PICO: Pipeline Inference Framework for Versatile CNNs on Diverse Mobile Devices

Xiang Yang, Zikang Xu, Qi Qi, Jingyu Wang, Haifeng Sun, Jianxin Liao, and Song Guo, *Fellow, IEEE*

Abstract—Distributing the inference of convolutional neural network (CNN) to multiple mobile devices has been studied in recent years to achieve real-time inference without losing accuracy. However, how to map CNN to devices remains a challenge. On the one hand, scheduling the workload of state-of-the-art CNNs with multiple devices is NP-Hard because the structures of CNNs are directed acyclic graphs (DAG) rather than simple chains. On the other hand, distributing the inference workload suffers from expensive communication and unbalanced computation due to the wireless environment and heterogeneous devices. This paper presents PICO, a pipeline cooperation framework to accelerate the inference of versatile CNNs on diverse mobile devices. At its core, PICO features: (1) a generic graph partition algorithm that considers the characteristics of any given CNN and orchestrates it into a list of model pieces with suitable granularity, and (2) a many-to-many mapping algorithm that produces the best pipeline configuration for heterogeneous devices. In our experiment with $2 \sim 8$ Raspberry-Pi devices, the throughput can be improved by $1.8 \sim 6.8\times$ under different CPU frequencies.

Index Terms—Mobile Computing, Pipeline Inference, Model Deployment.

✦

1 INTRODUCTION

R ECENT years witness an explosive growth of mobile devices. The huge number of mobile devices provides a large volume of data (images, videos, etc.). Meanwhile, versatile convolution neural networks (CNN) with pre-trained parameters become powerful tools to make intelligent decisions using these raw data (*CNN inference*). Embedding CNN with mobile devices enables many intelligent applications to become reality, such as smart home, intelligent factory, and even automatic driving [1], [2].

One obstacle to the embedding is the resource-limited mobile devices. Compared with datacenter, the computing capability of mobile devices is not enough to perform CNN inference alone. But on the contrary, the current wireless network is not prepared for transmitting the massive volume of raw data collected by these mobile devices. For example, an autopilot camera could capture more than 700 MB video record every second [3], and uploading all the video data to the datacenter will bring significant network latency. Moreover, uploading data from user devices to the cloud always brings concern about privacy [3].

Benefitted from the spatial independence of convolution operation, the input and output (*feature*) of convolutional (*conv*) layers can be split into several small tiles and executed on different devices. As a consequence, cooperative CNN inference on multiple mobile devices gains the attention of researchers recently [4], [5], [6]. During inference procedure, the data source (camera, sensors, etc.) captures raw data and splits it into tiles. These tiles are distributed to multiple

nearby mobile devices through a wireless network and executed independently using one or several layers. Then the data source is responsible to gather all the output tiles and stitch them to obtain the result. The procedure will be iterated multiple times until all layers are executed. Cooperative inference also protects user privacy since all the data stay in local. Moreover, the closer to the data source, the lower network latency it suffers. Compared with other strategies such as model compression and parameter pruning [7], [8], [9], cooperative inference neither losses the inference accuracy nor requires re-train the model.

However, despite all these benefits, there still leave some challenges that are not completely solved in previous works. Although the input feature can be parallel executed, (1) **the parallelism introduces redundant calculation** due to the property of CNN. The scalar in the output feature of one conv layer is calculated through a dot product with the conv kernel and a subregion of input feature. For most cases, the kernel size is bigger than 1×1, so that the input tiles of partitioned input feature will overlap with each other to guarantee the scalars at the edge of output tiles are correct. Moreover, the overlapped part will increase recursively when devices execute multiple layers during one iteration in the inference procedure, but the communication is expensive for mobile devices. As a consequence, the executed layers need to be carefully chosen. However, (2) **the structures of many CNNs are directed acyclic graphs** (DAG) rather than chains. ResNet34 [10] uses skip-connection technology that allows a layer to directly connects to another deeper layer. The structure of InceptionV3 [11] contains multiple branches to capture more information from the input feature. These complex structures lead to a huge number of possible choices. Previous works mainly focused on the chain structure [4], [5], [6], which is much easier than DAG. Compared with datacenter, (3) **the computing resources of mobile devices are diverse**, the heterogeneous environment also hinders the optimization for cooperative inference.

- *Xiang Yang, Zikang Xu, Qi Qi, Jingyu Wang, Haifeng Sun and Jianxin Liao are with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. E-mail: {yangxiang, xuzikang, qiqi8266, wangjingyu, hfsun, liaojx}@bupt.edu.cn.*
- *Song Guo is an IEEE Fellow (Computer Society) and an ACM Distinguished Member with the Department of Computing at The Hong Kong Polytechnic University. E-mail: cssongguo@comp.polyu.edu.hk*
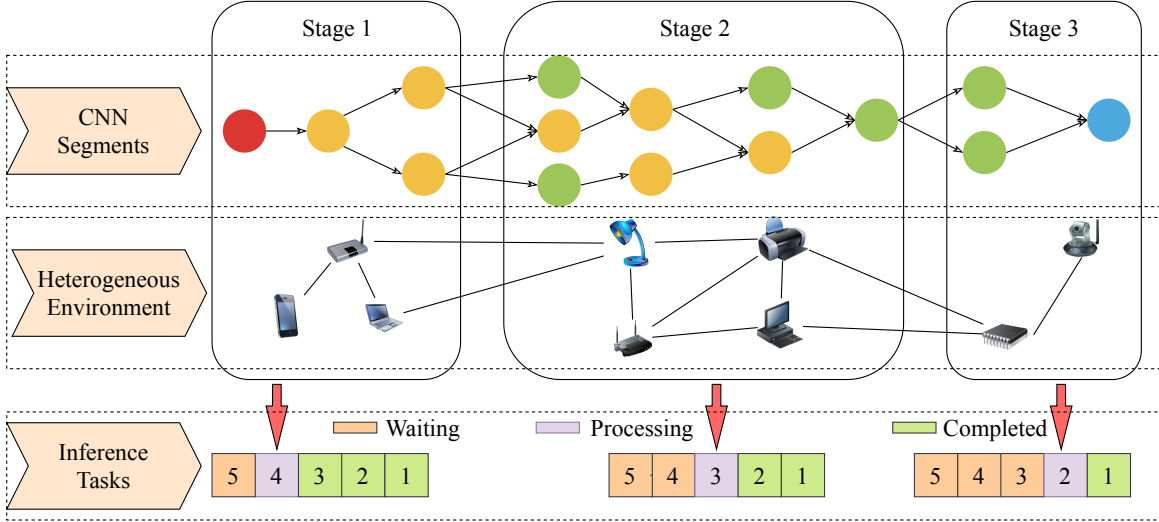- *Qi Qi and Jingyu Wang are the corresponding authors.*

Fig. 1: A diagrammatic sketch of pipeline inference.

In this paper, we explore previous works about parallelizing the CNN inference and propose a pipeline cooperation (PICO) framework for accelerating the inference on diverse mobile devices. Fig. 1 plots a diagrammatic sketch of our framework. PICO divides the entire CNN graph and mobile devices into 3 *stages*. These stages compose an efficient inference pipeline. Since each stage owns a small segment of original CNN and a subset of mobile devices, both communication overhead and the redundant calculation can be significantly reduced. There are two import metrics for pipeline: *latency* and *period*. The first term is the sum of inference latencies of all stages and the last term is the longest latency among stages. Obviously decreasing the period tends to increase the latency. Our optimization goal is to minimize the redundant calculation and period (maximize throughput) meanwhile to keep the latency of the pipeline under a certain value.

We first formulate the pipeline inference, then we analyze the complexity of the optimization problem and find that it is NP-Hard to directly obtain the optimal result. Based on our analysis, PICO uses a two-step optimization to maximize the throughput. In the first step, we orchestrate the CNN graph into a sequence of *pieces*. These pieces have the minimum redundant calculation inside and compose the original CNN graph in a chain structure. Then we choose the best partition for these pieces and devices to construct the inference pipeline. The algorithms used in the above procedures are based on dynamic programming.

In our experiment we use $2 \sim 8$ Raspberry-Pi devices to evaluate PICO framework. The throughput can be improved by $1.8 \sim 6.8\times$ under different CPU frequencies and number of devices.

In a nutshell, we make the following contributions:

- We present a pipeline cooperation (PICO) framework to accelerate CNN inference with diverse mobile devices.
- We propose an algorithm to split the complex CNN graph structure into more fine-grained pieces.
- We propose an algorithm to decide the optimal stage

settings for inference pipeline which maximize the throughput.
- We apply our technique on a cluster consisting of Raspberry-Pi-based hardware and evaluate image recognition and object detection CNN models.

The rest of this paper is organized as follows: Section 2 provides background information of CNN and different parallelization strategies in mobile devices. Section 3 formulates the inference process and gives a cost model for further optimization. Section 4 and 5 describe our approach to find near-optimal parallelization. Section 6 presents the results of our evaluation. Section 7 details the related work and Section 8 concludes.

2 BACKGROUND AND MOTIVATION

In this section, we briefly introduce the CNN inference and the current parallel schemes. Then we propose our pipeline cooperation scheme.

2.1 Procedure of CNN Inference

The convolution layer (*conv*) is the key module during CNN inference, Each conv layer owns a set of kernels. To produces the output feature, conv layers use their kernels to slide over the input feature received from the previous layer. Every movement of the kernel will produce a scalar in the output feature by a dot product between the weights of kernels and a small subregion of the input. The pooling layer (*pool*) performs a down-sampling operation. It is used to progressively reduce the number of parameters, memory footprint and amount of computation in the network.

Conv operation is the biggest bottleneck. Fig. 2 plots the computation and communication percentage by layer for two classic CNNs (VGG16 [12], YOLOv2 [13]). From the figure we can find conv layers dominate the consumption of computing resources. The conv operations occupy 99.19% of the computation in VGG16 and 99.59% in YOLOv2. How to efficiently execute conv operations is the key to accelerating CNN inference. Another finding is the variation. Since

(a) VGG16

(b) YOLOv2

Fig. 2: The communication and computation percentages of each layer.

different conv layers have different configurations (kernel size, padding, in and out channels), the communication or computation percentage also varies.

2.2 Parallelizing CNNs With Mobile Devices

Benefitted from the spatial independence of conv operations, the inference can be parallel executed by splitting the input feature into multiple tiles and distributing them to different mobile devices, as shown in Fig. 4. We refer this technology as *feature partition*. However, the partition of input feature overlaps with each other due to the property of conv operations. In Fig. 4, an input feature is split into four tiles and distributed to four devices. Assume the corresponding conv layer has a 3×3 kernel size, to obtain the correct value in P_1, the calculation with 3×3 kernel has to use more proportion (the edges of the yellow and pink region) of the input feature. This property leads to a *redundant calculation* and increases the difficulty of the design of parallel algorithm.

We next introduce the two parallelization schemes used in this paper. [4] is the first work that uses feature partition for cooperative CNN inference. For each layer, the basic idea is to split the input feature into tiles and distribute them to all devices, then gather them to obtain the output of this layer. We refer such a scheme as *layer-wise* parallelization. In a WLAN network, it can cause substantial network latency. The gain of layer-wise parallelization is significantly defeated by communication overhead. To reduce the communication among devices, *fused-layer* parallelization was introduced in [5] and [6]. This scheme fuses multiple layers instead of distributing the computation of every layer individually. Thus, mobile devices can execute the calculation of multiple layers without communication. But since the input will go through multiple layers, to obtain the correct value of output feature, the overlapped part of the input increases

recursively. In addition, all mobile devices need a full copy of original CNN for the two schemes, which increases the memory footprint.

2.3 The Structures Of CNNs

The structures of CNNs can be divided into three categories. We plot Fig. 3 to give an illustration. Note the norm layer and activation layer are ignored since they do not change the input and output shape and has less proportion of computation.

The earlier model such as VGG16 [12] and YOLOv2 [13] are built with the (1) **chain** structures. This structure is simple: neural layers inside the model are connected one by one, and the output of the previous layer is the input of the next layer. We plot the model structure of VGG16 in Fig. 3a for further explanation.

Later, the (2) **block** structure becomes popular in CNNs. Block structure enables CNNs to capture multiple features of input data to improve its performance using carefully designed blocks [11] and prevent the vanishing gradient problem when training deeper model [10]. It uses blocks to replace the layers in chain structure. All the blocks are still connected one by one, but inside the block, neural layers can be represented as an acyclic directed graph (DAG). Fig. 3b plots the 8th and 9th blocks in InceptionV3 [11]. Each block has multiple branches and contains several conv and pool layers, and these blocks are connected with the Contact operations that stacks the output of every sink layer of previous block in channel dimension and feeds the result to the next block.

To avoid manual design of the model structure, neural architecture search (NAS) is proposed. Compared with the previous two structures, the output structure of NAS is usually a complete graph, which can not be divided into sequence of blocks. We refer the structure as (3) **graph** structure. We plot a partition of NasNetMobile [14] in Fig. 3c, which has two source layers and three sink layers. The complex structure of CNN models is a big challenge for optimizing parallel strategy.

2.4 Motivation And Pipeline Inference

2.4.1 Motivation

In the above discussion, How to tackle the complex graph structure of CNN model and how to reduce the redundant calculation are keys to accelerating the CNN inference.

Tackle the complex structure: Cooperative inference needs to distribute the CNN model into multiple devices, but the structure of these model is complex and prevent more fine-grained optimization. Lots of previous works focus [4], [5], [6] on cooperative inference, they only consider chain structure. There lacks a solution for the more complex block and graph structures.

Reduce the redundant calculation: Communication is expensive in mobile environment. For layer-wise scheme, frequent communication among mobile devices causes inefficient performance. The redundant calculation also limits the cooperation of mobile devices for CNN inference. For fused-layer scheme, the redundant calculation quickly grows as the number of fused layers or devices increases.

(a) Chain Structure (*VGG16*)

(b) Block Structure (*InceptionV3*)
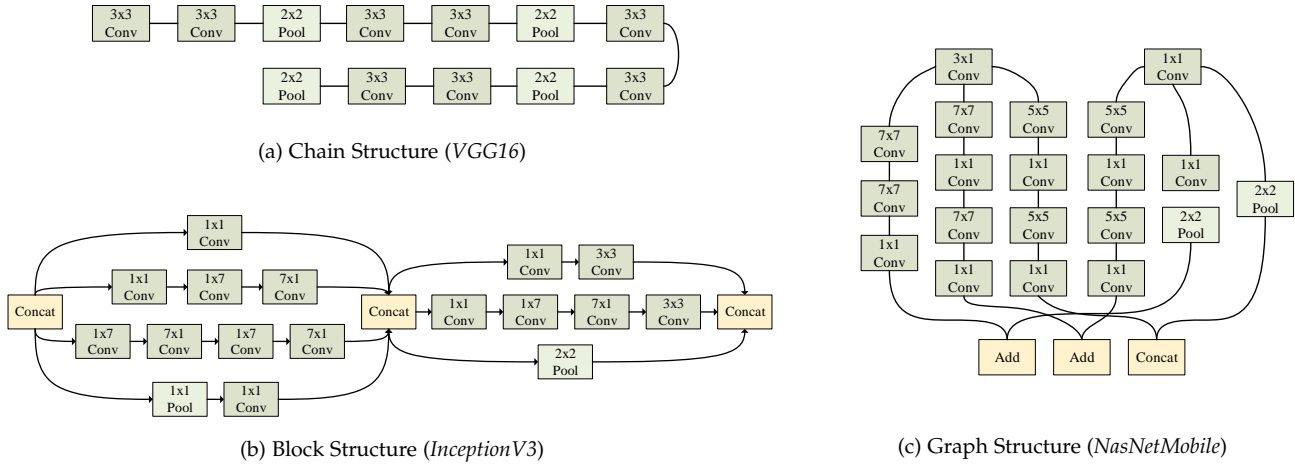

(c) Graph Structure (*NasNetMobile*)

Fig. 3: CNNs with different structures: The chain structure is the simplest one which just put the neural layers into a sequence. Block structure replaces the element in chain structure from neural layer to block, each block can be seen as a directed acyclic graph (DAG). Graph structure can not be partitioned into blocks, the entire structure is a huge DAG.
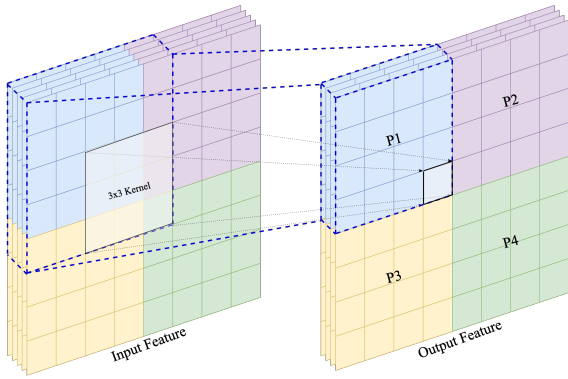

Fig. 4: Feature map partition strategy introduces redundant calculation.

(a) Per device overhead   (b) Total computation overhead

Fig. 5: Computation overhead with different partition settings.

To give a detailed explanation, we evaluate the required floating-point operations (FLOPs) for VGG16 with different numbers of fused layers and mobile devices. Fig. 5a presents the FLOPs per device meanwhile Fig. 5b shows the sum of FLOPs of all devices. We can find that fused-layer strategy performs well at the start, but when the numbers of fused layers or devices increase, the redundant calculation quickly grows.

2.4.2 Pipeline Inference

From the above discussion, the acceleration of CNN inference faces challenges when the number of devices or fused layers grows. For layer-wise scheme, the devices are idle in most time due to frequent communication and expensive network latency. On the contrary, devices can keep running with the fused-layer scheme, but it is whistling to the wind since the most computation is redundant. Here we introduce the *pipeline* scheme for parallelizing the CNN inference. This scheme divides both layers and mobile devices into several groups, as shown in Fig. 1. We refer such a group as *stage* in our description. The inference inside the stage uses fused-layer scheme and the entire CNN inference is performed stage by stage. If we set the number of stages to 1, The fused-layer scheme is a special case of the pipeline scheme. To maximize the inference throughput, the inference latency of every stage should be optimized as close as possible.

Using pipeline for inference has several advantages. (1) First, the communication overhead can be reduced since the calculated features only need to be synchronized among a subset of devices. (2) Second, the proportion of redundant calculation also decreases due to smaller numbers of layers and devices. (3) Third, each device owns a segment of CNN instead of the entire model, which reduces the memory footprint.

The concept of pipeline is widely adopted in task scheduling [15], [16] which maps multiple processors to an application composed of several tasks. However, pipeline meets difficulties when applied to CNN inference. The structure of CNN is a directed acyclic graph (DAG) rather than a chain, the mapping has to consider the data flow of DAG. Generally, the number of layers in CNN is more than the number of devices, thus the mapping is many-to-many, and different mapping strategy also changes redundant calculation. Moreover, the heterogeneous environment is also a big challenge.

TABLE 1: Notation definitions

Notation	Description
$\mathbb{G}:(\mathbb{V},\mathbb{E})$	CNN with graph structure.
$\mathbb{D}$	A heterogeneous cluster with D devices.
l_i	Layer i in model $\mathbb{G}$.
w_i, h_i	The width and height of the output frame of l_i.
k_i, p_i, s_i, c_i	Kernel size, padding, stride, and channel of l_i.
d_k	A device in cluster $\mathbb{D}$.
F_i^k	Input feature frame of layer l_i for device d_k.
$\mathcal{F}^k$	Set of all input feature for layers assigned to d_k.
$\mathcal{F}_{in}^k$	Input feature for source layers assigned to d_k.
$\mathcal{F}_{out}^k$	Output feature for sink layers assigned to d_k.
$b(d_h, d_k)$	Bandwidth between device d_h and device d_k.
$\mathcal{D}$	A subset of devices.
$\mathcal{M}:(\mathcal{V},\mathcal{E})$	Model partition deployed on $d_k \in \mathcal{D}$.
$\mathcal{S}:(\mathcal{M},\mathcal{D})$	A stage that belongs to the inference pipeline.
$\mathcal{M}_E$	Ending piece of CNN $\mathbb{G}$.
$\varphi(\mathcal{F}_k)$	Input frame size of F_k.
$\theta(\mathcal{M};\mathcal{F}^k)$	Required computing resources of $\mathcal{M}$.
$\vartheta(d_k)$	Computing capacity of device d_k.
$t_{comm}(d_k, \mathcal{F}^k)$	Communication time of device d_k.
$t_{comp}(d_f, d_k, \mathcal{F}^k)$	Computation time of device d_k.
$T(\mathcal{S})$	Time overhead for executing stage $\mathcal{S}$.
$T_{comm}(\mathcal{S})$	Communication time of stage $\mathcal{S}$.
$T_{comp}(\mathcal{S})$	Computation time of stage $\mathcal{S}$.
T_{lim}	Inference latency limit for optimization.
$\mathbb{S}$	Pipeline configuration containing all stages $\mathcal{S}$.
$\mathbb{S}^\star$	Optimal stage configuration.
$\mathcal{T}(\mathbb{G},\mathbb{D},\mathbb{S})$	Latency of the pipeline under configuration $\mathbb{S}$.
$\mathcal{P}(\mathbb{G},\mathbb{D},\mathbb{S})$	Period of the pipeline under configuration $\mathbb{S}$.

3 SYSTEM MODEL

In this section, we define our optimization problem for pipeline inference.

3.1 Problem Define

Generally speaking, our goal is to divide both CNN model with graph structure and mobile devices with heterogeneous computing resources into several stages properly, so that these stages could compose an inference pipeline that maximizes the throughput.

3.1.1 CNN With Graph Structure

We use an acyclic directed graph (DAG) $\mathbb{G}:<\mathbb{V},\mathbb{E}>$ to represent a given CNN model. The vertex set $\mathbb{V}$ contains all the neural layers and connector (e.g., Add and Contact in Fig. 3) $l_i \in \mathbb{V}$, and the elements (l_i, l_j) in the edge set $\mathbb{E}$ denotes the data flow of CNN model $\mathbb{G}$. In particular, $(l_i, l_j) \in \mathcal{E}$ means the output of layer l_i is the input of layer l_j. Since the CNN model will be executed as an inference pipeline with multiple stages, the $\mathbb{G}$ also needs to be split into multiple parts. We refer these parts as *segments*. A segment $\mathcal{M}:<\mathcal{V},\mathcal{E}>$ is a subset of original DAG $\mathbb{G}$, where $\mathcal{V} \subseteq \mathbb{V}$ and $\mathcal{E} \subseteq \mathbb{E}$.

Note the segment *is not* a regular smaller graph, since the edge set $\mathcal{E}$ contains some vertices that are not included by $\mathcal{V}$. Take Fig. 1 as an example, these segments on the top also contain edges that are connected with previous or next segments. Here we give some definitions of segment to simply our following modeling:

Definition 1. A subset $\mathcal{M}:<\mathcal{V},\mathcal{E}>$ is a *segment* of original graph $\mathbb{G}:<\mathbb{V},\mathbb{E}>$ if for all $e:(u,v) \in \mathbb{E}$, once u or v belongs to $\mathcal{V}$, e also belongs to $\mathcal{E}$.

Definition 2. For a segment $\mathcal{M}:<\mathcal{V},\mathcal{E}>$ and an edge $(u,v) \in \mathcal{E}$, if $u \notin \mathcal{V}$, then v is a *source* vertex of $\mathcal{M}$.

Definition 3. For a segment $\mathcal{M}:<\mathcal{V},\mathcal{E}>$ and an edge $(u,v) \in \mathcal{E}$, if $v \notin \mathcal{V}$, then u is a *sink* vertex of $\mathcal{M}$.

3.1.2 Optimization Goal

Given a heterogeneous cluster $\mathbb{D}$, where $d_k \in \mathbb{D}$ is a computing device in the cluster. We assume the computing capacity $\vartheta(d_k)$ of device d_k are known. In our practice, the $\vartheta(d_k)$ denotes floating point operations per second (FLOPS). We also assume the bandwidth between all mobile devices is the same and is known as b. This assumption covers most cases when these devices are under the same WLAN environment such as home and factory [6], [15].

For pipeline scheme, $\mathcal{D} \subseteq \mathbb{D}$ is a subset of heterogeneous devices. Each device $d_k \in \mathcal{D}$ owns a copy of model segment $\mathcal{M}$ but is assigned to produce different region $\mathcal{F}^k$ of the output feature map of all the sink vertex in $\mathcal{M}$. We use $\mathcal{F}$ to present the set of all $\mathcal{F}^k$ in $\mathcal{D}$. A stage $\mathcal{S}$ can be represented as a tuple $(\mathcal{M}, \mathcal{D}, \mathcal{F})$. Let $\mathbb{S}$ denote the set of stages composed by all the stages $\mathcal{S}$ we defined above, the optimization objective is to find such a $\mathbb{S}^\star$ that satisfies:

$$\mathbb{S}^\star = \arg\min_{\mathcal{T}(\mathbb{G},\mathbb{D},\mathbb{S}) \leq T_{lim}} \mathcal{P}(\mathbb{G},\mathbb{D},\mathbb{S}) \qquad (1)$$

where $\mathcal{T}(\mathbb{G},\mathbb{D},\mathbb{S})$ denotes the pipeline latency under specific stage configuration $\mathbb{S}$ and $\mathcal{P}(\mathbb{G},\mathbb{D},\mathbb{S})$ is the period of pipeline. T_{lim} is a hyperparameter that indicates the maximum inference latency we can accept.

3.2 Cost Model

Here we represent our cost model to guide the optimization. First, we quantify the essential input feature size for every device in a stage. Then, we formulate the inference latency of every stage. Finally, we get the inference period and latency of the entire pipeline using previous results.

3.2.1 The Input Feature Size For Devices

Every device d_k owns a segment $\mathcal{M}:<\mathcal{V},\mathcal{E}>$ and needs to produce correct output features $\mathcal{F}^k$. Once the $\mathcal{M}$ and $\mathcal{F}^k$ is given, we need to calculate the necessary input feature size for every layer $l_i \in \mathcal{M}$. The calculation had been discussed in [5], but it only considered models with chain structure. We will extend it into a more complex graph structure here with a top-down algorithm.

To calculate the input feature size of layer l_i, we need to find all the edges (l_i, l_j) start from l_i. We can assume the input feature sizes of all l_j is already calculated. Since the input of l_j is just the output of l_i, the necessary output feature size of l_i can be denoted as:

$$w_i = \max\{w_{i \to j}\}, \; h_i = \max\{h_{i \to j}\}. \qquad (2)$$

Here we use w_i and h_i to denote the necessary width and height of the output feature size of l_i, meanwhile, $w_{i \to j}$ and $h_{i \to j}$ is the input size of layer l_j.

Assume layer l_i has $k_i^w \times k_i^h$ kernel size and s_i stride size, once the output feature size is determined, the height h_i and width w_i of input feature can be calculated using the following equation:

$$w_{* \to i} = (w_i - 1)s_i + k_i^w, \; h_{* \to i} = (h_i - 1)s_i + k_i^h \qquad (3)$$

where $w_{*\to i}$ and $h_{*\to i}$ is the input feature size for l_i. Note this formula suits for both conv and pool layers.

Since the output feature size of all sink vertices of $\mathcal{M}$ is given (corresponding to $\mathcal{F}^k$), we can iteratively calculate all the output and input feature size of all layers in $\mathcal{M}$ with a top-down algorithm. The input feature size of all the source vertices of $\mathcal{M}$ is the input feature size needed by device d_k.

3.2.2 Inference Cost Of Devices

We use $f(l_i; F_i^k)$ to denote the required floating operations (FLOPs) of conv layer l_i when generating an output feature map F_i^k with size $c_i \times w_i \times h_i$. Assume layer l_i is a conv layer with $c_i' \times k_i^w \times k_i^h$ kernel size, c_i output channel and s_i stride size. Since each floating scalar in the output feature is calculated by sliding the kernel over the input feature, $f(l_i; F_i^k)$ can be given by:

$$f(l_i; F_i^k) = k_i^w k_i^h c_i' w_i h_i c_i. \tag{4}$$

Here we ignore the pool layers since they require far fewer FLOPs than conv layers (In Fig. 2).

Note the w_i and h_i in Eq. (3) denote the region of correct output feature. However, the F_i^k is the actual output feature size, which contains not only the correct output but also some redundant parts at the margin of F_i^k. Assume the size of F_i^k is known and (l_i, l_j) is an edge in $\mathcal{M}$, the output F_j^k of layer l_j can be calculated by:

$$w_j = \frac{w_i + 2p_j^w - k_j^w}{s_j^w} + 1, \ h_j = \frac{h_i + 2p_j^h - k_j^h}{s_j^h} + 1 \tag{5}$$

where p_j^w and p_j^h is the padding size of conv layer l_j. Since we have the input feature size for all source vertices in $\mathcal{M}$, to calculate the F_i^k for all layer $l_i \in \mathcal{M}$, we use a bottom-up algorithm similar with above, omit here.

Assume a device d_k is responsible to produce $\mathcal{F}^k$ with model segment $\mathcal{M}$, we can give the required FLOPs operation $\theta(\mathcal{M}; \mathcal{F}^k)$ with:

$$\theta(\mathcal{M}; \mathcal{F}^k) = \sum_{l_j \in \mathcal{M}} f(l_j; F_j^k). \tag{6}$$

Empirical studies by [17] have demonstrated that for specific layers and device, the computation time is proportional to the size of the input or output features, Therefore, the inference time $t_{comp}(d_k, F_k)$ for device d_k can be estimated by the following equation:

$$t_{comp}(d_k, \mathcal{F}^k)) = \alpha_k \frac{\theta(\mathcal{M}; \mathcal{F}^k)}{\vartheta(d_k)} \tag{7}$$

where $\vartheta(d_k)$ is the computing capacity (FLOPS) of device d_k. α_k is a coefficient computed by a regression model.

3.2.3 The Period and Latency Of Pipeline

As each device executes inference in parallel within stage, the computation time for stage $\mathcal{S} :< \mathcal{M}, \mathcal{D} >$ is determined by the maximum inference time among devices in $\mathcal{D}$:

$$T_{comp}(\mathcal{S}) = \max_{d_k \in \mathcal{D}} t_{comp}(d_k, \mathcal{F}^k). \tag{8}$$

Since each device $d_k \in \mathcal{D}$ will generate part of the calculation of stage $\mathcal{S}$, there exists a device d_f which is

TABLE 2: Optimization Complexity

Model Device	Chain	Block	Graph
Homogeneous	P	NP *	NP
Heterogeneous	NP	NP	NP

* [6] solves the optimization by considering the entire block as a special layer. However, this operation introduces lots of unnecessary calculations during inference.

responsible to distribute stage input and gather stage output from other devices. For a device $d_k \in \mathcal{S}$, the feature transferring time $t_{comm}(d_f, d_k, \mathcal{M})$ can be given by:

$$t_{comm}(d_f, d_k, \mathcal{F}) = \frac{\varphi(\mathcal{F}_{in}^k) + \varphi(\mathcal{F}_{out}^k)}{b(d_f, d_k)} \tag{9}$$

where $\varphi(\mathcal{F})$ is the feature size on a given input feature sizes $\mathcal{F}$. Here we use $\mathcal{F}_{in}^k$ and $\mathcal{F}_{out}^k$ to denote the input and output feature sizes of $\mathcal{M}$ owned by d_k. Sum the communication cost for each device d_k in stage $\mathcal{S}$, we define

$$T_{comm}(\mathcal{S}) = \sum_{\substack{d_k \in \mathcal{D} \\ d_k \neq d_f}} t_{comm}(d_f, d_k, \mathcal{F}^k) \tag{10}$$

as the communication cost of stage $\mathcal{S}$.

The cost function for each stage in pipeline inference is then defined as the total time of the frame transfer and layer computation:

$$T(\mathcal{S}) = T_{comp}(\mathcal{S}) + T_{comm}(\mathcal{S}) \tag{11}$$

Note the time for feature map partition and stitch is not discussed here. In practice, it is far less than the layer computation time $T_{comm}(\mathcal{S})$ and could be ignored.

Next, we define the optimization objective as:

$$\mathcal{P}(\mathbb{G}, \mathbb{D}, \mathbb{S}) = \max_{\mathcal{S} \in \mathbb{S}} T(\mathcal{S}), \ \mathcal{T}(\mathbb{G}, \mathbb{D}, \mathbb{S}) = \sum_{\mathcal{S} \in \mathbb{S}} T(\mathcal{S}) \tag{12}$$

where $\mathcal{P}(\mathbb{G}, \mathbb{D}, \mathbb{S})$, $\mathcal{T}(\mathbb{G}, \mathbb{D}, \mathbb{S})$ estimate the maximum execution time of stages in and inference latency in pipeline.

3.3 Analysis

The goal of our optimization algorithm is finding the best stage set $\mathbb{S}^\star$ that minimizes the maximum period $\mathcal{P}(\mathbb{G}, \mathbb{D}, \mathbb{S})$ of pipeline with heterogeneous clusters. Such an optimization faces the following challenges:

- The overhead of computation and communication of each layer in model varies and would be affected by the assigned feature map size F_i^k.
- The computing capacity $\vartheta(d_k)$ of every device in the heterogeneous cluster varies.
- For a specific stage $\mathcal{S}$, the number of devices $|\mathcal{D}|$ and the model segment $\mathcal{M}$ in stage also need to be configured.
- The structure of CNN model can be complex and hard to be partitioned.

In fact, we show the optimal solution can not be found in polynomial time unless $P = NP$.

Theorem 1. Given a CNN model $\mathbb{G}$ with chain structure, the problem of minimizing maximum stage execution time

$\mathcal{P}(\mathbb{G}, \mathbb{D}, \mathbb{S})$ with heterogeneous mobile devices under a constriction that $\mathcal{T}(\mathbb{G}, \mathbb{D}, \mathbb{S}) \leq T_{lim}$ is NP-hard.

Proof 3.1. Considering a scheduling problem defined as follows: Given L identical tasks that are needed to be executed one by one. All tasks can be paralleled to several processors without additional overhead. The goal is to assign these tasks to D heterogeneous devices and maximize the throughput. This problem is proven to be NP-hard by [16]. We can construct a CNN model with chain structure whose layers are identical and the kernel size of each layer is 1×1. This kernel size guarantees there is no overlapped partition when parallels the inference. If there exists a polynomial solution for this CNN model, obviously it can also be applied to the above task assignment problem. Thus, the optimization of $\mathcal{P}(\mathbb{G}, \mathbb{D}, \mathbb{S})$ is NP-hard. Here complete the proof.

Theorem 2. Given a CNN model $\mathbb{G}$ with graph structure, the problem of minimizing maximum stage execution time $\mathcal{P}(\mathbb{G}, \mathbb{D}, \mathbb{S})$ with homogeneous mobile devices under a constriction that $\mathcal{T}(\mathbb{G}, \mathbb{D}, \mathbb{S}) \leq T_{lim}$ is NP-hard.

Proof 3.2. We begin with introducing the problem of most balanced st-edge cuts (MBSTC). Given a graph G, founding an edge cut $[G, \bar{G}]$, which minimize $\max\{|G|, |\bar{G}|\}$ is NP-Hard [18]. Obviously, MBSTC is a special case of our optimization when the number of stages is 2. Thus, the problem is NP-Hard.

Given a heterogeneous edge environment, Theorem 1 shows find the optimal solution for CNNs with chain structure is NP-Hard. Theorem 2 shows that for CNNs with graph structure, even homogeneous edge environment is NP-Hard.

We summarize the result in Table 2, almost every situation is NP-Hard for optimization except chain structure model with homogeneous devices.

4 ORCHESTRATE THE MODEL STRUCTURE

In this section, we introduce our strategy to orchestrate the complex block and graph structures.

4.1 Insight

Ideally, we hope to directly divide CNN model $\mathbb{G}$ and mobile devices $\mathbb{D}$ into several stages $\mathbb{S}$. However, we can find there is no polynomial solution for $\mathbb{G}$ with block and graph structures from Table 2, neither with homogeneous nor heterogeneous environment. The only feasible situation to find the optimal strategy $\mathbb{S}$ is that the structure of model is a chain.

For block structure, a simple trade-off is to consider every block as a special layer. So that it could be optimized in polynomial time. However, this scheme introduces lots of redundant calculation inside blocks and can not be applied on these models with graph structures. Fig. 6 shows an extreme case of such a scheme. Considering a block with only two conv layers (l_a, l_b). The kernel size of l_a is 1×7 but the kernel size of l_b is 7×1. For layer l_a, according to Eq. (3) there is no redundant calculation on its width dimension since $k_a^w = 1$ (assuming the stride size is 1). Similarly, there is no redundant calculation on its length dimension for layer
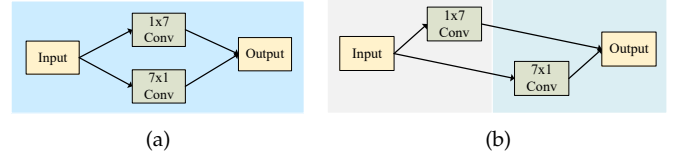

Fig. 6: A extreme case for a block with two layers. (a): consider the entire block as a special layer. (b): partition the block into more fine-grained pieces.
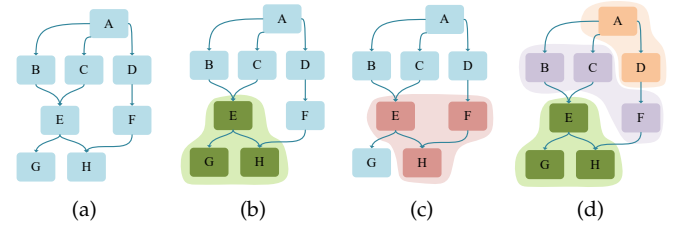

Fig. 7: The illustration of ending pieces. (a): The original graph $\mathbb{G}$. (b): $\{E, G, H\}$ is an ending piece of $\mathbb{G}$. (c): $\{E, F, H\}$ is not an ending piece of $\mathbb{G}$. (d): A partition of $\mathbb{G}$ using ending piece iteratively.

l_b. However, if we regard the block as a special whole layer, it will have redundant calculation on both width and length, as shown in Fig. 6a. But the block can be divided into two sequential *pieces*, one piece contains the input vertex and layer l_a, the other piece contains layer l_b and output vertex as shown in Fig. 6b. After this operation, there is no more redundant calculations inside the two pieces.

Here comes the insight. Given a CNN model $\mathbb{G}$, the goal is to transform it into a sequence of pieces. Since there may be some redundant calculation inside pieces, we need to minimize the redundancy inside every piece. After this operation, each piece can be regarded as a layer of original $\mathbb{G}$. Since these pieces construct a chain structure, the operation gives change for further optimization.

4.2 Partition Graph Into Pieces

We give a graph partition algorithm based on dynamic programming.

Here we will reveal the existence of the optimal substructure property of the problem of partitioning graph into pieces, which is necessary for dynamic programming. We first define the concept of *ending* piece of graph $\mathbb{G}$. Note since the piece of $\mathbb{G}$ is just a *smaller segment* defined in the previous section, we still use the notation $\mathcal{M}$ to denote these pieces.

Definition 4. An *ending piece* $\mathcal{M}_E$ is a special piece of $\mathbb{G}$ which for any edge $(u, v) \in \mathbb{G}$, if $u \in \mathcal{M}_E$, then $v \in \mathcal{M}_E$.

Fig. 7 gives an illustration of ending pieces. We plot a small graph $\mathbb{G}$ with 8 vertices in Fig. 7a and two different pieces in Fig. 7b and Fig. 7c. The green region $\{E, G, H\}$ in Fig. 7b is an ending piece of original graph $\mathbb{G}$. But the red region $\{E, F, H\}$ in Fig. 7c is not an ending piece since the edge E is a member of this piece but G is outside the red region. Graph $\mathbb{G}$ can be partitioned into pieces using

Algorithm 1 Partition graph into pieces

Require: F: map indexed with the hash $h(\mathbb{G})$ and return $F(\mathbb{G})$.
Require: R: map indexed with the hash $h(\mathbb{G})$ and return $\mathcal{M}$.
1: **function** PARTITION($\mathbb{G}, \mathcal{M}'_E$)
2:     compute the hash $h(\mathbb{G})$
3:     **if** F contains $h(\mathbb{G})$ **then**
       return $F[h(\mathbb{G})]$
4:     $min \leftarrow \infty$
5:     get $\mathcal{M} \subset h(\mathbb{G})$ which directly is connected with $\mathcal{M}'_E$
6:     **for** $\mathcal{M}_E \leftarrow DFS(\mathbb{G} - \mathcal{M})$ **do**
7:         $\mathcal{M}_E \leftarrow \mathcal{M}_E \cup \mathcal{M}$
8:         calculate the redundancy $C(\mathcal{M}_E)$
9:         $cur \leftarrow \max(\text{partition}(\mathbb{G} - \mathcal{M}_E, \mathcal{M}_E), C(\mathcal{M}_E))$
10:       **if** $min > cur$ **then**
11:          $F[h(\mathbb{G})] = cur$
12:          $R[h(\mathbb{G})] = \mathcal{M}_E$
13:          $min \leftarrow cur$
       return $F[h(\mathbb{G})]$
14: **function** OBTAIN($\mathbb{G}$)
15:     **if** $\mathbb{G} == \phi$ **then**
       return
16:     $\mathcal{M} \leftarrow R[h(\mathbb{G})]$
17:     print the piece $\mathcal{M}$
18:     obtain($\mathbb{G} - \mathcal{M}$)

---

the concept of ending piece recursively. Given a graph $\mathbb{G}$, the sketch of the procedure is to find an ending piece $\mathcal{M}_E$ and add it to the partition result (a sequence of pieces), then consider $\mathbb{G} - \mathcal{M}_E$ as a new graph and repeat the previous procedure recursively. Fig. 7d shows a partition result of graph $\mathbb{G}$.

Note partition $\mathbb{G}$ by ending piece can not guarantee that these pieces obtained are with a chain structure. If we assign vertex B in Fig. 7d from the middle piece $\{B, C, F\}$ to the first piece $\{A, D\}$, the partition result can also be obtained by the above procedure. However, such a result do not satisfy our goal since the first piece connects with two pieces at the same time. To prevent this, we add a constraint that all the vertices that are directly connect to the ending piece must belong to the ending piece in the next iteration. With this constraint, once $\{E, G, H\}$ is determined as ending piece, $\{B, C, F\}$ must belong to the same piece in the final result, which guarantees the obtained pieces a chain structure.

4.3 The Algorithm

Since our goal to minimize the redundant calculation of every piece, we need to quantify the redundant calculation cost $C(\mathcal{M})$ for a given piece $\mathcal{M}$. Assume $\mathcal{I}$ is the original input feature sizes of sources nodes in $\mathcal{M}$, and $\mathcal{I}'$ is the feature size with redundant parts that are calculated with Eq. (3). The value of $C(\mathcal{M})$ can be easily quantified by the difference of required FLOPs for the two input.

Here we give the state transfer equation for partitioning the graph structure:

$$F(\mathbb{G}) = \min_{\mathcal{M}_E \subset \mathbb{G}} \{\max\{F(\mathbb{G} - \mathcal{M}_E), C(\mathcal{M}_E)\}\}. \quad (13)$$

If $\mathbb{G}$ is partitioned into multiple pieces $\mathcal{M}$, the function $F(\mathbb{G})$ return the minimum FLOPs difference $C(\mathcal{M})$ among all partitioned pieces in $\mathbb{G}$. Algorithm 1 gives the pseudocode.

Line 2-4 checks whether $F(\mathbb{G})$ is already calculated, if true, the following computation can be skipped. Otherwise, we use a variable min to store the minimum value located in Eq. (13).

Line 5-7 adds a constraint to make sure the partitioned pieces follow a chain structure. Since the *partition* function uses recursion, the parameter $\mathcal{M}'_E$ stores the partitioned piece in its previous calculation. We use a *DFS* function to produce all the possible $\mathcal{M}_E$.

Line 8-13 is the core part of our proposed dynamic programming. It iterates all possible $\mathcal{M}_E$, and uses recursion to solve the optimization problem. Here we use a variable cur to store the best partition strategy for current $\mathcal{M}_E$. If the current strategy is better than the one we have recorded, we update the record variable min and map F and R.

Line 14-18 is the *obtain* function that receives the CNN $\mathbb{G}$ and shows the best partition strategy using the map R that is calculated in the *partition* function.

Note the *DFS* function can produce tons of available $\mathcal{M}_E$ for a given $\mathbb{G}$. Since iterating all of them leads to unfeasible complexity for optimization, we use a simple pruning strategy here. From the above discussion, it is clear that the more sequential layers we fuse, the more redundancy we get. In fact, we observe that the redundancy is intolerable when the *diameter* of $\mathcal{M}_E$ exceeds a specific number.

Definition 5. The diameter of piece $\mathcal{M}$ is the greatest distance of any vertex pair in $\mathcal{M}$.

With this observation, we limit the diameter of produced $\mathcal{M}_E$ in *DFS* function under a constant integer d. In practice, we set the value of d to 5.

5 PIPELINE COOPERATION FOR CNN INFERENCE

In this section, we present a pipeline cooperation (PICO) scheme aimed at efficiently executing CNN inference. PICO uses a heuristic algorithm based on dynamic programming to optimize the inference pipeline. We also implement an adaptive framework that automatically chooses suitable parallel scheme under dynamic workload.

5.1 Heuristic

For chain structure, although the polynomial algorithm for $\mathcal{P}(\mathbb{G}, \mathbb{D}, \mathbb{S})$ does not exist unless $P = NP$, the optimal solution can be found in polynomial time if these mobile devices are homogeneous, which leads to a heuristic two-step algorithm. We first find the optimal $\mathbb{S}^\star$ for a homogeneous cluster, then adapt the $\mathbb{S}^\star$ to a heterogeneous cluster using a greedy algorithm.

Since the CNN $\mathbb{G}$ is partitioned into L pieces, considering a specific stage $\mathcal{S} :< \mathcal{M}, \mathcal{D}, \mathcal{F} >$ that starts from i-th piece and ends at j-th piece. We can use the notation $\mathcal{S}_{i \to j}$ to represent it, so to the two notations $\mathcal{M}_{i \to j}$ and $\mathcal{D}_{i \to j}$.

5.1.1 Dynamic Programming

Based on the given cluster $\mathbb{D}$, we construct a new cluster $\mathbb{D}'$, which has the same number of devices of $\mathbb{D}$, but the computing capacity of each device is equivalent to the average of $\mathbb{D}$.

$$\vartheta(d'_k) = \frac{\sum_{d_k \in \mathbb{D}} \vartheta(d_k)}{|\mathbb{D}|} \; \forall d'_k \in \mathbb{D}', \quad |\mathbb{D}'| = |\mathbb{D}| \quad (14)$$

Algorithm 2 Dynamic programming for pipeline inference

Require: P, L: 3D arrays used to record the period and latency.
Require: S, R: 3D array used to trace the computed stage and sub-pipeline.

1: **function** DP(i, j, p, T_{lim})
2:     **if** $P[i][j][p]$ exists **then**
        return $P[i][j][p], L[i][j][p]$
3:     calculate $Ts[i][j][p]$ using (11)
4:     $P[i][j][p] \leftarrow Ts[i][j][p]$
5:     $T[i][j][p] \leftarrow Ts[i][j][p]$
6:     $S[i][j][p] \leftarrow (i, j, p)$
7:     **if** $m = 1$ or $j = i + 1$ **then**
        return $P[i][j][p], T[i][j][p]$
8:     **for** $s := i \rightarrow j - 1$ **do**
9:         **for** $m := 1 \rightarrow p - 1$ **do**
10:             calculate $Ts[s + 1][j][m]$ using (11)
11:             $T_{lim} \leftarrow T_{lim} - Ts[s + 1][j][m]$
12:             **if** $T_{lim} < 0$ **then**
13:                 **continue**
14:             $P[i][s][p-m], T[i][s][p-m] \leftarrow$ DP($i, s, p-m, T_{lim}$)
15:             **if** $T_{lim} < T[i][j][p - m]$ **then**
16:                 **continue**
17:             $period \leftarrow \max(P[i][s][p - m], Ts[s + 1][j][m])$
18:             **if** $period < P[i][j][p]$ **then**
19:                 $P[i][j][p] \leftarrow period$
20:                 $T[i][j][p] \leftarrow T[i][s][p - m] + Ts[s + 1][j][m]$
21:                 $R[i][j][p] \leftarrow (i, s, p - m)$
22:                 $S[i][j][p] \leftarrow (s + 1, j, m)$
    return $P[i][j][p], L[i][j][p]$
23: **function** BUILDSTRATEGY($(i, j, p), \mathbb{S}$)
24:     **if** $R[i][j][p]$ **then**
25:         BuildStrategy($R[i][j][p], \mathbb{S}$)
26:     calculate $\mathcal{S}_{i \rightarrow j}$ using $S[i][j][p]$
27:     $\mathbb{S} \leftarrow \mathcal{S}_{i \rightarrow j} \cup \mathbb{S}$

For any device d_k belongs to this stage, the output feature size $\mathcal{F}^k$ is equivalently partitioned. Thus, $\mathcal{F}^k$ can be determined by the size of stage. We denote $p = |\mathcal{D}_{i \rightarrow j}|$ for convenience.

The expression of stage can now be simplified as a three-element tuple (i, j, p). For the optimal pipeline $\mathbb{S}^\star$, it can now be broken into an optimal sub-pipeline consisting of pieces form 1 through s with $p - m$ mobile devices followed by a single stage with pieces $s + 1$ through j replicated over m workers. Then using the optimal sub-problem property, we can solve the optimization problem through dynamic programming:

$$P[i][j][p] = \min_{i \le s < j} \min_{1 \le m < p} \max \begin{cases} P[i][s][p - m] \\ Ts[s + 1][j][m] \end{cases} \quad (15)$$

where $P[i][s][p-m]$ is the time taken by the slowest stage of the optimal sub-pipeline between piece i and s with $p - m$ edge devices, $Ts[s + 1][j][m]$ is the time taken for a stage with model segment $\mathcal{M}_{s+1 \rightarrow j}$ with m devices. Obviously $P[1][L][D]$ is equivalent to $\mathcal{P}(\mathbb{G}, \mathbb{D}', \mathbb{S})$ in the homogeneous case. During optimization, we prune these solutions that exceed the inference limitation T_{lim}.

Algorithm 2 shows the pseudocode of our optimization algorithm which uses dynamic programming with memorization to find out the optimal parallelization strategy. Function DP computes the minimum period and records the optimal pipeline configuration in two 3D arrays R and S. The optimal parallelization strategy is built up through

Algorithm 3 Adjust stage configuration $\mathbb{S}$ for heterogeneous devices

Require: $\mathbb{S}'$: the optimal stage set for homogeneous cluster.
1: **function** ADJUSTSTAGE
2:     Initialize an empty $\mathbb{S}$
3:     Sort devices in $\mathbb{D}$ by computing capabilities $\vartheta(d_k)$
4:     **for** $d_k \in \mathbb{D}$ **do**
5:         Find the stage $\mathcal{S}'_{i \rightarrow j} \in \mathbb{S}'$ with minimum $\frac{\Theta'_{i \rightarrow j}}{|\mathcal{D}'_{i \rightarrow j}|}$
6:         Get $\mathcal{S}_{i \rightarrow j}$ from $\mathbb{S}$ or create $\mathcal{S}_{i \rightarrow j}$ with empty $\mathcal{D}_{i \rightarrow j}$
7:         $\mathcal{D}_{i \rightarrow j} \leftarrow d_k \cup \mathcal{D}_{i \rightarrow j}$
8:         Remove one device from $\mathcal{D}'_{i \rightarrow j}$
9:         **if** $|\mathcal{D}'_{i \rightarrow j}| = 0$ **then**
10:             Adjust feature partition $\mathcal{F}^k$ for every $d_k \in \mathcal{D}_{i \rightarrow j}$.
11:             $\mathbb{S} \leftarrow \mathcal{S}_{i \rightarrow j} \cup \mathbb{S}$
12:             Remove $\mathcal{S}'_{i \rightarrow j}$ from $\mathbb{S}'$
    return $\mathbb{S}$

---

function *BuildStrategy* by recursively iterating the calculated R and S, and adding the corresponding stage configuration $\mathcal{S}_{i \rightarrow j}$ to $\mathbb{S}$.

5.1.2 Adapt to the heterogeneity

We use a greedy algorithm to adapt the calculated $\mathbb{S}'$ in Algorithm 2 to the heterogeneous environment. For every stage $\mathcal{S}'_{i \rightarrow j} \in \mathbb{S}'$, we keep the model segment $\mathcal{M}_{i \rightarrow j}$ unchanged and choose a proper set of edge devices as $\mathcal{D}_{i \rightarrow j}$ from heterogeneous cluster $\mathbb{D}$. Let $\Theta_{i \rightarrow j}$ and $\Theta'_{i \rightarrow j}$ denotes the required computing resources of stage $\mathcal{S}_{i \rightarrow j}$ and $\mathcal{S}'_{i \rightarrow j}$:

$$\Theta_{i \rightarrow j} = \sum_{d_k \in \mathcal{D}_{i \rightarrow j}} \theta(\mathcal{M}_{i \rightarrow j}; \mathcal{F}^k), \quad (16)$$

We want $\Theta_{i \rightarrow j}$ to be as close to $\Theta'_{i \rightarrow j}$ as possible.

We initialize the stage set $\mathbb{S}$ with the same number of stages, each stage only the same number of workers and the same model fragment $\mathcal{S}_{i \rightarrow j}$. To achieve our goal, we sort the mobile devices by the computing capabilities $\vartheta(d_k)$ in reverse order and then iterate each device. In every iteration, we find the stage $\mathcal{S}'_{i \rightarrow j} \in \mathbb{S}'$ with maximum average computing requirement $\frac{\Theta'_{i \rightarrow j}}{|\mathcal{D}'_{i \rightarrow j}|}$. The current device d_k will be added to device set $\mathcal{D}_{i \rightarrow j}$. Once $\mathcal{D}_{i \rightarrow j}$ owns the same number of device in $\mathcal{D}'_{i \rightarrow j}$, we adjust the output feature size $\mathcal{F}^k$ for every device $d_k \in \mathcal{D}_{i \rightarrow j}$ with a *Divide And Conquer* algorithm. After this operation, we accomplish the presentation of stage $\mathcal{S}_{i \rightarrow j}$ and add it to $\mathbb{S}$. After all the iterations, we have a set of stages $\mathbb{S}$ for the heterogeneous cluster. The complete algorithm is shown in Algorithm 3.

5.1.3 Correctness

PICO can not guarantee the final configuration for the inference pipeline is optimal since the problem is NP-Hard. But we could show that both Algorithm 1 and 2 find the optimal solutions for the sub-problem they focus on.

Theorem 3. Given a CNN model $\mathbb{G}$, $F(\mathbb{G})$ in Eq. (13) returns the maximum amount of redundant FLOPs among those pieces in the optimal arrangement of $\mathbb{G}$.

Theorem 4. Assuming i and j is the start and end index of pieces that need to be deployed, $P[i][j][p]$ in Eq. (15) returns the minimal period of all possible pipeline configurations for p mobile devices.

The detailed proof can be found in the appendices.

5.2 Optimization Complexity

PICO aims to find a many-to-many mapping for various CNNs and heterogeneous mobile devices, which has been shown as NP-Hard in Section 3.3. Here we analyze the complexity of these algorithms proposed by PICO.

5.2.1 Analysis

PICO contains three novel algorithms, we analyze them one by one.

Algorithm 1 arranges $\mathbb{G}$ into sequential pieces. We first define the width of CNN to formulate the complexity of Algorithm 1:

Definition 6 (Width w of CNN). Given a CNN graph $\mathbb{G}$, w is the *width* of $\mathbb{G}$ if there are at most w neural layers in $\mathbb{G}$ such that there is no path connecting any two of them.

Since for every $\mathcal{M}$ generated in Line 6, Algorithm 1, the upper bound for every path in $\mathcal{M}$ is d since we limit the length of every path in $\mathcal{M}$ (see Section 4.3), the time complexity of Algorithm 1 can be given by Theorem 5.

Theorem 5 (Complexity of Algorithm 1). The time complexity of PICO is $O(wd(\frac{nd}{w})^w)$, where d is the upper bound for every path in $\mathcal{M}$, n is the number of vertices in $\mathbb{G}$, and w is the width of $\mathbb{G}$.

Proof 5.1. We only give a proof sketch here for ease of reading. The detailed proof is in the appendices. Line 8 in Algorithm 1 dominates the computation and the complexity for computing $C(\mathcal{M}_E)$ for every $\mathcal{M}_E$ is $O(wd)$, since there are up to wd vertices in $\mathcal{M}_E$. We need to count all the possible pairs $(\mathcal{G}, \mathcal{M}_E)$ during execution to analyze the complexity of the entire algorithm. Any directed acyclic graph $\mathbb{G}$ is also equivalent to a partially ordered set. By applying Dilworth's Theorem on $\mathbb{G}$, we can decompose $\mathbb{G}$ into w disjoint chains $\{\mathbb{V}_i\}$. Thus, any possible pair $(\mathbb{G}, \mathcal{M}_E)$ can be decomposed into a combination between $\mathcal{M}_E$ and these chains. Since $\mathcal{M}_E$ is an ending piece of $\mathbb{G}$, $\mathcal{M}_E \cap \mathbb{V}_i$ must be a suffix of $\mathbb{V}_i$. Therefore, the upper bound for the possible pair $(\mathbb{G}, \mathcal{M}_E)$ is $\Pi_{i \in \{1, \cdots, w\}} V_i d$, where V_i is the length of chain $\mathbb{V}_i$. And the maximum of $\Pi_{i \in \{1 \cdots, w\}} V_i d$ achieves when $V_i = V_j = n/w$. Thus, the number of all possible pairs $(\mathbb{G}, \mathcal{M}_E)$ is less than $(\frac{nd}{w})^w$, and the complexity of Algorithm 1 is $O(wd(\frac{nd}{w})^w)$.

Algorithm 2 generates an inference pipeline for D homogeneous mobile devices. The sub-problem of Algorithm 2 is to find the inference cost for a given stage, and the computational complexity is $O(nD)$. Assuming the Algorithm 1 partitions the CNN into L pieces, the total number of sub-problem of Algorithm 2 is $O(L^2 D)$, leading to a total time complexity of $O(nL^2 D^2)$.

Algorithm 3 is a simple greedy algorithm. The sorting operation in Line 3 has $O(D \log(D))$ complexity (e.g., quick sort). The for-loop in Line 4 repeats for D times and the line 6 has $O(\log(S))$ complexity for choosing $\mathcal{S}_{i \to j}$ from a tree set $\mathbb{S}$, where S is the size of $\mathbb{S}$. The complexity for Algorithm 3 is $O(D(\log(D) + \log(S)))$ and could be relaxed to $O(D \log(D))$ since $S \leq D$.

From the above discussion, we can deduce the complexity of PICO is $O(wd(\frac{nd}{w})^w + nL^2 D^2)$. The complexity is listed in Table 3 for summary. Note Algorithm 1 only needs
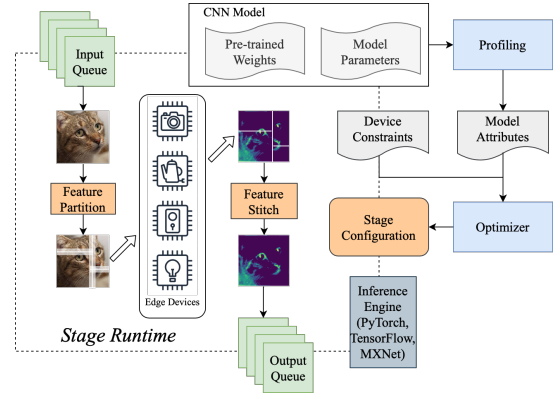

Fig. 8: The workflow of stages in an inference pipeline.

to run one time for every CNN and is not affected if the mobile environment changes.

TABLE 3: Computational complexity of PICO framework.

Algorithm 1	Algorithm 2	Algorithm 3	PICO
$O(wd(\frac{nd}{w})^w)$	$O(nL^2 D^2)$	$O(D \log(D))$	$O(wd(\frac{nd}{w})^w + nL^2 D^2)$

5.2.2 PICO in Practice

The computational cost of PICO in practice can be decomposed into two parts: the one-time cost $O(wd(\frac{nd}{w})^w)$ caused by Algorithm 1 and the ongoing cost $O(nL^2 D^2)$ caused by Algorithm 2 and 3.

The **one-time cost** caused by Algorithm 1 does not involve the specific edge environment or mobile device cluster, it can be executed on a powerful PC and the results can be directly used by mobile devices without additional processing. The **ongoing cost** caused by Algorithm 2 and 3 is lightweight and takes less than 1 second in the resource-limited Raspberry-Pi. Thus, the Algorithm 2 and 3 can be triggered if the mobile environment changes and immediately adapt to the new environment.

5.3 Implementation

The workflow of stages: We summarize the workflow of stages in Fig. 8. Each stage owns its configuration $\mathcal{S}_{i \to j}$ which is given by the previous optimization. The main thread of stage takes the feature map from the input queue, then splits it into small tiles with different sizes according to $\mathcal{F}$ and distributes them to those devices $\mathcal{D}_{i \to j}$. Once the computation finishes, the outputs of those devices are gathered and stitched together. There are two other threads responsible to put the receiving feature map into the input queue and send the output to the next stage.

Feature split and stitch: Most popular DL frameworks such as TensorFlow, PyTorch does not provide an efficient way to split feature map with overlapped parts, and using these high level provided by those frameworks to implement these operations brings intolerable latency. We accomplish the frame split and stitch operations by directly operate the frame tensor data point in the memory space through C++. In practice, after optimization, the time consumption of feature split and stitch can be ignored.

Fig. 9: The testbed in our experiment is composed of 8 Raspberry-Pi 4Bs, 2 Nvidia TX2 NXs, a Monsoon High Voltage Power Monitor (HVPM) and a Wi-Fi access point.

Fig. 10: The model structure of ResNet34 and InceptionV3.

Represent CNN into graph: We implement an automatic *GraphConvertor* module to convert a given CNN model file into a DAG. The module will record the input and output layers for every tensor during profiling. To achieve this, We modify the source file of PyTorch and add a new hook function *register_prev_forward_hook* as suggested in [19].

6 EXPERIMENT

We give the details of our evaluation bed for experiment and analyze the obtained result.

6.1 Environment Setup

Here we give the details of our evaluation setup.

Hardware: The mobile cluster for evaluating the PICO framework uses one Wi-Fi access point with 50Mbps bandwidth and 8 ARM based Raspberry-Pi 4Bs. Each Raspberry-Pi 4B has a Quad Core ARM Cortex-A73, which has 1.5 GHz max CPU frequency. It has 2 GB LPDDR2 SDRAM, and dual-band 2.4 GHz/5 GHz wireless for communication. To represent a realistic low-end mobile device cluster, we set these Raspberry-Pi 4B running with one CPU core during inference. Fig. 9 shows the test bed we used, the laptop is used to monitor and control this cluster. We limit the CPU frequency for each Raspberry-Pi using *Linux cGroup* to simulate the heterogeneous mobile cluster in the real world. The heterogeneous cluster has two Nvidia TX2 NX devices, which have a 2.2 GHz CPU frequency, and the six Raspberry-Pis that have three different CPU frequency settings: 1.5 GHz, 1.2 GHz, and 0.8 GHz, respectively.

Software: We implement PICO and other compared method using PyTorch with Gloo [20] as the communication backend. Due to differing output feature sizes on each device, we only used asynchronous P2P algorithms such as *isend()* and *irecv()* in Gloo. As for the network bandwidth, we use a method similar to MPEG-DASH [21], using the tool *ping* to send data of two different sizes and measure the response times. The rate is then determined by calculating the ratio between the difference in data size and the difference in response times.

Models overview: VGG16 [12] is a classic CNN classification model. It contains 13 conv layers, 5 pooling layers and 3 fc layers. You only look once version 2 (YOLOv2) [13] is a lightweight CNN used for real-time object detection system. It has deeper architecture compared with VGG-16. There are 23 conv and 5 pooling layers in YOLOv2, nearly twice as VGG16. Both VGG16 and YOLOv2 follow the chain structure. ResNet34 [10] and InceptionV3 [11] are two classic CNNs that use a block structure. ResNet34 uses a *skip-connection* strategy that allow the feature to skip several layers. Compared with ResNet34, InceptionV3 has more complex structure. The Inception block has multiple branches, and the conv layers also have many unbalanced (e.g., 1×7, 5×1) kernels.

Compared method: For VGG16 and YOLOv2, four different parallelization strategies are used in the evaluation: (1) Layer-wise (LW) scheme, which parallelizes the CNNs layer by layer; (2) Early-fused-layer (EFL) scheme, an extension to the implementation of DeepThings [5], which fuses and parallelizes the first few conv layers, then executes the rest layers in a single device; (3) Optimal Fused-layer (OFL) scheme, which selectively fuses convolution layers at different parts of a model; (4) CoEdge (CE) [22], the state-of-the-art parallelization scheme which extends the layer-wise scheme to heterogeneous environment and reduce the communication overhead by sorting devices into a list and limiting the communication to the neighbors for each device. Moreover, CoEdge uses a dynamic number of devices to process different layers to further reduces the impact of communication overhead instead of using all mobile devices. (5) Pipeline Cooperation (PICO) scheme, which is proposed in this paper. For ResNet34 and YOLOv2, we compare two different graph partition strategies: (1) Consider each block as a piece, which is used in [6], [17]. (2) Partition the entire graph into multiple pieces with suitable granularity, which is proposed in this paper.

6.2 CNN Graph Partition

Here we present some experimental results of our proposed graph partition algorithm.
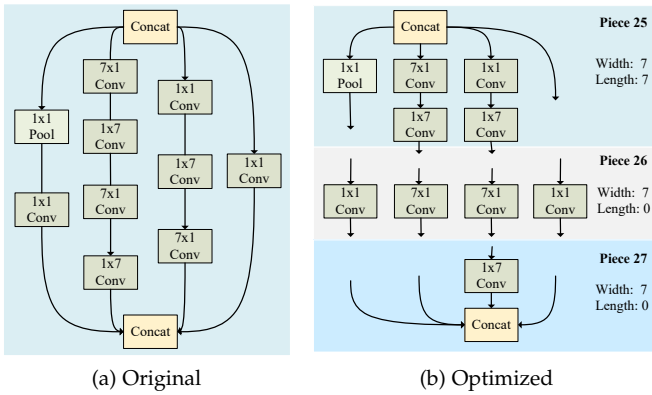
Fig. 11: An illustration of our graph partition algorithm. (a) Part of InceptionV3 model (InceptionC block). (b) The obtained pieces after optimization.
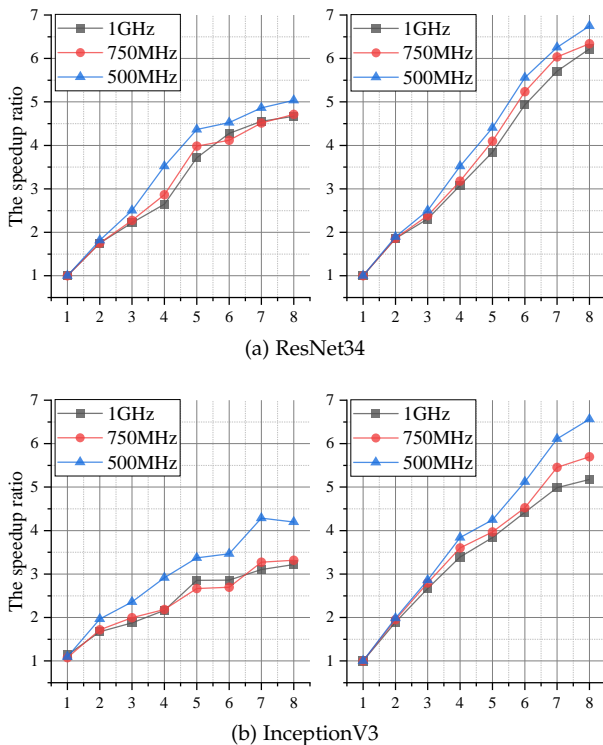
Model	n	w	$wd(\frac{nd}{w})^w$	Execution	Pieces
VGG16	19	1	4.7×10^2	0.10s	19
SqueezeNet	30	2	5.6×10^4	0.14s	29
ResNet34	38	2	9.0×10^4	0.28s	21
MobileNetV3	96	3	6.1×10^7	0.79s	31
InceptionV3	99	4	4.6×10^9	3.01s	38
NASNetL	570	8	1.1×10^{22}	> 5h	NaN
NASNetL-P	75×8	8	9.3×10^{14}	1.9h	34

TABLE 4: The performance of Algorithm 1 for various CNNs. NASNetL-P denotes the strategy which roughly partitions it into 8 parts.

shown in Fig. 11b (The full partition result is attached in the supplemental material). The entire InceptionV3 model is composed of 40 pieces, but due to the size of model, we can not plot the entire model here. The complete partition result is shown in the supplemental materials. These pieces have much smaller redundant calculation. Piece 25 has 7 pixel length redundancy, and Piece 26 and Piece 27 only have redundancy on only one dimension. Compared with Fig. 11a, the redundant calculation during inference can be significantly reduced. In addition, since we break block into pieces, the later optimization can make more fine-grained optimization.

6.2.2 Speedup After Partition

We can adapt PICO to those CNNs with non-chain structure by applying our graph partition algorithm at first. Here we compare the speedup ratio for ResNet34 and InceptionV3. Fig. 10 shows the structures of the two model, obviously they are constructed with the block structure. According to the figure, we can find the Inception block in InceptionV3 is more complex than Residual block used in ResNet34. Fig. 12 plots the speedup ratio under different CPU frequencies for ResNet34 and InceptionV3 with different strategy. The figures on the left part fuse the entire block into a whole, and the figures on the right show the results that adopt our graph partition algorithm, When executing CNN inference with 8 devices, PICO can achieve $6.8\times$ speedup for ResNet34 and $6.5\times$ for InceptionV3 after partitioning the CNN model. The speedup effect is more obvious with low CPU frequency since the limitation of computing resource is relieved when the number of mobile devices increases. As for the strategy of fusing the entire block, it achieves up to $5\times$ speedup for ResNet34, but only $4\times$ for InceptionV3 when the CPU frequency is low. We think it is caused by the difference in the number of layers in Residual and Inception blocks. Since the Inception block contains more layers than Residual block, the optimal model partition is more likely to exist within blocks.

Fig. 12: The speedup ratio for ResNet34 and InceptionV3. The left part shows the result by treating the entire block as a whole, and the right part uses graph partition algorithm.

6.2.1 Partitioned Pieces

Fig. 11 shows part of the partition result of InceptionV3 model. Fig. 11a plots the InceptionC block, which consists 4 branches and 10 neural layers. We can find if we consider the entire block as a layer [6], lots of redundant calculations will be introduced since there are many *unbalanced* conv kernels (e.g., 1×7 and 7×1). We can use Eq. (3) and Eq. (5) to quantify the redundant calculation. If we fuse the entire block into one piece (used in [6]), the devices have to introduce 13 pixel length on both the width and height dimensions. After running the partition algorithm, the block is split into three pieces (Piece 25, Piece 26, and Piece 27) as

6.2.3 Optimization Complexity

Algorithm 1 has $O(wd(\frac{nd}{w})^w)$ complexity to optimize the given CNN, as we analyzed in Section 5.2. Here we run Algorithm 1 on many popular CNNs on a PC equipped with Intel Core i9-10940X to give a comprehensive evaluation of its performance. The number of layers n, the width w and the upper bound $wd(\frac{nd}{w})^w$ for every tested CNN and the execution time are listed in Table 4. Note n only counts *conv* and *pool* layers, since other layers such as *BN* and *ReLU* do
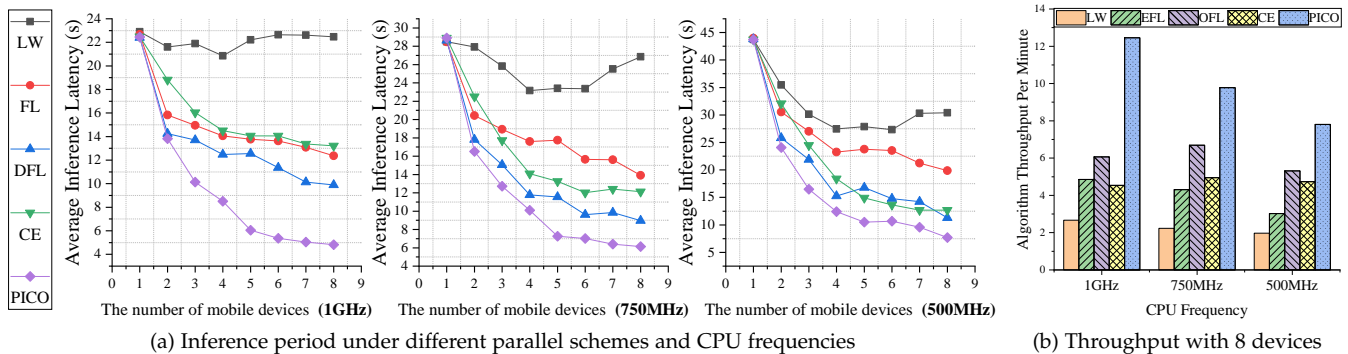
(a) Inference period under different parallel schemes and CPU frequencies

(b) Throughput with 8 devices

Fig. 13: The cluster capacity when executing VGG16.

(a) Inference period under different parallel schemes and CPU frequencies
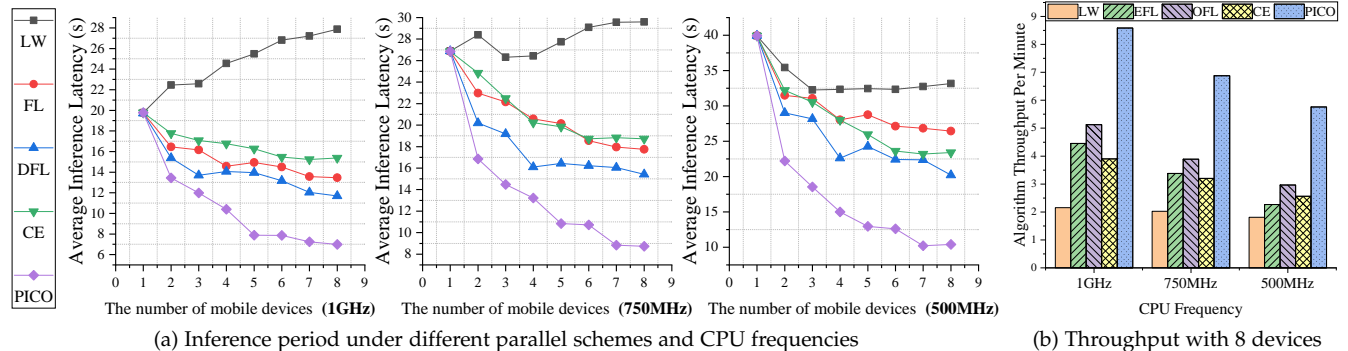
(b) Throughput with 8 devices
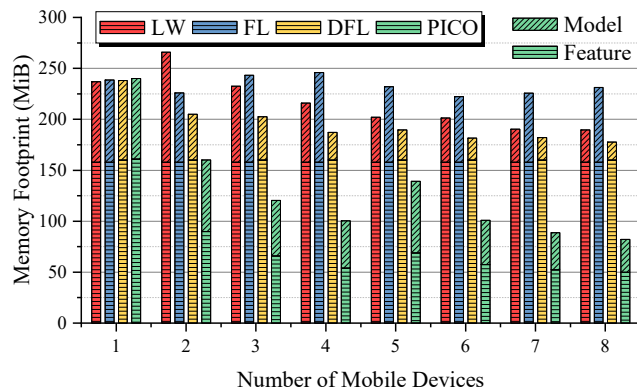
Fig. 14: The cluster capacity when executing YOLOv2.

Fig. 15: The memory footprints of different algorithms.

not change the output feature shape and require negligible computing resources. Additionally, the last column shows the number of pieces after optimization.

Many real-world CNNs are deep but narrow, which means they have small width w. Algorithm 1 is efficient and could be executed in less than one or several seconds for these models. We also add the NASNet-A-Large [23] (NASNetL) model to evaluate Algorithm 1 in an extremely complex case. NASNetL is constructed through *neural architecture search* technology. NASNetL is much broader ($w = 8$) and contains much more layers ($n = 570$) compared with the hand-designed models ($w \leq 4$ and $n \leq 100$). It is rare to deploy such a large-scale model on mobile devices. The

trade-off that considers a block as a layer [6], [17] has no effect since there is no block in NasNetL.

Directly applying PICO to NASNetL takes nearly infinite time to produce the final output considering the time complexity (1.1×10^{22}). We successfully adapt PICO to NASNetL using *divide-and-conquer*. Assume Algorithm 1 divides a model into L pieces, if we fuse the $L/2$ pieces from the input position into a smaller model and apply Algorithm 1 to it, then the smaller model must be divided into the same $L/2$ pieces as the original model (the property of dynamic programming). Inspired by this property, We cut a small part from the beginning of NASNetL, and apply Algorithm 1 on the smaller model to obtain several pieces. Only these pieces away from the cut line will be kept to make sure these pieces from different small model are still sequential. Then we apply the same strategy to the rest model until all the smaller models are partitioned into pieces. The last line in Table 4 shows the performance of the divide-and-conquer strategy. NasNetL is tackled with 8 parts and PICO produces the result in two hours. Since Algorithm 1 only needs to run once for every CNN regardless of specific mobile environment (see Section 5.2), the optimization cost is acceptable.

6.3 Pipeline Performance

We evaluate our proposed pipeline cooperation scheme with 2-8 Raspberry-Pi devices and measure some important metrics.

TABLE 5: The utilization, redundancy ratios and memory footnotes of heterogeneous mobile devices.

Model	Attributes	Methods	Type	Devices								Average
				NX@2.2	NX@2.2	Rpi@1.5	Rpi@1.5	Rpi@1.2	Rpi@1.2	Rpi@0.8	Rpi@0.8	
VGG16	Layers: 13 conv + 5 pool Input size: 244 × 244	CE	Utili.	80.87%	82.13%	69.37%	59.97%	57.91%	36.56%	23.11%	17.33%	53.40%
			Redu.	2.02%	1.93%	1.29%	2.06%	1.30%	1.41%	1.32%	0.77%	1.51%
			Mem.	195.0 M	183.0 M	162.0 M	158.0 M	147.0 M	149.0 M	134.0 M	137.0 M	158.1 M
		EFL	Utili.	32.43%	39.79%	72.58%	75.08%	94.23%	96.77%	64.09%	64.16%	67.39%
			Redu.	11.02%	11.60%	19.08%	19.83%	18.58%	19.22%	12.78%	13.42%	15.69%
			Mem.	142.0 M	147.0 M	169.0 M	179.0 M	173.0 M	183.0 M	151.0 M	165.0 M	163.6 M
		OFL	Utili.	38.90%	40.19%	60.87%	61.79%	85.34%	94.15%	76.54%	80.46%	67.28%
			Redu.	7.45%	7.67%	11.12%	11.39%	10.33%	10.53%	8.15%	8.31%	9.36%
			Mem.	149.0 M	149.0 M	158.0 M	159.0 M	154.0 M	155.0 M	151.0 M	152.0 M	153.4 M
		PICO	Utili.	91.21%	93.28%	83.12%	79.40%	47.63%	66.26%	68.17%	90.15%	77.40%
			Redu.	11.08%	10.97%	5.82%	3.83%	6.93%	5.55%	0.00%	3.83%	6.00 %
			Mem.	189.0 M	144.0 M	121.0 M	103.0 M	92.0 M	121.0 M	115.0 M	111.0 M	124.5 M
YOLOv2	Layers: 23 conv + 5 pool Input size: 448 × 448	CE	Utili.	76.85%	75.46%	65.81%	66.94%	46.32%	46.77%	22.49%	20.21%	52.61%
			Redu.	0.82%	0.76%	0.87%	0.83%	0.79%	0.71%	0.68%	0.61%	0.75%
			Mem.	265.0 M	260.0 M	255.0 M	246.0 M	245.0 M	240.0 M	235.0 M	239.0 M	248.1 M
		EFL	Utili.	37.85%	35.64%	67.24%	67.61%	96.01%	95.28%	75.87%	72.81%	68.54%
			Redu.	27.09%	27.09%	45.08%	45.08%	44.68%	44.68%	29.29%	29.29%	36.54%
			Mem.	189.0 M	178.0 M	208.0 M	208.0 M	208.0 M	207.0 M	178.0 M	178.0 M	194.3 M
		EFL	Utili.	39.28%	37.03%	69.47%	68.92%	97.02%	95.99%	77.61%	73.94%	69.91%
			Redu.	25.98%	25.98%	44.51%	44.51%	44.86%	44.86%	28.12%	28.12%	35.86%
			Mem.	193.0 M	182.0 M	212.0 M	212.0 M	212.0 M	211.0 M	182.0 M	182.0 M	198.3 M
		PICO	Utili.	89.37%	97.91%	89.96%	97.85%	89.44%	99.40%	91.89%	89.03%	93.11%
			Redu.	6.95%	2.27%	1.25%	9.18%	9.18%	5.89%	6.13%	5.05%	5.73%
			Mem.	188.0 M	135.0 M	108.0 M	116.0 M	113.0 M	122.0 M	159.0 M	157.0 M	137.3 M

6.3.1 Maximum Throughput

Fig. 13 and Fig. 14 plot the cluster capacity when executing VGG16 and YOLOv2 with different parallel schemes. The first three figures plot the inference period with different parallel schemes and CPU frequencies. The last figure plots the accomplished inference task per minute with 8 devices. It represents the throughput of different parallel schemes. PICO has the best performance as expected, since our optimization goal is to reduce the redundant computation and achieve minimum pipeline period. When the number of devices increases, the throughput of different strategies also improve except the executing YOLOv2 using LW with 1GHz CPU core. YOLOv2 has nearly twice number of layers compared with VGG16, which brings more communication overhead for Layer-wise strategy. Through CE also executes CNN layer by layer, CE uses a dynamic number of working devices during inference and reduces the traffic volume by only synchronizing overlapped features. Therefore, CE outperforms LW. When the computing resource is rich (1GHz), the gain brought by the increasing number of devices is offset by communication overhead. EFL and OFL fuse multiple layers into one model segment, and do not require communication among devices when they are executing one segment, thus the communication overhead is reduced. Since OFL optimizes the configuration of fused layers, it outperforms EFL which simply fuses the very early layers. However, when the number of devices is bigger than a certain number (4 for example), the improvement is very tiny due to the additional computation CPU redundancy.

6.3.2 Memory Footprint

Memory footprint is another important metric during inference. The inference latency will quickly grow when the required memory exceeds the onboard memory of the device, since the device has to use swap memory [17]. We use a python script to sample memory footprint from */proc/pid/status* for each inference process. Fig. 15 plots the average memory footprint of different algorithms. Here we ignore the performance of CE, since LW and CE have very similar performance when devices are homogeneous. According to our previous discussion, the memory footprint can be divided into two more fine-grained parts. The *Model* and *Feature* denote how much the model parameters and intermediate features take part in the memory footprint. We can find the memory footprint decreases as the number of mobile devices increases. Since LW, FL, DFL only split features, the whole model needs to be replicated on all mobile devices. This approach leads to the result that they can only decrease the memory footprint caused by intermediate features. Meanwhile, PICO distributes both models and features, thus PICO reduces the memory footprint significantly.

6.4 Impact of Heterogeneity

Here we evaluate the impact of heterogeneity on different parallel schemes. We monitor the CPU usage during inference on the heterogeneous mobile cluster and record the average computing resource utilization ratio (Utili.) for different parallel schemes. We also calculate the redundancy ratio (Redu.) and their memory footprint (Mem.) on every device during computation. The result is presented in Table 5. We remove LW scheme due to its bad performance in heterogeneous environment.

6.4.1 Load Balancing

The CPU utilization rates of these devices show the workload among these devices. PICO and CE will adjust the
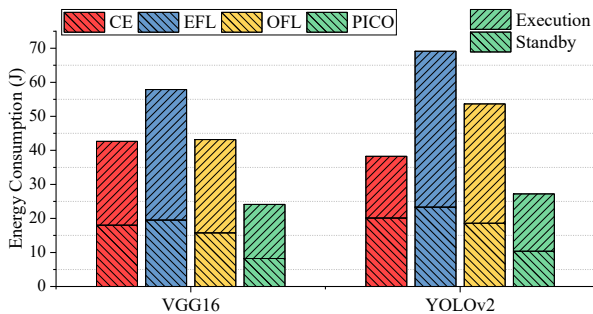
Fig. 16: The average energy consumption for every inference task with heterogeneous devices.

TABLE 6: Optimization time with graph-like CNN.

Branches, Layers, Devices	PICO	BFS (Optimal)
(2, 4, 4)	< 1s	1.58s
(2, 8, 6)	< 1s	18.23s
(3, 12, 4)	< 1s	11.96m
(3, 12, 6)	< 1s	45.24m
(3, 12, 8)	< 1s	> 1s
(4, 20, 4)	< 1s	> 1h
(4, 20, 6)	< 1s	> 1h

TABLE 7: Optimization time with heterogeneous devices.

Layers, Devices	PICO	BFS (Optimal)
(4, 4)	< 1s	< 1s
(8, 4)	< 1s	1.62s
(12, 4)	< 1s	3.84s
(16, 4)	< 1s	11.27s
(8, 6)	< 1s	4.35m
(10, 6)	< 1s	12.28m
(12, 6)	< 1s	> 1h
(8, 8)	< 1s	> 1h

feature partition size according to the specific devices. Thus, the workload of PICO and CE is better than EFL and OFL. We find both PICO and CE impose more percentages of workloads on these devices with higher CPU frequency (1.2 GHz). Take the CE as an example, the CPU utilization rate is up to 82.61% for the fastest devices, but drifts down 22.64% for the slowest devices when running VGG16. The reason is that CE uses a dynamic number of devices to process each layer. When the feature map is wide (e.g., 224 x 224), CE may use all devices to accelerate the execution. When the feature map is small (e.g., 7 x 7), CE may place all the workload on one powerful device to avoid redundant computation and communication. However, the computing resources of these slower devices are wasted. On the contrary, PICO can fully utilize the computing resources, thus having a better performance on load balancing.

6.4.2 Computation Efficiency

Because the input feature maps of different devices overlap with each other, the redundant computation can lead to inefficient performance. CE has the minimum average redundant computation, since CE synchronizes the feature map for every layer. But the frequent communication leads to low resource utilization and high inference latency. Fusing layers and executing them together can keep the devices busy, but will increase redundant computation. Especially for the EFL which has 46.54% percent redundancies executing YOLOv2. OFL uses dynamic programming to find a balance between communication and computation, but the redundancy ratio (12.08%) is still higher than PICO (7.64%) as PICO uses a subset of mobile devices instead of the entire cluster.

6.4.3 Energy Consumption

We measure the energy consumption for every inference task, the result is shown in Fig. 16. The energy consumption is composed of the inference execution and standby power consumption. EFL consumes the most energy, since EFL has the highest redundant computation compared with other schemes. Moreover, the redundant computation does not accelerate the inference, thus EFL also has high standby power consumption. OFL has a lower energy consumption compared with EFL since OFL reduces the redundant computation by synchronizing feature map periodically. CE executes the CNN layer by layer and has the lowest redundancy among all the schemes. However, the standby power consumption is the majority of energy consumption,

because CE has a long inference latency, especially executing YOLOv2. On the contrary, PICO has the lowest standby power consumption during inference task, since PICO can maximize the throughput during inference. Through PICO has more redundant computation compared with CE, the overall energy consumption is still lower than CE.

6.5 Comparing With Optimal Configuration

Because it is NP-Hard to find the best many-to-many mapping for graph-like CNN and heterogeneous devices, PICO can not guarantee finding the optimal inference pipeline configuration. Thus, we compare PICO with the optimal pipeline to further evaluate the performance. The optimal pipeline is obtained through a broad first search (BFS). We compare the optimization time for producing the pipeline configuration and the resource utilization of every mobile device during runtime.

6.5.1 Methodology

The main problem for the comparison is the possible solution space for BFS is over-complex. According to Table 2, finding the best many-to-many mapping for both chain-like CNN, heterogeneous devices and graph-like CNN, homogeneous devices are NP-Hard. But BFS tries to find the best many-to-many mapping for graph-like CNN and heterogeneous devices. We test the BFS with CNNs on 4-8 Raspberry-Pi devices, but all of them fail to produce the final output after several hours on a powerful PC. Therefore, we compare the performance of PICO and BFS from two sides. On the one side, (1) we compare PICO and BFS with graph-like CNN and homogeneous devices. On the other side, (2) we compare PICO and BFS with chain-like CNN and heterogeneous devices. Table 6 and 7 show the optimization overhead of PICO and BFS. Fig. 17 and 18 give the runtime performance.

6.5.2 Optimization Time

For all the situations listed in Table 6 and 7, PICO could accomplish the optimization within 1 second, But BFS requires much more time to give the output even on small
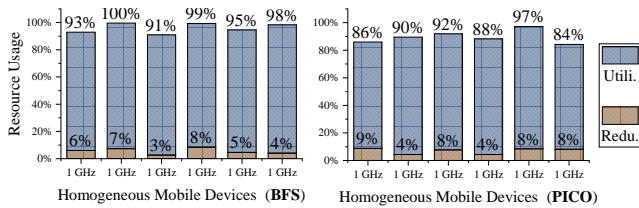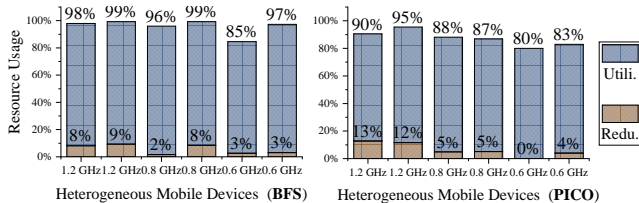
Fig. 17: Runtime performance with graph-like CNN.

Fig. 18: Runtime performance with heterogeneous devices.

scale problems. The optimization time dramatically grows on larger problems and BFS fails to finish the calculation on both sides. Moreover, these problems that BFS fails to solve are much easier (either chain-like CNN or homogeneous devices) than those that PICO has solved in the paper. Thus, BFS is not applicable in practice.

Another observation from Table 6 and 7 is that the changing of different parameters (branches, layers, devices) has different impacts on the optimization time. When the CNN is a graph and devices are homogeneous, increasing the number of layers has more impact than devices. Take the Table 6 as an example, row 3 and row 4 show that the optimization time increases from 11.96 minutes to 45.24 minutes when the number of devices increases from 4 to 6 (3.78×). But row 2 and row 3 show that the optimization time increases from 18.23 seconds to 11.96 minutes when the number of layers increases from 8 to 12 (39.36×) through the number of homogeneous devices decreases. On the contrary, increasing the number of devices has more impact when the CNN is a chain and devices are heterogeneous, as shown in Table 7. The optimization time increases from 1.62 seconds to 3.84 seconds (2.37×) when the number of layers increases from 4 to 8 (row 1 and row 2), but it increases from 1, 62 seconds to 4.35 minutes (161.11×) when the number of devices increases from 4 to 6 (row 2 and row 5). These two observations reveal the complexity of the many-to-many mapping when the CNN is a graph and devices are heterogeneous from sides.

6.5.3 Runtime Performance

We compare the runtime performance of PICO and BFS by plotting the computing resources utilization rate for each device. The result is plotted in Fig. 17 and 18. We also analyze the redundant computation during inference since high utilization rate does not lead to good performance [6].

Fig. 17 shows the runtime performance for a graph-like CNN and 6 homogeneous devices (1 GHz CPU frequency). The graph-like CNN used in the comparison contains 3 branches and 12 layers and is also used in row 4, Table 6. The optimal configuration found by BFS achieves 95% resource utilization rate. Meanwhile, the configuration found

by PICO has the similar performance (around 90%). The redundant computation of BFS is lower than PICO, but all redundant computation keep at a low level for both BFS and PICO. The performance for chain-like CNN and heterogeneous devices is plotted in Fig. 18. The CNN contains 10 layers and these devices have different computing resources, as shown on the x-axis (1.2 GHz, 0.8 GHz and 0.6 GHz). Similar to Fig 17, the optimal configuration (BFS) achieves great performance on these devices (up to 99%) except one (85%). As for PICO, the configuration places more workload of the inference to these devices who own rich computing resources, thus the resource utilization of them is similar to BFS (90% and 95% for the fastest devices). The average performance of the other devices is around 84.5%. Since PICO greatly reduces the computation complexity according to previous analysis, the performance of PICO is acceptable for most real world applications.

7 RELATED WORK

Along with the problem of enabling DNN-based intelligent applications, previous researches can be divided into two categories.

7.1 Inference Offloading

Due to the limited up-link of mobile devices, traditional way of uploading captured data to the cloud server is time-consuming [24], [25]. Researchers focus on offloading the computation of early layers to mobile devices (*Inference offloading*). To minimize the inference latency, Neurosurgeon [26] proposed to partition model between cloud server and mobile device according to the network situation. But [26] can only handle models with the chain structure. DADS [27] proposed a novel algorithm to partition DNN with graph structure using a min-cut algorithm. QDMP [28] noticed that directly applying min-cut on the entire graph is time-consuming. Based on the block structure, [28] proposed a divide-and-conquer algorithm to find the min-cut, which achieves a nearly linear complexity in their experiments. Meanwhile, Branchynet [29] propose *early exit* mechanism by adding exit layers at the midden of DNN. This mechanism enables mobile device not feature map to cloud server if the local accuracy already reaches a certain value. Considering the situation when server does not have the corresponding model, IoNN [30] an incremental offloading technology that significantly improves the inference performance.

7.2 Cooperative Inference

Recently researchers began to turn their attentions on executing inference completely at the edge with multiple mobile devices [4], [5], [6], [17], [22], [31], [32], [33], [34].

MoDNN [4] is the first work in this field. MoDNN equally partitions the out feature map for every layer and distributes these feature maps to homogeneous devices. In their following-up work MeDNN [31], they use an adaptive partition method for the heterogeneous devices. Both MoDNN and MeDNN need a master device to gather the entire output of every device for every layer. CoEdge [22] reduces the communication overhead by only sending the

overlapped feature map to the neighbors of devices. CoEdge also dynamically adjusts the number of working devices during inference to find the balance between communication and computation. EdgeFlow [33] introduces a forwarding table to overlap the communication with computation for CNNs with complex structures. The devices can execute one layer and receive the feature map required by other layers at the same time. All these works [4], [22], [31], [33] require devices to communicate with each other for every layer. However, the wireless environment can lead to considerable communication overhead using these works.

Deepthings [5] proposed to fuse the layers in the early stage of CNN to avoid communication during inference. But fusing layer increases overlapped feature maps among devices and harms the inference efficiency. DistrEdge [34] trains a deep reinforcement learning model to distribute the inference workload for heterogeneous devices. AOFL [6] uses a dynamic programming to find a trade-off between communication and computation. Devices need to synchronize the feature map after several layers using AOFL. DeepSlicing [17] propose a runtime scheduler to distribute the workload for heterogeneous devices. Both AOFL and DeepSlicing partition the CNN at the block level. Moreover, all these works [5], [6], [17] are at a loss for what to do when meeting some extremely complex CNN [14]. On the contrary, PICO breaks the block into smaller pieces to avoid additional redundant computation.

8 CONCLUSION AND FURTHER RESEARCH

In this paper, we propose a pipeline cooperation scheme (PICO) for efficiently executing inference with versatile CNN models and diverse mobile devices. This scheme improves the inference efficiency by reducing the redundant calculation. We first analyze the problem of partitioning CNNs and mobile devices into an inference pipeline. Using the analysis result, PICO uses a two-step strategy to build the pipeline. First, we orchestrate the graph structure of the given CNN into a sequence of pieces. Then we divide these pieces and devices into several stages. The input data is fed into the first stage and the inference result is produced at the last stage. These stages compose an inference pipeline. We adjust the partition size of features among devices according to their computing resources. The execution time of each stage is optimized to be as close as possible to gain maximum throughput. In our experiment with 8 Raspberry-Pi devices, the throughput can be improved by $1.8 \sim 6.8\times$ under various settings.

PICO has demonstrated strong performance across a range of heterogeneous clusters by adjusting the partitioned feature size for each device to accommodate varying computation capabilities. However, this approach is limited in addressing device-level imbalances within a given stage and is unable to address imbalances at the stage-level. This can result in failure if the computation capabilities of the devices are extremely varied. To address these challenges, we are actively pursuing the development of a novel algorithm that can better balance the workload across different stages. This is a critical area of focus for our ongoing research efforts.

REFERENCES

[1] G. Kour and R. Saabne, "Real-time segmentation of on-line handwritten arabic script," in *Frontiers in Handwriting Recognition (ICFHR)*. IEEE, 2014.

[2] ——, "Fast classification of handwritten on-line arabic characters," in *Soft Computing and Pattern Recognition (SoCPaR)*. IEEE, 2014.

[3] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, 2019.

[4] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017.

[5] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[6] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, "Adaptive parallel execution of deep neural networks on heterogeneous edge devices," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019.

[7] J. Li, Q. Qi, J. Wang, C. Ge, Y. Li, Z. Yue, and H. Sun, "Oicsr: Out-in-channel sparsity regularization for compact deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[8] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019.

[9] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE conference on computer vision and pattern recognition (CVPR)*, 2016.

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

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

[13] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[14] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.

[15] A. Benoit and Y. Robert, "Mapping pipeline skeletons onto heterogeneous platforms," *Journal of Parallel and Distributed Computing*, 2008.

[16] ——, "Complexity results for throughput and latency optimization of replicated and data-parallel workflows," *Algorithmica*, 2010.

[17] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu, "Deepslicing: collaborative and adaptive cnn inference with low latency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2175–2187, 2021.

[18] P. Bonsma, "Most balanced minimum cuts," *Discrete Applied Mathematics*, vol. 158, no. 4, pp. 261–276, 2010.

[19] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, G. R. Ganger, and P. B. Gibbons, "Pipedream: Pipeline parallelism for dnn training," in *Proceedings of the 1st Conference on Systems and Machine Learning (SysML)*, 2018.

[20] "Gloo: Collective communications library with various primitives for multi-machine training," https://github.com/facebookincubator/gloo.

[21] T. Stockhammer, "Dynamic adaptive streaming over http– standards and design principles," in *Proceedings of the second annual ACM conference on Multimedia systems*, 2011, pp. 133–144.

[22] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, "Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 595–608, 2020.

[23] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.

[24] J. H. Ko, T. Na, M. F. Amir, and S. Mukhopadhyay, "Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms," in *IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*. IEEE, 2018.

[25] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu, "Jalad: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution," in *International Conference on Parallel and Distributed Systems (ICPADS)*, 2018.

[26] 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*, 2017.

[27] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive dnn surgery for inference acceleration on the edge," in *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE.

[28] 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*, 2020.

[29] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *International Conference on Pattern Recognition (ICPR)*, 2016.

[30] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "Ionn: Incremental offloading of neural network computations from mobile devices to edge servers," in *ACM Symposium on Cloud Computing*, 2018.

[31] J. Mao, Z. Yang, W. Wen, C. Wu, L. Song, K. W. Nixon, X. Chen, H. Li, and Y. Chen, "Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017.

[32] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Towards collaborative inferencing of deep neural networks on internet of things devices," *IEEE Internet of Things Journal*, 2020.

[33] C. Hu and B. Li, "Distributed inference with deep learning models across heterogeneous edge devices," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 330–339.

[34] X. Hou, Y. Guan, T. Han, and N. Zhang, "Distredge: Speeding up convolutional neural network inference on distributed edge devices," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 1097–1107.

Zikang Xu is an undergraduate student majoring in Computer Science and Technology at Beijing University of Posts and Telecommunications. He obtained a postgraduate recommendation of State Key Laboratory of Networking and Switching Technology at Beijing University of Posts and Telecommunications. His research interests span broad aspects of machine learning, edge/cloud computing and distributed computing.

Qi Qi obtained her PhD degree from Beijing University of Posts and Telecommunications in 2010. Now, she is an associate professor of State Key Laboratory of Networking and Switching Technology at Beijing University of Posts and Telecommunications. She has published more than 30 papers in international journal, and obtained two National Natural Science Foundations of China. Her research interests include edge computing, mobile cloud computing, Internet of Things, ubiquitous services, deep learning, and deep reinforcement learning.
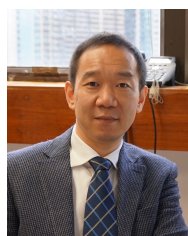

Jingyu Wang obtained his PhD degree from Beijing University of Posts and Telecommunications in 2008. He is currently a professor of State Key Laboratory of Networking and Switching Technology at Beijing University of Posts and Telecommunications. He has published more than 100 papers in international journal, including IEEE CMag, TVT, ISJ, TSC, TMM, TCC, IoT, TWC, and so on. His research interests span broad aspects of SDN/NFV, edge/cloud computing, IoV/IoT, big data processing and transmission, intelligent networks, and traffic engineering.

Haifeng Sun obtained his PhD degree from Beijing University of Posts and Telecommunications in 2017. He is currently a lecture of State Key Laboratory of Networking and Switching Technology at Beijing University of Posts and Telecommunications. His research interests span broad aspects of AI, NLP, big data analysis, object detection, deep learning, deep reinforcement learning, SDN, processing.

Jianxin Liao obtained his Ph.D degree at University of Electronics Science and Technology of China in 1996. He is currently the dean of Network Intelligence Research Center and the full professor of State Key laboratory of Networking and Switching Technology in Beijing University of Posts and Telecommunications. He has published hundreds of research papers and several books. His main research interests include cloud computing, mobile intelligent network, service network intelligent, networking architectures and protocols, and multimedia communication.

Xiang Yang received the B.E. degree in computer science and technology from Beijing University of Posts and Telecommunications, Beijing, China, in 2019. He is currently a PhD candidate of State Key Laboratory of Networking and Switching Technology at Beijing University of Posts and Telecommunications. His research interests span broad aspects of machine learning, distributed computing, edge/cloud computing and deep learning.

Song Guo is a Full Professor and Associate Head (Research & Development) in the Department of Computing at The Hong Kong Polytechnic University. His research interests are mainly in edge AI, big data and machine learning, mobile computing, and distributed systems. He has served on IEEE Fellow Evaluation Committees for both CS and ComSoc, and been named on editorial board of a number of prestigious international journals like IEEE TC, IEEE TPDS, IEEE TCC, IEEE TETC, ACM CSUR, etc.