Neural Network Partitioning for Fast Distributed Inference

Robert Viramontes
Department of Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, WI
rviramontes@wisc.edu

Azadeh Davoodi
Department of Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, WI
adavoodi@wisc.edu

2023 24th International Symposium on Quality Electronic Design (ISQED) | 979-8-3503-3475-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISQED57927.2023.10129343

***Abstract*—The rising availability of heterogeneous networked devices highlights new opportunities for distributed artificial intelligence. This work proposes an Integer Linear Programming (ILP) optimization scheme to assign layers of a neural network in a distributed setting with heterogeneous devices representing edge, hub, and cloud in order to minimize the overall inference latency. The ILP formulation captures the tradeoff between avoiding communication cost when executing consecutive layers on the same device versus the latency benefit due to weight *pre*-loading when an idle device is waiting to receive the results of an earlier layer across the network. In our experiments we show the layer assignment and inference latency of a neural network can significantly vary depending on the types of devices in the network and their communications bandwidths.**

***Index Terms*—t neural networks, distributed systems, optimization, latency minimization, heterogeneous systems**

I. Introduction

Neural networks have had an explosive growth in popularity over the past decade, particularly convolutional neural networks (CNN) which have demonstrated a strong accuracy in image recognition tasks. AlexNet [1] is an early CNN that demonstrated remarkable accuracy in the 2012 ImageNet competition and observed that the depth of the model was key to its success in the image recognition challenge. As a result, researchers have continued to investigate deeper networks to achieve greater image recognition accuracy.

At the same time, Internet of Things devices have proliferated and neural networks are a promising means to analyze the data collected by these devices. Distributed inference can take advantage of multiple devices that can communicate over a network to perform a single inference. Distributing the task allows opportunities to offload aspects of computation and take advantage of the different capabilities of heterogeneous devices. However, this presents a challenge in determining how to assign computation across devices to achieve goals such as minimizing the inference latency.

In this work, we propose an Integer Linear Program (ILP) optimization scheme for inter-layer partitioning of a complex neural network to assign its layers to heterogeneous devices which may include edge, hub, and cloud. Unlike prior work, we do not limit the number of transitions between devices, and allow a device to be utilized as many times as necessary during the distributed inference process. The goal of our optimization scheme is to explicitly minimize the inference latency in a global sense. One of our contributions is including latency models based on high-level device architecture such as memory of each device and the device's *internal* bandwidth to fetch the layer weights from its memory.

The ILP formulation also models device-to-device communication and allows pre-loading of weights by an idle device. This allows the optimizer to evaluate the tradeoff between single-device execution, which incurs no network communication but eliminates opportunities for pre-loading, and utilizing multiple devices, with pre-loading on idle devices.

We experiment with four popular CNNs and show the layer assignment solution and inference latency of the *same* CNN can significantly vary depending on the types of devices present in the network and their communication bandwidths.

This work was supported by the National Science Foundation under Grant 1608040.

II. Overview

A. Network Model and Initial Setup

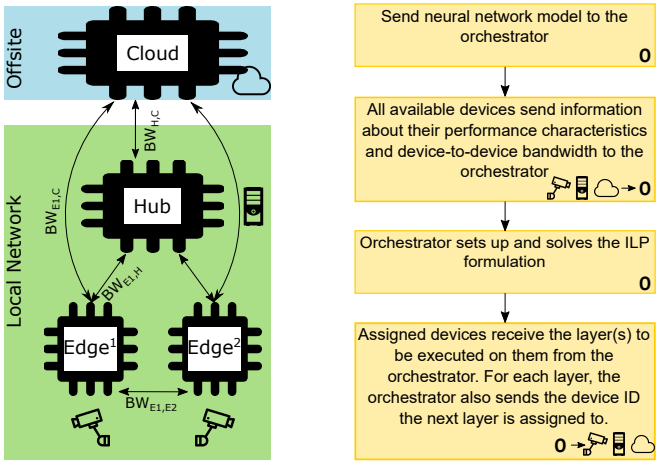
We describe a high-level overview of our framework to assign layers of a neural network to heterogeneous devices to minimize the inference latency in a distributed setting.

As shown in Fig. 1(a), in our network model the following devices may be present: (1) edge devices which may have sensing capability, (2) one or more hub devices, and (3) a cloud. We assume edge devices have the lowest compute capability, and the cloud has the highest one.

Additionally, a pair of devices have a unique communication bandwidth. In Fig. 1(a), we denote the bandwidth between the two edge devices E_1 and E_2 by the BW_{E_1,E_2}, bandwidth between E_1 and the hub device is denoted by $BW_{E_1,H}$, and bandwidth between the hub device and cloud is denoted by $BW_{H,C}$. These bandwidths could all be different but in practice: $BW_{E_1,E2} > BW_{E_1,H} > BW_{H,C}$.

Fig. 1(b) shows the steps to set up/generate the ILP for layer assignment. These steps are done prior to distributed inference and are executed on an *orchestrator* device which should have sufficient compute capability to solve the ILP.

First, in step 1, the neural network model is sent to the orchestrator. In step 2, the available devices in the network send information about their performance characteristics and



(a) (b)

Fig. 1. (a) A network may be comprised of edge, hub and cloud services with heterogeneous performance characteristics and varying device-to-device bandwidths; (b) Steps to set up and generate a layer assignment solution.

device-to-device bandwidth to the orchestrator. In step 3, the orchestrator creates and solves the ILP formulation based on the provided information in step 2 to determine a layer assignment. In step 4, the orchestrator sends the layer assignment information back to the devices. Next, each assigned device receives the layer(s) to be executed on it. In addition, for each layer, the ID of the "next device" will also be sent.

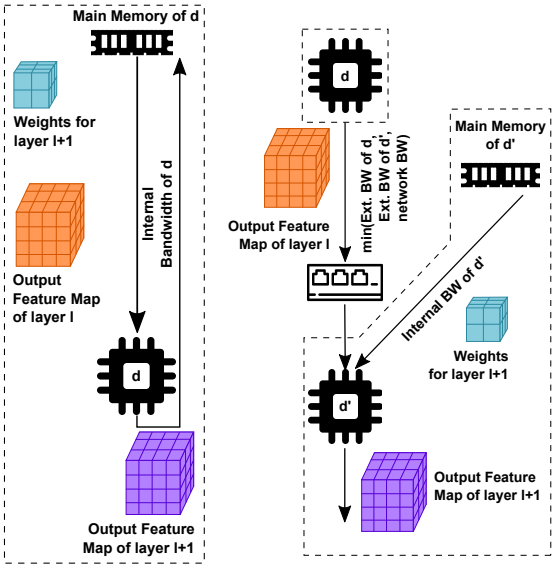
After the above steps, the layer assignment plan is complete and distributed inferencing may begin. While network conditions remain stable, the same plan can be re-used without invoking the orchestrator. As the network bandwidth may vary over time, the above steps should be repeated periodically to update the assignment under the new conditions, though methods to detect this are outside the scope of this work.

After the above steps, the layer assignment plan is complete and distributed inferencing may begin. While network conditions remain stable, the same plan can be re-used without invoking the orchestrator. If the network bandwidth varies over time, the above steps should be repeated periodically to update the assignment under the new conditions, though detecting such variance is outside the scope of this work.

Our ILP formulation generates a layer assignment solution by considering the individual performance characteristics of the available devices and the communication bandwidths to minimize the overall inference latency. It is capable of quickly generating flexible solutions such as (1) only performing the computation on one or more edge devices, (2) additionally utilizing the hub device(s), (3) utilizing the cloud.

B. Latency Models for Distributed Inference

We assume each device in the network has a main memory for storing weights and computation results for the neural network (NN) layers that are assigned to it. We refer to the *internal* bandwidth of a device to be the bandwidth of this main memory, as shown in Fig. 2(a). We differentiate the internal bandwidth from the *external* bandwidth of a device



(a) (b)

Fig. 2. Our latency models: (a) Single device model with access to main memory to load the assigned layer weights and/or results of previous computations at the rate of the *internal* bandwidth of the device; (b) Two device model which communicate across the network to send results of layer l (from device d) to compute layer $l+1$ (on device d'). The rate of transfer between d and d' is the minimum of the external bandwidths of the two devices and the network conditions. To minimize the latency of distributed inference, d' may pre-load the weights of layer $l+1$ while waiting for the results from d.

which is the one used to communicate (send or receive) with another device over the network, as shown in Fig. 2(b).

We consider three components of latency: (1) latency to compute a layer on a device, (2) latency to load the weights of a layer on a device, and (3) communication latency to send the results of a layer from one device to another.

1) Compute Latency: Let $t^c_{l,d}$ indicate the latency to compute layer l on device d. We estimate this latency as:

$$t^c_{l,d} = \frac{(\text{FLOP})_l}{(\text{FLOP/s})_d} \tag{1}$$

where $(\text{FLOP})_l$ indicates the number of floating point operations to compute layer l, and $(\text{FLOP/s})_d$ indicates the number of floating point operations per second that device d is capable of handling. The $(\text{FLOP/s})_d$ can be determined from the manufacturer specification. The $(\text{FLOP})_l$ is calculated by the tool `ptflops` [2], which reports operations as multiply-accumulate (MAC) so we double this number to estimate $(\text{FLOP})_l$. This is a useful and convenient metric for estimating performance [3], [4], especially in an ILP formulation, and is sufficient to capture latency among heterogeneous devices for layer assignment optimization purposes.

2) Weight Latency: Let $t^w_{l,d}$ denote the latency to load the weights of layer l on device d. It is determined by dividing the size corresponding to the weight parameters of layer l by the memory bandwidth of device d, as shown in Fig. 2(a).

$$t^w_{l,d} = \frac{w_l \times \alpha}{BW_{d,d}} \tag{2}$$

where w_l is the number of weights in layer l and α is the number of bytes required to store each weight. In the above equation, $BW_{d,d}$ refers to the internal bandwidth of device d.

3) Communication Latency: Assume layer l is computed on device d and its output should be communicated over the network to device d', as shown in Fig. 2(b). We denote $t^x_{l,d,d'}$ to be this communication latency and estimate it as:

$$t^x_{l,d,d'} = \frac{H_l \times W_l \times D_l \times \alpha}{BW_{d,d'}} \tag{3}$$

where H_l, W_l, and D_l are the height, width and depth of the output of l, respectively, and α is the number of bytes required to store each output element, as defined in Equation 2.

The parameter $BW_{d,d'}$ reflects the bandwidth between d and d'. We estimate it as the minimum of external bandwidths of d and d' and the bandwidth of the network itself, as in Fig. 2(b). Estimation of the network bandwidth itself is beyond the scope of this work; it has been an active topic of research, typically by utilizing dynamic probing techniques [5], [6], and more recently using machine learning techniques [7]. In case of a change in network bandwidth, the ILP can be set up and solved again quickly as shown in Fig. 1(b).

III. ILP Formulation

Given a standard feedforward neural network (NN) architecture[1], and heterogeneous networked devices, our formulation determines an assignment of each NN layer to a device.

A. Optimization Parameters and the NN Model

Let S_D denote the set of D available devices in the network. For device $d \in S_D$, let $t^c_{l,d}$ and $t^w_{l,d}$ denote the latency parameters to compute layer l and to load its weights, respectively. Also let $t^x_{l,d,d'}$ denote the communication latency to send the results of layer l from device $d \in S_D$ to $d' \in S_D$. These are computed using Equations 1-3.

Let S_L denote the set of L layers in the given NN. We also denote S_{L-1} to be set of all layers excluding the last layer of the NN, and $S_{2,L}$ denote the set of all layers excluding the first layer of the NN. Additionally, the size of the main memory of device d is denoted by M_d. Based on the application, we extend the NN by inserting *pseudo* layers, as explained below.

- We allow designating a device as the *source*, e.g., because it may be physically hooked to a camera for image capture. Here, the NN is extended to start with a pseudo layer with 0 FLOPs. The purpose of this pseudo layer is only to capture the communication latency from the source device to the device which executes the next layer.
- In case pre-processing needs to be done on the collected data before sending it to the NN, we insert a pseudo layer right before the first layer of the NN. This pseudo layer will have a compute latency given by Equation 1, to reflect the pre-processing task.
- We allow designating a device as *destination*, for example if it is required to receive the output computed from the last layer of NN in order to take action based on the inference result. This is modeled by extending the NN to have a pseudo layer of size $H_l = 1, W_l = 1, D_l = 1$ with 0 FLOPs at the end.

A pseudo layer is treated just as a layer in our formulation and modeled by adding it to the sets S_L, S_{L-1}, and $S_{2,L}$.

B. Optimization Variables

- Let binary variable $x_{l,d} = 1$ when layer l is assigned to device d and 0 otherwise.
- Let binary variable $y_{l,d,d'} = 1$ when layer l is assigned to device d and layer $l+1$ is assigned to device d'. This variable identifies when a communication latency between d and d' should be added to compute the overall inference latency. Note $y_{l,d,d'} \neq y_{l,d',d}$. It allows capturing solutions where two or multiple devices may compute the layers back and forth among each other.
- Let binary variable $z_{l,d} = 1$ when layers l and $l-1$ are both assigned to device d and 0 otherwise. This variable identifies when a latency for loading the weights of l on d should be added to compute the overall inference latency. This variable is defined to capture the impact of weight pre-loading on the latency.

C. Optimization Constraints and Objective

The constraints to our formulation are given below.

$$\sum_{d=1}^{D} x_{l,d} \leq 1 \qquad \forall l \in S_L \tag{4}$$

$$\sum_{l=0}^{L} w_l x_{l,d} \leq M_d \qquad \forall d \in S_D \tag{5}$$

$$0 \leq x_{l,d} + x_{l+1,d'} - 2y_{l,d,d'} \leq 1 \quad \forall l \in S_{L-1}\ \forall d \in S_D \tag{6}$$

$$0 \leq x_{l-1,d} + x_{l,d} - 2z_{l,d} \leq 1 \qquad \forall l \in S_{2,L}\ \forall d \in S_D \tag{7}$$

Constraint 4 indicates that each layer must be assigned to only one device. Constraint 5 ensures the weights of the layers assigned to a device fit within the size of its main memory.

Constraint 6 sets $y_{l,d,d'}$ when layer l executes on device d and the outputs are sent to d'. So $y_{l,d,d'} = 1$ only when $x_{l,d} = x_{l+1,d'} = 1$. This constraint (consisting of two inequalities) expresses a logical AND operation and is written for all combinations of devices in S_D and layers in S_{L-1}.

Constraint 7 sets $z_{l,d}$ when there is *not* an opportunity to pre-load the weights of l on device d because layer $l-1$ is also executed on d. So $z_{l,d} = 1$ only when $x_{l-1,d} = x_{l,d} = 1$. Recall, pre-loading reduces the inference latency when a layer is computed on a different device than its previous layer and is captured by this constraint. This constraint also expresses a logical AND operation and written for all combinations of devices in S_D and layers in $S_{2,L}$.

In addition to the above generic constraints, specific constraints may be added to pre-assign layers to devices. For example, if a source device is designated, the first psuedo-layer may be pre-assigned to the source. To pre-assign a layer l to device d, we simply set $x_{l,d} = 1$ in our formulation.

[1]Our formulation does not consider aspects such as state in a recurrent network. Such extensions are possible but beyond the scope of this paper.

Objective Function: The goal of our formulation is to minimize inference latency, hence our objective function is $\min(T)$, where T is the sum of computation, communication, and weight loading latencies $T = T_c + T_x + T_w$ given below:

$$T_c = \sum_{l=1}^{L} \sum_{d=1}^{D} t^c_{l,d} \; x_{l,d} \tag{8}$$

$$T_x = \sum_{l=1}^{L-1} \sum_{d=1}^{D} \sum_{d'=1}^{D} t^x_{l,d,d'} \; y_{l,d,d'} \tag{9}$$

$$T_w = \sum_{l=1}^{L} \sum_{d=1}^{D} t^w_{l,d} \; z_{l,d} \tag{10}$$

T_c expresses the compute latency for a given assignment of x. T_x expresses the cost to communicate the outputs of layers for a given assignment of x between devices across the network. T_w expresses the latency to load weights for a given assignment of z. Recall that pre-loading is utilized when $z_{l,d} = 0$, which means $t^w_{l,d}$ is *not added to* T_w because weight loading is hidden by a device that would otherwise be idle.

Note, in Equation 9, all possible device-to-device communication latencies are considered and added up if the corresponding y variable is assigned to 1. Note that since $y_{l,d,'d} \neq y_{l,d',d}$, both d and d' are varied across all the devices.

Overall, to minimize this objective, solving the formulation assigns the variables to take the most advantage of weight pre-loading, the network bandwidth between each pair of (heterogeneous) devices based on the network conditions, as well as the size of each layer in the neural network. As we show in our experiments, the choice of this assignment is not trivial and depending on factors such as network conditions and types of devices, the assignment of the layers of the *same* network could be different in a distributed setting.

D. Related Works

In this section, we explore related works on distributed inference systems, particularly those that utilize edge or mobile devices as part of the inference network. We propose that the approaches can be roughly categorized into two groups depending on their network partitioning strategy, either intra-layer or inter-layer, and explore these categories below.

In an *intra*-layer approach, each layer of the network is partitioned and chunks of work for each layer are sent to the networked devices. A technique for partitioning the computation of a convolution layer across heterogeneous devices is proposed in CoEdge [8]. This allows multiple edge devices to collaborate on the inference to reduce latency. MoDNN [9] is another intra-layer approach that demonstrates latency improvements by partitioning convolution and fully connected layers over multiple devices.

The reduced latency of the intra-layer approaches is primarily due to the increased parallel computation in a single layer. Both approaches incur some communication overhead to share layer information. However, intra-layer approaches are typically implemented in a cloud-less context, where the fast local networks mean this is small relative to the the parallel computation benefit. Because these works lack access to a cloud or other highly-performant device, they do not minimize latency in a global sense.

Our layer assignment work is inter-layer and minimizes the global latency. Combining inter-layer and intra-layer approaches may be possible to achieve the speedup benefits of both, but this has not been explored yet in prior work.

In an *inter*-layer approach, a single layer is executed by a single device, but sequential layers may be executed by different devices. Unlike intra-layer, these approaches typically assume environments with edge, hub, and cloud.

The work JALAD [3] finds an optimal point to partition a deep network in an edge-cloud structure. JALAD formulates the layer partitioning problem as an ILP to minimize the execution latency. However, its formulation assumes a strict edge-cloud architecture with only one partition point and does not consider an opportunity to overlap pre-execution work. In contrast, our formulation does not assume a fixed number of devices and can generalize to more complex sets of devices and adds a consideration for pre-loading of weights to find opportunities for utilizing multiple devices simultaneously.

The work ADDA [4] extends the partitioning approach to consider multi-branch networks, which have multiple exit points, and looks at finding an optimal partition between edge and cloud execution, considering that some paths may entirely execute on the edge. Similar to [3], ADDA's formulation is constrained to an edge-cloud architecture. In addition, it requires modifying the network to insert multiple branches. Under multiple network conditions, ADDA closely-matches the overall latency of a cloud-only execution model because it tends to pick a partition almost at the cloud-only point, while our approach can also find cases where an intermediate layer-split can further reduce latency.

Inter-layer approaches derive their latency reductions by finding optimal mappings of sequential work, but don't incur the same communication overheads as intra-layer techniques. This is more amenable to cloud-enabled contexts, where a relatively slow network discourages communication bottlenecks but the large performance asymmetry can reduce the overall latency. Our work is an inter-layer partitioning strategy for sequential layer execution, though we propose weight pre-loading as a way to increase parallelism.

IV. Experimental Results

In our experiments, we simulate varying network conditions by changing the device-to-device communication bandwidths. This approximates real-world conditions, such as a cellular (4G/5G) network which may vary in bandwidth due to other user's utilization, physical obstructions between radios, etc.

We incorporated devices which are representative of a typical cloud inference architecture to model a distributed system. This included (1) a data collection edge device with limited CPU processing (Raspberry Pi 3B); (2) a low-power hub device with low GPU compute capacity (NVIDIA Jetson Nano); and (3) a cloud service with high GPU compute capacity (NVIDIA 1080TI).

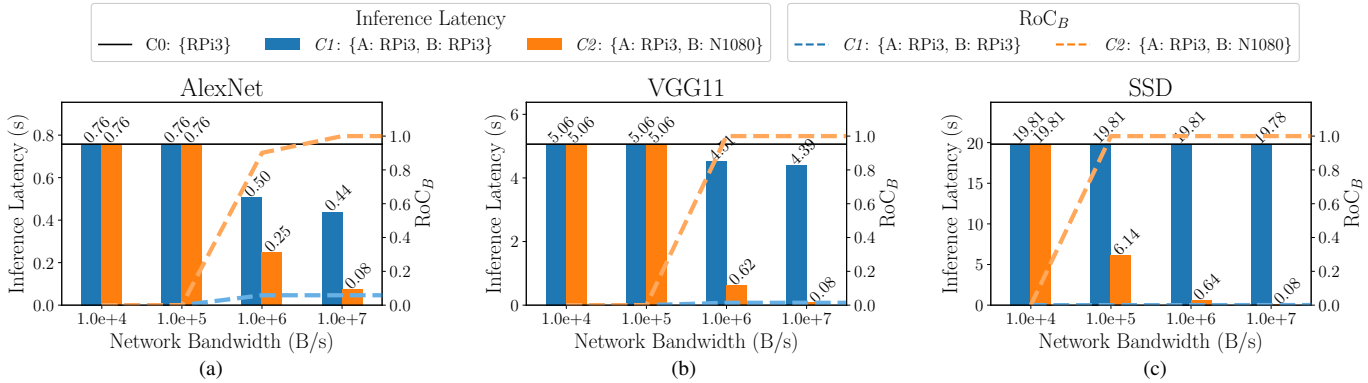


Fig. 3. Results for configurations C_0 (non-distributed), C_1 (two edge devices) and C_2 (edge device and a cloud). We vary network bandwidth between the two devices in each configuration, and report the inference latency and ratio of compute on device B. At the lowest network bandwidth, single-device execution is the optimal strategy. But at the highest bandwidths, configuration C_2 has the smallest inference latency, indicating, it is better to utilize a cloud.

Device parameters were determined from publicly-available benchmarks and/or the manufacturer specification and are summarized in the table below. Parameters include the GFLOP/s, internal and external bandwidths of a device which are used to compute the computation, weight loading, and communication latencies as defined in Section II-B.

Device	GFLOP/s	Int. BW B/s	Ext. BW B/s
Raspberry Pi 3B	3.62 [10]	719×10^6 [11]	1.11×10^7 [11]
NVIDIA Jetson Nano	236 [12]	25.6×10^9 [13]	1.25×10^8 [13]
NVIDIA 1080TI	11300 [14]	484.4×10^9 [14]	1.25×10^8

With these devices we define four distributed configurations.

C0) {*A*: Raspberry Pi3}: Non-distributed execution with a single lightweight edge device (for comparison purposes).

C1) {*A*: Raspberry Pi3, *B*: Raspberry Pi3}: Inference in a network of two homogeneous, lightweight edge devices.

C2) {*A*: Raspberry Pi3, *B*: NVIDIA 1080TI}: Inference with a lightweight edge device with access to a cloud GPU.

C3) {*A*: Raspberry Pi3, *B*: NVIDIA Jetson Nano, *C*: NVIDIA 1080TI}: Inference with a lightweight device, a performant device, and a cloud GPU. This configuration models presence of edge, hub, and cloud components.

In our experimental setup, we assume device A is the source and destination device for all the above configurations. This is done to model an application in which device A (i.e., Raspberry Pi 3) is always hooked to a camera for image capture and must receive the results of the inference in the end to take an action. As explained in Section III-A we model these by adding pseudo layers to the NN (with zero compute and weight loading latencies but non-zero communication latency) and preassign these pseudo layers to A.

Furthermore, we assume that the original image capture has a Standard HD dimensions of 3×1280×720 which represents the image quality of a typical commodity webcam or security camera connected to an edge device. Given that many image recognition CNNs operate on an input size of 3×224×224, we consider a preprocessing step to crop and downsample the HD image to this size before it is fed into the CNN. The preprocessing assumes one operation per pixel in each channel of the input image. This is also modeled as a pseudo layer in our formulation as explained in Section III-A, but this layer is not pre-assigned to a device.

We compare the assignment results for the image classification task on AlexNet [1], VGG11, SSD [15], and VGG16 [16]. For this analysis, we trained models from the PyTorch [17] `torchvision` library. The optimization formulation was solved using the Gurobi solver [18]. We used an Ubuntu 20.04 virtual machine with 4 vCPU and 4.0GB of memory. The runtimes for initial setup and solving the ILP formulation are extremely fast; for example for configuration C_1, averaged across 5 runs, AlexNet took 0.17s and VGG16 took 1.67s.

A. Inference Latency and Ratio of Compute

In this experiment, we first compare configurations C_0, C_1 and C_2 for AlexNet, VGG11, and SSD. (For these networks and configuration C_3, we did not notice a notable conclusion and results are not reported due to lack of space. Instead, we discuss C_3 with VGG16 which exhibits interesting results across *all* configurations.)

For C_1 and C_2 we vary the network bandwidth between the two devices in each configuration. For each bandwidth, we report the overall latency of distributed inference as measured using Equations 8-10 which includes weight (pre-)loading, compute, and communication latency.

We also report a ratio of compute (RoC) on each device. This metric measures the ratio of floating point operations performed on a device based on its assigned layers. More specifically, for configurations C_1 and C_2, we report the ratio of compute on device B and the rest of the computations are performed on device A. We denote these by $RoC_B^{C_1}$ and $RoC_B^{C_2}$, respectively. (For configuration C_0, we have $RoC_A^{C_0} = 1$ since everything is executed on a single A device.)

Fig. 3 shows the results for AlexNet, VGG11, and SSD. We make the following observations:

- In the lowest network bandwidth, non-distributed execution on a single device is the best strategy. For these bandwidths, our ILP assigns all computations to device A for C_1 and C_2 which can be seen because $RoC_B = 0$ for these two configurations and this bandwidth.

- As the network bandwidth increases, distributed execution has a clear advantage. Both configurations C_1 and C_2 have a significantly lower inference latency compared to C_0 for all the networks.
- In the highest network bandwidth, configuration C_2 which additionally takes advantage of a cloud device significantly outperforms C_1 in terms of inference latency.
- Ratio of compute on device B changes as a function of bandwidth for C_1 and C_2; with increase of bandwidth, more computation is assigned to device B. (This also indicates the ILP solution varies with network bandwidth.)
- The choice of the best configuration for a given bandwidth can be different across the NNs. For example, at bandwidth 10^5B/s, single-device execution is the optimal choice in AlexNet and VGG11. However, at the same bandwidth for SSD, utilizing the cloud in configuration C_2 results in significantly lower inference latency.

Fig. 4 compares these metrics for *all* configurations in VGG16. Recall in configuration C_3, a hub device is present in addition to the edge and cloud devices in configuration C_2. Here, for C_3 we report ratio of compute on devices B and C; the rest are executed on device A. We denote these by $RoC_B^{C_3}$ and $RoC_C^{C_3}$, respectively. We make the following observations:

- Configuration C_3 has the smallest inference latency in *all* bandwidths (e.g., 0.16s in C_3 versus 9.51s in other configurations for the 10^4 bandwidth).
- At the two lowest bandwidths (10^4 and 10^5), the minimum latency is when all computations are offloaded to the hub device in C_3, i.e., $RoC_B^{C_3} = 1$. This is because the hub device is more compute-efficient than the edge. Also communicating with the cloud in C_2 results in a higher latency.
- At the highest network bandwidth (10^7), the minimum latency is when offloading all computations to the cloud and not utilizing the hub ($RoC_C^{C_3} = 1$). (At this bandwidth, C_2 and C_3 generate effectively the same solution.)

Overall, from our experiments using four NNs, we can derive the following generic conclusions: Inference latency in a distributed setting can significantly vary depending on the network devices (e.g., availability of edge, hub, and cloud), device-to-device communication bandwidth, as well as the characteristics of the NN itself. Our ILP formulation is able to capture all the above factors to generate a layer assignment plan that minimizes the overall latency.

B. Effect of Pre-loading on Latency

Table I shows the breakdown of different contributors to the inference latency (denoted by L) in configuration C_1 at the peak network bandwidth of 10^7 B/s. The columns indicate the breakdown of the individual latency components, compute (T_c), communication (T_x), and loading of weights (T_w), as computed using Equations 8-10. These are reported without and with our proposed pre-loading technique to understand the source of pre-loading's benefit.

For AlexNet, we observe a 42% reduction in latency (from 0.76s to 0.44s) because we trade a 0.01s increase in communication (T_x) for a 0.33s reduction in weight loading (T_w). VGG16 and VGG11 show a similar trade off, though the reduction in latency is smaller, 7% and 13% respectively, because the bulk of latency is *computation* which does not benefit from pre-load. For SSD however, we do not see a significant advantage in pre-loading because the compute latency significantly dominates the pre-loading latency.

This capturing of pre-loading is another contribution of our work which was not done in prior work, and overall can be seen can result in significant benefit for some networks.

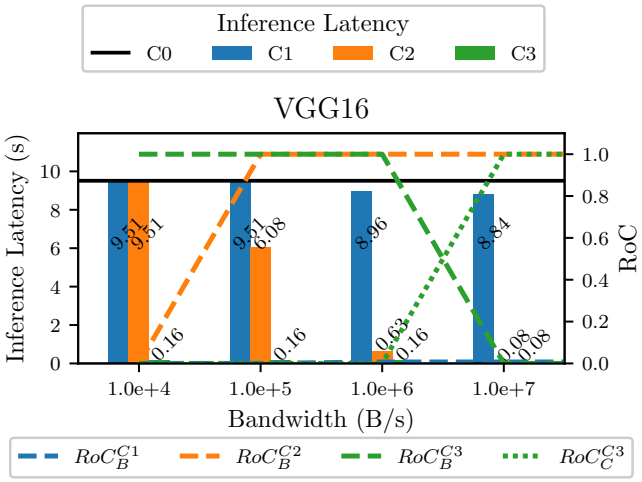


Fig. 4. Comparison of all configurations in VGG16. An interesting observation is that at the low and intermediate bandwidths (10^4 and 10^5), configuration C_3 shows a significantly smaller inference latency compared to all other configurations (e.g., 0.16s versus 9.51s in 10^4 bandwidth). At these bandwidths, utilizing a hub device is the optimal strategy compared to using the cloud and single-device execution.

TABLE I
INFERENCE LATENCY, WITHOUT AND WITH THE PRE-LOADING.

	No Pre-load				Pre-load			
	T_c	T_x	T_w	L	T_c	T_x	T_w	L (% decr.)
AlexNet	0.40	0.02	0.34	0.76	0.40	0.03	0.01	0.44 (42%)
VGG11	4.22	0.11	0.74	5.06	4.22	0.12	0.05	4.38 (13%)
SSD	19.29	0.33	0.19	19.81	19.29	0.35	0.14	19.78 (0.2%)
VGG16	8.57	0.18	0.77	9.51	8.57	0.19	0.08	8.84 (7%)

V. CONCLUSIONS

We proposed an ILP formulation to assign layers of a neural network to minimize the inference latency in a distributed setting with heterogeneous devices. To reduce the latency, we proposed an idle device can pre-load the weights of the layer that it is awaiting to execute, and captured it in the ILP.

We showed in our experiments that when minimizing inference latency in a distributed setting, the layer assignment solution can significantly vary depending on the network devices and device-to-device communication bandwidths. We also showed that the benefits of pre-loading the layer weights can be quite significant in NNs where the latency to compute a layer is comparable to the latency to load its weights.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25, 2012.

[2] "`ptflops`," 2021. [Online]. Available: https://github.com/sovrasov/flops-counter.pytorch

[3] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu, "JALAD: Joint Accuracy-And Latency-Aware Deep Structure Decoupling for Edge-Cloud Execution," in *IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, Dec. 2018, pp. 671–678.

[4] H. Wang, G. Cai, Z. Huang, and F. Dong, "ADDA: Adaptive Distributed DNN Inference Acceleration in Edge Computing Environment," in *IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, Dec. 2019, pp. 438–445.

[5] B. Melander, M. Bjorkman, and P. Gunningberg, "A new end-to-end probing and analysis method for estimating bandwidth bottlenecks," in *IEEE Global Telecommunications Conference (GLOBECOM)*, vol. 1, 2000, pp. 415–420.

[6] R. Lübben, M. Fidler, and J. Liebeherr, "Stochastic bandwidth estimation in networks with random service," *IEEE/ACM Trans. on Networking*, vol. 22, no. 2, pp. 484–497, 2014.

[7] S. K. Khangura, M. Fidler, and B. Rosenhahn, "Neural networks for measurement-based bandwidth estimation," in *IFIP Networking Conference*, 2018, pp. 1–9.

[8] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, "Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices," *IEEE/ACM Trans. on Networking*, vol. 29, no. 2, p. 595–608, Apr 2021.

[9] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for Deep Neural Network," in *Design, Automation & Test in Europe Conference (DATE)*, Mar. 2017, pp. 1396–1401.

[10] VMW Research Group. (2020, Oct) The gflops/w of the various machines in the vmw research group. [Online]. Available: http://web.eece.maine.edu/$\sim$vweaver/group/green_machines.html

[11] L. Hattersley. (2018, Mar) Raspberry Pi 3B+ specs and benchmarks. [Online]. Available: https://magpi.raspberrypi.com/articles/raspberry-pi-3bplus-specs-benchmarks

[12] "NVIDIA Jetson Nano developer forum," 2019. [Online]. Available: https://forums.developer.nvidia.com/t/help-question/71758

[13] *NVIDIA Jetson Nano System-on-Module Data Sheet*, NVIDIA, Feb 2020.

[14] TechPowerUp. GeForce GTX 1080 Ti specs. [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877

[15] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot MultiBox detector," in *Proc. of European Conference on Computer Vision (ECCV)*, vol. 9905, 2016, pp. 21–37.

[16] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014. [Online]. Available: https://arxiv.org/abs/1409.1556

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[18] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2021. [Online]. Available: https://www.gurobi.com