2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS) | 978-1-6654-4513-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICDCS51616.2021.00011

Towards Efficient Inference: Adaptively Cooperate in Heterogeneous IoT Edge Cluster

Xiang Yang*, Qi Qi*, Jingyu Wang*, Song Guo†, Jianxin Liao*
* *Beijing University of Posts and Telecommunications*
Email: {yangxiang, qiqi8266, wangjingyu}@bupt.edu.cn, jxlbupt@gmail.com
† *Hong Kong Polytechnic University*
Email: cssongguo@comp.polyu.edu.hk

***Abstract*—New applications such as smart homes, autonomous vehicles are leading an increasing research topic of convolutional neural network (CNN) based inference on IoT edge devices. Unfortunately, this scenario meets a huge roadblock caused by the limited computing resources owned by these devices. One popular solution is to execute inference on an edge cluster with parallelization schemes instead of on a single device. However, the heterogeneous edge devices and varied neural layers bring challenges to this process.**

In this paper, we propose a pipelined cooperation scheme (PICO) to efficiently execute CNN inference for edge devices. Our goal is to maximize throughput by reducing redundant computing meanwhile to keep the inference latency under a certain value. PICO divides the neural layers and edge 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. The execution time of each stage is optimized to approach the maximum throughput as close as possible. We also implement an adaptive framework to choose the best inference scheme under different workloads. In our experiment with 8 RaspberryPi devices, the average inference latency can be reduced by $1.7 \sim 6.5\times$ under different workloads, and the throughput can be improved by $1.8 \sim 6.2\times$ under various network settings.

***Keywords*-Edge Computing, Pipelined Inference, Model Deployment**

I. Introduction

The Internet of Things (IoT) with a huge number of sensing devices provides a massive amount of data (such as images, videos). Meanwhile, pre-trained convolution neural network (CNN) models become pervasive tools to make smart decisions using these data (*CNN inference*). Embedding CNN model with IoT devices provides an opportunity for frontier scenarios to become reality, such as smart home, intelligent factory, and even automatic driving [1], [2].

But new challenges arise for these scenarios. First, the current network is not prepared for the massive data collected by IoT devices. For example, an autopilot camera could capture more than 700 MB video record every second [3], and uploading such a large amount of data to the datacenter will bring unacceptable inference latency and create tremendous pressure on the network. Second, CNN inference is restricted by the resource-limited IoT devices. The computing capabilities of IoT devices are much limited compared with traditional datacenters. Executing CNN inference locally requires large computational resources and memory footprints that are usually not available in a single IoT device [4]. Third, uploading information from edge to cloud brings concern about privacy. In most cases, the inference is executed entirely within the cloud or through a hybrid combination of edge and cloud computing [4], [5]. However, users are resistant to upload information from their daily used devices to the cloud, neither the original data nor the pre-processed intermediate results at the edge [3].

As a consequence, efforts have been made to distribute the CNN inference from the cloud to the edge. Recently, collaborative inference among edge devices gains the attention of researchers [6]–[8]. In this scheme, a data source (camera, sensors, etc) splits the captured data into tiles and distributes them to an IoT edge cluster, and then these edge devices in the cluster collaboratively compute and return the final output to the data source. Such an approach has the benefit of protecting user privacy by keeping data staying local. Meanwhile, the closer location to its sources, the lower latency it responds. Since each device only processes part of the original data, the memory consumption and inference latency can be reduced. Compared with the model compression and parameter pruning approaches which aim at the trade-off between inference latency and accuracy [9]–[11], collaborative inference does not require adjusting the model architecture or re-train the model, without losing accuracy. Despite all these benefits, there still leave some challenges that are not completely solved yet in previous works.

On the one hand, due to the characteristic of CNN model, (1) **redundant calculation during collaboration cannot be neglected**. The input data tile of devices participating in the inference is partially overlapped with each other. Moreover, the overlapped part on a device increases when the number of assigned CNN layers to the devices and the number of devices grow. The detailed explanation is presented in

Qi Qi and Jingyu Wang are the corresponding authors.

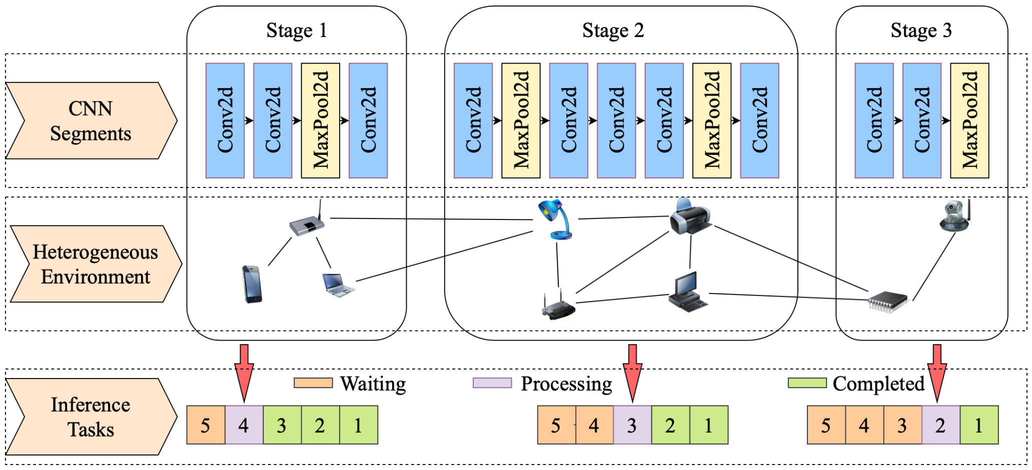


Figure 1: A diagrammatic sketch of pipelined inference.

Sec. II. On the other hand, (2) **communication overhead is expensive during collaborative inference**. Under the IoT environment bandwidth limits frequent communication among devices would affect the inference efficiency, but less communication brings more redundant calculation [8]. This phenomenon requires a optimization algorithm for a trade-off between computation and communication. But both the computing power of IoT devices and computation requirement of CNN layers varies. (3) **The heterogeneous edge environment also hinders the optimization for collaboration**. In the real-world IoT environment, the workload of executing inference tasks is diverse over time. Taking the devices in a smart home as an instance, these devices could be idle when occupants go to work, and busy when they return home. This demands an (4) **adaptive solution for the dynamic workload**.

In this paper, we explore previous efforts for parallel execution of CNN inference and propose a pipelined cooperation scheme (PICO) for multiple heterogeneous devices. Fig. 1 plots a diagrammatic sketch. PICO divides CNN model into contiguous neural layer segments. Each segment will be assigned to a sub-cluster of edge devices. We refer such a sub-cluster owning a neural layer segment as a *stage* and these stages compose an inference pipeline. There are 3 stages in Fig. 1. The input data is fed into the first stage, and the output result is produced at the last stage.

There are two import metrics in PICO, *pipeline latency* and *pipeline period*. The first term denotes the sum of time used for all stages and the last term is defined as the longest time used among stages. Obviously decreasing the period tends to increase the latency, thus our goal is to minimize pipeline period (maximize throughput) meanwhile to keep the pipeline latency under a certain value. PICO uses a lightweight algorithm based on dynamic programming to find out the partition strategy that meets our goal.

Note the inference latency of a task is composed by pipeline latency and the waiting time when task arrives. Since PICO improves the throughput, the waiting time can be greatly reduced when the workload is high, which decreases the inference latency in the end. We implement a framework that dynamically detects the current workload and automatically chooses the cooperation scheme with the lowest inference latency. In our experiment with 8 RaspberryPi devices, the average inference latency can be reduced by $1.7 \sim 6.5\times$ under different workloads, and the throughput can be improved by $1.8 \sim 6.2\times$ under various network settings.

In a nutshell, we make the following contributions:

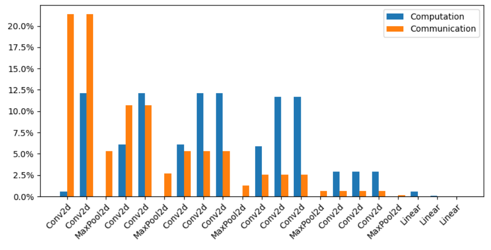
- We present a pipelined cooperation scheme (PICO) for CNN inference on a heterogeneous IoT edge cluster.
- We design a dynamic programming based algorithm to decide the optimal partition strategy which maximize the throughput.
- We propose an adaptive framework to automatically choose proper inference scheme under dynamic workloads.
- We apply our technique on an IoT edge cluster consisting of Raspberry-Pi-based hardware and evaluate image recognition and object detection CNN models.

II. Background And Motivation

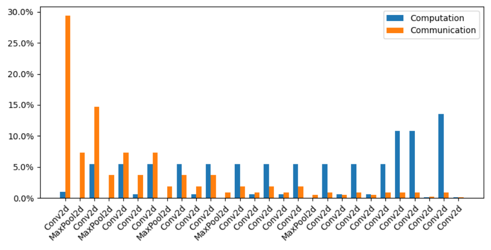
In this section, we briefly introduce the procedure of CNN inference and the current parallel scheme. Then we propose the pipelined cooperation scheme to improve the efficiency of inference.

A. Procedure of CNN Inference

During CNN inference, each convolution layer (*conv*) produces the output feature map by using a set of kernels to slide over the input volume or input feature map received from the previous layer. Every scalar in the output is



(a) VGG16



(b) YOLOv2

Figure 2: The communication and computation overhead of each layer.

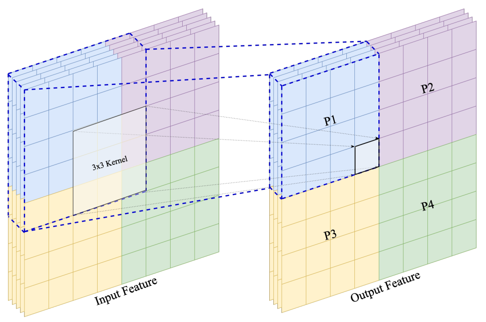
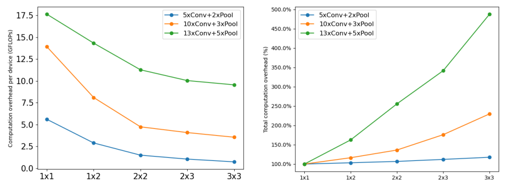


Figure 3: Feature map partition



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

Figure 4: Computation overhead with different partition settings.

calculated by a dot product between the weights of kernels and a small sub-region of the input. The pooling layer (*pool*) performs a down-sampling operation. It is used to progressively reduce number of parameters, memory footprint and amount of computation in the network.

Conv operation is the biggest bottleneck during inference. We measure two classic CNN models (VGG16 [12], YOLOv2 [13]) then plot the communication and computation percentage of each layer, as shown in Figure 2. Conv layers provide 99.19% computation in VGG16 and 99.59% in YOLOv2. And due to different settings own by each conv layer (kernel size, padding, in and out channels, etc), the communication or computation percentage also varies. How to efficiently execute conv operations is the key to accelerate CNN inference.

B. Parallelizing CNNs in IoT Devices

Fortunately these operations can be parallel executed on different devices at the same time by splitting the input into several tiles. In Fig. 3, four edge devices are assigned to produce different partitions $P_1 \ldots P_4$ of the output feature map of a specific layer with 3×3 kernel size. We refer this technology as *feature map partition*. But due to the property of conv operations, the partition of input feature map will overlap. In Fig. 3, to obtain the correct value in P_1, the calculation with 3×3 kernels have to use the more proportion of the input feature map. This property leadss to a redundant computation and increases the difficulty of parallel algorithm design.

We next introduce the two parallelization schemes used in this paper. [6] is the first work that uses feature map partition inf cooperative CNN inference. For each layer, the idea is to split the input feature map into pieces and distributes 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 wireless network, this results in substantial communication overhead. The benefits of layer-wise parallelization are generally defeated by communication costs. To reduce the communication among devices, *fused-layer* parallelization was introduced in [7] and [8]. Instead of distributing the computation of every layers individually, this scheme parallelizes multiple fused layers. Devices can execute the computation 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.

C. Motivation And Pipelined Inference

1) Motivation: Expensive communication cost in IoT environment and redundant computation limits the cooperation of edge devices for CNN inference. For fused-layer scheme, the redundancy quickly grows when the number of devices or fused layers increases. We measure the required floating-point operations (FLOPs) under different number of devices and fused layers using VGG16 [12]. Fig. 4a presents the FLOPs per device meanwhile Fig. 4b shows the sum of FLOPs of all devices. Fused-layer strategy works fine under small network, but the redundant computation quickly grows on deeper CNN. For layer-wise scheme, the redundant part is small, but the frequent communication among IoT devices causes inefficient performance.

2) Pipelined Inference: According to the above discussion, the CNN inference is hard to accelerate when the devices or layers in a CNN reaches a certain number. These devices is either idle due to expensive communication cost or whistling to the wind due to the redundant computation. To improve the resource utilization, we bring *pipeline* scheme into cooperative CNN inference. This scheme divides layers and devices into several disjoint subsets. The subset containing both layers and devices is referred as *stage* in our description. The CNN inference is performed stage after stage and the computation in each stage can be seen as a fused-layer scheme, as shown in Fig. 1. The fused-layer scheme can be seen as a special case of pipeline scheme when the number of stages is 1. To achieve the best resource utilization, the workloads of every stage should be as close as possible.

Pipeline is widely used in task scheduling where a bunch of processors is assigned to an application with pipeline structure [14], [15]. And many CNN models can be abstracted as such an application which each neural layer is a sub-task of pipeline. However, pipeline scheme meets difficulties when applied to edge inference. Traditional way of scheduling maps multiple processors to one sub-task. But generally the number of neural layers is more than devices, which requires a many-to-many mapping. And different mapping setting also changes the amount of parallelization overhead due to the redundant computation. Moreover, the heterogeneous edge devices and varied neural layers bring challenges to this scheme.

III. System Model

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

A. Problem Define

Given a CNN model $\mathbb{M}$ with L layers, where $l_i \in \mathbb{M}$ is a layer in the model. Let $\mathcal{M}_{i\to j}$ denotes a model segment from layer l_i to layer l_j. 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 computing capability. We also assume the bandwidth between all edge devices is the same and is known as b. This assumption covers most cases when these devices under the same WLAN environment such as home and factory [8], [14].

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

$$\mathbb{S}^\star = \underset{\mathcal{T}(\mathbb{M},\mathbb{D},\mathbb{S})\le T_{lim}}{\arg\min} \mathcal{P}(\mathbb{M},\mathbb{D},\mathbb{S}) \tag{1}$$

where $\mathcal{T}(\mathbb{M},\mathbb{D},\mathbb{S})$ denotes the pipeline latency under specific stage configuration $\mathbb{S}$ and $\mathcal{P}(\mathbb{M},\mathbb{D},\mathbb{S})$ is the period of pipeline.

B. Cost Model

Let $f(l_i; F_i^k)$ denotes the required FLOPs of conv layer l_i when generates an output feature map F_i^k. Assume F_i^k is a $c_i \times w_i \times h_i$ floating tensor, where w_i and h_i is the width and height of output feature. If layer l_i is a conv layer with $k_i \times k_i$ kernel size, c_i output channel and s_i stride. Since each floating number is calculated by the $k_i \times k_i$ kernel with different channel c_{i-1}, $f(l_i; F_i^k)$ can be given by:

$$f(l_i; F_i^k) = {k_i}^2 c_{i-1} w_i h_i c_i \tag{2}$$

where c_{i-1} is the output channel of data produced by the previous layer l_{i-1}. Note here we ignore the pool layers since they require far fewer FLOPs than conv layers.

The height h_i and width w_i of feature map F_i^k can be calculated recursively using the feature map F_{i+1}^k of the next layer l_{i+1}:

$$h_i = (h_{i+1}-1)s_{i+1} + k_{i+1},\ w_i = (w_{i+1}-1)s_{i+1} + k_{i+1}. \tag{3}$$

This formula suits for both conv and pool layers.

So far if a device d_k is responsible to produce F_k^j with model segment $\mathcal{M}_{i\to j}$, we can give the required FLOPs operation $\theta(\mathcal{M}_{i\to j}; F_k^j)$:

$$\theta(\mathcal{M}_{i\to j}; F_j^k) = \sum_{i'=i}^{j} f(l_{i'}; F_{i'}^k). \tag{4}$$

And the inference time $t_{comp}(d_k, F_k)$ for device d_k can be estimated by the following equation:

$$t_{comp}(d_k, F_j^k) = \alpha_k \frac{\theta(\mathcal{M}_{i\to j}; F_j^k)}{\vartheta(d_k)} \tag{5}$$

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

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

$$T_{comp}(\mathcal{S}_{i\to j}) = \max_{\substack{d_k \in \mathcal{D}_{i\to j} \\ F_j^k \in \mathcal{F}_j}} t_{comp}(d_k, F_j^k). \tag{6}$$

Let d_f denote the device responsible for frame partition and stitch in stage $\mathcal{S}_{i\to j}$. d_k is a computing device, the input/output feature map transfer time $t_{comm}(d_f, d_k, \mathcal{M}_{i\to j})$ can be given by:

$$t_{comm}(d_f, d_k, \mathcal{M}_{i\to j}) = \frac{\varphi(F_i^k) + \varphi(F_j^k)}{b} \tag{7}$$

where $\varphi(F_i^k)$ is the feature size on a given input feature map F_i^k. Sum the communication cost for each device d_k in stage $\mathcal{S}_{i\to j}$, we define

$$T_{comm}(\mathcal{S}_{i\to j}) = \sum_{\substack{d_k \in \mathcal{D}_{i\to j} \\ F_k \in \mathcal{F}_{i\to j}}} t_{comm}(d_f, d_k, F_k) \tag{8}$$

as the communication cost of stage $\mathcal{S}_{i\to j}$.

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

$$T(\mathcal{S}_{i\to j}) = T_{comp}(\mathcal{S}_{i\to j}) + T_{comm}(\mathcal{S}_{i\to j}) \tag{9}$$

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

Next we define the optimization objective as:

$$\mathcal{P}(\mathbb{M}, \mathbb{D}, \mathbb{S}) = \max_{\mathcal{S}_{i\to j} \in \mathbb{S}} T(\mathcal{S}_{i\to j}) \tag{10}$$

$$\mathcal{T}(\mathbb{M}, \mathbb{D}, \mathbb{S}) = \sum_{\mathcal{S}_{i\to j} \in \mathbb{S}} T(\mathcal{S}_{i\to j}) \tag{11}$$

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

C. Analysis

The goal of our optimization algorithm is finding the best stage set $\mathbb{S}^\star$ that minimizes the maximum period $\mathcal{P}(\mathbb{M}, \mathbb{D}, \mathbb{S})$ 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 $\mathcal{S}_{i\to j}$, the number of devices $|\mathcal{D}_{i\to j}|$, the start i and end point j of model segment $\mathcal{M}_{i\to j}$ in stage can also be configured.
- the number of stages in optimal $|\mathbb{S}^\star|$ is uncertain.

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

Theorem 1: Given a constriction $\mathcal{T}(\mathbb{M}, \mathbb{D}, \mathbb{S}) \leq T_{lim}$, the problem of minimizing maximum stage execution time $\mathcal{P}(\mathbb{M}, \mathbb{D}, \mathbb{S})$ with a heterogeneous cluster is NP-hard.

Proof: Considering a scheduling problem defined as follows: Given L identical tasks can be paralleled to several processors without additional overhead, assign these tasks to D heterogeneous devices. The goal is to maximize the throughput. This problem is proven to be NP-hard by [15]. We can construct a CNN model 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 for the above task assignment problem. Thus the optimization of $\mathcal{P}(\mathbb{M}, \mathbb{D}, \mathbb{S})$ is NP-hard. Here complete the proof. ■

IV. Pipelined Cooperation Scheme for CNN Inference

In this section, we present a pipelined 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 which automatically chooses suitable parallel scheme under dynamic workload.

A. Heuristic

Although the polynomial algorithm for optimizing $\mathcal{P}(\mathbb{M}, \mathbb{D}, \mathbb{S})$ does not exist unless $P = NP$, the optimal solution can be found in polynomial time if the IoT cluster is homogeneous. which lead 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.

1) Dynamic Programming: Given a 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 $\mathbb{D}$.

$$\vartheta(d'_k) = \frac{\sum_{d_k \in \mathbb{D}} \vartheta(d_k)}{|\mathbb{D}|} \; \forall d'_k \in \mathbb{D}', \quad |\mathbb{D}'| = |\mathbb{D}| \tag{12}$$

Considering a specific stage $\mathcal{S}_{i\to j}$. Fir any device d_k belongs to this stage, the output feature map F_j^k is equivalently partitioned. Thus, $\mathcal{F}_j$ can be determined by the size of stage. We denote $p = |\mathcal{D}_{i\to j}|$ 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 layer form 1 through s with $p - p'$ edge devices followed by a single stage with layers $s + 1$ through j replicated over p' workers. Then using the optimal sub-problem property, we can solve the optimization problem through dynamic programming:

$$P[i][j][p] = \min_{i \leq j' < j} \min_{1 \leq p' < p} \max \begin{cases} P[i][j'][p - p'] \\ Ts[j' + 1][j][p'] \end{cases} \tag{13}$$

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

Algorithm 1 Dynamic programming for pipelined 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 (9)
 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 $p' := 1 \rightarrow p - 1$ do
10:             calculate $Ts[s+1][j][p']$ using (9)
11:             $T_{lim} \leftarrow T_{lim} - Ts[s+1][j][p']$
12:             if $T_{lim} < 0$ then
13:                 continue
14:             $P[i][s][p-p'], T[i][s][p-p'] \leftarrow DP(i, s, p-p', T_{lim})$
15:             if $T_{lim} < T[i][j][p-p']$ then
16:                 continue
17:             $period \leftarrow \max(P[i][s][p-p'], Ts[s+1][j][p'])$
18:             if $period < P[i][j][p])$ then
19:                 $P[i][j][p] \leftarrow period$
20:                 $T[i][j][p] \leftarrow T[i][s][p-p'] + Ts[s+1][j][p']$
21:                 $R[i][j][p] \leftarrow (i, s, p-p')$
22:                 $S[i][j][p] \leftarrow (s+1, j, p')$
        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:     $\mathbb{S} \leftarrow \mathcal{S}_{i\rightarrow j} \cup \mathbb{S}$
```

Algorithm 1 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 function *BuildStrategy* by recursively iterating the calculated R and S, and adding the corresponding stage configuration $\mathcal{S}_{i\rightarrow j}$ to $\mathbb{S}$.

2) Adapt to the heterogeneity: We use a greedy algorithm to adapt the calculated $\mathbb{S}'$ in Algorithm 1 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 $\mathcal{S}'_{i\rightarrow j}$:

$$\Theta_{i\rightarrow j} = \sum_{d_k \in \mathcal{D}_{i\rightarrow j}} \theta(\mathcal{M}_{i\rightarrow j}; F_j^k), \tag{14}$$

We want $\Theta_{i\rightarrow j}$ are 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 edge 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 map F_j^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 2.

Algorithm 2 Adjust stage configuration $\mathbb{S}$ for heterogeneous devices

```
Require: $\mathbb{S}'$: the optimal stage set for homogeneous cluster.
 1: function ADJUSTSTAGE
 2:     Initialize a empty $\mathbb{S}$
 3:     Sort devices in $\mathbb{D}$ by compute 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 partition $\mathcal{F}_j$ using Divide-And-Conquer.
11:             $\mathbb{S} \leftarrow \mathcal{S}_{i\rightarrow j} \cup \mathbb{S}$
12:             Remove $\mathcal{S}'_{i\rightarrow j}$ from $\mathbb{S}'$
    return $\mathbb{S}$
```

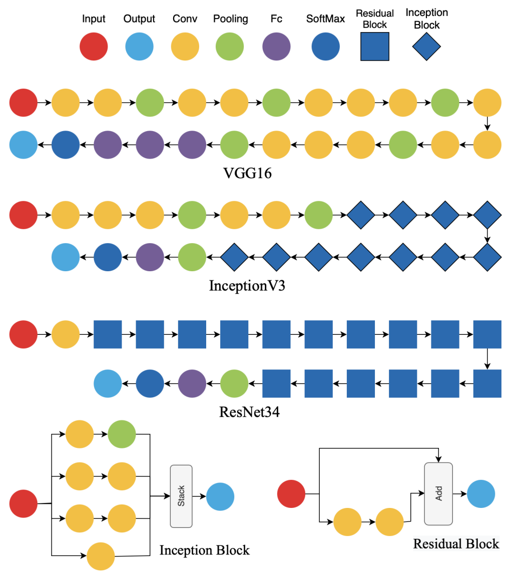


Figure 5: The typical CNNs with both chain and graph structures.

B. Generalize to graph-based CNN

So far those CNNs in our discussion are assumed with chain structure. In recent years researchers tend to structure CNN with high-level blocks instead of using layers directly. We plot three neural network structures in Fig. 5. VGG16 is a typical chain CNN which is composed by continuous layers. ResNet34 [16] and InceptionV3 [17] represent the common graph-based CNNs. Instead of using continuous layers, these CNNs build different kinds of blocks which can be abstracted as a small directed acyclic graph with several paths from the input to the output feature map, such as Residual block in ResNet34, Inception block in InceptionV3. As these blocks are still continuously connected, we use a trade-off to generalize our algorithm to these CNN models by considering each block as a special layer. In order to ensure the correctness of inference, we first calculate the partition of input feature map for every path in one block, and then combine them into a bigger one which is used as the input partition for this block.

C. Adaptive Parallel Scheme Switching

In the real world, the arriving time of every inference task is uncertain. When the IoT cluster is under a heavy workload, pipelined inference could greatly improve the throughput and thus reduces the average waiting time of each task. But if the workload is light, there may be only one working stage at one time, which causes an unefficient pipeline. For this situation, though those one-stage schemes [6]–[8] has more redundant computation, we still should switch to them since they use all devices to executing one inference task.

Theorem 2: If there are λ inference tasks arrive per unit time follows the *Poisson distribution*, and the parallel scheme has a period p and executing latency t. The average inference latency for each task is $\frac{p(2-p\lambda)}{2(1-p\lambda)} + t$.

Proof: The problem is an instance of *M/D/1 queue model*, and the proof can be found in queuing theory [18], omitted. ■

Once the set of stage configurations $\mathbb{S}$ is determined, The pipeline latency $\mathcal{T}(\mathbb{M}, \mathbb{D}, \mathbb{S})$ and period $\mathcal{P}(\mathbb{M}, \mathbb{D}, \mathbb{S})$ can be calculated using the cost model in III-B. Then we can estimate the average inference latency of pipeline with Theorem 2. As for those one-stage scheme p is equal to t. In our experiment, we choose [8] as the one-stage scheme.

Another problem is to measure the workload λ of cluster. It is hard for the edge cluster to capture the realtime workload directly, thus we use a moving average method to estimate the current workload λ_t:

$$\lambda_t = \beta\hat{\lambda} + (1-\beta)\lambda_{t-1} \tag{15}$$

where λ_t is the predicted workload of edge cluster at time t, $\hat{\lambda}$ is the measured workload in the last time $t-1$, and β is a hyper-parameter used to denote the impact of current workload $\hat{\lambda}$.

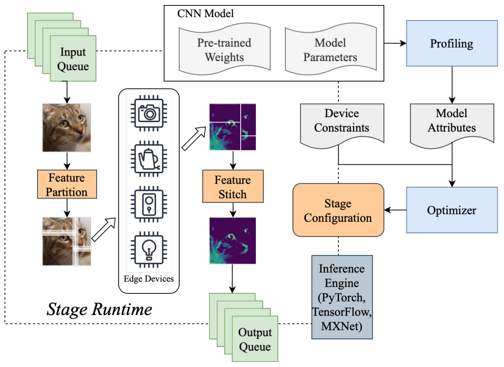


Figure 6: The workflow of stages in an inference pipeline.

D. Implementation

The workflow of stages: We summarize the workflow of stages in Fig. 6. 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 size according to $\mathcal{F}_j$ and distributes them to those devices $\mathcal{D}_{i\to j}$. Once the computation finishes, the outputs of those devices is gathered and stitched together. There are two other threads responsible to put the receiving feature map into the input queue and to send the output to the next stage.

Backend inference engine: Darknet [19] is commonly adopted as inference engine at the edge, but currently it does not support conv layers with non-square kernel size (e.g., 1×3, 5×1), which is widely used in InceptionV3. We turn to use LibTorch (A C++ version of PyTorch) instead of Darknet to execute convolution layers, and implement a distributed framework integrated with network communication modules using C++ extension and TCP/IP with socket. We use NNPACK to accelerate the inference on the ARM chips.

Feature split and stitch: Most popular DL framework such as TensorFlow, PyTorch does not provide an efficient way to split feature map with overlapped parts, and use these high level provided by those frameworks to implement the 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.

V. Experiment

We introduce the evaluation bed of our experiment and plot the result including inference latency, throughput and resource utilization rate of different parallel schemes.

A. Environment Setup

Hardware: We build an IoT cluster for evaluating our method using the 8 ARM based Raspberry-Pi 4Bs and one

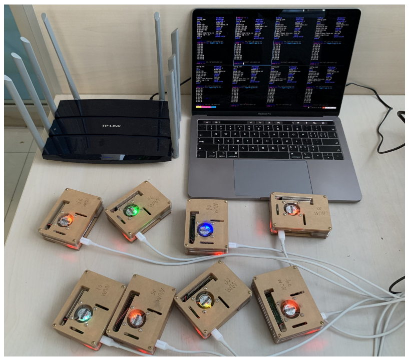


Figure 7: The testbed in our experiment composed by 8 Raspberry-Pi 4Bs and one WiFi access point.

WiFi access point with 50Mbps bandwidth. Each Raspberry-Pi 4B consists of a Quad Core ARM Cortex-A73, 2 GB LPDDR2 SDRAM and dual-band 2.4 GHz/5 GHz wireless. These Raspberry-Pi 4B are fixed to run with one CPU core in order to represent a realistic low-end edge device cluster. We plot this IoT cluster in Fig. 7, a laptop is used to monitor this cluster.

Models overview: VGG-16 [12] was the champion in the 2014 ImageNet ILSVRC challenge [20]. It revealed that one of the critical attributions of the neural network model is the depth, in other words, the number of layers. The network contains 13 conv, 5 pooling and 3 fc layers. You only look once (YOLO) [13] is a real-time object detection system. Compared with VGG-16, it has deeper architecture but used smaller memory. There are 23 conv and 5 pooling layers in YOLO, nearly twice of VGG-16. One insight of YOLO is used conv layer with 1x1 kernel size to replace the fc layer, which significantly reduced the number of parameters. ResNet34 and InceptionV3 are typical graph-based CNNs that uses several blocks instead of layers.

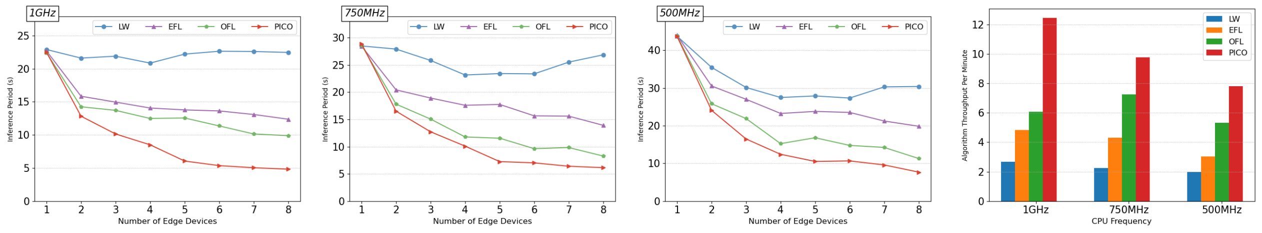
Compared method: We compare four different parallelization strategies in our evaluation: (1) Layer-wise (LW) scheme, which parallelizes the networks layer by layer; (2) Early-fused-layer (EFL) scheme, an extension of to the implementation of DeepThings [7], which fuses and parallelizes the first few conv layers and 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) Pipelined Cooperation (PICO) scheme, which assign different layers to different devices by dynamically adapting to the available computing resources and network condition in an IoT cluster.

Inference task arrival scheme: We consider the inference tasks arrive to the cluster in an online fashion and discrete timesteps. More specifically, these tasks arrive following a *Poisson distribution*. We define the cluster capacity as the throughput of Early-Fused-layer scheme when processing tasks. The average task arrival rate is chosen to make the average workload varies between 40% and 150% of cluster capacity. We also consider a simple task arrival scheme such that each task arrives immediately once the last task was complete. This scheme is used to measure the maximum throughput of different parallel schemes.

B. Runtime Performance

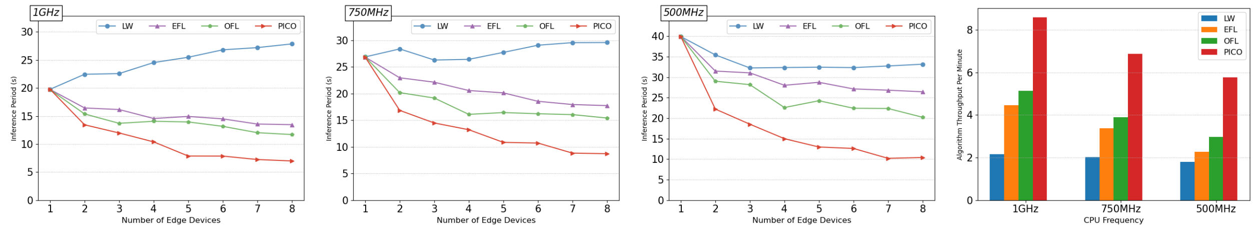
Maximum throughput: Fig. 8 and Fig. 9 plot the cluster capacity when executing VGG16 and YOLOv2 with different parallel schemes. The first three figures plot the inference period with different parallel schemes and CPU frequencies. The last figure plots the accomplished inference task per minute with 8 devices. It represents the throughput of different parallel schemes. PICO has the best performance as excepted, 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 layer-wise scheme with 1GHz CPU core. YOLOv2 has nearly twice number of layers compared with VGG16, which brings more communication overhead for Layer-wise strategy. When the computing resource is rich (1GHz), the gain brought by the increasing number of devices is offset by communication overhead. Early-fused-layer and optimal-fused-layer schemes 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 optimal-fused-layer scheme optimizes the configuration of fused layers, it outperforms early-fused-layer 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.

Average inference latency: Fig. 10 and Fig. 11 plots the average inference latency of VGG16 and YOLOv2 under different workloads and CPU frequencies when tasks arrive following a Poisson distribution. This experiment is carried out using 8 devices. We execute the inference process for 10 minutes and repeat them 3 times with different parallel schemes. The average inference latency includes the task waiting time and the actual processing time. Note we remove layer-wise strategy from comparison due to its poor performance. We also add the adaptive parallel scheme switching strategy to PICO and refer it as APICO. When the workload increases, the average inference latency also goes up. However, PICO and APICO keep a stable average inference latency compared with early-fused-layer and optimal-fused-layer schemes. According to Theorem 2, the inference period greatly affects the task waiting time under heavy workload, thus the latency of early-fused-layer scheme grows fastest due to its longest inference period. When the workload of cluster is light, there is unlikely to have new task arriving when the last task is not finished. In this situation, PICO may be only one active stage at a time,



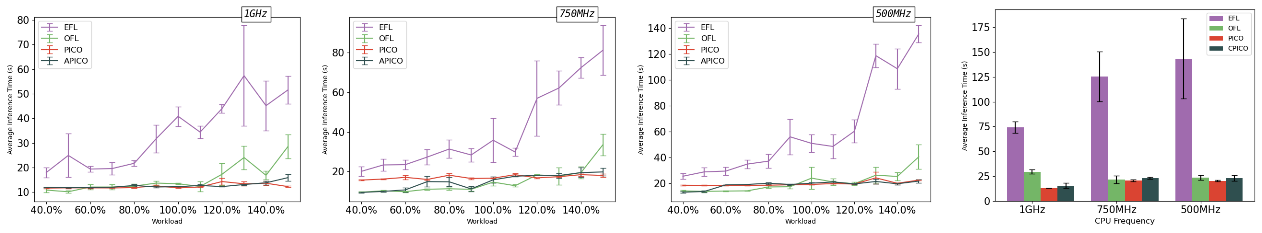
(a) Inference period with under different parallel scheme and CPU frequencies (b) Throughput with 8 devices

Figure 8: The cluster capacity when executing VGG16.



(a) Inference period with under different parallel scheme and CPU frequencies (b) Throughput with 8 devices

Figure 9: The cluster capacity when executing YOLOv2.



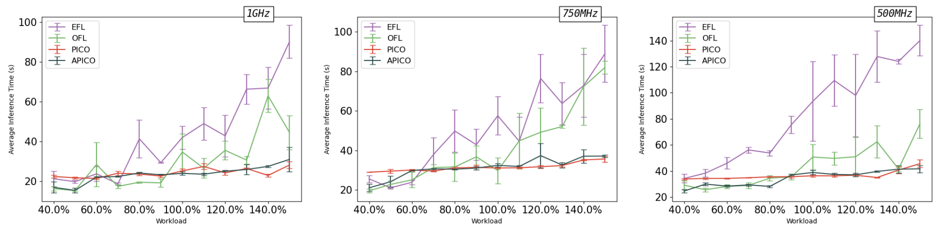
(a) Average inference latency under different workloads (b) Latency under 100% workload

Figure 10: The average latency of different algorithms for VGG16.

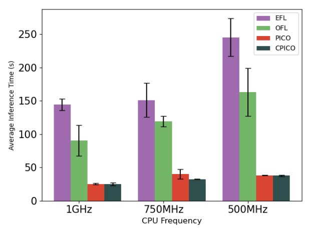
and thus has an inefficient inference. We can observe it in both Fig. 10 and Fig. 11. The average latency of PICO may higher than both or one fused-layer schemes. Since these parallel schemes use the entire cluster to execute the current inference task. We can also find the APIO uses fused-layer scheme in the light workload, and then switches to pipeline scheme when the workload grows.

Speedup for graph-based CNNs: We can adapt PICO to those graph-based CNNs by considering the block structure as a special layer. Fig. 12 plot the speedup ratio under different CPU frequencies for ResNet34 and InceptionV3. When executing CNN inference with 8 devices, PICO can achieve nearly $5\times$ speedup for ResNet34 and $4\times$ for InceptionV3. The speedup effect is more obvious with low CPU frequency since the limitation of computing resource is relieved when the number of edge devices increases. We can also find the speedup effect for ResNet34 is better than InceptionV3. We think it is caused by the different among layers in residual and inception blocks, as shown in Fig. 5. Since the inception block contains more layers than residual block, the optimal model partition is more likely to exist within blocks. And PICO currently does not support such a partition, which leads to a smaller speedup ratio.

Resource utilization: We monitor the CPU usage during inference on a heterogeneous edge cluster and record the average computing resource utilization ratio (Utili.) for different parallel scheme. We also calculate the redundancy ratio (Redu.) during computation. The result is presented in Table I. With the greedy algorithm 2, the PICO can adjust the proper partition size for each device, thus the resource utilization of different heterogeneous devices keeps at a high level (77.18% and 94.89% for VGG16 and YOLOv2). In the same time, the redundant computation is kept in low percentages. Layer-wise scheme has the minimum average



(a) Average inference lantency under different workloads

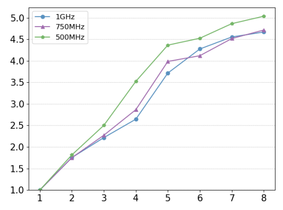


(b) Latency under 100% workload

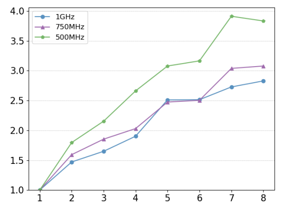
Figure 11: The average latency of different algorithms for YOLOv2.

Table I: The utilization and redundancy ratios among heterogeneous devices with different parallel schemes.

Model	Attributes	Methods	Type	Devices								Average
				1.2GHz	1.2GHz	800MHz	800MHz	600MHz	600MHz	600MHz	600MHz	
VGG16	Layers: 13 conv + 5 pool Input size: 244 × 244	LW	Utili Redu	30.35% 2.35%	23.66% 2.35%	32.41% 1.82%	32.64% 1.82%	44.08% 1.93%	45.85% 1.93%	43.50% 2.23%	44.99% 2.23%	37.19% 2.08%
		EFL	Utili Redu	33.64% 13.54%	34.87% 14.14%	74.19% 22.78%	76.76% 23.66%	95.96% 22.28%	98.63% 23.10%	66.77% 15.07%	66.86% 15.75%	68.46% 18.79%
		OFL	Utili Redu	40.30% 8.77%	41.58% 8.99%	62.79% 12.91%	63.79% 13.18%	88.31% 12.14%	97.08% 12.35%	79.27% 9.62%	83.07% 9.78%	69.53% 10.97%
		PICO	Ratio Redu	96.49% 10.44%	95.46% 10.33%	79.03% 5.22%	77.61% 0.00%	64.77% 6.33%	40.76% 4.95%	70.67% 3.22%	92.65% 3.22%	77.18% 5.47%
YOLOv2	Layers: 23 conv + 5 pool Input size: 448 × 448	LW	Utili Redu	26.77% 0.54%	22.55% 0.54%	29.50% 1.07%	29.39% 1.07%	45.66% 0.95%	45.11% 0.95%	44.74% 0.66%	44.07% 0.66%	35.97% 0.80%
		EFL	Utili Redu	37.85% 27.09%	35.64% 27.09%	67.24% 45.08%	67.61% 45.08%	96.01% 44.68%	95.28% 44.68%	75.87% 29.29%	72.81% 29.29%	68.54% 36.54%
		OFL	Utili Redu	47.34% 11.14%	47.96% 11.14%	71.59% 13.02%	70.87% 13.02%	95.32% 14.53%	94.98% 14.53%	87.31% 9.64%	87.61% 9.64%	75.37% 12.08%
		PICO	Utili Redu	88.01% 8.30%	96.84% 0.00%	88.50% 3.48%	96.75% 13.27%	99.00% 13.27%	98.96% 7.89%	90.97% 8.17%	98.30% 6.76%	94.89% 7.64%



(a) ResNet34



(b) IncetionV3

Figure 12: The speedup ratio for graph-based CNNs.

redundant computation but also has the worst performance on resource utilization. Those fused-layer schemes keep the devices busy, but many computations are redundant. Especially for the early-fused-layer scheme which has 46.54% percent redundancies executing YOLOv2. Through optimal-fused-layer scheme optimizes the configuration, the redundancy ratio is still higher than PICO since PICO uses a subset of edge devices instead of the entire cluster.

C. Comparing with optimal strategy

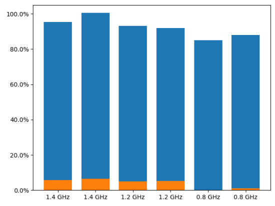
To find out the gap of $\mathbb{S}$ computed by PICO and the optimal result $\mathbb{S}^\star$, we implement a Breadth-First-Search based algorithm (BFS) and several toy models with different numbers of layers. We compared the optimization time and the resource utilization of the two algorithms.

Optimization cost: We measure the time cost during optimization for both algorithms. The result is presented in Table II. The time cost of BFS grows shapely when the numbers of layers and devices grow. For a model with 10 layers prepared for deploying on 6 devices, BFS takes 12.28 minutes to give the optimized result. When the number of layers increases to 12, the solutions can not be given within 1 hour. Since most CNNs are with more than 10 or even 100 layers nowadays, the BFS is impractical to adapt.

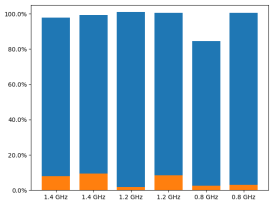
Resource utilization: We write a tiny model with 8 conv layers and 2 pooling layers and deploy it on a heterogeneous cluster with 6 devices. The model receives input images from the standard 64×64 MINIST dataset. The resource utilization and redundant computation are plotted in Figure

Table II: The execution cost of algorithms

Layers, Devices	PICO (Heuristic)	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



(a) PICO



(b) BFS

Figure 13: The comparison of resource utilization and redundant computation for PICO and BFS.

13. Those orange parts denote the redundant computation. The resource utilization ratios of all 6 devices are above 80%, as shown in Figure 13a. Through BFS achieves around 95% percentage, considering the time taken by PICO and BFS, we think the performance of PICO is acceptable.

VI. Related Work

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

Inference offloading: Due to the limited up-link of edge devices, traditional way of uploading captured data to the cloud server is time-consuming [21], [22]. Researchers focus on offloading the computation of early layers to edge devices (*Inference offloading*). To minimize the inference latency, Neurosurgeon [5] proposed to adaptively partition DNN between server and edge device according to the network situation. DADS [8] extended the idea to graph-based DNN by using a min-cut algorithm. Based on [8], QDMP [4] noticed the graph-based DNNs have more cut vertex than common graphs and proposed a divide and conquer algorithm, which achieves a nearly linear complexity in their experiments. Meanwhile, Branchynet [23] propose *early exit* mechanism by adding exit layers at the midden of DNN. This mechanism enables edge 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 [24] an incremental offloading technology which significantly improves the inference performance.

Collaborative inference locally: Recently researchers began to turn their attentions on executing inference completely at the edge with small IoT cluster [6]–[8], [25]. MoDNN [6] is the first work in this field. MoDNN considers such a situation where several edge devices collaborative inference under a WiFi access point using layer wise strategy. In their following up work MeDNN [26], they used an adaptive feature map partition method according to the heterogeneous devices. Deepthings [7] proposed to fuse the early layers of DNNs. Note that the feature map between layers are partitioned horizontally in [6], [26], whereas Deepthings partitions the feature map into 2D grids to further reduce memory overhead. This method reduced the communication overhead, but increased the redundancy calculation. AOFL [8] use a dynamic programming based algorithm to find a trade off between communication overhead and computation redundancy. [27] discussed using pipelined inference to achieve maximum throughput. But they only consider scheduling the number of input samples since the DNN they considered is already split into model segments, and they do not mention the redundant computation when partitioning the feature map.

VII. Conclusion

In this paper, we explore the recent parallel schemes for deploying CNN models and propose a pipelined cooperation scheme (PICO) to efficiently execute inference with heterogeneous IoT edge device. PICO divides the neural layers and edge 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. This scheme improves the inference efficiency by reducing the redundant computing. The execution time of each stage is optimized to be as close as possible to gain maximum throughput. We also implement an adaptive framework to choose the proper algorithm for executing CNN according to different workloads. In our experiment with 8 RaspberryPi devices, the average inference latency can be reduced by $1.7 \sim 6.5\times$ under different workloads, and the throughput can be improved by $1.8 \sim 6.2\times$ under various network settings.

Acknowledgment

This work was supported in part by the National Key R&D Program of China 2020YFB1807805, in part by the National Natural Science Foundation of China under Grants 62071067, 61771068, 61872310, in part by the Hong Kong RGC Research Impact Fund (RIF) with the Project No. R5060-19 and R5034-18, General Research Fund (GRF) with the Project No. 152221/19E and 15220320/20E, Collaborative Research Fund (CRF) with the Project No. C5026-18G, and Shenzhen Science and Technology Innovation Commission (R2020A045).

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] G. Kour and R. Saabne, "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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[15] A. Benoit and Y. Robert, "Complexity results for throughput and latency optimization of replicated and data-parallel workflows," *Algorithmica*, 2010.

[16] 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.

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

[18] G. Donald, F. S. John, M. T. James, and M. H. Carl, *Fundamentals of Queueing Theory*. Wiley-Interscience, 2008.

[19] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[20] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] J. M. Tarnawski, A. Phanishayee, N. Devanur, D. Mahajan, and F. Nina Paravecino, "Efficient algorithms for device placement of dnn graph operators," *Advances in Neural Information Processing Systems*, 2020.