Toward Collaborative Inferencing of Deep Neural Networks on Internet-of-Things Devices

Ramyad Hadidi, Jiashen Cao, Michael S. Ryoo, and Hyesoon Kim, *Member, IEEE*

***Abstract*—Recent advancements in deep neural networks (DNNs) have enabled us to solve traditionally challenging problems. To deploy a service based on DNNs, since DNNs are compute intensive, consumers need to rely on compute resources in the cloud. This approach, in addition to creating a dependency on the high-quality network infrastructure and data centers, raises new privacy concerns because of the sharing of private data. These concerns and challenges limit the widespread use of DNN-based applications, so many researchers and companies are trying to optimize DNNs for fast in-the-edge execution. Executing DNNs is further pushed to the edge with the widespread use of embedded processors and ubiquitous wireless networks in Internet-of-Things (IoT) devices. However, inadequate power and computing resources of edge devices, along with the small number of local requests, limit the use of prevalent optimization techniques such as batch processing. In this article, we enable the utilization of the aggregated computing power of several IoT devices by creating a local collaborative network for a subset of DNNs, visual-based applications. In this approach, IoT devices cooperate to conduct single-batch inferencing in real time while exploiting several new model-parallelism methods, which will be introduced in this article. Our approach enhances the collaborative system by creating a balanced and distributed processing pipeline while adjusting the tasks in real time. For experiments, we deploy a system with up to 10 Raspberry Pis and execute state-of-the-art visual models, such as AlexNet, VGG16, Xception, and C3D.**

***Index Terms*—Computer vision, distributed system, edge computing, Internet of Things (IoT), real-time system.**

I. Introduction and Motivation

DEEP and convolution neural networks (DNNs/CNNs) have shown extraordinary power in understanding large-scale data that are massively diverse and complex in several applications, such as computer vision and video recognition [1]. DNN-based applications are extensively being researched and applied to our daily lives. Because of their proximity to the data, conventional consumer-level devices, such as Internet-of-Things (IoT) devices, are a great candidate for the in-the-edge inferencing of DNNs [2]. However, compared to high-performance computing (HPC) data centers, IoT devices lack the required performance to execute DNNs [3], [4]. Nevertheless, in-the-edge inferencing of DNNs is gaining ground because of the widespread availability of IoT devices [5], affordability of embedded processors, and ubiquitous wireless networks. In-the-edge execution is also not dependent on cloud services and the high-quality networks [6], [7] that are not accessible in several scenarios (e.g., drones). Additionally, in-the-edge inferencing improves the privacy of users since it does not rely on the privacy policy of companies (e.g., offering smart home security cameras but requiring one to upload 24/7 recordings; offering private photographs storage but internally using the photographs as a training set). In this article, we focus on an environment that already has several IoT devices. Since not all devices are busy at one time, our aim is to aggregate the computational power of these devices to perform faster in-the-edge execution.

This article utilizes a local and distributed system of IoT devices for performing the entire computation of DNNs for CNN-based visual models with single-batch inferences. A single IoT device alone cannot effectively handle the entire computations of a DNN [8], [9]. Although with some optimizations, such as weight pruning [10] and precision reduction [11], [12], we can run limited versions of the current models on IoT devices [13] with the advancement of DNNs and the emergence of generalized models, the increase in the demand of computing power for DNNs is not expected to stop [14]. Therefore, exploring the efficient distribution of DNN computations is essential. As discussed, since IoT devices are a great candidate for DNN-based applications by moving the computations of DNNs closer to the edge, we can achieve the following: 1) reducing the dependence on cloud resources and high-quality network infrastructure for scenarios with limited Internet connectivity, such as drones and robots in a disaster area; 2) improving the privacy of private data since the data are not exposed outside the local network; and 3) providing an alternative solution to understand raw data locally than the current *de facto* solution of offloading to the cloud. Our discussions in this article focus on CNN-based visual models that have several use cases in IoT applications, such as video analytics and monitoring services.

In this article, we target CNN-based computer vision models and the computations of their layers, fully connected (`fc`) and convolution (`conv`) layers. Although computer vision models have been studied extensively for HPC machines, compared to the cloud, their in-the-edge execution changes important assumptions. First, since the requests are local and

Manuscript received May 9, 2019; revised December 2, 2019; accepted February 2, 2020. Date of publication February 6, 2020; date of current version June 12, 2020. This work was supported by NSF under Grant CNS 1815047 and Grant CNS 1814985. *(Corresponding author: Ramyad Hadidi.)*

Ramyad Hadidi, Jiashen Cao, and Hyesoon Kim are with the School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: rhadidi@gatech.edu; jcao62@gatech.edu; hyesoon@gatech.edu).

Michael S. Ryoo is with the Department of Computer Science, AI Institute, Stony Brook University, Stony Brook, NY 11794 USA (e-mail: mryoo@cs.stonybrook.edu).

Digital Object Identifier 10.1109/JIOT.2020.2972000

real-time performance is important, we might not have enough data to process in parallel (i.e., no immediate data-level parallelism). This means we cannot batch many requests to amortize the expensive costs of memory operations. Second, besides low computation power, IoT devices have significantly smaller memories, compared to HPC machines. If the memory requirement of a small computation task is not satisfied, the execution performance suffers considerably. Performance loss in such situations occurs because the device uses off-chip storage as swap memory. Third, by locally processing DNN computations, we avoid the high cost of offloading images/videos to cloud servers, which requires high uploading bandwidth.

Our contributions in this article are as follows.

1) We introduce several model-parallelism techniques for CNN-based DNN models, mainly used in computer vision, to reduce the memory footprint per device and divide their computations.
2) We generate, deploy, and monitor a balanced data-processing pipeline that efficiently processes the computations of DNNs. Our heuristic requires significantly less profiling and exploration than the previous work [8].
3) We study prevalent CNN models, such as image recognition (AlexNet [15], VGG16 [16], residual neural network (ResNet) [17], and Xception [18]) and video recognition (C3D [19]).
4) We propose a system in which collaborative and resource-constrained IoT devices perform the single-batch computation of DNNs in a distributed fashion. We deploy an example of such a system on an interconnected network of up to 10 Raspberry Pi 3s (RPis) [20].

II. Prior Work

Recently, with the prevalence of large DNN models, distributing a single model has gained the attention of researchers [8], [21]–[25]. Large models need more memory, and when the memory requirement of a DNN model is larger than the system's memory, the performance of a model suffers noticeably. More important, when executing DNNs on IoT devices, compared with HPC machines, two important criteria change: 1) we cannot batch several requests and make use of data parallelism and 2) we do not have access to machines with high memory capacities. This is why several released tools try to the alleviate memory and computation footprint of DNNs [4], such as ELL library [26], Tensorflow Lite [27], and TensorRT [28]. With the increasing importance of privacy, several companies have released specialized hardware for the edge, such as edgeTPU [29] and JetsonNano [30]. Besides these endeavors, designing efficient (in terms of memory and computation footprints) DNN models is also an ongoing effort [31], [32]. Our methods are orthogonal to these techniques since our aim is to distribute the computation of DNNs. In fact, such techniques are applicable to our distributed system to accelerate the execution even further.

We extend our previous work [23], in which several robots collaborate to perform distributed DNN computations, with new model-parallelism methods and a faster heuristic. Compared to the previous work that introduced one model-parallelism method for `fc` layers, we introduce additional methods for `conv`. Additionally, we study the characteristics of these methods. Although we provided an algorithm to distribute the tasks in [23], we find that our new heuristics with online monitoring tools significantly shorten the time to find the same near-optimal distribution. This is because the previous algorithm needs access to the entire profiled data, which takes a long time to gather and does not always cover all cases. On the other hand, the new heuristics shorten the exploration time by reducing possible cases using online monitoring tools. We use our previous work to implement the same dynamic allocation of tasks during execution with IP table files. (Refer to [23] for a detailed explanation.) Neurosurgeon [24] statically partitions a DNN model between a *single* edge device and the cloud. The partitioning is always between the cloud and only *one* edge device. DDNN [22] also tries to partition the model between the edge devices and cloud, but model retraining is necessary for each setting. In DDNN, sensor devices perform only the first few layers in the network, and the rest of the computation is offloaded to the cloud.

Another general direction is to reduce the overhead of DNNs using techniques, such as weight pruning [10], [33], resource partitioning [34], [35], quantization and low-precision inference [11], [12], [36], [37], binarizing weights [38]–[40], and designing specific models for mobile phones [31], [32]. Although these techniques reduce the overhead of DNNs, they require several additional steps that decrease the accuracy and enforce retraining of the model. This article could be applied on top of these techniques to increase the final performance as well. In summary, the following are the main differences of this article compared to the previous studies: 1) we study resource-constrained and IoT devices with limited memory space; 2) we increase the real-time performance of single-batch DNN inferencing; and 3) we introduce several model-parallelism methods for `conv`.

III. Background

A. Layers Overview

Fully Connected Layer: In a dense or `fc` layer, the value of each output element, or activation, is calculated from the weighted sum of all inputs as $a_j = \sum_i x_i w_{ij} + b_j$, in which i is the input, j is the output number, inputs are denoted as x_i, weights as w_ij, b_j as biases, and a_j as activations. This formula may also be written using matrix notations as $a = Wx + b$. W and b are defined during training and are fixed during inferencing.

`conv`: In computer vision models, usually all the layers except the last ones are `conv`. A `conv` applies a set of filters to a subset of inputs by sweeping each filter (i.e., kernel) over them. Each filter creates a channel, or depth (i.e., z-axis), of the output (Fig. 1). The spatial dimensions (i.e., x-axis and y-axis) of the output are defined by four parameters: 1) the size of input; 2) filter; 3) stride; and 4) padding. In this article, we use the same padding, which means the output size of a `conv` is same as the input size. In other cases, one can

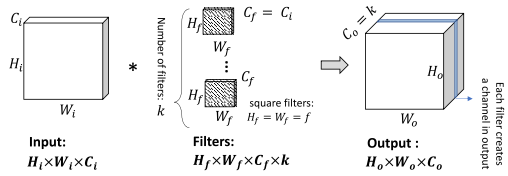

Fig. 1. conv—the computations consist of several filters, each of which creates a channel in the output.

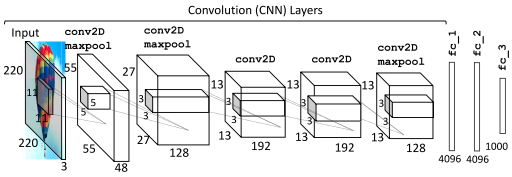

Fig. 2. AlexNet—architecture of the AlexNet image-recognition model [15].

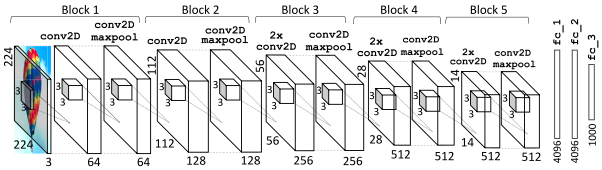

Fig. 3. VGG16—architecture of the VGG16 image-recognition model [16].

simply replace the output dimensions with the appropriate formulas. Similarly, we can extend these concepts to larger input dimensions.

Other Layers: To introduce nonlinearity, an activation layer (φ), such as ReLU, is applied on the output to create the input to the next layer, or $h_j = \varphi(a_j)$. This allows a model to learn complex functions. In addition, often a pooling layer downsamples the data and reduces the dimensions, such as a max-pooling (maxpool) layer. These layers, compared to fc and conv layers, are much less compute intensive, so we group them with their corresponding parent layer.

B. Models Overview

AlexNet: In the 2012 ImageNet large-scale visual recognition challenge (ILSVRC), AlexNet [15] significantly outperformed all the prior competitors. Fig. 2 illustrates the model of a single-stream AlexNet, which consists of five conv and three fc layers.

VGG16: Fig. 3 depicts the VGG16 model [16], which has a total of 16 layers: 13 convolution and 3 fc layers. As seen, VGG16 has a structured model; deeper conv has more filters and smaller spatial dimensions.

ResNet: The ResNet [17] introduced "skip connection" for training deeper networks in 2016. In this article, we used ResNet50 with 50 layers (Fig. 4). Additionally, Fig. 5(a) illustrates the basic blocks for ResNet that are used in Fig. 4. This model is residual in the sense that shortcut connections skip some blocks, which makes training easier for deep models.

Xception: The Xception [18] model is based on Inception V3 [41]. The Xception module independently processes the correlations in cross-channel and spatial features. Therefore, Xception introduces a special convolution unit, shown in

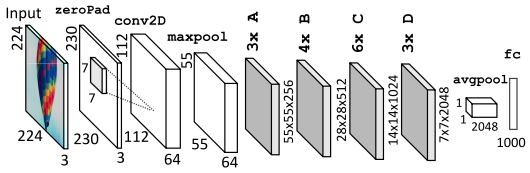

Fig. 4. ResNet50—architecture of the ResNet50 image-recognition model [17].

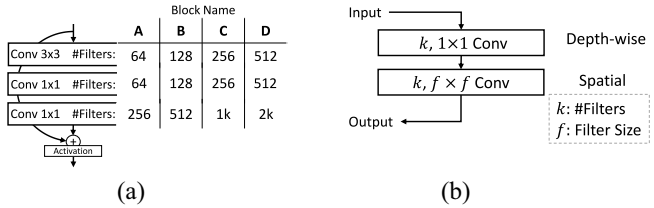

Fig. 5. ResNet50 and Xception blocks. (a) ResNet50 bottleneck block with a skip connection [17]. (b) Xception separable convolution block [18].

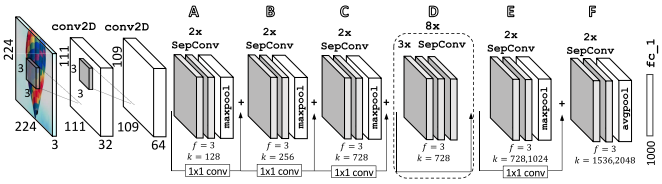

Fig. 6. Xception—architecture of the Xception image-recognition model [18].

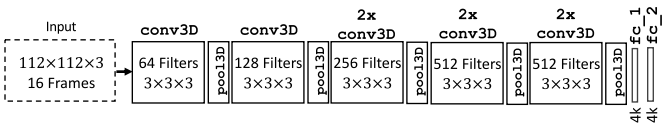

Fig. 7. C3D—architecture of the C3D action-recognition model [19].

Fig. 5(b), the separable convolution unit that decouples the mapping of cross-channel and spatial features. Separable convolution first performs cross-channel (i.e., depth-wise) convolution over input channels, and then performs an independent spatial convolution on each of the outputs. Fig. 6 shows the Xception model with 34 separable conv.

C3D: The C3D [19] model is designed to process videos and has been used in action recognition and scene classification tasks. To learn spatiotemporal features, the C3D model uses 3-D convolutions, which produce an output volume instead of a 2-D output per filter. Compared to a conventional conv, an additional sweep along the z-axis creates a volume in the output. Fig. 7 shows the C3D model, which consists of eight 3-D conv.

IV. Parallelizing and Distributing Inference Methods

In this section, we present our methods for distributing and parallelizing the computations of single-batch inferencing in fc and conv. We examine two general directions: 1) data and 2) model parallelism. In data parallelism, the presence of many requests at the same time enables us to increase the number of inferences per second (IPS) by independently serving each inference separately on each device. In model

TABLE I
CHARACTERISTICS OF MODEL PARALLELISM METHODS FOR FC LAYERS FOR A LAYER OF INPUT DIMENSION d_i, OUTPUT DIMENSION d_o, AND NUMBER OF DEVICES n

Name	#Devices	Distributed Activation	Multiplication (per device)	Reduction (per device)	Weights (per device)	Communication (total-per inference)	Merge Operation
No Splitting	1	N/A	$d_i d_o$	d_o	$d_i d_o$	$d_i + d_o$	N/A
Output Splitting	n	✓	$\frac{d_o}{n} d_i$	d_o/n	$n d_i$	$n d_i + d_o$	Concat
Input Splitting	n	✗	$\frac{d_i}{n} d_o$	$d_o[\lvert(d_i/n - 1)]$	$n d_o$	$d_i + n d_o$	Sum

parallelism, which is applicable to the computations required for a single input, the inference computations are distributed and parallelized over multiple devices. Data parallelism is realized because of batch processing and grouping several requests together. However, as discussed, IoT devices serve a limited number of requests. Moreover, these devices have limited memory to process the entire computation of an input in a tight schedule. Therefore, using only data parallelism might not be applicable to all situations in the edge.

To apply both model and data parallelism, we first divide a DNN model on multiple devices by layers (or a group of layers) and create a processing pipeline. These layers process the input sequentially, and the output of each layer is dependent on the output of its previous layer(s). Thus, we must correctly maintain the dependence between layers. By utilizing this processing pipeline, we can increase the throughput of computation, while the latency for each computation remains the same. In this article, we improve the performance further by applying model parallelism on top of this processing pipeline to parallelize the computation of the bottleneck layers.

Data parallelism was already introduced in [8] and [23] for `fc` and `conv` for DNN models. But, applying only data parallelism would not always work for resource-constrained and IoT devices. This is because data parallelism duplicates the same amount of computations on another device. Since computations are the same but on a different input data, the memory and computation footprints are not reduced. In detail, data parallelism alone cannot ensure high performance in IoT devices because:

1) for large layers, just the duplication of devices does not provide a performance benefit because the entire data are not loaded to the memory. Thus, a device still pays a high cost for accessing the off-chip storage (i.e., swap);
2) data parallelism needs a stream of input data, whereas in several scenarios, the frequency of input data is low;
3) to create a balanced and efficient data-processing pipeline in a distributed system, a balanced work assignment is required.

However, data parallelism is not flexible in adjusting the amount of computation per device. Model parallelism, on the other hand, exploits intralayer independence of computations in DNNs to create fine-grained divisions of work. Thus, it solves the mentioned shortcomings of data parallelism. However, compared to the data parallelism, employing such deeper level parallelism needs knowledge of how each layer does its computations and how parallelism affects data communication, computations, and aggregation. We endeavor to address this knowledge gap in this article.

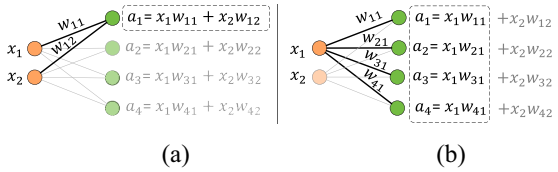

Fig. 8. `fc` layer model parallelism—applying model parallelism methods on a simple `fc` layer: (a) output splitting, in which each output is independently calculated and (b) input splitting, in which partial outputs are calculated based on a part of the input.

A. Model Parallelism for Fully Connected Layers

In an `fc` layer, since the computations of each activation (a_j) are independent of other activations, we can parallelize its computations. We describe two methods specific to `fc` layers: 1) output and 2) input splitting, shown in Fig. 8(a) and (b), respectively. In output splitting, we parallelize the computation of each activation while transmitting all input data to each device. Fig. 8(a) highlights a device and its computations to derive its activation. Each device holds the weights corresponding to its activations. Later, when the computations of each device are done, we merge the results by concatenating values in the correct order. We can apply an activation function either on each device or after the merging.

In input splitting, a device computes a partial part of *all* the activations. Fig. 8(b) illustrates an example in which a device computes half of the required multiplications for all the activations. In this method, a part of the input is transmitted to each device. Each device holds the weights corresponding to its input split. Later, when the computation of each device is finished, a merge operation adds all of the partial sums. Contrary to the output-splitting method, we cannot apply an activation function before the merge.

A more detailed summary of these methods is presented in Table I. These methods trade communication with the memory footprint. This is because each device holds part of the weights but needs to transmit more variables. A more detailed examination is shown in the table, where n is the number of devices, and d_i and d_o are input and output dimensions, respectively. As seen, both methods somehow divide the memory footprint (i.e., saved weights) and the number of multiplications. Input splitting slightly increases the number of reductions because computing the partial sums is necessary on

TABLE II
MODEL PARALLELISM METHODS FOR CONV (ASSUMING THE SAME PADDING)

Name	Division Factor	#Nodes	Distributed Activation	Weights (per device)	Input (per device)	Filters (per device)	Output (per device)	Communication (total-per inference)	Merge Operation
Baseline	N/A	1	N/A	kC_if^2	$H_iW_iC_i$	kC_if^2	H_iW_ik	$(C_i+k)(H_iW_i)$	N/A
Channel	k' $\frac{\text{filters}}{\text{node}}$	$\lceil \frac{k}{k'} \rceil$	✓	$k'C_if^2$	$H_iW_iC_i$	$k'C_if^2$	H_iW_ik'	$(\lceil \frac{k}{k'} \rceil C_i+k)(H_iW_i)$	Concat
Spatial	d $\frac{\text{part}}{\text{dimension}}$	d^2	✓	kC_if^2	Eq.2	kC_if^2	$\frac{1}{d^2}H_iW_ik$	$(d^2Eq.2+k)(H_iW_i)$	Concat
Filter	C_b batches	$\lceil \frac{C_i}{C_b} \rceil$	✗	kC_bf^2	$H_iW_iC_b$	kC_bf^2	H_iW_ik	$(C_i+k\lceil \frac{C_i}{C_b} \rceil)(H_iW_i)$	Sum

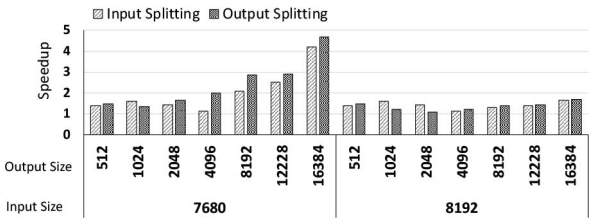

Fig. 9. Model-parallelism methods performance—input and output splitting performance on two RPis for `fc` layers.

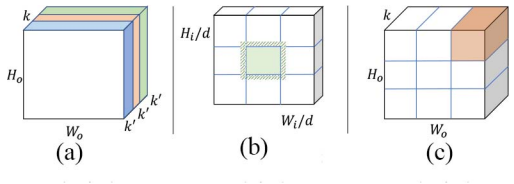

Fig. 10. Convolution model parallelism I. (a) Channel-splitting output. Spatial-splitting (b) input and (c) output.

each device. Furthermore, output- and input-splitting methods have a communication overhead of $(n-1)d_i$ and $(n-1)d_o$, respectively. Depending on the size of the input and output, we can find the most optimum choice based on our device and communication.

As an example, in Fig. 9, we run a series of dense layers on an RPi and their distributed versions on two RPis (in total four devices, with an initial sender and a final receiver). We cover a range of 512–16 384 in output sizes, and two input sizes, 7680 (not a power of two) and 8192 (a power of two). As seen, for the input size of 7680 and large output sizes, we achieve super-linear speedups. This is because in these cases, slow off-chip storage (i.e., swap) is used. However, for the input of 8192, the baseline DNN framework can optimize accesses and avoid swap activities by tiling. The baseline DNN framework optimizes the swap space accesses; however, it cannot always hide such costs for the input size of 7680. Furthermore, if off-chip storage activities are not occurring in the baseline case, as seen, speedup values are less than the ideal value of two. This is because each distribution has a communication cost associated with it. We examine these costs and their impact on our distribution in Section V. Fig. 9 shows that the input-splitting method has mostly lower performance than output splitting. This is because the input-splitting method cannot apply activations locally. Therefore, input splitting cannot benefit from a reduced number of values to transfer, compared to output splitting. The reduction of values occurs because activation functions (such as ReLU), set every negative value (or close to zero values) to zero. Thus, in sum, fewer values are transferred after activation.

B. Model Parallelism for Convolutional Layers

In a `conv`, each filter creates a channel in the output data. As Fig. 1 illustrates, assume the dimensions of input, filters, and output are $H_i \times W_i \times C_i$, $H_f \times W_f \times C_f \times k$, and $H_o \times W_o \times C_o$, respectively. The depth of the filters is defined by the depth of the input, or $C_f = C_i$. Here, without loss of generality, we assume square filters, $H_f = W_f = f$. The number of channels in the output is defined by the number of filters, or $C_o = k$. Each filter contains C_if^2 weights that are set during training. *Per output element*, each filter performs C_if^2 multiplications of its weights and input values, and one reduction operation. So, for k filters in a `conv`, per output element, we perform kC_if^2 multiplications and k reductions. Therefore, the total number of multiplications and reductions in a `conv` for *all elements* is

$$\begin{aligned} \text{Multiplications: } & H_oW_okC_if^2 \xrightarrow{\text{Same Padding}} H_iW_ikC_if^2 \\ \text{Reductions: } & H_oW_ok \xrightarrow{\text{Same Padding}} H_iW_ik. \end{aligned} \tag{1}$$

For a single inference, the amount of communication is the sum of the number of input and output elements, or $(H_iW_iC_i)+(H_iW_ik) = (C_i+k)(H_iW_i)$.

In the rest of this section, we describe our specific methods of model parallelism for `conv`. Since each method has advantages and disadvantages, Table II provides a detailed overview of the discussions in this section.

Channel Splitting: In channel splitting, each device calculates a nonoverlapping set of channels in output. In other words, each device processes only k' filters that $k' \leq k$. Fig. 10(a) shows an example output of this method with three devices. Since k' filter is processed per device, a total of $\lceil k/k' \rceil$ devices required. Each device needs not only its set of k' filters but the entire input data. So, filter's weights are divided across devices, or $k'C_if^2$ per device. The total number of multiplications and reductions remains the same, and each device handles $\lceil k/k' \rceil^{-1}$ part. In the end, when every device is finished, their data are concatenated depthwise, which is in $O(k)$. For the output, the total number of output elements to be transferred is H_iW_ik. We have the option to apply the activation function on each device or after the merging. In total, as shown in Table II, communication overhead is $(\lceil k/k' \rceil C_i - 1)H_iW_i$, since we need to transmit a copy of the input to all devices.

TABLE III
COMPARISONS OF MODEL PARALLELISM METHODS FOR CONV

	Channel Splitting	Spatial Splitting	Filter Splitting
Input	Entire input is copied	Input is divided spatially	Input is divided channel-wise
Filters	Some filters are saved	All filters are saved	Part of all filters are saved
Output	Each node calculates a channel	Each node calculates a spatial region	Each node calculates a partial output
Overhead	Input is copied across all devices	Input overlapping elements	Output partial sums

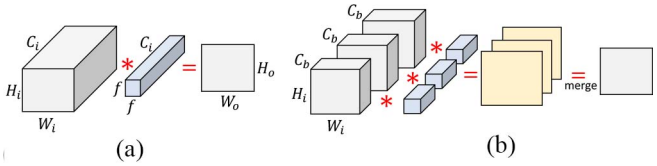

Fig. 11. Convolution model parallelism II. Illustration of (a) one filter convolution and (b) its corresponding filter splitting.

Spatial Splitting: Spatial splitting splits the input spatially, in the x-axis and y-axis. Assume that each split dimension is in d parts, so there are a total of d^2 parts,[1] as shown in Fig. 10(b). Each part of the input is transmitted to a device. Furthermore, each region is extended for $\lfloor f/2 \rfloor$ more overlapping elements with neighboring parts, so that we can do convolution on the borders. Therefore, the number of input data elements to be transmitted per device is

$$\left\lceil \frac{1}{d^2} \right\rceil H_i W_i C_i + 4\lfloor f/2 \rfloor \left(d^2 - d\right) \tag{2}$$

in which the first term represents the split input, and the second term represents the numbers of extra overlapping elements. Compared to channel splitting in which a copy of input is transmitted to all devices, spatial splitting only pays extra overhead for the overlapping elements. Since each device processes all filters and each needs a copy of all weights. Hence, the total number of filter elements to be transmitted is $d^2 k C_i f^2$. Note that this is a one-time cost for all inferences. The total number of multiplications and reductions is the same in total and each device processes only $1/d^2$. When the computation of each device is finished, their output is concatenated spatially. Similar to the previous method, the total number of output elements to be transferred is $H_i W_i k$. We have the option to apply the activation function either on each device or after the merging. As discussed, the communication overhead for spatial splitting is only for overlapping parts, which approximately is $4d^2\lfloor f/2 \rfloor (d^2 - d)$. Since the filter size is usually small, this overhead is not significant.

Filter Splitting: In filter splitting, both input and filter are split channelwise in batches of size C_b. Fig. 11(a) illustrates the base case in the convolution of one filter, which produces a single channel in the output. Fig. 11(b) illustrates the same filter divided into three parts with their corresponding input. Since there is a one-to-one correspondence between input and filter elements, each device computes a partial output. In the end, to create the final output, we sum all corresponding elements and apply the activation function. By denoting the input channel size as C_i, we need a total of $\lceil C_i/C_b \rceil$ devices. Since the input is split channelwise, the total number of input element transfers is without an overhead, or $H_i W_i C_i$. Similarly, each device saves only its dedicated channels of all filters, so the memory footprint is also divided. But, since each device sends a partial output to the merging device, there is an overhead of $(k\lceil (C_i/C_b) \rceil - 1)(H_i W_i)$ for transmitting output elements compared to the baseline. To create the final output, we need to perform $k\lceil (C_i/C_b) \rceil$ reductions. The concatenation is in $O(C_i/C_b)$.

Methods Comparison: Table III presents a comparison of the described methods. Channel splitting has an overhead of copying the input, whereas filter splitting has to transmit partial sums. The impact of these differences on the performance is defined by the properties of a `conv`. As illustrated in Fig. 12, we run a `conv` with the kernels 3×3, 5×5, and 9×9, filter depths 128 and 512, and various input depths with 128×128 inputs. We distribute this layer on three RPis using the mentioned splitting methods (in total five devices, with an initial sender and a final receiver). Speedups are relative to single-device execution. We see that in the kernel 3×3 and filter depth 128, smaller input depths have no speedup. This is because the amount of computation per device after the distribution is small. However, for the large input depths, since the amount of computation after the distribution is more balanced, we see a speedup. Furthermore, in most cases, spatial splitting performs better. This is because, contrary to other methods, spatial splitting has less communication overhead. However, for larger 9×9 kernels, since the number of overlapping elements increases, the advantage of spatial splitting compared to other methods decreases.

V. WORK DISTRIBUTION

To understand why distributing and parallelizing DNN computations are necessary for IoT devices, Fig. 13 shows the memory usage and time to process an input (i.e., latency) of some layers in C3D and VGG16. As seen, `fc` layers of both models have an extremely large memory footprint that causes long latency (in order of minutes, not shown). For `fc` layers, this large memory footprint and low compute intensity activate the use of swap space. This behavior is true for almost all visual models since after extracting visual features using `conv`, these models need to flatten the features for classification. Such conversion from visual features to categorical features, which are implemented with `fc` layers,

[1]For simplicity, we divide each dimension to equal parts here. In our implementations, any number of divisions is possible.

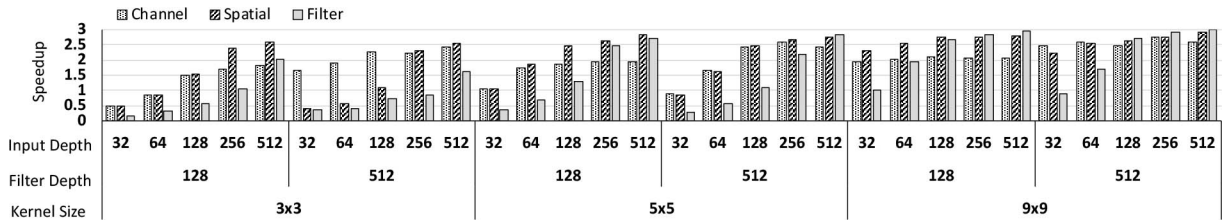

Fig. 12. Convolution model-parallelism methods performance—performance comparisons of model-parallelism methods for conv distributed on three RPis. Speedup is measured against a single RPi. As seen, depending on input depth, filter depth, and kernel size, the best distribution method varies.

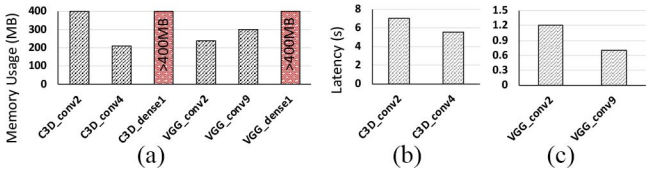

Fig. 13. Per-layer memory and latency—memory usage and latency of some layers in VGG16 and C3D models on an RPi during an inference.

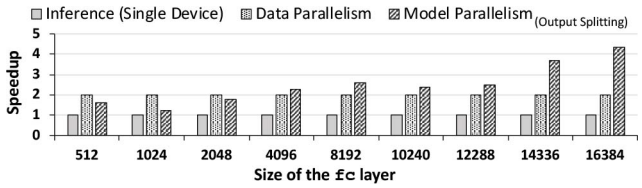

Fig. 14. Model and data parallelism performance—speedup of model and data parallelism for fc layers, normalized over single device, with different sizes and input size of 7680 on two RPis [8].

causes high memory usage with low computational density. Hadidi *et al.* [8] performed an experiment, the result of which is shown in Fig. 14 that shows model parallelism has higher performance benefit than data parallelism for these layers. These results, with results in Fig. 9, show how distribution achieves higher performance.

As discussed, applying model parallelism is necessary for layers with a large memory footprint, such as first fc layers [Fig. 13(a)], to bypass swap space usage. conv, on the other hand, has a much smaller memory footprint; but with a few layers on a device, we will eventually exceed the available memory of the device and face the same issue. For conv, it is also possible that the latency of a single layer is long and not suitable for real-time processing because of its large computation load. To this end, we present some examples in Fig. 13 that show the latency of some convolution layers in VGG16 and C3D models. As illustrated, even for a single layer, the latencies are not suitable for real-time processing. In addition, as shown in Fig. 12, we see that model parallelism is able to provide us with a performance benefit. Therefore, in contrast with the previous work [8], which has not analyzed model-parallelism methods for conv, we found such model-parallelism methods to be useful in DNN models.

Note that most DNN models have more than ten layers, and until now, as examples, we have shown only the statistics for single layers. The mentioned challenges are exacerbated with more layers. In summary, the total latency of executing the entire model on a single device is longer because the latencies of all layers are accumulated because there are no parallelization opportunities. Model-parallelism methods help us to solve these challenges because they reduce the memory footprint and exploit more compute resources by introducing parallelism among devices.

Model- and data-parallelism methods help us to distribute and parallelize the DNN computations. But, how can we find a near-optimal distribution for a given number of devices? The distributed system that we study is essentially a processing pipeline for the DNN model; each device processes a part of the computation and offloads the rest to the next devices. Our goal is to find a distribution that achieves a near-the-optimal number of IPS (higher is better) or the lowest latency (lower is better). In general, if we have $\mathcal{W}$ amount of work and n workers, our speedup compared to a single node case is

$$\text{Speedup} = \frac{\mathcal{W} + \text{overhead}_{\text{single}}}{\mathcal{W}/n + \text{overhead}_{\text{pipeline}}} \tag{3}$$

in which the $\text{overhead}_{\text{pipeline}}$ entails communication overhead ($\propto$ data size), and some fixed overhead such as the network set-up time between devices. Similarly, $\text{overhead}_{\text{single}}$ shows the overhead associated with the single-node execution, such as swap space activities. If the communication overhead dominates our distribution, and single-device execution does not have significant overhead, we experience a slowdown after the distribution. Several examples of such layer configurations can be found in Fig. 12. To avoid such scenarios, we need to: 1) avoid unnecessary distribution to reduce the amount of communication overhead and 2) associate enough work per node so the benefit of parallelizing exceeds the communication overhead. To do so, we merge less compute-intensive layers on a single node. As an online load-balancing technique, we also monitor idle nodes and combine the layers to increase the utilization of each node, thereby achieving a balanced pipeline. However, if the $\text{overhead}_{\text{single}}$ is significant, such as swap memory activities in fc layers, in an acceptable range of $\text{overhead}_{\text{pipeline}}$, we experience speedups with distribution, as observed in Figs. 9 and 14.

Generating a Balanced Pipeline: To create a near-optimal distribution, the latency of each device should be similar to that of other devices. Thus, the amount of work per device, or $\mathcal{W}/n$, should be the same. Model parallelism helps us gain access to smaller granularities of work during distribution. On the other hand, data parallelism does not change the amount of work per device, but increases the throughput. With model parallelism, the throughput of a task increases, so the effective latency seen

Procedure 1 Heuristics for Distributing a DNN Model

procedure GENERATEDISTRIBUTION
Inputs: list of layers $\mathbb{L}$, #Nodes n
mem_{size}: Memory size per node
Regression models or profiling database, $\mathbb{D}$
Outputs: dictionary of node IDs to a set of its tasks, $\mathbb{T}$
Step1: Check memory usage all layers in $\mathbb{L}$ using $\mathbb{D}$, if larger than mem_{size}, add that layer to the model parallelism list.
Step2: Using latency of layers in $\mathbb{D}$ and their split version, and by ensuring sequential dependency of layers, try to create groups of layers with the same latency. Create $\mathbb{T}$.
Step3: Deploy $\mathbb{T}$. Monitor queue occupancy and latency on each device. Goto Step2.

TABLE IV
RPI 3 SPECIFICATIONS

CPU	1.2 GHz Quad Core ARM Cortex-A53
Memory	900 MHz 1 GB RAM LPDDR2
GPU	No GPGPU Capability

TABLE V
NVIDIA JETSON TX2 SPECIFICATIONS [42]

CPU	2.00GHz Quad Core ARM Cortex-A57 2.00GHz Dual Denver 2
Memory	1600 MHz 8 GB RAM LPDDR4
GPU	Pascal Architecture - 256 CUDA Core

by the next devices decreases. By considering these properties, to generate a distribution, first, we create a database with a mix of: 1) regression models based on the amount of work and type of layers and 2) profiled data from some layers and their split versions (similar to the results in Section IV). Then, we study our given DNN model layer by layer. If the memory footprint is large and causes swap activities, for that layer, we have to first use model parallelism. After that, we try to group fewer compute-intensive and sequential layers to reduce the communication overhead. The grouping is done in a way that the average latency for processing an input on each device would be similar. After deploying such an initial distribution, we monitor the queue occupancy and latency of each device. With these gathered new data, we repeat the above steps to tune the distribution in runtime by creating a more balanced pipeline. Procedure 1, in $O(n)$, summarizes these steps. The initial execution time (number of iterations of the procedure) until the system adjusts the performance depends on the complexity of the model. For the model in this article, it takes less than 5 min, or around 25 iterations.

To give an unbiased view, the limitations of this approach are the following. First, for the initial deployment, there should be some initial measurements close to the size of each layer. Second, our procedure currently is evaluated in systems with the same type of devices. Third, devices might lose some data points when the system is dynamically configured for a new distribution. Finally, this article is focused on DNN models for computer vision tasks. Note that although discussions in this article are about the execution of a single DNN model, one can extend our methods to multiple concurrent DNN models. However, the user needs to ensure that the system can handle the computation loads of concurrent models either by introducing more devices or designing reactive event-based systems.

VI. SYSTEM EVALUATION

We evaluate our method on a distributed system with RPi [20]. Table IV presents the specifications of an RPi. To provide a comparison reference, we also execute DNN models on Nvidia Jetson TX2 [42], the specifications of which are in Table V. TX2 is a high-performance embedded platform with both a CPU and GPU with 8 GB DDR4. In contrast, RPis are an edge device with no GPU and less than 1 GB DDR2. On an RPi distributed system, to show how our distribution heuristics provide better performance, we compare our results with a distributed system that deploys simple pipelining. A simple pipeline ensures correctness, but would not necessarily be an optimal design. For our implementations, we created a software stack with Docker containers. We use Keras 2.1 [43] with the TensorFlow backend (version 1.5) [44]. For RPC calls and serialization, we use Apache Avro [45]. We use an IP table file to assign tasks to each device. A local WiFi network with the measured bandwidth of 94.1 Mb/s and a measured client-to-client latency of 0.3 ms for 64 B is used. All trained weights are loaded to each Pi's storage (16 GB storage in our system), so each Pi can be assigned to execute any part of a layer. More details on the implementation can be found in [23]. Note that each Pi has an SD card storage, for storing the weights, which is relatively inexpensive compared to the main memory. If local storage is limited, the assigned weight can also be shared in the network from a network-storage filesystem. This approach makes a tradeoff between how fast the switching time between different models can be and per-device storage usage.

After finding a distribution of computations, we create a single file containing a Python dictionary of the IP addresses and their assigned computation. We upload the file to all devices, and each device, by reading the model description and its assigned computation, finds its position in the pipeline. After handshaking, which takes less than 1 min, the system is ready. During runtime, each device reports its latency and request queue occupancy. By collecting such status, we are able to find bottleneck devices in our pipeline and create a more balanced pipeline, as Procedure 1 describes.

AlexNet and VGG16: We deploy AlexNet and VGG16 models, including the last `fc` layers, on various distributed systems. Since the first `fc` layer in AlexNet faces a limited memory issue on an RPi, all of our distributions perform output splitting for this layer. The rest of the `conv` are allocated to idle devices. Our two near-optimal systems have four and six devices and achieve higher than 2× speedups compared to distributed systems with a simple pipeline [Fig. 15(b)]. Because AlexNet layers all have low computation requirements, we could not get more benefit by distributing the computations. Fig. 15(a) presents a more detailed performance measurement for AlexNet. Compared with TX2 with a GPU and CPU, the six-device distribution has a higher performance.

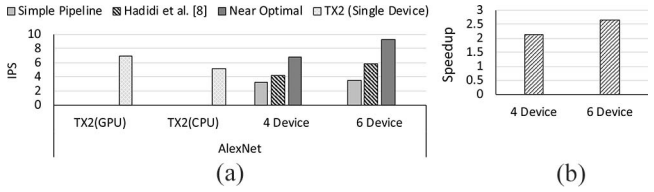

Fig. 15. AlexNet deployment results. (a) IPS. (b) Speedup over the near-optimal case for four and six devices.

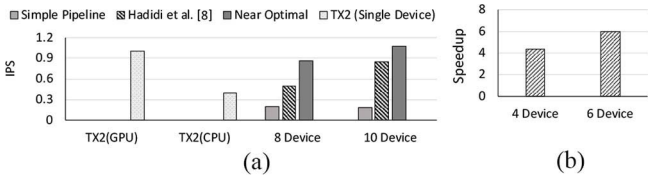

Fig. 16. VGG16 deployment results. (a) IPS. (b) Speedup over the near-optimal case for four and six devices.

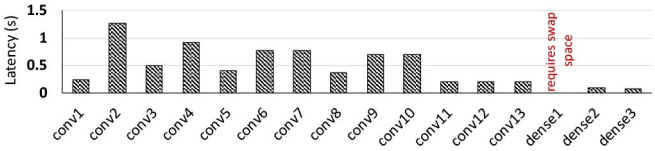

Fig. 17. VGG16 layer-wise latency—VGG16 measured layer-wise latency on an RPi for an inference.

The VGG16 model consists of more computationally intensive layers compared to the layers of AlexNet. Therefore, we use eight and ten devices for distribution to achieve up to 6×speedup compared to the simple pipelining scenario, as shown in Fig. 16(b). Moreover, as shown in Fig. 16(a), with more performance details, both of our distributions have higher performance than TX2 CPU. Our ten-device distribution also achieves similar performance to TX2 GPU. It is worth noting that both of our near-optimal distributions have higher performance than the TX2 CPU and a simple pipelining scenario. Similar to AlexNet, since we include the first `fc` layer, all of our distributions perform output splitting for this layer. For other layers, to gain a better insight, in Fig. 17, we measured the layer-wise latency of VGG16 layers that are executed on RPi. Except for the first `fc` layer, we are able to run all other layers on a single RPi. But, some layers have extremely long latencies, so we are bounded by such layers in our simple pipelining scenario (e.g., second `conv`). On the other hand, in our eight- and ten-device systems with the near-optimal distribution, we bypass this bottleneck by using the model-parallelism methods for `conv`, that are proposed in Section IV.

C3D: The C3D model, as discussed in Section III-B, incorporates 3-D `conv`. To understand this model behavior, we analyze the layer-by-layer latency of C3D models on the RPi in Fig. 18(a). As shown, the first layers of C3D are quite heavy for IoT devices. For instance, the latency of the second `conv` is 18 s. This high latency is caused by the high computational demands of 3-D convolutions. Model-parallelism methods for `conv` are particularly useful in distributing this among all devices. We apply our three methods of model parallelism

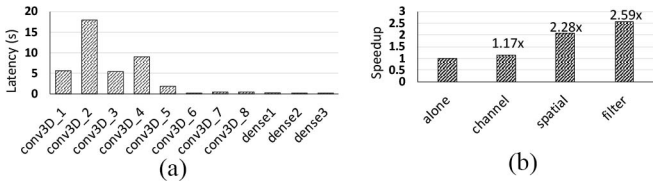

Fig. 18. C3D Results I. (a) C3D layer-wise latency of a single inference. (b) Achieved performance speedup after applying model-parallelism methods on the heaviest layer (`conv3D_2`).

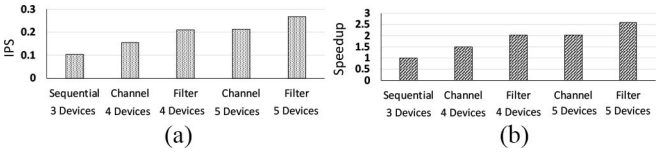

Fig. 19. C3D Results II—performance of C3D first three layer deployments on various systems. (a) IPS. (b) Speedup over three-device sequential.

on three devices for the second (heaviest) `conv`. As seen, we attain up to a 2.6×speedup by using three devices for this layer. Note that the spatial- and filter-splitting methods achieve higher performance than the channel-splitting method. This is because the size of the input is large, and therefore, methods such as channel splitting, which does not divide the input, have a high overhead for communicating the copies to all devices, whereas, both spatial- and filter-splitting methods have a lower overhead due to the split input.

To get an estimation of the overall performance of C3D, we select the heaviest layers of the C3D model (`conv3D_2`, `conv3D_2`, and `conv3D_4`) and deploy them on a distributed system using our heuristics. The first system, our baseline, is simply the sequential execution. By introducing extra devices, our heuristics split the computations of `conv3D_2`, similar to Fig. 18(b). The results for both filter- and channel-splitting methods for four and five devices are shown in Fig. 19. As shown, with a higher number of devices, the performance gain also increases. In all variations, the filter-splitting method, as observed in Fig. 12 and discussed in the previous paragraph, achieves higher performance than channel splitting.

ResNet50 and Xception: The ResNet50 and Xception models have similar building blocks, as shown in Figs. 5–6. For practical reasons of the limited number of devices, we choose to experiment with Xception. Since the building blocks of both models are similar, our observations are extendable to ResNet models as well. We measure the layer-wise latency of layers in Xception during single-batch inferencing, shown in Fig. 20. As seen, in comparison with AlexNet and VGG16, for which the final `fc` layers were the most compute-intensive and resource-hungry layers, in Xception, some `conv` are more compute intensive and resource hungry. To better understand the aggregated processing time for Xception, we measured the total latency of different blocks in Xception (as shown in Fig. 6), when they are executed on a single RPi. Fig. 21 depicts the measured latencies. As seen, block C has the longest latency among other blocks. Since Xception is a large model, we

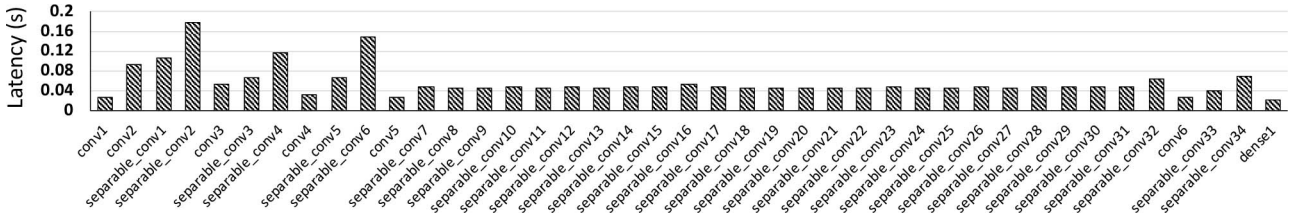

Fig. 20. Xception layer-wise latency—Xception measured layer-wise latency on an RPi for a single inference.

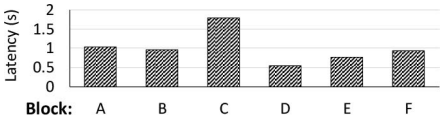

Fig. 21. Xception blockwise latency—execution latency of Xception per block on an RPi during a single inference.

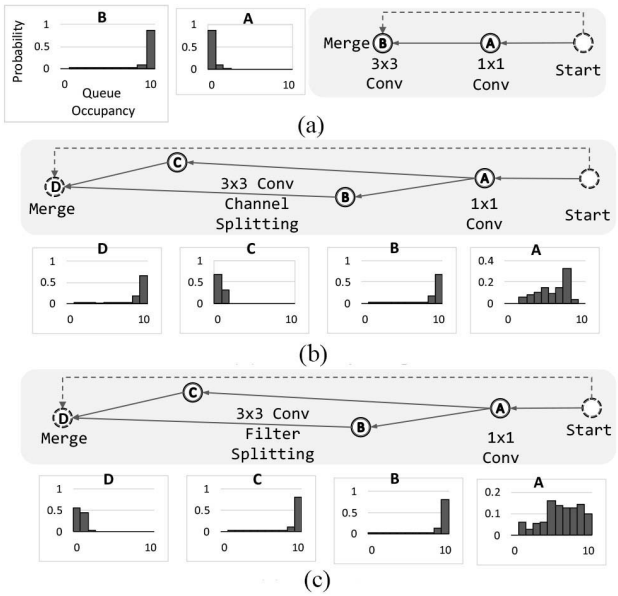

Fig. 22. Reported queue occupancy for Xception—systems executing Xception block C (see Fig. 6) in (a) sequential, (b) channel splitting on two devices, and (c) filter splitting on two devices modes. Comparing with the performance results presented in Fig. 23, monitoring tools help us solve the performance bottlenecks of the system online.

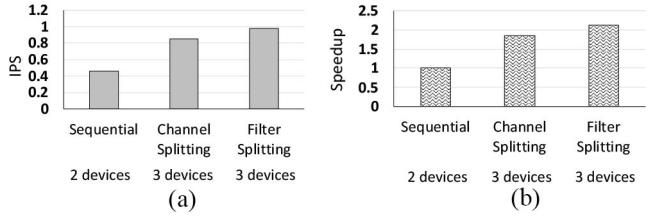

Fig. 23. Experiments on Xception block C. (a) IPS. (b) Performance speedup for the systems shown in Fig. 22, consisting of multiple RPis.

deployed only one block in our system. We chose the heaviest block (i.e., block C) and deployed it on three different systems, shown in Fig. 22.

The system shown in Fig. 22(a) shows a simple sequential distribution, in which each device processes a layer. Fig. 22(b) shows a system that uses channel-splitting method for the heaviest `conv` in block C. Similarly, Fig. 22(c) illustrates a system in which the heaviest `conv` in the block C is distributed using the filter-splitting method. The performance comparisons of these systems are shown in Fig. 23. As seen, by including another device, our system can achieve up to a $2\times$ speedup.

Fig. 22 also depicts the queue occupancy of the devices that is extracted from our monitoring tools. The histograms in the figure show the queue occupancy of the devices. Note that queue size per device is limited to ten requests. As seen, in Fig. 22(a), the queue of device B is always full. Therefore, our heuristics apply splitting to the work that is performed in device B. Fig. 22(b) and (c) shows such splitting for the channel- and filter-splitting methods, respectively. Although we still see a close-to-full occupancy for devices B and C, which perform the split job, we observe that device A occupancy has shifted to the right. This shows that our method was successful in creating a more balanced work distribution, but did not have enough available devices to create the best distribution. Note that as discussed, our heuristics have access to a database of similar experiments that are done in Fig. 12. Therefore, it does not need to perform both splittings to find the best performing one. Here, we are showing both as an example.

VII. Conclusion

In this article, we proposed several new model-parallelism methods for single-batch inferences of DNNs. We focused on DNNs for visual applications that consist mostly of CNN-based models. As discussed in this article, with the aid of these methods, we can move the computations of DNNs closer to the edge and IoT devices. These methods divide the memory and computation footprint of DNN models and distribute them among several devices. We deployed our heuristics for several state-of-the-art visual DNN models while measuring their performance on a cluster of RPis. We planed to extend this article to heterogeneous nodes by introducing IoT-tailored cluster managing tools such as Kubernetes [46]. As another direction for future work, we plan to extend this article to more than visual DNNs, such as long short-term memories (LSTMs), covering areas, such as translation and speech recognition. Furthermore, we are studying the possibility of various methods in alleviating the communication overhead such as bypassing the dependencies between the layers, compression, and using coded distribution [47].

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015.

[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, 2013.

[3] M. L. Merck *et al.*, "Characterizing the execution of deep neural networks on collaborative robots and edge devices," in *Proc. ACM Practice Exp. Adv. Res. Comput. Rise Mach. Learn. (PEARC)*, 2019, pp. 1–6.

[4] R. Hadidi *et al.*, "Characterizing the deployment of deep neural networks on commercial edge devices," in *Proc. IISWC*, 2019, pp. 35–48.

[5] Gartner Inc. (2015). *Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015*. Accessed: Dec. 2, 2019. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2015-11-10-gartner-says-6-billion-connected-things-will-be-in-use-in-2016-up-30-percent-from-2015

[6] F. Biscotti *et al.*, *The Impact of the Internet of Things on Data Centers*, vol. 18, Gartner Res., Stamford, CT, USA, 2014.

[7] I. Lee and K. Lee, "The Internet of Things (IoT): Applications, investments, and challenges for enterprises," *Bus. Horizons*, vol. 58, May 2015, pp. 431–440.

[8] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Distributed perception by collaborative robots," *IEEE Robot. Autom. Lett.*, vol. 3, no. 4, pp. 3709–3716, Oct. 2018.

[9] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "ERIDANUS: Efficiently running inference of DNNs using systolic arrays," *IEEE Micro*, vol. 39, no. 5, pp. 46–54, Sep./Oct. 2019.

[10] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding," in *Proc. Int. Conf. Learn. Represent.*, 2016. [Online]. Available: arXiv:1510.00149.

[11] Y. Gong *et al.*, "Compressing deep convolutional networks using vector quantization," 2014. [Online]. Available: arXiv:1412.6115.

[12] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on CPUs," in *Proc. NIPS*, vol. 1, 2011, pp. 1–8.

[13] *Compiling AI for the Edge*, Ofer Dekel Microsoft Res., Redmond, WA, USA, 2019.

[14] J. Devlin *et al.*, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018. [Online]. Available: arXiv:1810.04805.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Neural Inf. Process. Syst.*, 2012, pp. 1106–1114.

[16] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Represent.*, 2015. [Online]. Available: arXiv:1409.1556.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[18] F. Chollet, "XCeption: Deep learning with depthwise separable convolutions," 2016.

[19] D. Tran, L. D. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3D convolutional networks," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 4489–4497.

[20] Raspberry Pi Foundation. (2017). *Raspberry Pi 3*. Accessed: Dec. 2, 2019. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[21] J. Mao, X. Chen, K. W. Nixon, C. D. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Proc. Design Autom. Test Europe*, 2017, pp. 1396–1401.

[22] S. Teerapittayanon, B. McDanel, and H. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 328–339.

[23] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Real-time image recognition using collaborative IoT devices," in *Proc. ReQuEST Workshop ASPLOS*, 2018, p. 4.

[24] Y. Kang *et al.*, "NeuroSurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2017, pp. 615–629.

[25] R. Hadidi *et al.*, "Musical chair: Efficient real-time recognition using collaborative IoT devices," 2018. [Online]. Available: arXiv:1802.02138.

[26] Microsoft. (2017). *Embedded Learning Library (ELL)*. Accessed: Dec. 2, 2019. [Online]. Available: https://microsoft.github.io/ELL/

[27] Google. (2017). *TensorFlow Lite*. Accessed: Dec. 2, 2019. [Online]. Available: https://www.tensorflow.org/mobile/tflite/

[28] Nvidia. *NVIDIA TensorRT*. Accessed: Dec. 2, 2019. [Online]. Available: https://developer.nvidia.com/tensorrt

[29] Google. (2019). *Edge TPU*. Accessed: Dec. 2, 2019. [Online]. Available: https://cloud.google.com/edge-tpu/

[30] Nvidia. (2019). *Jetson Nano*. Accessed: Dec. 2, 2019. [Online]. Available: https://www.developer.nvidia.com/embedded/jetson-nano-developer-kit

[31] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017. [Online]. Available: arXiv:1704.04861.

[32] M. Tan *et al.*, "MnasNet: Platform-aware neural architecture search for mobile," 2018. [Online]. Available: arXiv:1807.11626.

[33] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Proc. Neural Inf. Process. Syst.*, 2017, pp. 2181–2191.

[34] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *Proc. Int. Symp. Comput. Archit.*, 2017, pp. 535–547.

[35] J. Guo, S. Yin, P. Ouyang, L. Liu, and S. Wei, "Bit-width based resource partitioning for CNN acceleration on FPGA," in *Proc. IEEE Symp. Field Program. Custom Comput. Mach.*, 2017, p. 31.

[36] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplication," 2014. [Online]. Available: arXiv:1412.7024.

[37] U. Köster *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Proc. Neural Inf. Process. Syst.*, 2017, pp. 1742–1752.

[38] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," 2016. [Online]. Available: arXiv:1605.04711.

[39] M. Courbariaux *et al.*, "Binarized neural networks: Training deep neural networks with weights and activations constrained to $+1$ or -1," 2016. [Online]. Available: arXiv:1602.02830.

[40] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.*, 2016, pp. 525–542.

[41] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.

[42] NVIDIA. (2017). *Nvidia Jetson TX2*. Accessed: Dec. 2, 2019. [Online]. Available: https://developer.nvidia.com/embedded/jetson-tx2-developer-kit

[43] F. Chollet *et al.* (2015). *Keras*. [Online]. Available: https://github.com/fchollet/keras

[44] M. Abadi *et al.* (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. [Online]. Available: https://www.tensorflow.org/

[45] TAS Foundation. (2017). *Apache AVRO*. Accessed: Dec. 2, 2019. [Online]. Available: https://avro.apache.org

[46] R. Hadidi *et al.*, "An edge-centric scalable intelligent framework to collaboratively execute DNN," in *Proc. SysML Demo*, 2019, p. 2.

[47] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Robustly executing DNNs in IoT systems using coded distributed computing," in *Proc. ACM DAC*, 2019, p. 234.