Adaptive Parallel Execution of Deep Neural Networks on Heterogeneous Edge Devices

Li Zhou
zholi@cse.ohio-state.edu
The Ohio State University

Mohammad Hossein Samavatian
samavatian.1@osu.edu
The Ohio State University

Anys Bacha
bacha@umich.edu
University of Michigan–Dearborn

Saikat Majumdar
majumdar.42@osu.edu
The Ohio State University

Radu Teodorescu
teodores@cse.ohio-state.edu
The Ohio State University

ABSTRACT

New applications such as smart homes, smart cities, and autonomous vehicles are driving an increased interest in deploying machine learning on edge devices. Unfortunately, deploying deep neural networks (DNNs) on resource-constrained devices presents significant challenges. These workloads are computationally intensive and often require cloud-like resources. Prior solutions attempted to address these challenges by either introducing more design efforts or by relying on cloud resources for assistance.

In this paper, we propose a runtime adaptive convolutional neural network (CNN) acceleration framework that is optimized for heterogeneous Internet of Things (IoT) environments. The framework leverages spatial partitioning techniques through fusion of the convolution layers and dynamically selects the optimal degree of parallelism according to the availability of computational resources, as well as network conditions. Our evaluation shows that our framework outperforms state-of-art approaches by improving the inference speed and reducing communication costs while running on wirelessly-connected Raspberry-Pi3 devices. Experimental evaluation shows up to 1.9× ∼ 3.7× speedup using 8 devices for three popular CNN models.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; *Massively parallel algorithms*; • **Computer systems organization** → *Embedded hardware.*

KEYWORDS

deep learning, edge devices, parallel execution, inference

ACM Reference Format:
Li Zhou, Mohammad Hossein Samavatian, Anys Bacha, Saikat Majumdar, and Radu Teodorescu. 2019. Adaptive Parallel Execution of Deep Neural Networks on Heterogeneous Edge Devices. In *SEC '19: 4th ACM/IEEE Symposium on Edge Computing, November 7–9, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3318216.3363312

SEC '19, November 7–9, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6733-2/19/11…$15.00
https://doi.org/10.1145/3318216.3363312

1 INTRODUCTION

Deep neural networks (DNNs) are rapidly becoming indispensable tools for solving complex problems that include computer vision [28], natural language processing [10], machine translation [5], and many others. Advancements in hardware [15, 23, 36] and lightweight frameworks [4, 45] are making the deployment of machine learning algorithms to new environments possible. In addition, various emerging applications are driving the need for deploying machine learning algorithms to edge devices in smart homes, smart cities, autonomous vehicles, and healthcare [6, 31, 35]. However, in most cases, the bulk of the computation, even for inference problems, is performed entirely within the cloud or through a hybrid combination of edge and cloud computing. In other words, inputs are collected on edge devices, but the processing is mostly offloaded to powerful servers in the cloud.

Cloud-based processing of user-generated data faces several challenges and limitations. For instance, uploading user data to the cloud raises privacy concerns. Consumers are becoming increasingly aware of the privacy implications associated with online services and are likely to be more concerned about devices uploading audio and video data to the internet for further processing. Examples include baby monitors or other in-home cameras, voice assistants, etc. Furthermore, many IoT applications require frequent decision making that render cloud-based computing impractical due to the communication latency it brings.

In response to the aforementioned concerns, efforts have been made to push machine learning inference from the cloud to the edge. Edge processing has the benefit of keeping data closer to its source to provide real-time responses while protecting the privacy of the end-user [17, 29]. Unfortunately, edge computing for machine learning workloads faces challenges of its own. Most machine learning algorithms are computationally demanding making edge devices inadequate for handling such workloads due to their constrained performance, energy, and memory capacities. To this end, a significant amount of research has investigated efficient approaches for deploying DNNs to the edge. This includes collaborative computation between edge devices and the cloud [18, 26, 44], model compression and parameter pruning [11, 19, 46, 48], or customized mobile implementations [21, 23, 36, 50]. Despite all these efforts,

having the ability to scale existing DNNs without sacrificing the model accuracy and processing the collected data streams in real time present ongoing challenges to the deployment of machine learning across edge devices.

In this work, we explore parallel execution of DNN inference across multiple heterogeneous devices that are energy-constrained. This doesn't require extra model tuning efforts, or new hardware deployment. A possible application that we envision for our work is local smart home processing. Instead of relaying collected data that includes voice commands, sensor readings, and video streams from a camera to the cloud or a local edge server/cloudlet, our solution leverages other smart home devices, such as speakers, light switches, and hubs to resolve the request. This approach creates a number of research challenges that need to be addressed: (1) how to partition the workload efficiently across devices; (2) how to optimize execution across heterogeneous devices possessing different compute capabilities; (3) how to account for the higher communication latency in wirelessly connected devices.

To answer these questions, we develop a framework for partitioning DNN inference in an optimal way across multiple heterogeneous devices. A simple model of the available devices and their compute capabilities is generated first. Next, our framework uses parameterized performance prediction models for multiple DNN layer types. The framework uses these models to predict the execution time for each layer based on available devices and communication latency, and selects the best partition points and parallelization strategies for each partition. To accommodate the heterogeneity of the computing resources, partition sizes are chosen to match device capabilities. Then at runtime, the partitions are distributed and executed across multiple devices. The framework is deployed and evaluated on the VGG-16 and ResNet-50 image recognition models [20, 42], and the YOLOv2 object detection model [39], achieving speedups of 1.9× ~ 3.7×, relative to the models running on a single device.

Some prior work [17, 33] explored parallelizing DNN inference on edge devices. Their approach, however, relies on layer-granularity parallelism that result in large amounts of intermediate feature maps that are transferred across devices. When communication bandwidth is low, this significantly impacts performance. Other work [51] proposed fusing some of the DNN layers to reduce the communication overhead. Their approach is, however, limited to fusing the first several layers in the DNN. Our work generalizes the problem of layer fusion and proposes a mechanism for choosing the groups of DNN layers to be fused in an optimal way. This is also the first work we are aware of that considers device heterogeneity in this context.

In this work, we mainly focus on solving the three above mentioned challenges. More sophisticated issues include how to design a framework for different types of hardware or OS, how to adapt to the dynamic changes of device compute capabilities and network latency. We leave these questions for future work. Overall, this paper makes the following contributions:

- We discuss the trade-offs in partitioning DNN inference among multiple lightweight edge devices, and propose an Adaptive Optimal Fused-layer (AOFL) parallelization to reduce the communication cost in an IoT network.

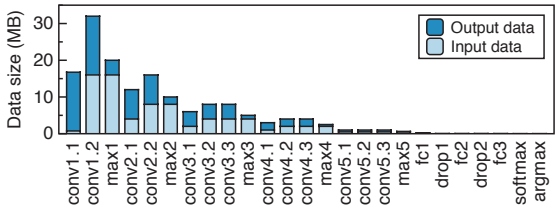


Figure 1: The per-layer data size of input and output in VGG-16.

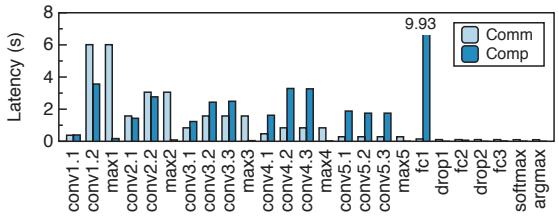


Figure 2: The per-layer latency of input communication and layer computation in VGG-16 with a single-core of Raspberry-Pi3 running at 1 GHz in a 25 Mbps local WiFi network.

- We design a dynamic programming-based search algorithm to decide the optimal partition and parallelization for a DNN model based on the compute capabilities and networks, The algorithm works for both edge-edge and edge-cloud collaborations.
- We present a collaborative CNN acceleration framework that adapts to the computing resources and network condition in the presence of heterogeneity of compute capabilities.
- We apply our technique on a distributed IoT cluster consisting of Raspberry-Pi3-based hardware and evaluate image recognition and object detection DNN models.

The rest of this paper is organized as follows: Section 2 provides background information. Section 3 explores different parallelization strategies in IoT devices and their trade-offs. Section 4 describes our approach for finding near-optimal parallelization. Section 5 presents the results of our evaluation. Section 6 details the related work and Section 7 concludes.

2 BACKGROUND

In this section, we give an overview of CNNs and popular DNN models that we use in the evaluation section.

CNN layers. A convolutional neural network consists of an *input*, an *output* layer, and multiple hidden layers. The hidden layers typically consist of a stack of convolution layers, normalization layers, ReLU layers (i.e., activation functions), pooling layers, and fully-connected layers. The **Convolution (*conv*) layer** is the core building block of a CNN. It has of a set of learnable filters (kernels), which convolve across the spatial dimension (width and height) of the input volume and generate a 2-dimensional feature map for each filter by computing the dot product between their weights

and a small sub-region of the input. As a result, each layer generates a successively higher level abstraction of the input data. The **Batch normalization layer** normalizes features across spatially grouped feature maps to reduce internal covariate shift. To increase non-linearity, an **activation layer** applies an element-wise activation function such as rectified-linear unit (*relu*) to the input data, which removes negative values from a feature map by setting them to zero. Both layers are less compute intensive and are often optimized to compute with a previous operator. As such, we group them with the corresponding previous layer that produces their input. The **Pooling layer** (e.g., max pooling (*max*)), performs a down-sampling operation along the spatial dimensions. It is used to progressively reduce the spatial size of the representation, number of parameters, memory footprint and amount of computation in the network. Hence, it also controls overfitting. Finally, after several convolution and max pooling layers, the high-level reasoning in the neural network is done via a dense or **fully-connected (fc) layer**. Neurons in a fully-connected layer are connected to all activations in the previous layer. The value of each output is calculated from the weighted sum of all inputs. The **Softmax** and **argmax** layers produce a probability distribution over the classes for classification and select the one with highest probability as the prediction. fc and *conv* layers are among the most compute- and data-intensive layers of a CNN model.

Models overview. VGG-16 was the runner-up in the ImageNet ILSVRC challenge [40] in 2014. Most importantly, it showed that the depth of the network is a critical component for good performance. The network contains 16 $conv/fc$ layers and features a homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from beginning to end. A downside of VGG-16 is that it uses a lot more memory and parameters in the first fully-connected layer. **Residual neural network (ResNet)** [20] features special skip connections and makes heavy use of batch normalization. It introduces a so-called "identity shortcut connection" that skips one or more layers to overcome the vanishing gradient issue, enabling much deeper networks with good performance. **You only look once (YOLO)** [38] is a real-time object detection system that is tasked with determining the location of certain objects on a given image, as well as classifying those objects. Similar to ResNet, YOLO is also missing heavy fully-connected layers at the end of the network.

Key observations. We use VGG-16 to highlight some of the computation and communication characteristics that are typical in CNN models. Figures 1 and 2 show the per-layer input and output data sizes, as well as the input communication and layer computation latency. This data was collected by executing each VGG-16 layer across two Raspberry-Pi3 devices that had a single active core running at 1 GHz and communicated over a 25 Mbps local WiFi network. Overall, we make the following observations from this experiment:

(1) Convolution layers dominate the computation time. In VGG-16, 73.8% of the total computation time is spent on *conv* layers. For other models without heavy fc layers at the end of the network like YOLOv2, *conv* layers consume up to 99.93% of the computation time.

(2) IoT environments generally rely on wireless communication, which can be slow due to network delays or slow on-device networking hardware. As a result data transfers can be more costly than computation for certain layers as shown in Figure 2.

(3) The first few layers in the network generate the most output data and are therefore the most communication-intensive. In VGG-16, the cumulative input data size of the first 6 layers represents 66.7% of the total input data size. Figure 1 shows the data consumed and produced by each layer in VGG-16, ordered by depth from left to right. We can see that the relative amount of input/output data transferred between layers decreases substantially for deeper layers.

Based on the aforementioned observations, we focus on improving the performance of the convolution layers through parallelization. In addition, device-to-device communication should be avoided for the first few layers of the CNN model where the input and output data sizes are large. Finally, when parallelizing *conv* layers across multiple IoT devices, we need to carefully consider the trade-off between the reduced computation time and the increased communication cost.

3 DISTRIBUTING AND PARALLELIZING DNN INFERENCE AT THE EDGE

We envision the deployment of our system in an environment such as a smart home, in which devices of different types and compute capabilities collaborate to solve a joint task. This creates a number of research challenges that need to be addressed: (1) how to partition the workload efficiently between devices; (2) how to factor in the heterogeneity in compute capabilities of devices; (3) how to account for the higher communication latency in devices that connect wirelessly.

Figure 3 provides an overview of the approach we take to solve these challenges. First, a simple model of the available devices and their compute capabilities is generated. Next, our framework uses parameterized performance prediction models for multiple DNN layer types. At runtime, the framework predicts the execution time for each layer based on available devices and communication latency, and selects the best partition points and parallelization strategies for each partition. To accommodate the heterogeneity of the compute environment, partition sizes are chosen to match device capabilities. Then, the partitions are distributed and executed across multiple devices.

3.1 Model Parallelism and Partitioning

In this work we employ model parallelism to partition DNN inference among multiple lightweight edge nodes. Model parallelism subdivides the DNN parameters into partitions that can be assigned to multiple devices. Each partition generates a subset of the output feature maps. The partial outputs are then aggregated to form the final output for each layer. This approach can reduce computation latency for a single input and is therefore a good fit for parallelizing machine learning inference.

Partitioning methods. Figure 4 shows two partitioning methods that can be used to parallelize a 2-dimensional convolution layer: (1) *channel partitioning* and (2) *spatial partitioning*. As shown in

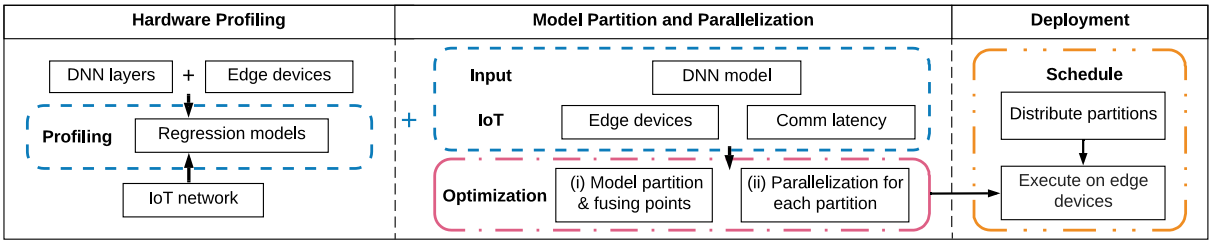


Figure 3: Design overview.

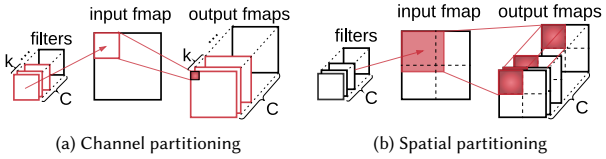


Figure 4: Examples of parallelizing a 2D convolution layer (note that, each filter and its corresponding input/output feature map have the same channel size and are not shown in the plots).

Figure 4a, in a *conv* layer each filter generates a feature map (i.e, a channel of the output feature maps). The output feature maps can be partitioned along the *channel* dimension such that each device computes a subset of the output feature maps. This requires mapping the corresponding set of filters to each device. The input maps have to be replicated across all the devices. Channel partitioning is generally more beneficial for training because it reduces communication costs associated with synchronizing network parameters. The need to fully replicate inputs can, however, add substantial communication overhead making channel partitioning less practical for inference.

An alternative approach is to partition the output feature maps spatially (i.e., by height and width) and assign them to multiple devices. Each device keeps a copy of the network and computes a subset of the output feature maps. As Figure 4b shows, compared with channel partitioning where the entire input has to be transferred to all devices, in spatial partitioning each device only requires a subset of the input. This greatly reduces communication costs in edge networks that rely on wireless communication.

Note that due to the nature of the convolution operation, for filter sizes that are greater than 1, input partitions are not completely disjoint. Each partition has to be extended by $\lfloor f_i/2 \rfloor$ (f_i is the size of filters of layer i) along both dimensions to include inputs from neighboring partitions that are required in the computation of the output partition.

3.2 Tradeoffs in Parallelizing CNNs in IoT Devices

We now introduce two parallelization strategies used in our framework to parallelize a DNN model.

Layer-wise parallelization. Prior work [27] has shown that the best parallelization strategy depends on the characteristics of the DNN (i.e., layer type, shape of feature maps and size of filters). As a result, layer-wise parallelization [25] has been proposed to allow each layer to be parallelized independently using the appropriate technique for each layer to obtain the best performance. However, in layer-wise parallelization, each device computes part of the output of the current layer and all the subsets of the output need to be merged and re-partitioned before the execution of next layer. This requires output to be gathered by a host node, partitioned and re-sent to all client devices. In a wireless network this results in substantial communication overhead, as we have shown in Figure 2. For our system, the benefits of layer-wise parallelization are generally defeated by the communication costs. We use layer-wise parallelization as a baseline for comparison.

Fused-layer parallelization. To reduce the data movement between layers, we propose using fused-layer parallelization. The concept of layer fusion was first proposed in [2] as a method to reduce off-chip data movement in a CNN accelerator. The idea is to send the output of one layer directly to the input of the next layer without going through memory. We propose extending this concept by parallelizing multiple fused layers instead of single layers individually.

Figure 5 illustrates this concept on the VGG-16 network. A layer-wise parallelization example is shown in Figure 5(a) for layers $conv2.1$ and $conv2.2$. The input maps for $conv2.1$ are partitioned by processor P_1 and assigned to processors $P_1 - P_n$ for computation. The outputs are then gathered at P_1, merged, partitioned and assigned as inputs to $P_1 - P_n$ in order to compute $conv2.2$.

Figure 5 (b) shows an example of fused-layer parallelization. Two convolution layers ($conv5.1$ and $conv5.2$) are parallelized as a single fused-layer block. Partitioning is performed layer-by-layer starting from the last layer in the fused block. Each layer's input is the output of the previous layer. In this example, both layers are divided into 2x2 disjoint subsets. The required input elements for each partition are calculated based on its output elements. For *conv* layer, we also need to extend each partition's input by $\lfloor f_i/2 \rfloor$ on height and width for overlapping elements. The process is applied recursively up to the first layer in a fused block ($conv5.1$ in our example). Note that the overlap of the input partitions (highlighted in purple in Figure 5) leads to some redundant computation as well as additional communication overhead. As the number of fused layers increases so does the overlap in the input feature map partitions. This is

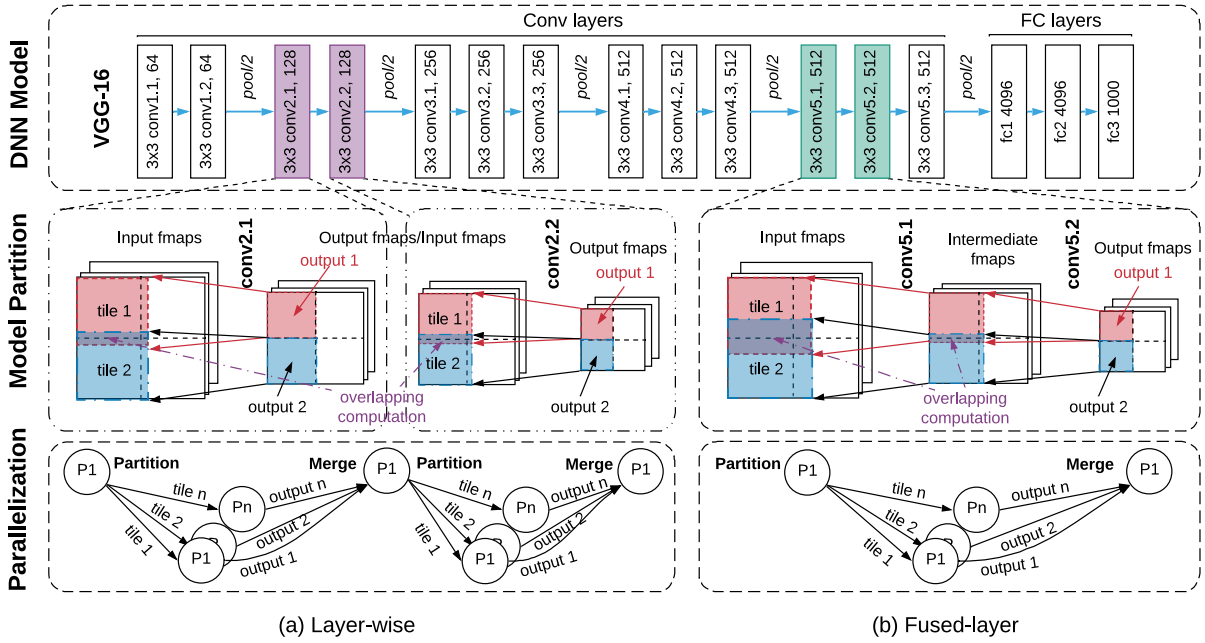
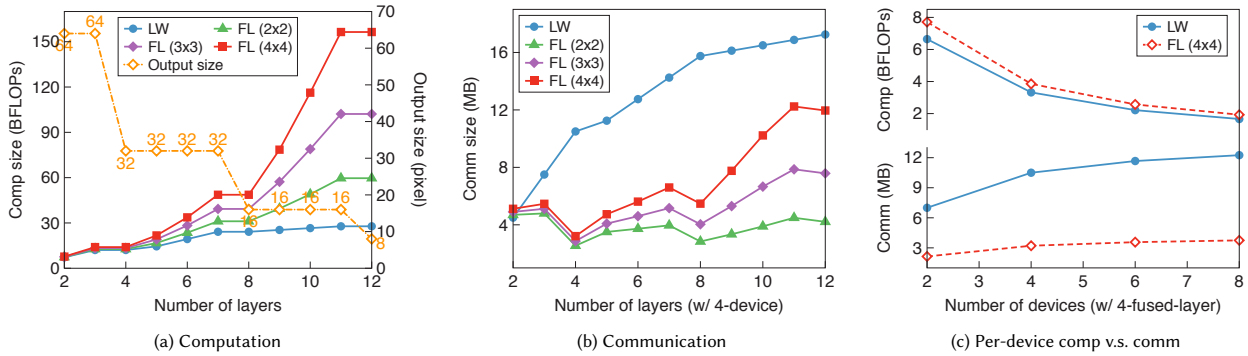


Figure 5: Illustration of layer-wise parallelization (a) vs layer fusion (b) for VGG-16.



(a) Computation (b) Communication (c) Per-device comp v.s. comm

Figure 6: Comparison of (a) computation size (BFLOPs) and (b) communication size (MB) for Layer-wise (LW) and Fused-layer (FL) parallelization over 12 layers starting from *conv*2.1 of VGG-16. (c) compares reduced per-device computation size and communication size for 4 layers on 4 devices.

because each input partition has to include all the input elements required to compute the last output partition of the fused block.

The partitions are next distributed to processors $P_1 - P_n$ for computation. All convolution layers in the fused block will now be computed locally and only the output of the last layer will be merged at P_1. This reduces communication costs because only the input of the first layer and the output of the last layer need to be communicated between devices. The more layers are fused, the more communication costs are reduced. However, layer fusion introduces additional costs compared to layer-wise partitioning. The overlap in input partitions adds to the communication cost of distributing those partitions, and adds redundant computation to each node. This cost increases with the number of fused layers. As a result, finding the optimal number of layers to be included in a fused block requires carefully balancing the costs and benefits of layer fusion.

Computation vs. communication trade-off. Figure 6 compares computation and communication costs for layer-wise (LW) and fused-layer (FL) parallelization in VGG-16. Beginning at *conv*2.1, fused-layer parallelization fuses between 2 and 12 layers at different partitioning granularities (i.e., 4 (2x2), 9 (3x3), and 16 (4x4)). Computation load is represented in billions floating point operations

per second (BFLOPs). We can see that computation increases as expected with the number of layers. When the number of fused layers is small (2-4) the amount of computation performed by FL and LW is very close. This is because the overhead of redundant computation is small as the first couple of layers have a relatively large spatial output dimension (64x64). The output dimension is shown by the yellow line in Figure 6a. As the number of fused layers increases, and the output spatial dimension decreases ($\leq$ 16x16) the cumulative computation overhead increases dramatically, up to $3\times \sim 5\times$ when fusing 12 layers. It also leads to more overlapping elements in the first input layer which increases communication costs.

The communication costs of FL are approximated using a 4-device setup by measuring input and output data sizes of a fused block. In LW, the communication cost is calculated as the cumulative input data size of each layer. Figure 6b highlights communication costs for the two techniques measured as the amount of data transferred between devices, as a function of the number of layers. We can see that in all cases, FL results in substantially lower communication overhead compared to LW. For instance, when fusing 8 layers, the communication cost of LW is 3× higher than that of FL. This is because FL data is transferred only for the first and last layer in the fused block, while LW requires device-to-device data transfers between all layers.

The degree of parallelism (i.e., number of parallel devices) is another factor that affects performance. Different layers or fused blocks, have different computation needs and communication costs, which affect the optimal degree of parallelism. Our target environment is especially sensitive to communication cost which often limits the optimal number of devices. Figure 6c shows the estimated per-device computation and total communication costs for a 4-fused-layer block with different number of devices. As the number of devices increases, the per-device computation decreases while the total communication overhead increases. The communication overhead for FL increases more slowly with the number of devices compared to LW, and it is also lower overall. This suggests that FL is likely exhibit better scalability with the number of devices.

Deploying fused-layer parallelization in our environment in an optimal way requires answering the following questions: (1) which layers should be fused, (2) how many layers to include in each fused block, (3) for each fused and unfused block, how many partitions/devices offer the optimal performance and (4) how to match the partition size to the capability of the device in a heterogeneous environment?

4 ADAPTIVE FUSED-LAYER PARTITION AND PARALLELIZATION

Answering the aforementioned questions requires solving a multivariate optimization problem. We present a dynamic programming based search algorithm to find the parameter values that are projected to achieve the lowest execution time under our cost model.

4.1 Problem Definition

Given a CNN model $\mathbb{G}$ with n layers, where $l_i \in \mathbb{G}$ is a layer in the model and edge (l_i, l_j) is a tensor that is an output of layer l_i and an input of layer l_j. The model runs on a list of devices $\mathbb{D}$, and we assume the communication bandwidth of each connection between two devices (d_i, d_j) is known. For layer-wise parallelization, a parallelization strategy $\mathcal{S}^{lw}$ includes a configuration c_i for each layer l_i which consists of a combination of parameters from {*height*, *width*, *channel*}, and a subset of devices $\mathcal{D} \in \mathbb{D}$. For layer fusion we add the grouping of layers as an additional dimension to the optimization. For each fused block, a parallelization strategy $\mathcal{S}^{fl}$ is generated. The optimization objective is to find a parallelization strategy $\mathbb{S} = \mathcal{S}^{lw} \bigcup \mathcal{S}^{fl}$ such that the execution time $\mathcal{T}(\mathbb{G}, \mathbb{D}, \mathbb{S})$ is minimized.

Table 1: Symbol definitions.

Symbol	Description
$\mathbb{G}$	A CNN model with n layers
$\mathbb{D}$	A list of devices, with known computation abilities and communication latency
$\mathbb{S}$	Partition and parallelization configurations for all layers in model $\mathbb{G}$
$\mathcal{G}^{lw}$, $\mathcal{G}^{fl}$	Layers in model $\mathbb{G}$ using layer-wise or fused-layer parallelization, $\mathbb{G} = \mathcal{G}^{lw} \bigcup \mathcal{G}^{fl}$
$\mathcal{S}^{lw}$, $\mathcal{S}^{fl}$	Partition and parallelization configurations for layers using layer-wise or fused-layer parallelizations, $\mathbb{S} = \mathcal{S}^{lw} \bigcup \mathcal{S}^{fl}$
$l_i, l_{(i,j)}$	Layer(s) in model $\mathbb{G}$, $l_i, l_{(i,j)} \in \mathbb{G}$
$e = (l_i, l_j)$	A tensor e from layer l_i to layer l_j
d_i	A device in IoT cluster, $d_i \in \mathbb{D}$
c_i	A parallelization configuration for layer i, $c_i \in \mathbb{S}$
$t_l(i)$	The cost function for layer l_i using layer-wise parallelization
$t_f(i, j)$	The cost function for a fused block that fuses j consecutive layers from layer i

4.2 Cost Model

We develop a performance model that will be used to guide the optimization search. To construct the model for each layer type, we vary the configuration parameters of the layer and measure the latency for each configuration. Using the profiles, we build a regression model for each layer type to predict execution latency. To predict the communication cost, we use a similar approach by varying the data transfer size and measuring the latency. We define two basic cost functions:

- For each layer l_i and its parallelization configuration c_i from {*height*, *width*, *channel*}, $t_c(l_i, c_i, \mathcal{D})$ is the time to process layer l_i under configuration c_i on devices $\mathcal{D} \in \mathbb{D}$. This only includes the inference time which is estimated by processing the layer under the configuration multiple times on the devices and measuring the average execution time.
- For each $e = (l_i, l_j)$, $t_x(e, d, k)$ is time to transfer the tensor e between two devices d and k using the size of the data and the known communication bandwidth. $t_x(e, d, \mathcal{D}) = \sum_{k \in \mathcal{D}} t_x(e_k, d, k)$ is the total time of transferring the tensor e_k from a local device d to remote devices $k \in \mathcal{D}$.

The cost functions for each layer l_i in layer-wise parallelization is then defined as the total time of the input/output tensors transfer and layer computation:

$$t_l(i) = t_x(e_{in}, d, \mathcal{D}) + t_x(e_{out}, d, \mathcal{D}) + t_c(l_i, c_i, \mathcal{D}) \quad (1)$$

Using this cost function, we define

$$\mathcal{T}(\mathbb{G}, \mathbb{D}, \mathcal{S}^{lw}) = \sum_{l_i \in \mathbb{G}} t_l(i) \quad (2)$$

$\mathcal{T}(\mathbb{G}, \mathbb{D}, \mathcal{S}^{lw})$ estimates the execution time of a single inference under layer-wise parallelization strategy $\mathcal{S}^{lw}$.

Next, assuming several consecutive convolution layers are grouped under fused-layer parallelization strategy $\mathcal{S}^{fl}$, where $\mathcal{S}^{lw}(i)$ of layer l_i in the fused block will be replaced with $\mathcal{S}^{fl}(i, j)$. The cost function for a fused block $l_{(i,j)} \in \mathcal{G}^{fl}$ (i.e., fusing j consecutive layers from layer i) is then defined as the total data transfer time of the first layer's input tensor, the last layer's output tensor, and the sum of the computation time of all grouped layers, running on $\mathcal{D}$ devices:

$$t_f(i, j) = t_x(e_{in}, d, \mathcal{D}) + t_x(e_{out}, d, \mathcal{D}) + \sum_{i \le k < i+j} t_c(l_k, c_k, \mathcal{D}) \quad (3)$$

Next we define

$$\mathcal{T}(\mathbb{G}, \mathbb{D}, \mathbb{S}) = \sum_{l_i \in \mathcal{G}^{lw}} t_l(i) + \sum_{l_{(i,j)} \in \mathcal{G}^{fl}} t_f(i, j) \quad (4)$$

where $\mathcal{T}(\mathbb{G}, \mathbb{D}, \mathbb{S})$ estimates the total execution time of a single inference for a model $\mathbb{G} = \mathcal{G}^{lw} \bigcup \mathcal{G}^{fl}$ on a list of devices $\mathbb{D}$ under a hybrid layer-wise/fused-layer parallelization strategy $\mathbb{S} = \mathcal{S}^{lw} \bigcup \mathcal{S}^{fl}$.

4.3 Dynamic Programming-based Optimization

The optimization goal for our framework is finding parallelization strategy $\mathbb{S}$ that minimizes the runtime $\mathcal{T}(\mathbb{G}, \mathbb{D}, \mathbb{S})$. Given the relatively small optimization space, we use dynamic programming-based search through the space of possible solutions.

We start by determining the optimal parallelization configuration for each layer, and each fused block. We find the optimal $\mathcal{S}^{lw}$ and the optimal $\mathcal{S}^{fl}$ with two tables of cost $t_l(.)$ and $t_f(.)$ with a list of devices $\mathcal{D}$ by varying the used number of devices from 1 to the maximum. Then we use the following dynamic programming based search algorithm to find the optimal placement of fused blocks in a model.

We define the *fused-layer partitioning problem* as follows. Given a CNN model $\mathbb{G}$ with n layers and tables of cost $t_l(.)$ and $t_f(.)$, determine the minimum cost $t_o(i, j)$ (equivalent to $\mathcal{T}_o(\mathbb{G}, \mathbb{D}, \mathbb{S})$ when $i = 0, j = n$) that can be achieved through layer fusion.

We can partition a n-layer model in 2^{n-1} ways, since we have an independent option of fusing, or not fusing, at distance j from the first layer, for $j = 1, 2, ..., n$. We denote a decomposition into parts using ordinary additive notation. If an optimal solution partitions the model into k parts, for some $1 \le k \le n$, then an optimal decomposition $n = i_1 + i_2 + ... + i_k$ of the model into fused layers $i_1, i_2, ..., i_k$, for a minimum time:

$t_o(0, n) = t_f(0, i_1) + t_f(i_1, i_2) + ... + t_f(\sum_{1 \le j \le k-1} i_j, i_k)$.

Algorithm 1 Adaptive Optimal Fused-layer (AOFL) Parallelization Strategy Search Algorithm

Input $\mathbb{G}$: a CNN model with n layers
$\mathbb{D}$: a list of devices
$\mathcal{S}^{lw}(.)$ and $\mathcal{S}^{fl}(.)$: precomputed parallelizations
$t_l(.)$ and $t_f(.)$: precomputed cost functions
Output $\mathbb{S}$: a parallelization strategy that minimizing $\mathcal{T}(\mathbb{G}, \mathbb{D}, \mathbb{S})$
$t_o(.) \leftarrow MAX,\ fl_o(.) \leftarrow 1$
function AOFL(i, j)
  if $t_o[i][j] < MAX$ **then**
    return $t_o[i][j]$
  if $j = 0$ **then**
    $t_{min} \leftarrow 0,\ l_{opt} \leftarrow 0$
  else if $j = 1$ **then**
    $t_{min} \leftarrow t_l(i),\ l_{opt} \leftarrow 1$
  else
    $t_{min} \leftarrow MAX$
    for $k \in [1, j]$ **do**
      $t \leftarrow t_f(i, k) + AOFL(i + k, j - k)$
      if $t < t_{min}$ **then**
        $t_{min} \leftarrow t,\ l_{opt} \leftarrow k$
  $t_o[i][j] \leftarrow t_{min},\ fl_o[i] \leftarrow l_{opt}$
  return $t_o[i][j]$
function BuildStrategy($\mathcal{S}^{lw}(.)$, $\mathcal{S}^{fl}(.)$, $fl_o(.)$)
  $i \leftarrow 0$
  while $i < n$ **do**
    if $fl_o(i) = 1$ **then**
      Add $\mathcal{S}^{lw}(i)$ to $\mathbb{S}$
    else
      Add $\mathcal{S}^{fl}(i, fl_o(i))$ to $\mathbb{S}$
    $i \leftarrow i + fl_o(i)$
  return $\mathbb{S}$

More generally, we can frame the optimal value $t_o(0, n)$ for $n \ge 1$ in terms of optimal cost from fusing layers:

$t_o(0, n) = \min(t_f(0, n),\ t_o(0, 1) + t_o(1, n - 1),\ t_o(0, 2) + t_o(2, n - 2), ...,\ t_o(0, n - 1) + t_o(n - 1, 1))$.

The first argument, $t_f(0, n)$, corresponds to fusing all the layers of the model. The other $n - 1$ arguments correspond to the minimum time obtained by making an initial partitioning of the model into two groups of layers i and $n - i$, for each $i = 1, 2, ..., n - 1$, and then recursively searching for the optimal subpartions of those layer groups. We may view every decomposition of a j-layer part starting from layer i as fusing the first part followed by some decomposition of the reminder, and simplify the general equation for dynamic programming as shown below:

$$t_o(i, j) = \begin{cases} 0 & j = 0, \\ t_l(i) & j = 1, \\ \min\limits_{1 \le k \le j} (t_f(i, k) + t_o(i + k, j - k)) & otherwise. \end{cases} \quad (5)$$

Algorithm 1 shows the pseudocode of our optimization algorithm which uses dynamic programming with memoization to find out

Algorithm 2 Update Cost Functions $t_l(.)$ and $t_f(.)$

Input $\mathbb{G}$ and $\mathbb{D}$
Output $t_l(.)$, $t_f(.)$ and $\mathcal{S}^{lw}(.)$, $\mathcal{S}^{fl}(.)$
function UPDATE($\mathbb{G}, \mathbb{D}$)
  $\mathcal{D} \leftarrow \emptyset,\ t_l(.) \leftarrow MAX,\ t_f(.) \leftarrow MAX$
  Rank $\mathbb{D}$ by $freq$ and then by bw between source dev
  ▷ Iterate $\mathbb{G}$ as follow:
  for $i \in [0, n-1]$ **do**
    for $j \in [1, n-i]$ **do**
      $\mathcal{D}_l(.) \leftarrow \emptyset,\ \mathcal{D}_f(.) \leftarrow \emptyset$
      for $d \in \mathbb{D}$ **do**
        $\mathcal{D}' \leftarrow \mathcal{D} \bigcup d$
        Adjust the size of partitions in c_i, if applicable
        if $j = 1$ **then**
          Predict $t_l(i)^{\mathcal{D}'}$ with regression model
          if $t_l(i)^{\mathcal{D}'} < t_l(i)$ **then**
            $t_l(i) \leftarrow t_l(i)^{\mathcal{D}'}, \mathcal{D}_l(i) \leftarrow \mathcal{D}'$
            Update $\mathcal{S}^{lw}(i)$ with c_i and $\mathcal{D}_l(i)$
        Predict $t_f(i,j)^{\mathcal{D}'}$ with regression model
        if $t_f(i,j)^{\mathcal{D}'} < t_f(i,j)$ **then**
          $t_f(i,j) \leftarrow t_f(i,j)^{\mathcal{D}'}, \mathcal{D}_f(i,j) \leftarrow \mathcal{D}'$
          Update $\mathcal{S}^{fl}(i,j)$ with c_i and $\mathcal{D}_f(i,j)$

the optimal parallelization strategy. Function *AOFL* computes the minimum time and a list of optimal number of fused layers starting from a given layer. The optimal parallelization strategy is built up through function *BuildStrategy* by iterating the model from the beginning with steps of a list of optimal number of fused layers, and adding the corresponding parallelization configuration to $\mathbb{S}$.

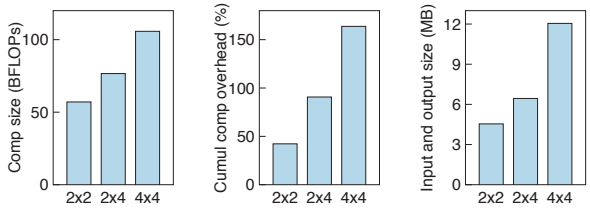
Optimizing for heterogeneity. In order to adapt to the heterogeneity in compute capabilities of the IoT network, we need to balance the workload assigned to each device such that they all finish execution at roughly the same time. To that end, the algorithm ranks participating devices $\mathbb{D}$, first by clock frequency ($freq$) then by network bandwidth (bw) between the source and destination devices. The two cost tables of $t_l(.)$ and $t_f(.)$ are updated using function *Update* described in Algorithm 2. Within a fused block, the computation and communication costs of each partition are functions of its input size $\mathcal{P}$, and the execution time of each partition can be represented as

$$t_f(.)_i = t_{comm}(\mathcal{P}_i) + t_{comp}(\mathcal{P}_i) \tag{6}$$

where $i = 1, 2, .., N$. The ratio of input sizes of two partitions in a fused block then can be calculated with the regression models by setting $t_f(.)_i$'s equal. To obtain the best performance, a top device is used when predicting the execution time when adding one more device.

4.4 Implementation

We use Darknet [37] with NNPACK [13] as backend inference engine kernel to execute convolution layer, and implement a distributed framework integrated with network communication modules using TCP/IP with socket.



(a) Total comp size (b) Total comp overhead (c) Total comm size

Figure 7: An example of total computation size (BFLOPs), computation overhead (%), and communication data size (MB) with same fused blocks in VGG-16 at three different partitioning granularities.

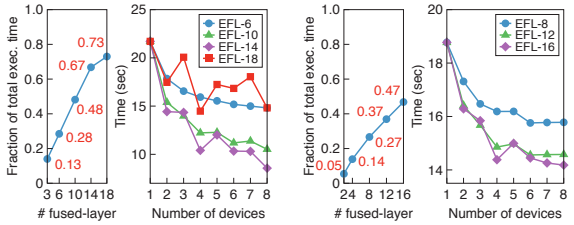
Linear regression performance model. We used a linear regression model to predict the runtime of different convolution layers with different configurations and various input shapes based on the partitioning scheme. The model is trained on samples with different convolution related parameters like input size and channel, number of filters, size of the filters, padding and stride. We randomly selected appropriate values from the range of interest for each parameter. We run each sample and get its runtime to generate the training and test sets for linear regression.

Deployment and device mapping. Assuming a CNN model is divided into blocks, and each block includes one or more fused layers respectively. The local device that captures the input (e.g., a video camera or smart speaker) is responsible for partitioning the input tensor, and distributing the partitioned input among devices. Note that it is possible that the optimal configuration for some blocks is to run on a single device, in which case they will be deployed on a single node. The remote devices start to execute once a task is received and send the results back once the job is completed. All results will be merged at the local device and re-partitioned for the next block.

Reconfiguration. The performance of edge devices in IoT networks fluctuates. Our system periodically recalculates the optimal partition points to adapt to changes in computational capabilities and network conditions. Once there is a change in the model parallelization, we adjust the configurations of partitions and reschedule them. Since the proposed algorithm is lightweight, it only takes about several seconds to recalculate the optimal partition points and reconfigure the framework. Note that the interval for recalculating the optimal partitions should be carefully selected to avoid performance degradation due to rescheduling overhead. We leave this exploration to future work.

5 EVALUATION

In this section, we conduct experimental evaluations to validate the effectiveness and robustness of the proposed Adaptive Optimal Fused-layer (AOFL) parallelization. We first explore the performance trade-off at different partitioning granularities and compare the runtime of different parallelization strategies. We then demonstrate the robustness of our solution to heterogeneous network and processing conditions.



(a) VGG-16 (b) YOLOv2

Figure 8: Performance of Early Fused-layer (EFL) parallelization with different number of fused layers.

5.1 Experimental Setup

Workloads. We focus on parallelizing a total of 13 convolution layers from VGG-16 and 23 layers from YOLOv2, accounting for 73.8% and 99.3% of their total execution time, respectively.

Testbed. We build two IoT clusters for running our experiments using the Raspberry-Pi3 B+ model. Each Raspberry-Pi3 B+ consists of a 1.4 GHz Quad Core ARM Cortex-A53, 1 GB LPDDR2 SDRAM and dual-band 2.4 GHz/5 GHz wireless. The first cluster (Cluster-1) is dedicated to performance evaluation and consists of 8 single core nodes that are fixed to run at 1 GHz in order to represent a realistic low-end edge device cluster. The second cluster (Cluster-2) is configured to represent a heterogeneous edge environment. We use it to evaluate the framework's ability to deal with heterogeneous devices that have different core frequencies between 400 MHz and 1 GHz. In addition, to evaluate the impact of different network conditions, we test with three network configurations consisting of two routers that support 2.4 GHz/5 GHz WiFi. (1) WiFi-1, a fast network with a measured bandwidth of 93.7 Mbps. (2) WiFi-2, a medium speed network with a measured bandwidth of 62.7 Mbps. (3) WiFi-3, a slow network with the measured bandwidth of 25.1 Mbps. Finally, we assume that each device reserves enough memory before it processes any assigned tasks and that all the trained weights are loaded into each device's storage.

Schemes. We compare four different parallelization strategies in our evaluation: (1) Layer-wise (LW) parallelization, which parallelizes the networks layer by layer; (2) Early Fused-layer (EFL) parallelization, an extension of to the implementation of DeepThings [51], which fuses and parallelizes the first few convolution layers of a model and executes the remaining layers in a single device; (3) Optimal Fused-layer (OFL) parallelization, which selectively fuses convolution layers at different parts of a model; (4) Adaptive Optimal Fused-layer (AOFL) parallelization, which extends OFL by dynamically adapting to the available computing resources and network condition in an IoT network.

5.2 Partitioning Granularity in Fused-layer Parallelization

The performance of fused-layer parallelization depends on partitioning configurations that include the points of interest for layer fusion in a CNN model, the partitioning granularity, and the number of devices. Given the fused blocks in a model, the partitioning granularity of each block affects the computation and communication size. In our analysis, we quantify computation in billion floating-point operations per second (BFLOPs), and communication in terms of transferred data size in megabytes (MB). For an IoT network where the computational power of devices and the network bandwidth are known, the partitioning granularity represents a qualitative measure of the computation to communication ratio.

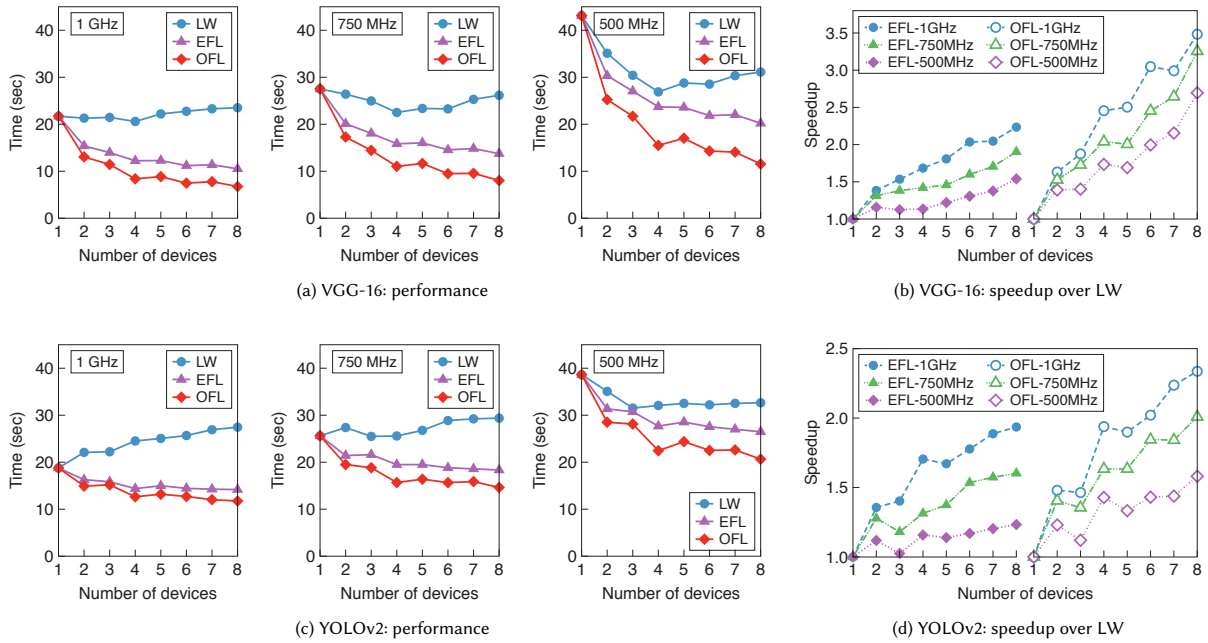
Figure 7 shows an example of the impact of different partitioning granularities on computation and communication with two fused blocks in VGG-16. The first seven and last six convolution layers in the model are fused. Finer granularity results in lower average computation, however its total computation is increased, as well as the computation overhead. Increasing the granularity from 2x2 to 2x4, the average computation size is reduced by 35.4%, but the total computation size is increased by 34.3% as shown in Figure 7a. Because finer granularity creates more redundant computation from overlaps among partitions. Figure 7b shows increasing the granularity from 2x4 to 4x4 increases total overhead from 90.7% to 163.7%.

Figure 7c shows similar trend for communication size. We observe that with the same fused blocks, the difference of total transferred data size between two partitioning granularities is from the cumulative overlaps among partitions in the input layer of fused blocks. Switching granularity from 2x2 to 2x4, the communication size is decreased by 29.1% on average but increased by 41.8% in total.

We test the three partitioning granularities with Cluster-1 and WiFi-1, and observe that granularity of 2x2 achieves the best performance on 4 devices since it has the least computation and communication cost. Overall, enabling more devices requires increasing the number of partitions. As such, our design adapts to different partitioning granularities by adjusting the fused blocks to reduce the computation cost. For instance, at granularity of 2x4 on 8 devices, breaking the second fused block into two smaller blocks with 3 convolution layers in each can effectively reduce the computation and communication cost by 32.3% and 4.4% respectively. In general, the choice of the partitioning granularity is a trade-off between computation and communication which varies based on the available computational resources and the condition of the network. Our design uses Algorithm 1 to determine the optimal parallelization configurations.

5.3 Runtime Performance

We compare the execution time of our Optimal Fused-layer (OFL) parallelization with two baselines: Layer-wise (LW) and Early Fused-layer (EFL) parallelizations on 1 to 8 devices for VGG-16 and YOLOv2. We set the partitioning granularity to be the same number of used devices. To find the optimal fusing point for EFL, we vary the number of fused layers from the beginning of each model. As shown in Figure 8, the performance of EFL initially improves when the number of fused layers and devices is increased. However, EFL doesn't scale well with deeper CNNs. We observe that fusing 14 and 18 layers in VGG-16 results in similar execution times. The same is observed with fusing 12 and 16 layers in YOLOv2. In some cases, the performance may even degrade if too many layers are fused. For instance, fusing 18 layers in VGG-16 runs slower than



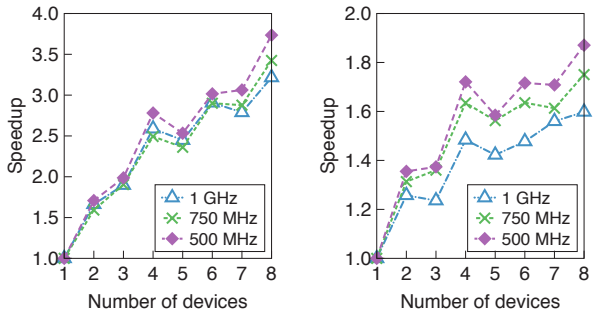
(a) VGG-16: performance

(b) VGG-16: speedup over LW

(c) YOLOv2: performance

(d) YOLOv2: speedup over LW

Figure 9: Performance of different parallelizations running VGG-16 and YOLOv2 at different frequencies. The speedup is compared with Layer-wise (LW) parallelizaiton at the corresponding frequency.



(a) VGG-16

(b) YOLOv2

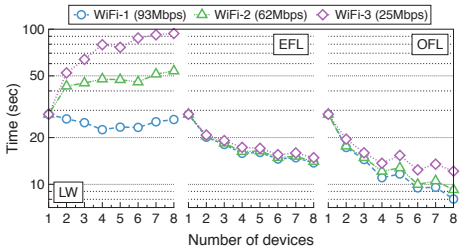
Figure 10: Speedup of Optimal Fused-layer (OFL) over a single edge device in WiFi-1 (93 Mbps).

fusing 6 layers. Another drawback of EFL relates to the fraction of layers that can be fused within a given model. EFL only works on convolution layers, leaving 27% ∼ 33% of VGG-16 and 54% ∼ 63% of YOLOv2 execution serialized.

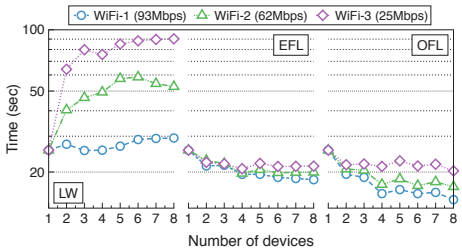
Figure 9 shows the execution time of LW, EFL and OFL at different frequencies. In LW, the communication cost increases linearly with the number of devices due to the per layer data distribution and processing synchronization. The latency reaches a minimum on 4 devices across different frequencies in VGG-16. However, the latency increases in YOLOv2 at higher frequencies when using more devices since the model has lightweight convolution layers. Overall, the performance gain from parallelizing computation is easily canceled out because of the the high communication cost. As such, we conclude that LW is not suitable for networks that have a low computation to communication ratio.

By contrast, EFL and OFL reduce the communication cost through layer fusion. Despite the computation overhead, both of them scale their performance with more devices. For example, at 750 MHz EFL reduces the latency by 27.7% ∼ 47.5% on 2 to 8 devices relative to LW. OFL further reduces the latency by 14.1% ∼ 41.5% on 2 to 8 devices relative to EFL. We observe that OFL always performs better than EFL for the same partitioning granularity due to its flexibility in fusing layers. Unlike EFL, OFL is able to harness parallelization for deeper layers in a model and adapt the fused blocks as the number of devices changes. As a result, OFL is 1.2× ∼ 1.7× faster than EFL. We also observe that OFL shows better improvement over EFL with slower frequencies. This makes it more suitable for low-end edge devices.

Increasing the number of available devices to boost performance requires the consideration of two factors. First, the execution time is reduced as more devices contribute to the computation. However, this increases the communication time since more data is sent over the network. Second, the overhead introduced by overlapping partitions also increases since we need to use finer partitioning granularity. Overall, as Figures 9 (b) and (d) show, OFL exhibits much better scalability with the number of devices compared to EFL, especially for VGG-16.



(a) VGG-16



(b) YOLOv2

Figure 11: Execution time under different WiFi settings.

5.4 Analysis of Optimal Partitioning Strategies

We analyze the optimal layer fusion configurations under our cost model for VGG-16 and YOLOv2 and observe that OFL fuses more layers from the beginning CNN layers that have large input and output sizes as the number of devices increases. It uses the added compute power to offset the increased computation overhead and achieve large reduction in communication. For instance, we observe that the first 10 and 16 layers in both models are fused respectively. Second, deeper layers in a CNN model have smaller height/width dimensions but a larger channel dimension. This introduces higher computation overhead due to the overlapping of partitions with finer granularity. As a result, OFL fuses a smaller number of layers to balance the computation overhead and communication cost in deeper layers. In addition, large fused blocks are broken into smaller blocks to reduce the computation overhead when the frequency of the devices is decreased.

5.5 Robustness

We examine the robustness of our proposed Adaptive Optimal Fused-layer (AOFL) parallelization in terms of scalability, network conditions and heterogeneity.

Scalability. Figure 10 shows the scalability of Optimal Fused-layer (OFL) parallelization as a function of devices with different frequencies in WiFi-1. The speedup relative to a single device increases as more devices are added, but is limited by the high communication cost. For VGG-16, OFL runs 1.6× ~ 3.7× faster on 2 to 8 devices than a single device. The speedup with YOLOv2 is less and ranges from 1.3× ~ 1.9×. This is because the computation of convolution layers in YOLOv2 are lightweight and the communication represents a larger fraction of the runtime. The speedup is also improved when the frequency drops and the ratio of computation increases.

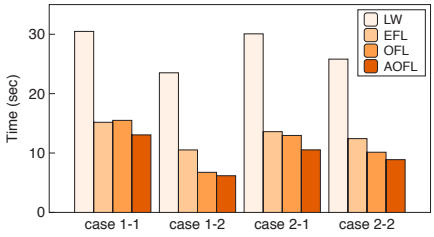


Figure 12: Execution time on different configurations of the heterogeneous Cluster-2.

For instance, with 8 devices, the speedup increases from 3.2× to 3.7× and 1.6× to 1.9× in each model when doubling the frequency from 500 MHz to 1 GHz. This shows that OFL scales better with higher computation to communication ratio, and can obtain higher speedup with less powerful devices. It is especially suitable for computation-intensive models that operate in a slow network, as well as low-end edge devices.

Impact of the network. Figure 11 shows how our framework adapts to changes in the network conditions. LW is heavily impacted by changes in the bandwidth. The execution time increases by 59.8% and 86.4% on average when the network slows down from 93 Mbps to 25 Mbps (WiFi-1 to WiFi-3). On the other hand, EFL is not sensitive to the network conditions. An average slow down of 4.8% is measured when switching from the faster network to the slow one (WiFi-1 to WiFi-3). Since EFL only requires distributing and collecting inputs and outputs once, the change of the communication cost due to different network conditions has a lesser effect on the total execution time. Finally, OFL experiences latency increases in 14.3% increments as the network is slowed down. However, it still performs better than EFL despite the higher communication cost. This also indicates that OFL takes better advantage of faster networks by distributing the computation of more layers in a model in order to improve performance.

Impact of heterogeneity. In these experiments, we focus on the heterogeneity in computational ability that can lead to inconsistent progress among the workers and slow down the speed of the inference. We use different device speeds available in the heterogeneous Cluster-2 in order to evaluate the robustness of our method in the presence of heterogeneity.

Table 2: Case study for heterogeneous IoT.

Cases	Devices in Custer 2: *#Devices* x *Frequency* x *#Cores (= 1)*
1-1	2 x 1 GHz, 6 x 400 MHz
1-2	1 x (1 GHz x 4), ...
2-1	2 x 1 GHz, 2 x 800 MHz, 2 x 600 MHz, 2 x 400 MHz
2-2	2 x 1 GHz, 4 x 800 MHz, 2 x 600 MHz

Table 2 lists the different heterogeneous configurations we consider in our study. The results of this study are summarized in Figure 12 and can be divided into two main categories.

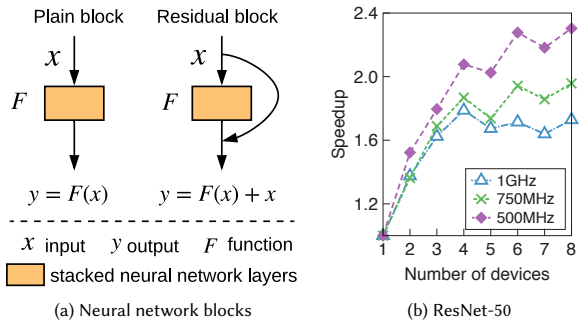
Case 1 represents scenarios that prompt our solution to use a small number of fast devices to achieve better performance. This

can be further divided into the sub-cases: 1-1) where the optimal performance of the model is parallelized with the two fastest devices while the rest of the devices are unused. For instance, with two 1 GHz devices, AOFL reduces the runtime by 15.8% compared to a configuration that uses all 8 devices. 1-2) represents a configuration that mimics local server approach where a much faster device is available. In this sub-case, we use a four-core device running at 1 GHz. We observe that the framework recognizes this and sends the entire input to such a device for execution. This indicates that our proposed algorithm also works for edge-cloud collaboration.

Case 2 represents scenarios that prompt our solution to adapt to the availability of devices with different frequency distributions. This can be further divided into the sub-cases: 2-1) when the frequencies of devices span over a wide range, AOFL optimally assigns tasks across all the devices with the exception of the slowest two devices running at 400 MHz. This configuration results in an 18.7% reduction in runtime. 2-2) when the frequencies of devices are within a narrow range, the input partition sizes are adjusted to accommodate the frequency, so each device can spend a similar time in computation. This configuration results in all 8 devices being used leading to a 12.3% reduction in runtime.

5.6 Generalizing to More DNNs

Our framework is able to accommodate various CNNs with complex structures, including bypass connections and parallel layers.



(a) Neural network blocks (b) ResNet-50

Figure 13: (a) Illustration of two different neural network blocks. (b) Speedup of Optimal Fused-layer (OFL) for ResNet-50 over a single edge device in WiFi-1 (93 Mbps).

Figure 13a shows two types of neural network blocks. Plain block stacks convolution layers and goes deeper to get better performance. Both VGG-16 and YOLOv2 have consecutive convolution layers, and achieve significant performance improvement using fused-layer parallelization. However, deep neural networks have the problem of vanishing gradients, which may stop the neural network from further training. To overcome the vanishing gradient issue, ResNet features special skip connections to reuse activations from previous layers until the adjacent layer learns its weights, enabling much deeper networks with good performance. The output tensor in the residual block is calculated with element-wise operations from the outputs of previous layers. In fused-layer parallelization, fusing layers in different residual blocks are not allowed unless the input of the current residual block and the output of the next residual block are within the same fused-layer block. This means fusing all the layers in two consecutive residual blocks. Then the bypass connections are duplicated and run on each partition, since there are no dependencies across different partitions. Figure 13b shows the performance of Optimal Fused-layer (OFL) parallelization for ResNet-50 as a function of devices with different frequencies in WiFi-1. In our experiments, one or more residual blocks are fused based on the search algorithm, and similar performance gain and scalability are observed. OFL runs 1.7× ∼ 2.3× faster on 2 to 8 devices than a single device. The results also indicate better speedup when the frequency drops and the ratio of computation increases.

Fused-layer parallelization may be limited by the available fusable convolution layers and model structures. For example, GoogLeNet [43] features inception module where filters with multiple sizes are used to operate on the same level. The outputs of all filter branches are concatenated depth-wise and sent to the next inception module. The network essentially would get a bit wider rather than deeper. A single filter branch may not benefit from fused-layer parallelization due to its number of fusable convolution layers. However, the structure of inception module itself implies a possible parallelism among the filter branches. DenseNet [22] features dense blocks where each layer is connected to every other layer in feedforward fashion. It alleviates vanishing gradients, strengthens features propagation, and encourage features reuse. In order to use fused-layer parallelization, it requires to fuse the entire dense block, and the performance depends on the depth of a dense block. We will explore the applicability to more DNNs in future work.

Model compression, including parameter pruning or quantization, may also benefit from the fused-layer parallelization when the system works under high computation to communication ratio. In the 5G era, much faster communication speed will be available, which can potentially change today's computation and communication ratio at the edge. The benefits of fused-layer parallelization would be further increased by the faster 5G network. However, when network speed is slow, it may be more suitable to run DNNs on a single device. Our algorithm can predict the best parallelization options based on the model characteristics, devices capacity and network condition.

6 RELATED WORK

Current techniques enabling DNN-based intelligent applications fall into two categories: *cloud-based* and *edge-only* approaches. Cloud-based approaches [3, 16, 24, 26, 34, 44, 49] fully (i.e., cloud-only) or partially (i.e., edge-cloud collaboration) offload the computation to the cloud. Neurosurgeon [26] proposes to distribute a DNN model between edge devices and the cloud by deciding a single partition point. In case the model is not pre-installed when offloading remotely, IONN [24] designs incremental model uploading with multiple partition points to overlap the local client and server executions.

Edge-only approaches [14, 18, 19, 23, 33, 36, 47, 51] execute the DNNs on a single edge device with specialized hardware or a small IoT cluster. Many customized accelerators for CNNs have been proposed [1, 7–9, 12, 30, 32, 41]. They focus on improving the performance of CNNs by exploring the potential for resources sharing,

and optimizing basic operations. Emerging memory technologies have been explored due to the high memory requirements of machine learning applications.

Our work falls into the category of distributing DNN inference in a small IoT cluster, which have gained increasing research interests recently. NestDNN [14] proposes that multiple compressed models can be dynamically selected based on available runtime resources. Collaborative computing enables a small IoT cluster to run larger models or speedup inference by employing the available idle devices. MoDNN [33] uses layer-wise parallelization, however the MapReduce-like execution results in a large amount of intermediate data to be transferred across devices. Collaborative perception [18] pipelines the computation by partitioning a DNN model and distributing the partitioned blocks to multiple edge devices. Follow-up work in [17] introduces model parallelism methods for both dense and convolution layers in order to address the memory overhead due to model replication. DeepThings [51] reduces the communication cost by fusing the early convolution layers and parallelizing these layers in multiple devices. Our work generalizes the fused-layer approach, proposes a mechanism for finding optimal layer fusion and device partitioning configurations, and is optimized for heterogeneous IoT environments, which were not addressed in prior work.

7 CONCLUSION AND FUTURE WORK

In this paper, we explore the trade-offs of distributing machine learning applications in IoT, and propose an Adaptive Optimal Fused-layer (AOFL) parallelization for improving the DNN inference at the edge. We design a dynamic programming based search algorithm to find the optimal partition and parallelization for a CNN model on a list of edge devices. We implement a CNN acceleration framework that adapts to the changes of computational resources and network conditions. Experimental evaluations show up to 1.9× ∼ 3.7× speedup can be achieved on 8 devices for three popular CNN models.

In future work we plan to expand this framework to other devices, including low-power accelerators such as the Google Edge TPU. We will also develop fault-tolerance solutions that can cope with compute nodes unpredictably failing or being temporarily unavailable.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their feedback. This work was supported in part by NSF XPS Award 60053525.

REFERENCES

[1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 1–13.

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 22.

[3] Amazon. [n.d.]. Machine Learning on AWS. https://aws.amazon.com/machine-learning/.

[4] Apple. [n.d.]. Core ML. https://developer.apple.com/documentation/coreml.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations (ICLR)*.

[6] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).

[7] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 269–284.

[8] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 367–379.

[9] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 609–622.

[10] Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning (ICLR)*. 160–167.

[11] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[12] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. 92–104.

[13] Marat Dukhan. 2018. NNPACK. https://github.com/Maratyszcza/NNPACK.

[14] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of the 24th ACM International Conference on Mobile Computing and Networking*. 115–127.

[15] Raspberry Pi Foundation. [n.d.]. Raspberry Pi. https://www.raspberrypi.org/.

[16] Google. [n.d.]. Cloud Machine Learning Engine. https://cloud.google.com/ml-engine/.

[17] Ramyad Hadidi, Jiashen Cao, Micheal S Ryoo, and Hyesoon Kim. 2019. Collaborative Execution of Deep Neural Networks on Internet of Things Devices. *arXiv preprint arXiv:1901.02537* (2019).

[18] Ramyad Hadidi, Jiashen Cao, Matthew Woodward, Michael S Ryoo, and Hyesoon Kim. 2018. Distributed Perception by Collaborative Robots. *IEEE Robotics and Automation Letters* 3, 4 (2018), 3709–3716.

[19] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[21] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[22] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[23] Intel. [n.d.]. Movidius Neural Compute Stick. https://software.intel.com/en-us/movidius-ncs.

[24] Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. 2018. IONN: Incremental Offloading of Neural Network Computations from Mobile Devices to Edge Servers. In *Proceedings of the ACM Symposium on Cloud Computing*. 401–411.

[25] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. *arXiv preprint arXiv:1802.04924* (2018).

[26] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[27] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[29] He Li, Kaoru Ota, and Mianxiong Dong. 2018. Learning IoT in edge: deep learning for the internet of things with edge computing. *IEEE Network* 32, 1 (2018), 96–101.

[30] Robert LiKamWa, Yunhui Hou, Yuan Gao, Mia Polansky, and Lin Zhong. 2016. RedEye: Analog ConvNet Image Sensor Architecture for Continuous Mobile Vision. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 255–266.

[31] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, Bram Van Ginneken, and Clara I Sánchez. 2017. A survey on deep learning in medical image analysis. *Medical image analysis* 42 (2017), 60–88.

[32] Dao-Fu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Temam, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. PuDianNao: A Polyvalent Machine Learning Accelerator. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 369–381.

[33] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. 2017. Modnn: Local distributed mobile computing system for deep neural network. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1396–1401.

[34] Microsoft. [n.d.]. Azure Machine Learning service. https://azure.microsoft.com/en-us/services/machine-learning-service/.

[35] Mehdi Mohammadi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. 2018. Deep learning for IoT big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials* 20, 4 (2018), 2923–2960.

[36] Nvidia. [n.d.]. Jetson Nano. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/.

[37] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[38] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[39] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: better, faster, stronger. *arXiv preprint* (2017).

[40] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[41] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 14–26.

[42] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[43] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[44] Surat Teerapittayanon, Bradley McDanel, and HT Kung. 2017. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 328–339.

[45] TensorFlow™. 2019. TensorFlow for Mobile and IoT. https://www.tensorflow.org/lite.

[46] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. 2011. Improving the speed of neural networks on CPUs. In *in Deep Learning and Unsupervised Feature Learning Workshop, NIPS*. Citeseer.

[47] Zirui Xu, Zhuwei Qin, Fuxun Yu, Chenchen Liu, and Xiang Chen. 2018. DiReCt: Resource-Aware Dynamic Model Reconfiguration for Convolutional Neural Network in Mobile Systems. In *Proceedings of the ACM International Symposium on Low Power Electronics and Design*. 37.

[48] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN pruning to the underlying hardware parallelism. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 548–560.

[49] Qi Zhang, Lu Cheng, and Raouf Boutaba. 2010. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications* 1, 1 (2010), 7–18.

[50] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6848–6856.

[51] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.