corresponding solutions in Section IV-B. Note that the replicated vertices in the converted graph (e.g., vertices v_2 and v_4) are only used to generate the scheduling order of each path. Only one of the replicated vertices will be processed and all other replicated vertices will be skipped in the actual DNN computation and communication processing.

V. Evaluation

A. Device Information

We implement our proposed scheduling algorithms on a typical computation offloading setting. Specifically, we use a Raspberry Pi 4 model B as the local end device, which has a quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz and 4 GB RAM. We set up an edge server by using a desktop in our lab which has a six-core CPU (i7-8700) @ 3.20GHz, a GTX 1080 GPU, and 32 GB RAM. The operating system and key software that we use are Ubuntu 20.04, Python 3.8, Torch 1.7, CUDA 11.0, and cudnn 8.0.3.

B. System Setup

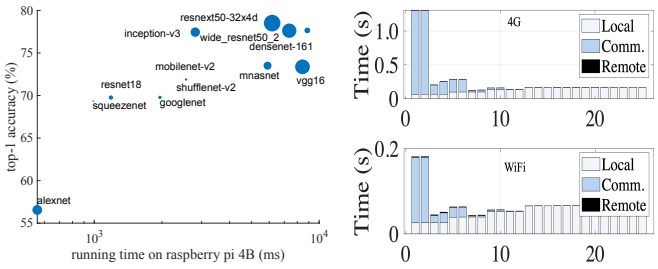
In the experiments, each DNN model is partitioned based on the network partition decision. We use `gRPC`, an open source flexible remote procedure call interface for inter-process communication. `torch.save()` function is called to conduct serialization for the intermediate result of the local device and measure the size of the generated data (e.g., `sys.getsizeof()`). The intermediate result is then saved into an `BytesIO` which uses memory buffer for quick transmission without disk read/write operation.

In our implementation, the local device will send a gRPC request to the server, including (1) a `string` which indicates DNN model and partition information, so that the server can resume with second part of the DNN model correctly (the string encodes the name of the DNN model and pre-defined IDs of partitioned layers instead of the model parameters to reduce communication volume), and (2) a `byte array` data contains the serialized intermediate result. Once the server receives a gRPC request, it will decode the request message to get the DNN model information from the string and decode the `byte array` data with `torch.load()` function. The server will send back a gRPC message after finishing the second part of DNN models with two pieces of information, (1) the DNN result and (2) the remote processing latency. With the remote processing time returned, the local device can further calculate communication latency.

The proposed schedule algorithm is conducted on local end device based on the latency measurement of three stages. Specifically, the latency measurement results is conducted offline and stored into a lookup table. As a result, the local device can choose the best schedule in real-time based on the current network environment. We test the prototype under different wireless connection conditions by varying the bandwidth between local and remote servers. Particularly, we emphasize three common network conditions, including, 3G, 4G and WiFi environments. Typical bandwidths of 3G, 4G and WiFi network are set to 1.1 Mbps, 5.85 Mbps and 18.88 Mbps, respectively, same as [8].

C. Algorithm Comparison

To test the effectiveness of the proposed algorithms, we compare the following three baselines: (1) Local-Only (LO) algorithm: the entire DNN is processed on the local device. (2) Remote-Only (RO) algorithm: the entire data is offloaded, and the remote server is used to process the entire DNNs. (3) DNN Surgery Light (DSL) algorithm: this algorithm is proposed in [8], which doesn't consider the layer-wise processing pipeline.



(a) inference latency vs. accuracy (b) layer-wise latency

Fig. 6. Latency and accuracy analysis in computation offloading.

D. Performance Evaluation

1) Validating the Necessity of the Proposed Research: Fig. 6(a) shows the inference time of STOA DNN models on Raspberry Pi, and the size of each point denotes the corresponding DNN model size. From Fig. 6(a), a clear trade-off between DNN inference accuracy and running time can be observed. The deep learning accuracy improves in recent models, such as Inception-v3 and ResNeXt50. However, the running time for these models is also very large, i.e., $10s$, which limits them to be applied into local devices. It is worth noting that DNNs models on the right-up region of Fig. 6(a) are general DAG DNNs, and thus the proposed algorithms in this paper is very suitable for them. In Fig. 6(b), we validate the computation and communication latency of VGGNet in 4G and WiFi environments. The results clearly show that cooperative computation offloading can reduce communication traffic significantly; for example, the communication size reduces greatly after the first six layers in Fig. 6(b).

2) Improvement over Local Processing: We compare the proposed EJ and NEH algorithms with two baselines under 3G, 4G, and WiFi environments and the results are shown in Table I. It is worth noting that NEH and EJ algorithms always achieve the same performance and the reason is that the five networks used in the experiments are not very complex. Results show that the naive computation offloading may not be able to reduce the inference latency but significantly increase the inference delay due to the slow communication link. For instance, in 3G environment, RO algorithm has the worst performance for all five networks in our experiments. For large DNN models, such as Multi-stream, the RO algorithm ends with 5x latency. Cooperative computation offloading can dynamically adjust its offloading strategy based on the communication bandwidth and always achieve no worse performance compared with LO and RO algorithms. The proposed algorithms achieve the best performance in all three environments. In Table I, we can clearly find a trend that the impact layer-wise schedule can reduce the inference latency significantly for AlexNet-Parallel, GoogLeNet, Siamese Network (MobileNet v2) and Multi-steam Network (CaffeNet/AlexNet).

3) Improvement over SOTA Cooperative Offloading Schedule: We use the speedup ratio, which is the ratio between the processing time of LO algorithm and the processing time of other algorithms for quantitative evaluation wit the DSL

TABLE I
COMPARISON OF THE INFERENCE LATENCY OF TYPICAL DNN MODELS UNDER DIFFERENT OFFLOADING STRATEGIES (MS).

Model	LO	3G		4G		WiFi	
		RO	Our	RO	Our	RO	Our
AlexNet-P	406	4422	**406**	877	**255**	337	**187**
GoogLeNet	848	4475	**848**	879	**728**	352	**352**
ResNet18	871	4463	**871**	946	**810**	341	**341**
Siamese	3919	8783	**1998**	1702	**1087**	613	**469**
Multi-stream	1198	13146	**931**	2513	**692**	891	**317**

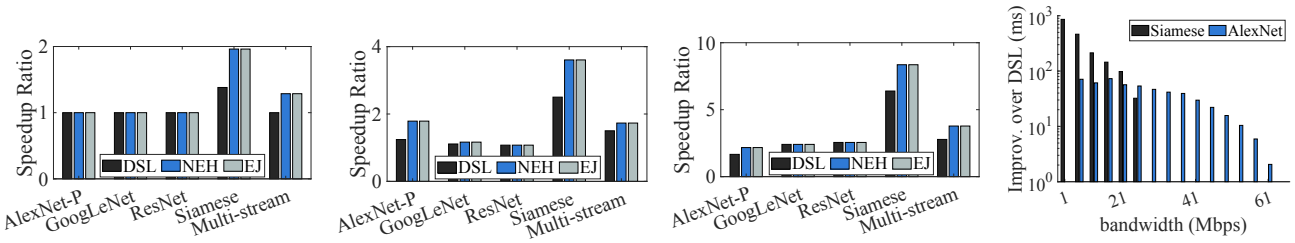
algorithms. The results are shown in Figs. 7(a), 7(b), and 7(c). The results show that the proposed algorithms further reduce the inference latency compared with the DSL algorithm and achieve 20%, 70%, and 1x more speedup than DSL algorithm in Multi-stream and Siamese networks under 3G, 4G, and WiFi environments, respectively. It is because that computation and communication load in these networks are relatively large and thus there is a large room for scheduling optimization.

4) Various Communication Settings: In Fig. 7(d), we investigate the advantage of the proposed method over the DSL algorithm in a wide range of network environments. Particularly, we test AlexNet-Parallel and Siamese Network under a communication range of [1, 80] Mbps. In Fig. 7(d), we directly compare the latency improvement over DSL for the proposed algorithms. The results show that the proposed schedule methods can further reduce up to 1 second over a wide range of bandwidths. It is worth noting that AlexNet-Parallel can further reduce latency in a wider range compared with Siamese Network. The reason is that if the communication bandwidth is larger than 26 Mbps, the bandwidth is fast enough so that the optimal strategy is RO algorithm.

5) Impact of Overhead: We evaluate the overhead introduced by cooperative computation offloading. There are two types of overhead. First, the proposed schedule algorithms cause extra scheduling latency. Fig. 8(a) shows the results of the introduced overhead of the proposed two algorithms. We run the proposed algorithms 50 times to get the average running time at the Raspberry Pi. The results show that both algorithms can finish within the order of microseconds. It is negligible compared with the inference latency, which has the order of milliseconds. Overall, EJ algorithm is relatively faster compared with NEH algorithm. Second, offloading intermediate data to the server causes extra communication latency, which is between 5 to 20 ms in our experiments. As a result, the communication overhead should not be ignored.

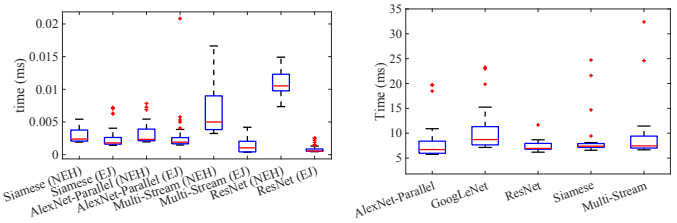
VI. CONCLUSION

In this paper, we discuss the importance of layer-wise processing pipeline on the inference completion time and prove that it is NP-hard. We categorize SOTA DNNs into three different architectures and develop the optimal solution in line and certain multi-path architectures, and efficient offloading solutions in all other architectures to minimize the processing latency. We validate the proposed schemes via real-world implementations on SOTA DNNs and the results show that the proposed schedule methods can further reduce inference latency by more than 8x in the best case.



(a) 3G (b) 4G (c) WiFi (d) various bandwidth

Fig. 7. Speedup ratio of five DNNs under different network environments.



(a) schedule overhead (b) communication overhead

Fig. 8. computation offloading overhead.

VII. Acknowledgements

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

References

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

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

[3] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[4] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, "Revisiting unreasonable effectiveness of data in deep learning era," in *IEEE ICCV*, 2017.

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

[6] H. Touvron, A. Vedaldi, M. Douze, and H. Jégou, "Fixing the train-test resolution discrepancy: Fixefficientnet," *arXiv preprint arXiv:2003.08237*, 2020.

[7] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

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

[9] 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.

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

[11] Directed acyclic graph (dag) network for deep learning. [Online]. Available: https://www.mathworks.com/help/deeplearning/ref/dagnetwork

[12] P. Aleksic, M. Ghodsi, A. Michaely, C. Allauzen, K. Hall, B. Roark, D. Rybach, and P. Moreno, "Bringing contextual information to google speech recognition," 2015.

[13] Tensorflow lite. [Online]. Available: https://www.tensorflow.org/lite

[14] 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.

[15] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *ECCV*. Springer, 2016.

[16] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *IEEE CVPR*, 2017.

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

[18] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, "Adaptive parallel execution of deep neural networks on heterogeneous edge devices," in *ACM/IEEE SEC*, 2019.

[19] C. Canel, T. Kim, G. Zhou, C. Li, H. Lim, D. G. Andersen, M. Kaminsky, and S. R. Dulloor, "Scaling video analytics on constrained edge nodes," *arXiv preprint arXiv:1905.13536*, 2019.

[20] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition on mobile devices," in *ACM SenSys*, 2015.

[21] 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 *IEEE INFOCOM*, 2020.

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

[23] B. Lin, Y. Huang, J. Zhang, J. Hu, X. Chen, and J. Li, "Cost-driven offloading for dnn-based applications over cloud, edge and end devices," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 8, 2019.

[24] J. Wu, *Distributed system design*. CRC press, 1998.

[25] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *IEEE CVPR*, 2016.

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

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

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

[29] M. A. Ponti, L. S. F. Ribeiro, T. S. Nazare, T. Bui, and J. Collomosse, "Everything you wanted to know about deep learning for computer vision but were afraid to ask," in *IEEE SIBGRAPI-T*, 2017.

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

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

[32] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *IEEE CVPR*, 2018.

[33] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *IEEE CVPR*, 2017.

[34] H. Fan and H. Ling, "Parallel tracking and verifying: A framework for real-time and high accuracy visual tracking," in *IEEE ICCV*, 2017.

[35] 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.

[36] B. Chen, C. A. Glass, C. N. Potts, and V. A. Strusevich, "A new heuristic for three-machine flow shop scheduling," *Operations Research*, vol. 44, no. 6, pp. 891–898, 1996.

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

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

APPENDIX

A. Proof of Theorem 1

Proof. We can reduce the 3-machine flow-shop problem to the DeepInference-L. The 3-machine flow-shop problem is a well-known NP-hard problem [38]. We prove that, for any instance of 3-machine flow-shop problem with m jobs, we can reduce it to a special case of our DeepInference-L problem by building a special DAG which has m separate computational paths as shown in Fig. 9. Any path will have three stages in such a special DNN. It is worth noting that if we consider the network partition in a path-based perspective, we use p_1, p_2, and p_3 to denote the corresponding local processing time, output transmission time, and the remote processing time for that path in three stages, respectively. Particularly, (1) $p_1 = \sum_{l=b}^{c} t_l$, (2) $p_2 = s_c$, and (3) $p_3 = \sum_{l=c+1}^{e} t'_l$, where v_b, v_c, and v_e are the beginning vertex, the last vertex processed locally, and the last vertex of this path, respectively. We can build layers so that p_1, p_2, and p_3 are equal to the processing times of job i in three machines in polynomial time.

3-machine flow shop ⇒ DeepInference-L: Assume that there exists an optimal schedule for 3-machine flow shop. Then, the optimal solution of 3-machine flow shop can be used for the DeepInference-L to get the minimum computation time in the above-mentioned special case.

DeepInference-L ⇒ 3-machine flow shop: Similarly, assume that we find the optimal solution of this special case of DeepInference-L. Then, we can derive the optimal solution for the corresponding 3-machine flow shop problem based on the above reduction.

DeepInference-L is as hard as 3-machine flow shop. □

B. Proof of Theorem 2

Proof. Theorem 2 can be proved via contradiction. Assume that there exists an optimal transmission schedule $\sigma^\star$, where v_j's output is transmitted earlier than that of v_i's output and $v_i \prec v_j$. The completion time for $v_{j'}$, $C_{j'}(\sigma^\star)$, is

$$\sum_{k=1}^{j} s_k + \sum_{k=1}^{j'-1} \max\{C_k(\sigma^\star) - \sum_{l=1}^{k'} s_l - t'_k, 0\} + t'_{j'}, \quad (6)$$

where the first part is the data transmission time up to $v_{j'}$, the second part is the idle time due to speed mismatch between remote processing and communication, and the third part is the remote computation time of $v_{j'}$. Then, if we compare $C_{j'}(\sigma^\star)$ with schedule σ's result $C_{j'}(\sigma)$, where the only difference between σ and $\sigma^\star$ is that the transmission orders of v_i and v_j's are swapped, we will get the following result,

$$\begin{aligned}
&C_{j'}(\sigma) - C_{j'}(\sigma^\star) = \\
&\sum_{k=1}^{i'-1} \max\{C_k(\sigma) - \sum_{l=1}^{k'} s_l, 0\} - \sum_{k=1}^{i'-1} \max\{C_k(\sigma^\star) - \sum_{l=1}^{k'} s_l, 0\} \\
&+ \max\{C_{i'}(\sigma) - \sum_{l=1}^{k'} s_l, 0\} - \max\{C_{i'}(\sigma^\star) - \sum_{l=1}^{k'} s_l, 0\} + \\
&\sum_{k=i'+1}^{j'-1} \max\{C_k(\sigma) - \sum_{l=1}^{k'} s_l, 0\} - \sum_{k=i'+1}^{j'-1} \max\{C_k(\sigma^\star) - \sum_{l=1}^{k'} s_l, 0\},
\end{aligned}$$

where the three parts are completion time difference between all vertices before $v_{i'}$, the vertex $v_{i'}$, and all vertices between

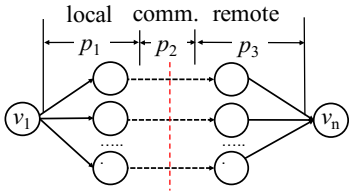


Fig. 9. An example of NP-hard reduction.

$v_{i'}$ and $v_{j'}$. Particularly, the first part is 0 since both schedules are exactly the same until $v_{i'}$. The second part is no larger than 0 since the v_i does not need to wait for s_j in schedule σ. For the third part, the difference is also no larger than 0 since

$$\begin{aligned}
&\sum_{k=i'+1}^{j'-1} \max\{C_k(\sigma) - \sum_{l=1}^{k'} s_l, 0\} - \sum_{k=i'+1}^{i'-1} \max\{C_k(\sigma^\star) - \sum_{l=1}^{k'} s_l, 0\} \\
&= \sum_{k=i'+1}^{j'-1} \max\{C_k(\sigma) - C_k(\sigma^\star) - s_j, 0\} \le 0.
\end{aligned}$$

$C_k(\sigma) - C_k(\sigma^\star) - s_j$ is no larger than 0 due to overlap between communication and remote computation. Then, we find a counter-example. □

C. Proof of Theorem 3

Proof. This theorem can be proved with contradiction. A non-preemptive path-based schedule can be denoted as $(p(1), p(2), \ldots, p(m))$ at a stage, where the index i to denote the ith scheduled path. By a slight abuse of notation, we ignore the subscript information which denotes the stage in this proof to simplify notation. Assume that the optimal schedule $\sigma^\star$ is preemptive, which means that there exists a path, e.g., $p(i)$, which is split into two sub-stages, denoted as $p_1(i)$ and $p_2(i), p(i) = p_1(i)||p_2(i)$. If $p_1(i)$ and $p_2(i)$ are scheduled consecutively, it is equivalent to a non-preemptive schedule. Therefore, we can demote this schedule as the following sequence, $(\cdots, p_1(i), \cdots, p_2(i), \cdots)$. without loss of generality, we assume that the starting time of $p(1)$ is 0 and the idle time of path $p(k)$ is I_k. Then, the completion time of $p(j)$ is $\sum_{k=1}^{j}(p(k) + I_k)$. If we delay $p_1(i)$ and process it together with $p_2(i)$ and keep all other paths' schedule, we get a new schedule σ, $(\cdots, p_1(i), p_2(i), \cdots)$, and the completion time of each path should be updated as follows. For any path $p(j)$, $p(j) \prec p_1(i)$, the completion time of $p(j)$ should still be $\sum_{k=1}^{j}(p(k) + I_k)$ since the schedule sequence is the same in both schedules. For any path $p(j)$, $p_1(i) \prec p(j) \prec p_2(i)$, the completion time of $p(j)$ should still be $\sum_{k=1}^{j}(p(k) + I'_k) - p_1(i) - I_{1i}$. It is worth noting that $I'_j \le I_j, \forall j$ due to the fact that

$$I'_k = \begin{cases} 0 & \text{stage 1} \\ I_k - \max\{p'(k) - p(k-1), 0\} & \text{stages 2 and 3} \end{cases}$$

where $p'(k)$ is the processing time in the previous stage. Therefore, each path gets a smaller completion time, including $p_2(i)$. For any path $p(j), p_2(i) \prec p(j)$ in σ, the processing time becomes smaller which is because that paths between $p_1(i)$ and $p_2(i)$ have a smaller completion time and thus all the following paths can start earlier. □