Technical Report

Distributed Convolutional Neural Network Training on Resource-Constrained Mobile and Edge Clusters

Pranav Rama
Andreas Gerstlauer

UT-CERC-24-02

May 2024

Computer Engineering Research Center
Chandra Family Department of Electrical and Computer Engineering
The University of Texas at Austin

The University of Texas at Austin
Chandra Department of Electrical and Computer Engineering
Cockrell School of Engineering

Distributed Convolutional Neural Network Training on Resource-Constrained Mobile and Edge Clusters

Pranav Rama, Andreas Gerstlauer

Electrical and Computer Engineering
The University of Texas at Austin

Abstract

The training of deep convolutional neural networks (DNNs/CNNs) is traditionally done on servers with powerful CPUs and GPUs. Recent efforts have emerged to localize machine learning tasks fully on the edge. This brings advantages in reduced latency and increased privacy, but necessitates working with resource-constrained devices. Approaches for inference and training in mobile and edge devices based on pruning, quantization or incremental and transfer learning require trading off accuracy. Several works have explored distributing inference operations on mobile and edge clusters instead. However, there is limited literature on distributed training on the edge. Existing approaches all require a central, potentially powerful edge or cloud server for coordination or offloading.

In this report, we describe an approach for distributed CNN training exclusively on mobile and edge devices. Our approach is beneficial for the initial CNN layers that are feature map dominated. It is based on partitioning forward inference and back-propagation operations among devices through tiling and fusing to maximize locality and expose communication and memory-aware parallelism. We also introduce the concept of layer grouping to further fine-tune performance based on computation and communication trade-off, and we describe an algorithm to determine the optimal grouping profile.

Results show that for a cluster of 2-6 quad-core Raspberry Pi3 devices, training of an object-detection CNN provides a 2x-15x speedup with respect to a single core and up to 8x reduction in memory usage per device, all without sacrificing accuracy. Optimally grouping offers up to 1.5x speedup depending on the reference profile and batch size. Compared to traditional data and model parallelism, on an 8-core setup, our approach provides up to 4x speedup for a batch size of one.

Table of Contents

List of Figures

Chapter 1: Introduction

Traditionally, training and inference of deep learning (DL) models is performed in the cloud. This requires a large amount of data to be collected and sent to a centralized infrastructure, introducing latency, privacy, and real-time concerns. Various approaches have proposed to partition the processing between mobile, edge, and cloud resources [1]. However, such approaches still rely on a remote cloud for partial processing. To address the latency and privacy concerns when communicating with the cloud, recent efforts have emerged to localize DL tasks fully on mobile or edge devices [2, 3, 4]. However, this brings the challenge of performing compute and memory-intensive inference and training operations on such resource-constrained devices.

A wide range of approaches have been proposed to address limited memory and computing capabilities in mobile and edge settings. Techniques such as pruning and quantization focus on decreasing the complexity of the model by removing weights and neurons or reducing the bit precision during inference and/or training [5, 6, 7]. Other approaches such as incremental and transfer learning start from a pre-trained model and only partially update the model to save computational resources and reduce training time [8, 9]. However, all of these approaches trade-off accuracy for decreased computational complexity.

Alternatively, several methods have been proposed recently to utilize parallelism in DL models by partitioning them across multiple devices, preserving the original model and accuracy. Federated learning [10, 11] exploits data parallelism, but still requires a central server for coordination as well as storing and processing of complete models in each device, which is often infeasible given memory constraints. Other approaches partition and distribute the model itself across a cluster of edge devices [12, 13, 4, 14]. In addition to exploiting available multi-device parallelism,

this allows for reducing both the computational and storage requirements on each device. However, such approaches have only been demonstrated for inference so far.

In this report, we present an approach for distributed CNN training exclusively on communication- and memory-constrained mobile and edge clusters. Our approach targets feature map-dominated early CNN layers. We adopt a tiling and fusing-based partitioning scheme that has previously been demonstrated for inference [15, 12, 16] and extend it to apply to both forward and back-propagation training tasks. The scheme tiles feature maps to reduce memory footprint and expose model parallelism, then fuses matching tiles of consecutive layers into independent execution stacks placed on each device to maximally exploit locality. Furthermore, groups of layers are formed among tiles where synchronization of feature data shared among neighboring tiles is performed only at group boundaries. At the end of a single training pass, the final weight updates of all stacks are aggregated. This approach can confirm to arbitrary memory constraints imposed by each edge device while exposing parallelism, minimizing communication, and exploiting the locality inherent in convolutional and pooling layers. Our distributed training approach includes the following contributions:

1. We propose a novel method for tiling and fusing of backpropagation tasks that considers memory and communication constraints, while exploiting parallelism for distributed CNN training on resource-constrained device clusters.

2. We apply the concept of layer grouping, and we develop a cost model and propose an algorithm to determine optimal groups of forward inference and back-propagation tasks in order to further fine tune computation and communication overhead based on the grouping profile of the layers.

3. We evaluate our approach on distributed training of Yolov2 [17], a common CNN used for object detection, distributed across a network of quad-core Rasberry Pis. Results show a 2-15x speedup compared to a single core and with up to 8x

reduction in memory usage per tile, where optimal grouping can achieve up to 1.5x speedup depending on the reference profile and batch size.

The rest of the report is organized as follows: Chapter 2 details related works, Chapter 3 provides a brief overview of our distributed training approach, Chapter 4 details our training and partitioning strategies, Chapter 5 describes a cost model and optimal grouping, Chapter 6 then describes our experimental setup and results, and Chapter 7 provides a summary of our work.

Chapter 2: Related Works

Performing inference on resource-constrained edge and mobile devices has received significant attention. Pruning and quantization techniques involve removing unnecessary weights and/or neurons, as well as reducing the range of values representing network weights or features [5, 6]. However, these methods trade-off accuracy for decreased model complexity.

By contrast, approaches for distributed inference in edge settings preserve accuracy and instead exploit inherent parallelism to partition a model and distribute it across multiple devices. [13, 14, 4, 12, 3]. These methods can be applied to the forward inference pass in distributed training. We adopt tiling and fusing strategies from distributed edge and hardware accelerated inference [15, 12, 16] in our work and extend it to the back-propagation pass in order to support fully distributed training.

Training CNNs requires additional memory compared to inference due to the need to store input data, gradients, and activation values for each layer [18]. This normally requires partitioning of the workload involving powerful edge servers or the cloud [1, 19, 20]. Multiple approaches exist for training a DL model on a single edge device [7]. These typically employ simplified model architectures [21] or use reduced bitwidths for training [8]. Alternatively, approaches for incremental or transfer learning take a pre-trained model and only update a subset of weights [22] or the last layers of a model with every training sample [9]. However, all of these approaches trade off accuracy for reduced model complexity.

Multi-device solutions that rely on federated learning exploit data parallelism to collaboratively train a model, with each device training a local model on its own data and devices exchanging weight updates as different variants of a distributed gradient descent [10, 11]. Other multi-device approaches rely on approximate gradient prediction methods that trade off accuracy [23]. However, mobile and edge devices,

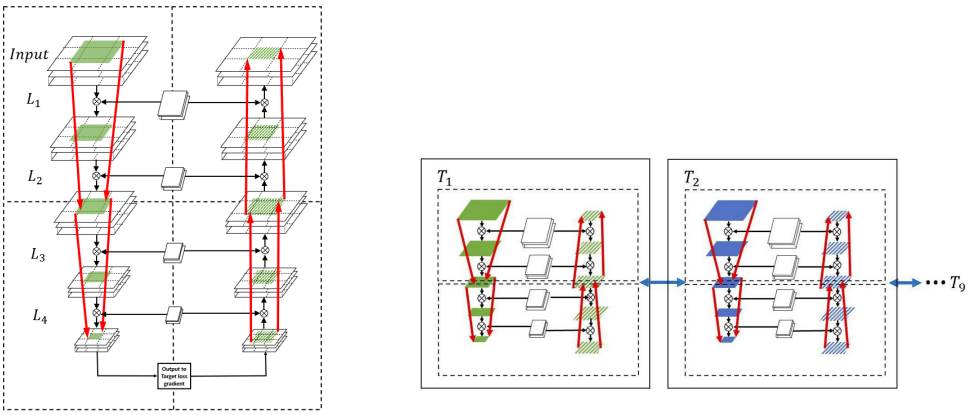
e.g. in the IoT space, often lack sufficient memory to store an entire local model. In [19, 20], federated learning is hierarchically combined with model partitioning in each local cluster. However, these approaches use partitioning schemes commonly used in cloud settings, which are not memory- and communication-optimal. Distributed training approaches optimized for cloud environments [24, 25, 26, 27] do not account for the greater resource limitations found in mobile and edge settings, and typically exploit mini-batch data parallelism and layer-based pipelining, which may not be optimal especially when batch sizes are small. Our work fits in as a form of partitioning in a more granular form to extract parallelism within each data sample rather than between complete samples/mini-batches.

Chapter 3: Overview

Fig. 3.1 gives an overview of our approach. We partition feature data and delta gradient maps in forward inference and back-propagation passes, respectively, into tiles in a grid-wise fashion along their width and height dimensions. In both passes, output tiles of each layer are computed from input tiles through convolutions with filter data or through simple pooling operations. Exploiting the inherent locality in these operators, all intermediate matching tiles on forward and backward passes are fused into independent execution stacks and tasks that stay local on one device. Tiling exposes parallelism and reduces storage requirements proportional to the tiling granularity, while fusing maximizes locality and thus minimizes communication overhead.

Each output tile is computed by pooling or convolving a certain dependent input region with the filter data. This dependent region includes the corresponding input tile along with some boundary data, which depends on the filter size and stride. The boundary data has to be communicated between the neighboring devices prior to starting the convolutions in both the forward and backward passes.

Alternatively, we can further combine multiple convolutional and pooling layers to form groups where communication of boundary data with neighboring tiles is done only at the beginning of each group. Within each group, any required intermediate data is locally computed from input data collected at the beginning of the group and no further communication is needed within the group. Fig. 3.1 shows 2 groups each in the forward and backward pass. In this case, communication is done at the feature-map inputs of layer L_1 and L_3 in the forward pass and at the delta gradient inputs of layer L_4 and L_2 in the backward pass. These feature and gradient maps serve as synchronization points where all tiles share boundary data with the neighboring tiles. Grouping introduces a trade off between communication and computation overhead. Larger groups have more redundant computation since the boundary data



(a) CNN with 3x3 tiling split into 2 groups

(b) Independent execution tasks with communication boundaries

Figure 3.1: Distributed training overview.

grows with the group size as illustrated by the funneling red arrows in Fig. 3.1. At the same time, larger groups synchronize less frequently whereas smaller groups have more communication and synchronization overhead. We will discuss optimal grouping strategies later.

The only points at which the entire partition needs to be communicated is when receiving the input training sample at the first layer and the initial delta gradient loss at the last layer. Once this is received, the forward pass and backward pass can be completed with just intermediate group boundary synchronization, which is a much smaller overhead.

Each task and device requires access to a complete copy of all filters. In order to update the filter weights during back-propagation, partial weight gradients computed by each task for each tile must be summed across all tiles to get the final weight updates. This requires the devices to communicate their entire partial weight update sets with each other or a common central device for summing at the end of the training cycle for each batch. Such weight updates are only required once at the end of each batch, and can stay local on each tile until then. For the early

feature-map dominated layers, filters are relatively small and storing local copies in each device as well as communicating updates between tiles carries a small overhead in comparison to the computation and memory benefits we get from feature and gradient map partitioning, as will be shown in Chapter V and VI.

Chapter 4: Distributed Training

In the following, we describe details of our distributed tiling approach for a single layer followed by a discussion of fusing and grouping across multiple layers.

4.1 Single-Layer Tiling

Fig. 4.1 illustrates the tiling process of a forward pass, backward pass and weight update at layer l for a 2x2 tile partition. X_l and X_{l+1} are the input and output feature maps of layer l respectively. Assuming a tiling into an NxM grid, in the forward pass, each of the NxM tiles in X_l, with the necessary boundary data, are convolved with the filter F_l, to produce the NxM tiled output feature-maps X_{l+1}.

For back-propagation, we need to compute two gradients, the delta loss gradients and the weight updates. The delta loss gradients are obtained through recursive back-propagation starting with the loss gradients at the output of the last layer. To calculate the loss gradients $\frac{\partial Loss}{X_{l+1}}$, each output tile of the next layer's loss gradients, $\frac{\partial Loss}{X_{l+2}}$, together with the necessary boundary data is convolved with the 180° rotated filter to produce the corresponding tile in $\frac{\partial Loss}{X_{l+1}}$.

Finally, to compute the weight updates, the feature map tiles of X_l are first convolved with the corresponding tiles of the delta loss gradient $\frac{\partial Loss}{X_{l+1}}$ to produce NxM filter gradient sets, one for each tile. These NxM weight updates are partial sums pertaining to the region of the map the tile is associated with, and the final weight gradients $\frac{\partial Loss}{F_l}$ can simply be obtained by summing them up.

This final gradient can then be used to update F_l as illustrated in the figure. The summation requires each device to communicate their partial sums to a common device that performs the summation and transmits the updated weight gradients back to each tile. To minimize overhead, the summation can be done once for all filters in all layers at the end of the training cycle of a single batch.

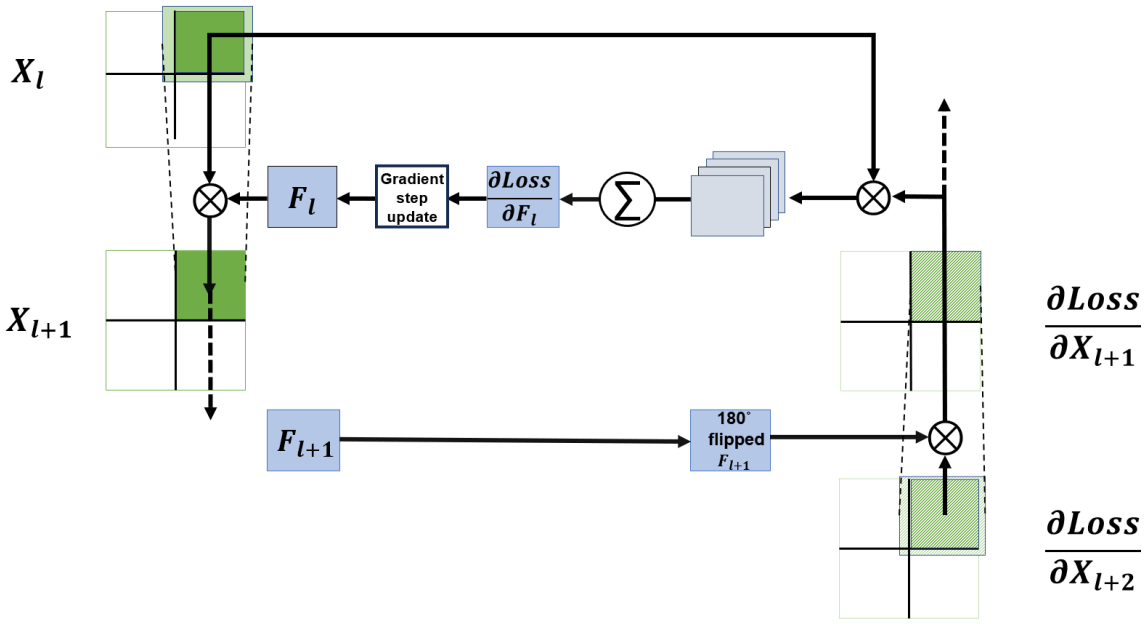
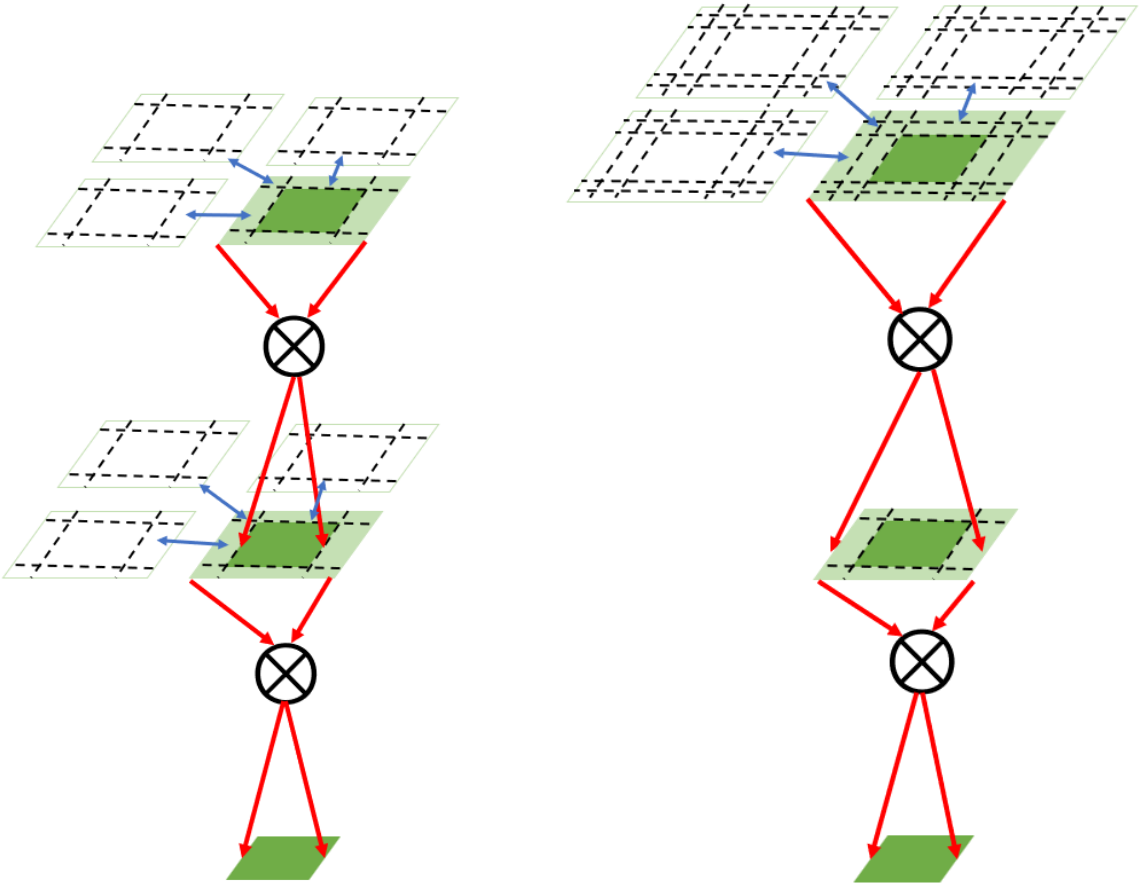


Figure 4.1: Single-layer tiled forward inference and back-propagation.

4.2 Fusing and Grouping

As introduced earlier, matching tile partitions in the forward and backward passes are fused across all convolutional and pooling layers in that they stay local on the same device. Fig. 4.2 illustrates the fusing and grouping across 2 layers for a forward pass (the backward pass is symmetrical). The center region of each tile (dark green) is fused across layers, stays local on the device and is never exchanged with other devices (in both forward and backward passes). However, the devices also exchange some neighboring boundary data (light green portion) required to complete the convolutions/pooling. Fig. 4.2(a) shows the case without grouping where boundary exchange occurs at the input to both layers thus having minimal redundant computation and storage. By contrast, Fig. 4.2(b) shows the case where the exchange only occurs at the input to first layer. However, in this case, the amount of shared boundary data per tile increases leading to more storage and redundant computation on each tile. This motivates the need for optimal grouping where sets of intermediate layers are aggregated to form groups and boundary data is exchanged at the input



(a) Fusing with 2 groups (b) Fusing with 1 group

Figure 4.2: Fusing and grouping illustration.

layers to the groups. Having more groups reduces the redundant computation and storage at the expense of additional communication and synchronization overhead, i.e. there is a trade-off between computation and communication that determines an optimal grouping profile.

Fig. 4.3 illustrates a granular view of communication at the group boundary. Each device both transmits and receives the required boundary data to/from up to 8 neighboring tiles, where devices transmit data from the internal border of the locally computed tile while receiving boundary data external to it. These exchanges happen

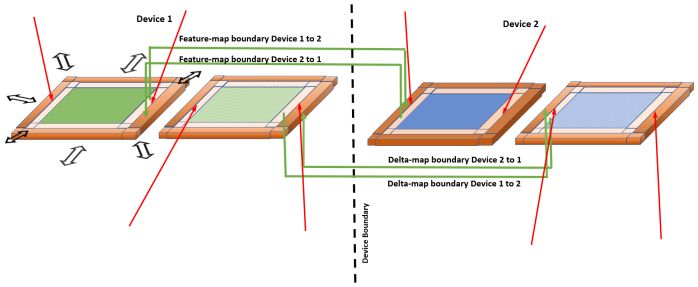


Figure 4.3: Group boundary communication illustration

at the group input layers in forward and backward passes. While they are shown to be the same in the figure, the group input layers do not have to be common in forward and backward passes. The double ended arrows at the feature map in device 1 indicate that similar exchanges occur with the other 7 neighboring tiles, if present. The same happens at group input delta maps and feature maps across all tiles.

To represent a tile in the partitioned grid at a layer, we use the notation $t_{l,(i,j)}$, where (i, j) represents the tile index and l represents the layer index in the network. The span of each tile with boundary data at layer l can be represented by its top-left $(x1_{l,(i,j)}, y1_{l,(i,j)})$ and bottom-right $(x2_{l,(i,j)}, y2_{l,(i,j)})$ co-ordinates. $t_{l,(i,j)} = [(x1_{l,(i,j)}, y1_{l,(i,j)}), (x2_{l,(i,j)}, y2_{l,(i,j)})]$. Furthermore, we represent the filter/kernel size and stride at layer l as $K_l \times K_l$ and S_l, respectively. A group starting at the input to layer s and ending at the input to layer e (output of layer $e-1$) is represented as a tuple (s, e). We define a grouping profile to be the vector $\mathbf{G}$ consisting of the list of start and end layer index tuples (s, e).

Suppose we have a grid of $M \times N$ tiles and want to create the grouping profile in the forward pass for an L layer network. To derive required boundary and tile data in the forward pass, we begin at the feature-map output of the last layer of the last group and recursively traverse backward among intermediate layers and groups. The feature map output of the last layer of the network is partitioned length and breadth wise equally among the tiles such that each tile has dimension $w_{L+1} \times h_{L+1} = \lceil \frac{W_{L+1}}{M} \rceil \times \lceil \frac{H_{L+1}}{N} \rceil$ where $W_{L+1} \times H_{L+1}$ is the width and height of the

feature map at the output of the last layer, L, of the network. We can zero-pad as necessary on outer tiles of the grid such that all tiles have uniform dimension. Then, for any group, (s, e), we recursively compute the dependent region in the previous layers to produce the required feature map for each tile in each intermediate layer l within the group, where $s < l \leq e$. Given the tile co-ordinates at the input to layer l (output of layer $l-1$), the required tile region at the input to layer $l-1$ is

$$x1_{l-1,(i,j)} = x1_{l,(i,j)} \times S_{l-1} - \lfloor \frac{K_{l-1}}{2} \rfloor \tag{4.1a}$$

$$y1_{l-1,(i,j)} = y1_{l,(i,j)} \times S_{l-1} - \lfloor \frac{K_{l-1}}{2} \rfloor \tag{4.1b}$$

$$x2_{l-1,(i,j)} = x2_{l,(i,j)} \times S_{l-1} + \lfloor \frac{K_{l-1}}{2} \rfloor + (S_{l-1} - 1) \tag{4.1c}$$

$$y2_{l-1,(i,j)} = y2_{l,(i,j)} \times S_{l-1} + \lfloor \frac{K_{l-1}}{2} \rfloor + (S_{l-1} - 1) \tag{4.1d}$$

when layer $l-1$ is convolutional, and

$$x1_{l-1,(i,j)} = x1_{l,(i,j)} \times S_{l-1} \tag{4.2a}$$

$$y1_{l-1,(i,j)} = y1_{l,(i,j)} \times S_{l-1} \tag{4.2b}$$

$$x2_{l-1,(i,j)} = x2_{l,(i,j)} \times S_{l-1} + (S_{l-1} - 1) \tag{4.2c}$$

$$y2_{l-1,(i,j)} = y2_{l,(i,j)} \times S_{l-1} + (S_{l-1} - 1) \tag{4.2d}$$

when layer $l-1$ is a pooling layer.

For the backward pass, computing group boundary data bounds is similar except that we go in the opposite direction, i.e. we start computing the co-ordinates from the delta gradient map output of the first layer of the network. We equally partition this delta map among the tiles and recursively compute the dependent regions in the later layers to produce the required segment of the delta gradient at the end of the first group. We then do the same for the later groups. For any group, (s, e), given the tile co-ordinates of the delta map at intermediate layer l, where $s \leq l < e$, the tile co-ordinates of the delta map at layer $l+1$ are

$$x1_{l+1,(i,j)} = \lceil \frac{x1_{l,(i,j)} - \lfloor \frac{K_l}{2} \rfloor}{S_l} \rceil \tag{4.3a}$$

$$y1_{l+1,(i,j)} = \lceil \frac{y1_{l,(i,j)} - \lfloor \frac{K_l}{2} \rfloor}{S_l} \rceil \tag{4.3b}$$

$$x2_{l+1,(i,j)} = \lfloor \frac{x2_{l,(i,j)} + \lfloor \frac{K_l}{2} \rfloor}{S_l} \rfloor \tag{4.3c}$$

$$y2_{l+1,(i,j)} = \lfloor \frac{y2_{l,(i,j)} + \lfloor \frac{K_l}{2} \rfloor}{S_l} \rfloor \tag{4.3d}$$

when layer l is convolutional, and

$$x1_{l+1,(i,j)} = \lfloor \frac{x1_{l,(i,j)}}{S_l} \rfloor \tag{4.4a}$$

$$y1_{l+1,(i,j)} = \lfloor \frac{y1_{l,(i,j)}}{S_l} \rfloor \tag{4.4b}$$

$$x2_{l+1,(i,j)} = \lfloor \frac{x2_{l,(i,j)}}{S_l} \rfloor \tag{4.4c}$$

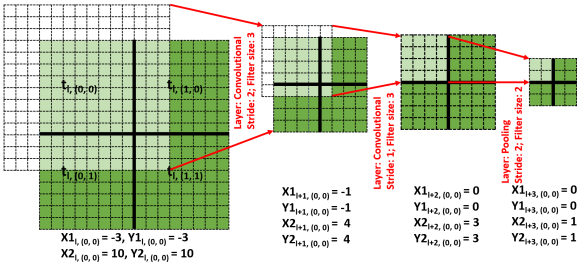
$$y2_{l+1,(i,j)} = \lfloor \frac{y2_{l,(i,j)}}{S_l} \rfloor \tag{4.4d}$$

when layer l is a pooling layer.

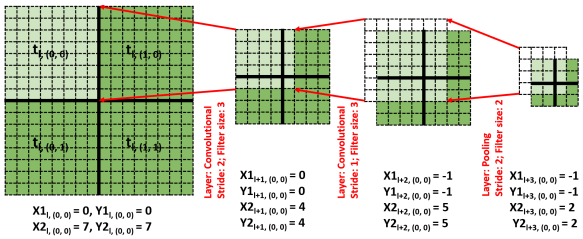
For tiles on the outer edge and corners that are not fully surrounded by neighbors, we pad zeros in place of the boundary data. Communication is not necessary in that case.

Fig. 4.4 illustrates examples of finding the co-ordinates for a 3-layer group in the forward and backward passes for a 2x2 tile setup. In the forward pass, the 4x4 output of layer $l+2$ is partitioned evenly into 2x2 tiles and the above equations for inference grouping are applied to get the dependent region at the input to layer l. Likewise, on the backward pass, the delta gradient output of layer l has a total dimension of 16x16 and is partitioned evenly into 8x8 tiles. Applying the equations for back-propagation grouping, we find the dependent region in the delta map input to layer $l+2$.

After completing the forward and backward passes, the partial filter gradients are computed by convolving the corresponding delta gradients with the feature-maps as described in the previous section. For this, just $\lceil \frac{K_l}{2} \rceil$ element wide boundary data would be required in the feature-map at layer l. However, this data is already gathered



(a) Tile co-ordinates example for a 3-layer group in forward pass



(b) Tile co-ordinates example for a 3-layer group in backward pass

Figure 4.4: Grouping examples.

during the forward pass and can be re-used to avoid additional communication for this step.

4.3 Distributed Training Framework

Algorithm 1 summarizes the flow of training a single sample on one device. The initial requirements for the device include the grid tile partitioning dimensions, the index of the current tile in the grid, the network parameters, grouping profiles in forward and backward passes and the ID of the gateway device responsible for distributing samples and updating partial weights. An assigned gateway device distributes the desired slices of input samples and the output layer delta losses for the batch to the devices. Each tile then iterates through the forward groups, exchanging boundary data for each group and computing forward inference for each layer in each group. Likewise, the backward pass is executed. If the device is the assigned gateway,

Algorithm 1 Distributed training cycle of a single sample

Require: -

1. Device ID - index of current tile partition (i, j)
2. Grouping profiles in forward and backward pass - $\mathbf{G_f}$ and $\mathbf{G_b}$
3. Network parameters - number of layers L
4. Network configuration - $N_l, 1 \leq l \leq L$.
5. Initial filter weights $F_l, 1 \leq l \leq L$
6. Gateway device ID - (i_{gw}, j_{gw})

Receive input feature map data X_1 from gateway device
for each $(s, e) \in \mathbf{G_f}$ **do**
    for each device (m, n) in Neighbors(i, j) **do**
        ExchangeSharedBoundaryData(m, n, i, j)
    end for
    for each $s \leq l < e$ **do**
        X_{l+1} = ForwardInference(l, N_l, X_l, F_l)
    end for
end for
Send output feature map X_{L+1} to gateway
Receive output delta map $\frac{\partial Loss}{\partial X_{L+1}}$ from gateway
for each $(s, e) \in \mathbf{G_b}$ **do**
    for each device (m, n) in Neighbors(i, j) **do**
        ExchangeSharedBoundaryData(m, n, i, j)
    end for
    for each $e \geq l > s$ **do**
        $F_{l,partial}, \frac{\partial Loss}{\partial X_{l-1}}$ =BackPropagation$(l, N_l, \frac{\partial Loss}{\partial X_l}, F_l)$
    end for
end for
if $(i, j) = (i_{gw}, j_{gw})$ **then**
    for each device $(m, n) \neq (i_{gw}, j_{gw})$ in cluster **do**
        for each $1 \leq l \leq L$ **do**
            $F_{l,partial}$+=ReceivePartialWeightUpdates(m, n, l)
        end for
    end for
    for each $1 \leq l \leq L$ **do**
        F_l = UpdateWeights$(F_{l,partial})$
    end for
    for each device $(m, n) \neq (i_{gw}, j_{gw})$ in cluster **do**
        for each $1 \leq l \leq L$ **do**
            SendUpdatedWeights(m, n, l, F_l)
        end for
    end for
else
    for each $1 \leq l \leq L$ **do**
        SendPartialWeightUpdates$(i_{gw}, j_{gw}, l, F_{l,partial})$
    end for
    for each $1 \leq l \leq L$ **do**
        F_l = ReceiveUpdatedWeights(i_{gw}, j_{gw}, l)
    end for
end if

it waits to receive the partial weight updates from other devices while accumulating the partial weight updates as it receives them to avoid storing multiple copies. The updated weights are then sent back to the devices. If the device is not the gateway, it simply sends its partial weights to the gateway and waits to receive the updated weights. The training cycle on the next batch can then begin.

In addition, CNNs include activations and typically some form of batch normalization, and we discuss how the approach can be easily extended to include these features.

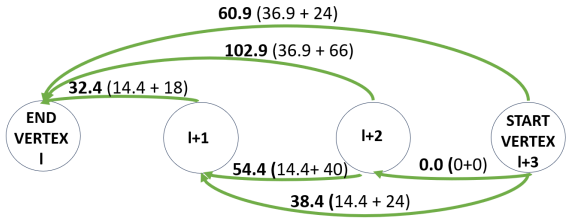
Since activations are just functions of individual outputs in feature maps, the default model can include activation layers by simply applying the function (forward and backward gradient) independently in each tile without any communication.

CNN training also generally includes batch normalization after each convolutional layer and bias updates after training a batch. This involves computing the mean and variance of feature-maps and delta gradients. The overall mean and variance can be computed by taking the mean and variances of each individual tile. Thus, each tile can compute its mean and variance, then communicate the 2 values to a common tile that aggregates the values from all tiles and sends the overall mean and variance back to each tile. Thus, this process is similar to filter updates, but the communication overhead is very small since there are only 2 values for each output channel to be communicated by the tiles.

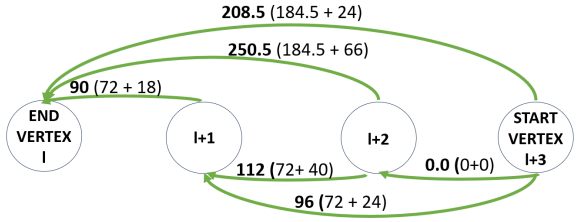
Chapter 5: Grouping Optimization

We described how our distributed tile partitioning approach works to complete forward and backward passes while maximizing locality and minimizing communication overhead. As discussed earlier, this approach avoids the need to communicate entire feature-maps in intermediate layers. However, as also discussed previously, boundary data shared between tiles in each layer has to be either shared and communicated or redundantly computed in each device. Grouping determines at which layers boundary data is communicated vs. computed. The grouping profile, i.e. how many groups to have and at what layers to share boundary data in forward and backward passes determines the tradeoff between computation and communication. Having more groups always lowers redundant computation and adds fixed synchronization overhead. While grouping generally shifts boundary data communication from intermediate to group input layers (see Fig. 3), the effect of grouping on the amount of boundary data communicated, however, is more nuanced since there is variation due to factors such as varying depths (channels) of feature maps across layers and varying filters sizes and stride.

In our distributed model, there are also some fixed computation and communication costs such as filter updates and distribution of input image and output delta samples among tiles. These fixed costs are independent of the grouping profile. What we can control through the grouping profile is the fixed synchronization, shared boundary data communication and redundant computation overheads. In this chapter, we introduce a cost model that takes in computation and communication costs of the devices as well as a grouping profile as inputs and returns the total cost to perform one cycle of training. We then propose an algorithm that finds an optimal grouping profile using the cost model. Finally, we evaluate the cost model and grouping profiles on the reference Yolov2 network and discuss some implications of communication and computation overhead, the relative significance of each in the total cost and expected



(a) comp cost = 0.1; comm cost = 2



(b) comp cost = 0.5; comm cost = 2

Figure 5.1: Grouping graph algorithm examples.

performance improvements.

5.1 Cost Model

We use a simple model to account for computation and communication cost per tile during optimization. The computation cost for a given tile in a given layer is proportional to the floating-point multiply accumulate (MAC) operations performed by the tile in that layer. By contrast, the communication cost for a group is proportional to the total boundary elements exchanged at the input to the group. For any tile (i, j) in layer l, we can derive its feature map dimension $w'_l \times h'_l \times D_l$ including shared boundary data using the co-ordinate equations (1)-(4) from Chapter 4, where $w'_l = (x2_{l,(i,j)} - x1_{l,(i,j)} + 1)$, $h'_l = (y2_{l,(i,j)} - y1_{l,(i,j)} + 1)$ and D_l is the feature map depth. Additionally, following what was described in Chapter 4, we define the dimensions of the feature map without boundary data to be $w_l \times h_l \times D_l$

The computation cost for tile (i, j) in layer l with filter dimensions $K_l \times K_l \times D_l$, D_{l+1} filters and stride S_l, is equivalent to the number of MAC operations $O_l =$

$w'_l \times h'_l \times D_l \times K_l \times K_l \times D_{l+1}/(S_l \times S_l)$. We divide by $S_l \times S_l$ since we assume that the convolution is implemented to avoid multiplying by intermediate zeros. To model the total computation cost, P_g, for a group $g = (s, e)$ with start and end layer indices s and e, respectively, we compute the weighted sum of the computation costs of all layers in the group: $P_g = \sum_{l=s}^{l=e-1}(c_p \cdot O_l)$, where scalar c_p is the fixed computation cost per operation. The backward pass will have a similar cost for the delta gradient back-propagations across each group.

For estimation of communication costs, the amount of shared boundary data to be communicated is found by subtracting the dimension of the original feature map (without boundary data) from the dimension that includes boundary data. Thus, the number of shared boundary elements B_g for group $g = (s, e)$ is simply $B_g = D_s \times ((w'_s \times h'_s) - (w_s \times h_s))$. The data has to be communicated across the entire depth, D_s, of the feature map. This gives a group communication cost of $C_g = c_c \times B_g$, where c_c is the fixed communication cost per element.

The total cost for a given grouping profile, $\mathbf{G}$, would then just be the sum of computation and communication costs for each group. The problem of finding an optimal grouping profile can be setup as an optimization problem using this total cost model:

$$\arg\min_{\mathbf{G}} \sum_{g \in \mathbf{G}} (P_g + C_g + c_f), \tag{5.1}$$

where $\mathbf{c_f}$ is an additional term that represents the fixed overhead per communication transaction (synchronization overhead). We can apply the cost model independently in forward and backward passes.

5.2 Grouping Algorithm

It is infeasible to find an analytical solution to the above optimization problem due to the combinatorial nature of how many groups there are and where each group starts and ends.

Algorithm 2 Optimal Grouping Algorithm

```
1: N = Number of feature maps (including output of last layer)
2: Initialize Vertex set V = N indices of feature maps
3: for i = (1, N − 1) do
4:     for j = (i + 1, N) do
5:         Edge E(i, j) = TotalGroupCost i → j
6:     end for
7: end for
8: Invoke Dijsktra's shortest path for the graph G(V, E)
9: Sequence of nodes returned is the optimal grouping profile
```

We instead present an algorithm for finding optimal grouping profiles. The problem can be fundamentally represented as a graph, where each feature map in the network is a node/vertex in the graph. A directed edge (i, j) connecting the feature map i to the feature map j is assigned a weight that equals the total computation plus communication cost of traversing a group that starts at the input to layer i and terminates at the output to layer $j - 1$. Given such a graph, the optimal grouping profile is determined by the shortest path through this graph. The vertices of the shortest path in this weighted graph from start vertex (input to initial layer of the network) to end vertex (output of final layer of the network) are the layers at which a new group begins and boundary exchange communication occurs. This would be the optimal grouping profile for the given computation and communication cost.

Algorithm 2 shows how we find the optimal grouping solution. To first construct the graph, we find the costs of all possible groups in the network. This can be done by iterating through every layer and finding the cost of the groups starting at the input feature map to that layer (boundary synchronization point) and terminating at each successive feature map in the network. Weighted edges are inserted into the graph using these group costs. The shortest path itself can be found using Dijsktra's algorithm [28]. The graph and algorithm can similarly be applied in the backward pass and in this case the synchronization points (vertices) would be the delta maps.

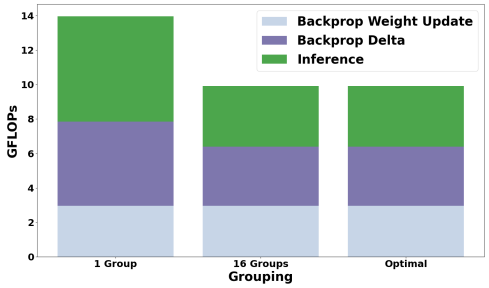
Figure 5.1 illustrates the transformation of the 3-layer network example from Fig. 4.4(b) into the corresponding cost graph for the backward pass. Fig. 5.1(a) shows

a case where we assume the computation and communication costs per element to be $c_p = 0.1$ and $c_c = 2$, respectively. We ignore any fixed synchronization overhead ($c_f = 0$) in this example and assume the case of a generic tile surrounded by all 8 neighbors. For the case of having 1 group (synchronizing only at input delta map $l+3$), the total cost is 60.90. The communication cost is determined by the amount of boundary data needed at the input delta map $l+3$, which consists of 12 elements as can be seen from Fig. 4.4(b). With this, we get a total communication cost of $c_c \times (12)$ = 24. For computation, the group starting a delta map $l+3$ and ending at map l will incur computation costs as the sum of the costs for all three layers. Layer $l+2$ is a pooling layer that simply assigns the delta map elements to appropriate indices in the output. We ignore the cost for any pooling layers since there are no multiply-accumulate operations and their cost is negligible compared to that of convolutional layers. Hence, the computation cost for the third layer is 0. For layer $l+1$, a 7x7 tile ($l+2$ with boundary data) produces a 5x5 tile ($l+1$). With a 3x3 filter and stride 1, this translates into 5x5x3x3 = 225 MAC operations. Similarly, the output delta map $l+1$ of layer $l+1$ serves as the input map to layer l, and for the filter size 3 with stride 2, we find the MAC operations here to be 8x8x3x3/(2x2) = 144. Thus, the total MAC operations are 225 + 144 = 369. Multiplying this by the fixed computation cost per operation of 0.1, we get total computation cost for the group to be 36.90. The total group cost is then 24+36.90 = 60.90.

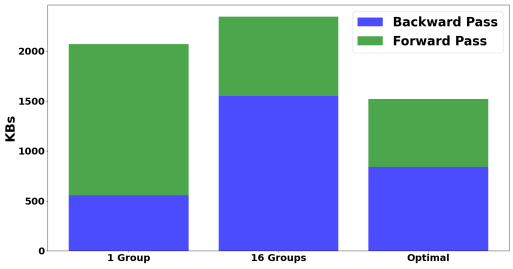
We can likewise programmatically iterate and find the other group costs and construct the graph. It turns out that the 1 group we worked out is optimal in this case. If we, however, increase the computation cost to say 0.5 (Fig. 5.1(b)), we find that the optimal solution is to synchronize at $l+3$ and $l+1$ (2 groups).

5.3 Grouping Analysis

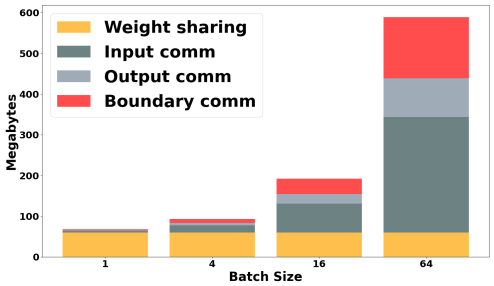
To further demonstrate and analyze grouping tradeoffs, we applied our model on the Yolov2 network, assuming a 3x3 tile partitioning. In particular, the model



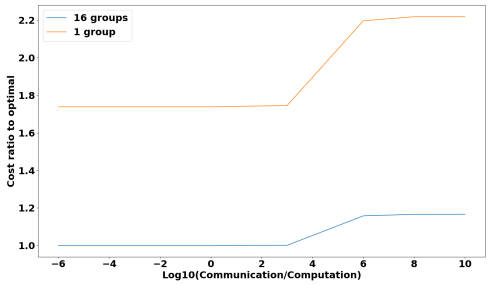
(a) Computation cost with grouping



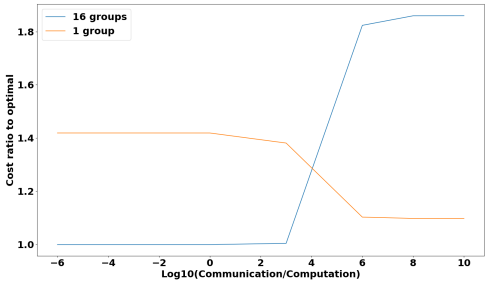
(b) Boundary communication



(c) Communication components with batch size



(d) Grouping profile comparison for cost-ratios in forward pass



(e) Grouping profile comparison for cost-ratios in backward pass

Figure 5.2: Predicted per tile costs for a 3x3 tile partition on Yolov2 for various grouping profiles.

was evaluated on the first 16 layers of Yolov2. These are feature-dominated layers and include 12 convolutional and 4 pooling layers. The input dimension of the image sample was fixed to be 608x608x3. For analysis of grouping profiles and their impact, we consider the optimal solution returned by the algorithm along with two corner

cases of grouping - having one large group (all boundary data collected at the first layer) and having 16 groups (where boundary data is synchronized at each of the 16 layers).

Fig. 5.2(a) illustrates the computation cost estimate for a single sample. From a computation standpoint, we see that having 16 groups performs better than having one large group. The grouping profile with minimum redundant computation is optimal for computation. We can intuitively see even without applying the algorithm that synchronizing at every layer results in the solution with the least redundant computation. Thus, for devices limited by processing speed, it is optimal to synchronize at every layer.

However, synchronizing every layer is not necessarily the best solution in terms of boundary communication. Fig. 5.2(b) shows an expanded view of the boundary data communication cost with grouping. We again compare the overhead of having 1 group, 16 groups and the optimal solution returned by our algorithm from a communication standpoint. We see that the optimal grouping solution (a solution with 5 groups) does better than having 1 or 16 groups in terms of communication.

To get the overall optimal solution, we can input the relative ratio of computation to communication cost into the algorithm. However, boundary communication overhead is only part of the total communication cost in distributed CNN training. Fig. 5.2(c) shows the amount of communication for each component in training Yolo2 as a function of batch size. For a single sample (batch size 1), the fixed overhead of filter communication is the limiting factor, and the input/delta sample distribution communication overhead is also greater than that of the boundary communication. Thus, any optimization of boundary data by the algorithm based on grouping profiles would be insignificant. However, with increasing batch sizes, the filter communication overhead stays the same while the input/output samples and boundary communication scales linearly with batch size, becoming the limiting factor for large batch sizes. The fixed input/output sample communication still has potentially greater overhead

than that of boundary communication and this cannot be optimized by the algorithm. Communication of initial and final input and output samples is, however, implementation-dependent. Assuming that input and output data is pre-distributed or does not need to be aggregated, respectively, i.e. does not need to be communicated in real time, boundary communication becomes the only remaining component. In this case, for high-latency networks, the boundary communication optimization by the algorithm can be meaningful.

Finally, we did some analysis on the effect of varying communication to computation cost ratio for the grouping profiles on the overall performance. Fig. 5.2(d) shows the cost factor for forward pass on a single sample for 1 and 16 groups with varying communication to computation cost ratio. The base of 1 represents the optimal solution and the plots of 1 and 16 group costs represent the cost with respect to the optimal. We see that regardless of the ratio, 1 group does worse than 16 groups (ignoring fixed synchronization overhead). This is expected because having 1 group needs more redundant computation and more communication since there is more boundary data to accumulate at the larger initial layer as opposed to at successive smaller layers. Hence, unless we are trying to minimize fixed synchronization overhead, having 1 group has heavy communication and computation cost for the forward pass. For very high communication costs, there is a solution having 5 groups that minimizes communication overhead that is optimal for communication limited networks. For the backward pass in 5.2(e), (ignoring filter-update and fixed synchronization overhead), we see that there is actually a trade-off in the grouping profiles. For communication limited setups (high ratio), having 1 group is better since although it has greater redundant computation, it accumulates all data at the last layer (smallest delta map) thus minimizing communication overhead.

Chapter 6: Experiments and Results

We implemented our distributed training approach in C on top of the Darknet framework from [29, 17] and validated our model using the first 16 layers of the Yolov2 CNN. Our primary experimental test-bed consisted of 6 Raspberry-Pi3 devices with quad-core ARM Cortex-A53 CPUs and 1 GB of RAM each running a Linux kernel. Each tile was executed as an individual linux process and we allocated upto 4 tiles per device to run on the 4 cores. The devices were all part of a local 100Mbps Ethernet network. For communication between processes within the same device, we used shared memory and local sockets to minimize overhead. TCP network sockets were used to communicate between processes across devices on the network.

6.1 Speedup

Fig. 6.1 shows the execution time results for a single training sample (batch size of 1) across different combinations of devices and cores ranging from 1 to 6 devices, each using 1 to 4 of their cores. The number of tiles in a given device/core combination is the total number of cores across all devices. Each tile was scheduled as an independent process. A baseline single device with 1 tile (1 process - single core) took around 7 minutes to finish the training cycle (forward pass, backward pass and weight updates) on a single sample. The speedup factors for the different configurations are shown with respect to this baseline reference. Since filter weight updates are only done once per batch, we show 2 speedup factors for a baseline batch size of 1, where weight communication overhead dominates, and for infinite batch size where weight update cost is negligible compared to other components and excluded. The actual speedup should be between these 2 depending on the batch size.

We observe that computation times dominate for small number of devices and tiles, but scale down with increasing number of devices and cores (more tiles).

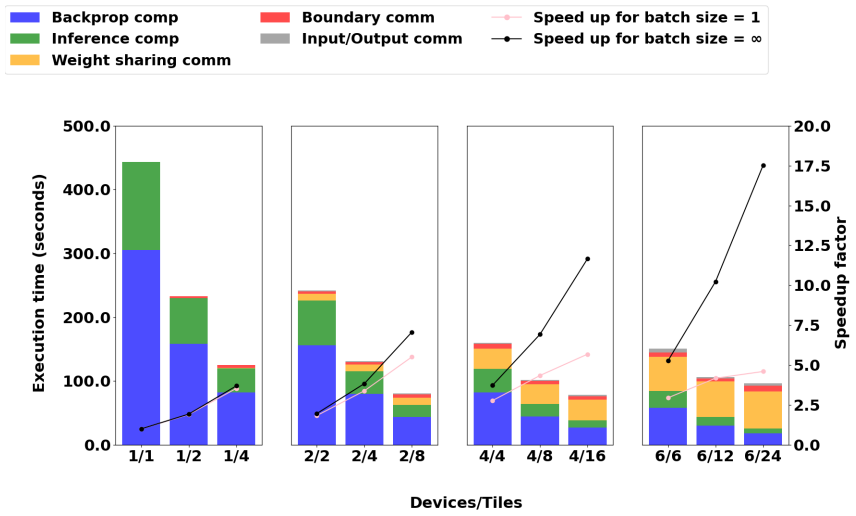


Figure 6.1: Execution time split and speedup with number of tiles and devices

Due to the shared memory implementation within devices, there is no overhead for communication between tiles on the same device. Consequently, the communication overhead is uniform across different numbers of cores with the same number of devices. However, boundary data and weight communication overhead increases with more devices, where overhead for weight updates dominates for a larger number of devices. This limits speedup for small batch sizes and can outweigh savings in computation times, where 6 devices perform worse than 4. At the same time, we do observe strong scaling in the speedup for large batch sizes. We will further analyze results with varying batch size later.

6.2 Memory

Fig. 6.2 shows the peak physical memory utilization per tile measured while the training cycle of a single sample on the Raspberry-Pis was in progress. The figure also shows the split of the major memory usage components - feature maps, delta maps, filters and other implementation-related components such as a preallocated buffer for intermediate computation, communication buffers and code space. The

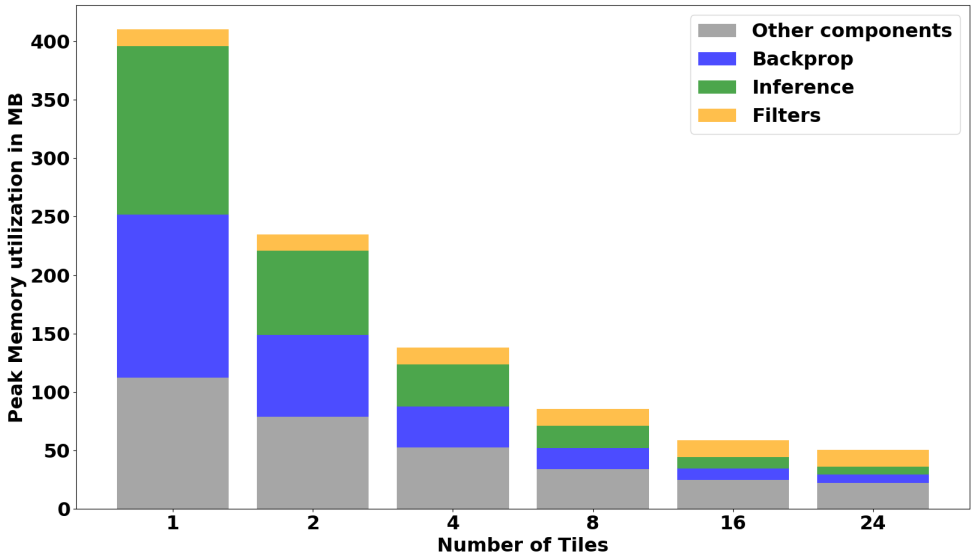


Figure 6.2: Memory utilization with number of tiles

memory consumption is ∼400 MB per tile and drops to ∼50 MB per tile when using 24 tiles. In general, by tiling in a finer granularity, memory requirements per tile and hence per device are reduced. However, while memory requirements for feature, delta maps and other buffers decreases linearly with the number of tiles, filter memory usage is constant, which leads to diminishing returns.

6.3 Batching and Grouping

We further conducted experiments on a batch of samples of various sizes. We also performed a comparison of different grouping profiles with a single group and 16 groups (1 layer per group). Fig. 6.3 shows the result of running the training cycle on a batch size of 1 to 8 samples. We conducted this experiment using all 4 cores on the 6 Rapberry Pi devices using 24 tiles. We observe that synchronizing every layer (16 groups) performs significantly better than having a single group across all batch sizes. In case of the Raspberry Pis, total execution time is dominated by

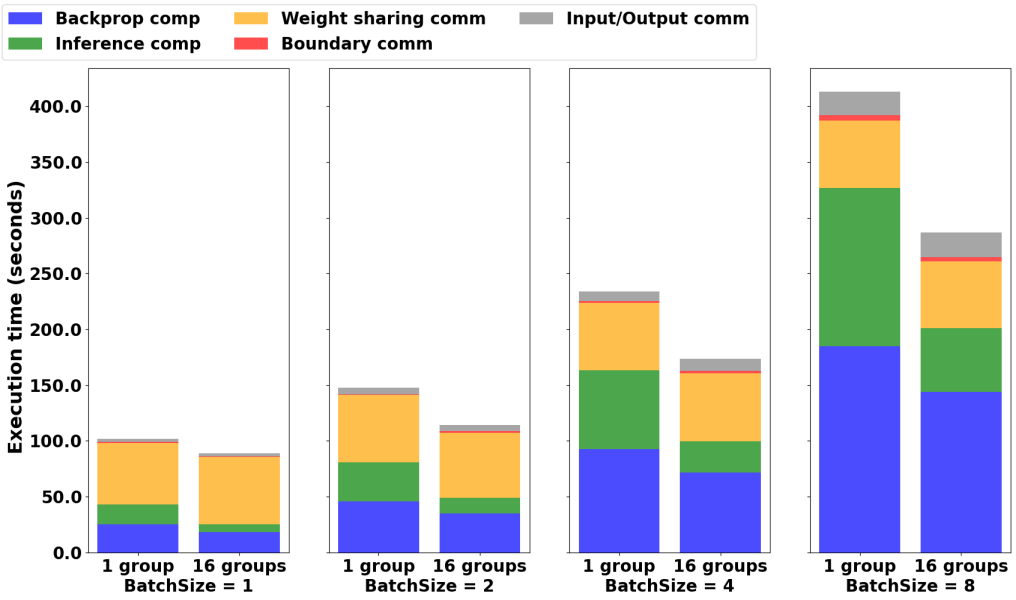
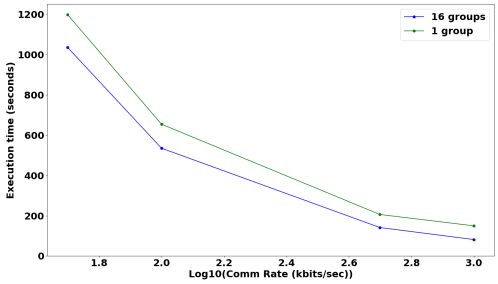


Figure 6.3: Comparison with batch size and grouping

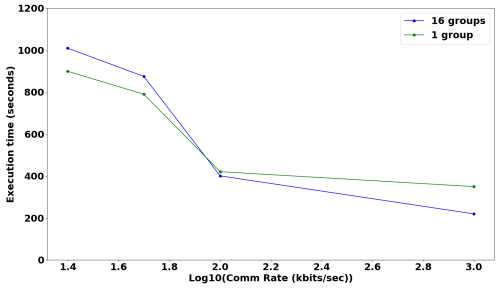
computation times or weight updates, and the improvement comes from optimization of computation as predicted by our cost model analysis earlier. Computation costs scale proportionally with the number of samples in the batch, but the filter updates are done once per batch and take roughly the same time across batch sizes. As such, the relative contribution of weight update costs decreases with larger batches. At the same time, the boundary communication and input/output communication overhead increases with larger batch size, but is negligible compared to computation cost. Overall, Raspberry-Pi devices are computation limited and hence the synchronizing at every layer to minimize redundant computation is optimal.

6.4 Network-Emulator Experiments

Additionally to test the effect of varying communication to computation ratio, we conducted experiments on the Raspberry-Pi setup using the Linux network emulator tool, NetEm. Fig. 6.4 shows the results of individually running a single



(a) Forward pass



(b) Backward pass

Figure 6.4: Grouping experiments with communication rate control via network emulator

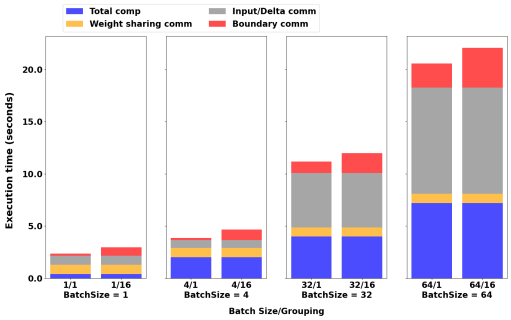
sample of forward pass and backward pass for different communication rates. The communication rate was varied from 50 kilobits/sec to 1 Megabit/sec with identical computation setup. We see that in the forward pass, Fig. 6.4(a), having 16 groups and synchronizing every layer consistently performs better than having 1 group as predicted earlier due to the higher communication and redundant computation cost in forward pass when using larger groups. However, in the backward pass, we see the trade-off predicted that after increasing communication cost beyond a certain threshold, the 1 group solution performs better. We do not include the fixed filter update overhead in this analysis, and it just illustrates the forward and backward pass grouping performance with communication rate.

6.5 GPU Experiments

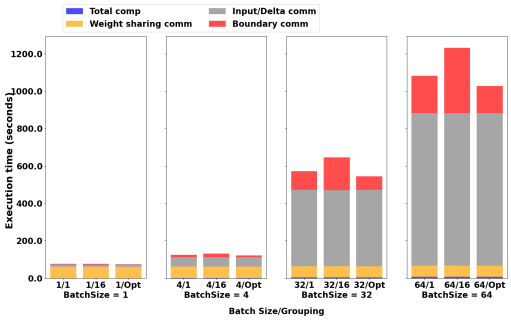
We also conducted experiments on a pair of Nvidia-Jetson Nano boards to illustrate the case of a communication-limited setup. Each board had a quad-core ARM Cortex-A57 CPU and a Maxwell architecture GPU with 128 CUDA cores. The 2 boards were connected using a 10Gbps Ethernet link.

Fig. 6.5(a) illustrates the single batch training cycle time for different batch sizes for a 2-tile setup (each board training on the GPU). Here, the inference plus backprop computation is very fast, and the communication and synchronization overhead is the limiting factor. In this case, the difference in boundary communication overhead among different groupings though small is noticeable. The case of 1 group performs better than 16 groups since it synchronizes less frequently. On the GPUs, the extra redundant computation in the 1 group case has negligible effect on computation time. By contrast, the extra communication and frequent synchronization, which includes transferring data to and from the GPU incurs a relatively larger overhead. Hence, it is more optimal to synchronize less frequently, and having 1 group is optimal.

Fig. 6.5(b) illustrates results of an experiment where the communication rate



(a) 10 Gbps communication bandwidth



(b) 2 Mbps communication bandwidth

Figure 6.5: GPU experiments: 2 GPUs to make a 2 tile setup

was further limited to 2 Mbps with the Linux NetEm network emulator tool. Due to the slow communication rate, the data communication overhead dominates computation and any fixed synchronization overhead. In this case, the grouping that minimizes the total data to be shared performs better than having 1 or 16 groups. The optimal solution has 5 groups, and although it synchronizes more frequently than in the case of 1 group, the amount of shared boundary data it exchanges is lower. Since the fixed synchronization overhead is not as significant, the 5 group solution performs better.

It is important to emphasize here that the input and delta communication overhead is much more significant than that of the boundary communication/synchronization. Hence, the overall difference in performance is minor regardless of grouping. However, as discussed earlier, the method in which the samples are distributed is implementation-dependent, and if we assume that the input and output data samples are pre-distributed/loaded onto the devices through other means and not done in real time, the only communication would be boundary and filter updates. In this case, the effect of boundary communication optimization would be significant.

6.6 Comparision to Traditional Approaches

Finally, we conducted an experiment to compare the performance of our distributed edge training framework with approaches that are based on existing parallelization and partitioning methods used in cloud settings. We implemented and compared against solutions using data parallelism and layer-partitioning on Yolov2. In the case of data-parallelism, each device contained model weights of all 16 layers. The batch samples were evenly distributed among the number of devices and cores. For layer partitioning, we split the layers across devices and realized a pipelined implementation.

Fig. 6.6 shows the comparision of our approach with the data- and layer-partitioning implementations on Nvidia Jetson CPUs. 8 cores were utilized across 2

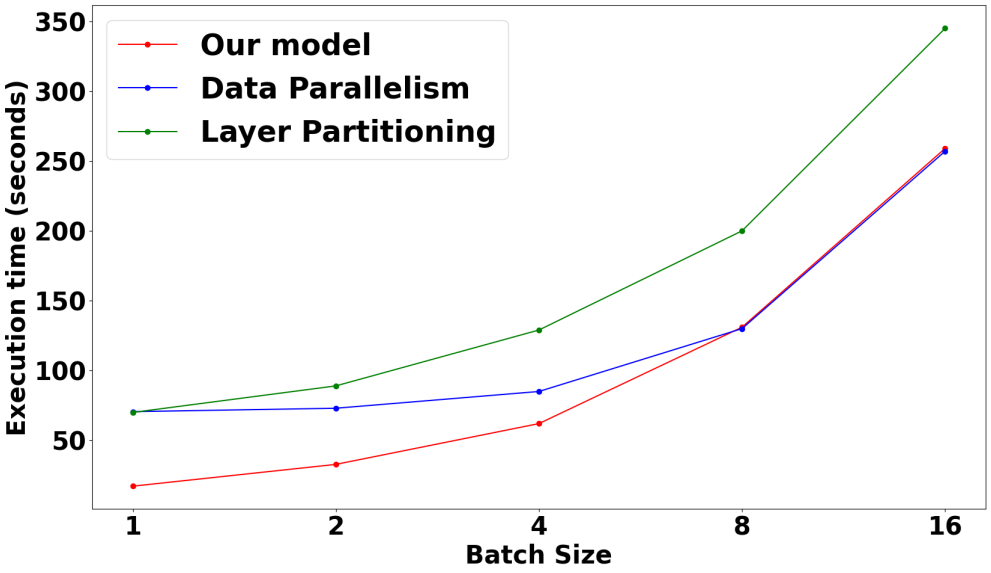


Figure 6.6: Comparision of our model to layer partitioning and data parallelism with 8 cores for varying batch sizes

devices to distribute 8 tiles. For layer partitioning, the 16 layers were split among the 8 cores to distribute the computation as evenly as possible. Likewise, for data parallelism, the samples in the batch were evenly distributed among the 4 cores.

We observe that for small batch sizes, our approach does significantly better than existing approaches since it provides parallelism within each sample thus minimizing latency at a more granular level. Once the number of samples in the batch equals or exceeds the number of cores, data parallelism performance becomes comparable. Data parallelism is expected to be slightly more efficient in this case since it does not have some of the overheads such as boundary synchronization. However, data parallelism incurs significantly higher memory requirements. In case of memory-limited devices, this can make data parallel solutions infeasible or lead to high overheads for swapping.

Both a data parallel and our solution perform better than layer partitioning since the parallelism in layer partitioning only comes in full effect when all cores are

computing and the pipeline is full. Additionally, layer partitioning has higher communication overhead since entire feature maps and delta maps have to be communicated in intermediate layers.

Chapter 7: Summary and Conclusions

In this report, we proposed a method for distributed mobile and edge training in feature-map dominated convolutional and pooling layers. Our method exploits locality in convolutional layers to partition feature maps and the delta gradients in forward and backward passes. It parallelizes training at a granular within each sample as opposed to mini-batch level parallelism in traditional approaches. All intermediate layers are fused in that the core feature maps and delta maps remain local on the device with only a small overhead of shared data communication between neighboring tiles. Layers are further grouped based on a grouping profile that affects tradeoffs between computation, shared boundary communication and synchronization overhead, where we proposed a cost model and algorithm to find the optimal grouping profile.

We evaluated our model on the Yolov2 network. Results showed that for a cluster of 2-6 quad-core Raspberry Pi3 devices, training of an object-detection CNN provided a 2x-15x speedup with respect to a single core and up to 8x reduction in memory usage per device. Optimally grouping offered up to 1.5x speedup depending on the reference profile and batch size. Compared to traditional data and model parallelism, on an 8-core setup, our approach provided up to 4x speedup for a batch size of one.

Future work will explore weight partitioning techniques and how to extend our approach to the later weight dominated layers.

References

[1] W. Ren, Y. Qu, C. Dong, Y. Jing, H. Sun, Q. Wu, and S. Guo, "A Survey on Collaborative DNN Inference for Edge Intelligence," *arXiv preprint arXiv:2207.07812*, 2022.

[2] R. Stahl, A. Hoffman, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "DeeperThings: Fully Distributed CNN Inference on Resource-Constrained Edge Devices," *International Journal of Parallel Programming*, vol. 49, no. 4, pp. 600–624, 2021.

[3] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, "Adaptive Parallel Execution of Deep Neural Networks on Heterogeneous Edge Devices," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. Association for Computing Machinery, 2019.

[4] J. Du, M. Shen, and Y. Du, "A Distributed In-Situ CNN Inference System for IoT Applications," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020.

[5] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, 2021.

[6] P. Joshi, M. Hasanuzzaman, C. Thapa, H. Afli, and T. Scully, "Enabling All In-Edge Deep Learning: A Literature Review," *IEEE Access*, vol. 11, pp. 3431–3460, 2023.

[7] P. P. Ray, "A review on TinyML: State-of-the-art and prospects," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 4, pp. 1595–1623, 2022.

[8] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, "On-device training under 256KB memory," in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[9] H.-Y. Chiang, N. Frumkin, F. Liang, and D. Marculescu, "MobileTL: On-device Transfer Learning with Inverted Residual Blocks," *arXiv preprint arXiv:2212.03246*, 2022.

[10] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated Learning: Challenges, Methods, and Future Directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[11] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, "When Edge Meets Learning: Adaptive Control for Resource-Constrained Distributed Machine Learning," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018.

[12] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

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

[14] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu, "DeepSlicing: Collaborative and Adaptive CNN Inference With Low Latency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2175–2187, 2021.

[15] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," *MICRO-49: The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, no. 22, 2016.

[16] R. Stahl, D. Mueller-Gritschneder, and U. Schlichtmann, "Fused depthwise tiling for memory optimization in tinyml deep neural network inference," 2023.

[17] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," *arXiv preprint arXiv:1612.08242*, 2016.

[18] Hecht-Nielsen, "Theory of the backpropagation neural network," in *International 1989 Joint Conference on Neural Networks*, 1989.

[19] Z. Wang, H. Xu, Y. Xu, Z. Jiang, and J. Liu, "CoopFL: Accelerating federated learning with dnn partitioning and offloading in heterogeneous edge computing," *Comput. Netw.*, vol. 220, p. 109490, 2023.

[20] T. Sen and H. Shen, "Distributed Training for Deep Learning Models On An Edge Computing Network Using Shielded Reinforcement Learning," *arXiv preprint arXiv:2206.00774*, 2022.

[21] M. M. Grau, R. P. Centelles, and F. Freitag, "On-device Training of Machine Learning Models on Microcontrollers With a Look at Federated Learning," in *Proceedings of the Conference on Information Technology for Social Good*. Association for Computing Machinery, 2021.

[22] H. Cai, C. Gan, L. Zhu, and S. Han, "TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning," *arXiv preprint arXiv:2007.11622*, 2021.

[23] Y. Chen, K. Zhao, B. Li, and M. Zhao, "Exploring the Use of Synthetic Gradients for Distributed Deep Learning across Cloud and Edge Resources," in *HotEdge*, 2019.

[24] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: generalized pipeline parallelism for DNN training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, 2019.

[25] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[26] S. Akintoye, L. Han, H. Lloyd, X. Zhang, D. Dancey, H. Chen, and D. Zhang, "Layer-wise Partitioning and Merging for Efficient and Scalable Deep Learning," *Future Generation Computer Systems*, vol. 149, 2023.

[27] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," 2018.

[28] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

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