Distributed Convolutional Neural Network Training on Mobile and Edge Clusters

Pranav Rama, Madison Threadgill, and Andreas Gerstlauer

Electrical and Computer Engineering
The University of Texas at Austin, Austin TX, USA
{pranavrama9999, madison.threadgill, gerstl}@utexas.edu

Abstract. The training of deep and/or 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 paper, 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. 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. Grouping offers up to 1.5x speedup depending on the reference profile and batch size.

Keywords: Distributed edge computing · machine learning

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, 2]. 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 [3–5]. 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. 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. These approaches trade-off accuracy for decreased computational complexity.

Several complementary methods have recently been proposed to utilize parallelism in DL models by partitioning them across multiple devices while preserving the original model and accuracy. Federated learning [6] 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 [5, 7–9]. 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 paper, 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 [7, 10, 11] 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 of forward inference and backpropagation 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, a common CNN for object detection, distributed across a network of quad-core Rasberry Pis.

2 Related Work

Performing inference on resource-constrained edge and mobile devices has received significant attention. Approaches for distributed inference in edge settings exploit inherent parallelism to partition a model and distribute it across multiple devices [4, 5, 7–9]. 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 [7, 10, 11] in our work and extend it to the back-propagation pass in order to support distributed training.

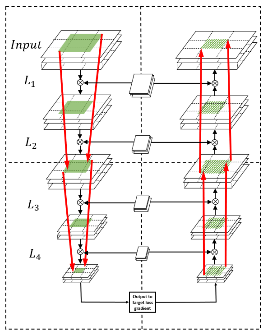
Training CNNs requires additional memory compared to inference due to the need to store input data, gradients, and activation values for each layer. This normally requires partitioning of the workload involving powerful edge servers or the cloud [1,12,13]. Multiple approaches exist for training a DL model on a single edge device [14]. These typically employ simplified model architectures [15] or use reduced bitwidths for training [16]. Alternatively, approaches for incremental or transfer learning take a pre-trained model and only update a subset of weights [17] or the last layers of a model with every training sample [18]. 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 [6]. Other multi-device approaches rely on approximate gradient prediction methods that trade off accuracy [19]. However, mobile and edge devices, e.g. in the IoT space, often lack sufficient memory to store an entire local model. In [12, 13], 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 optimized for the greater memory and communication limitations in mobile and edge settings.

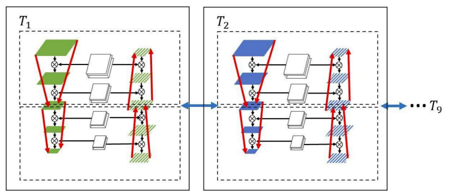
3 Overview

Fig. 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 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.



(a) 3x3 tiling w/ 2 groups



(b) Independent execution tasks with communication boundaries

Fig. 1: Distributed CNN training overview.

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. 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 grows with the group size as illustrated by the funneling red arrows in Fig. 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.

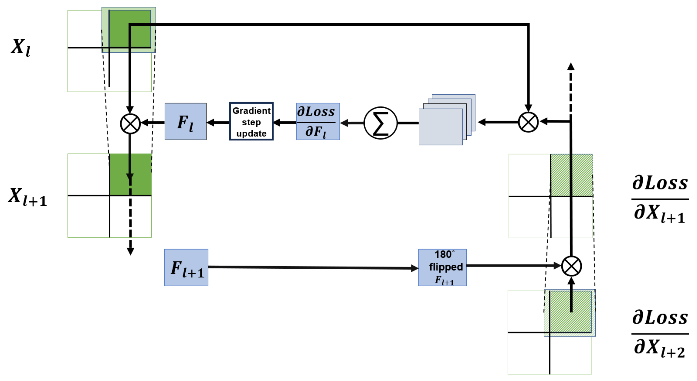

X_l
X_{l+1}
F_l
Gradient step update
$\frac{\partial Loss}{\partial F_l}$
$\frac{\partial Loss}{\partial X_{l+1}}$
$\frac{\partial Loss}{\partial X_{l+2}}$
F_{l+1}
180° flipped F_{l+1}

Fig. 2: Single-layer tiled forward inference and back-propagation.

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.

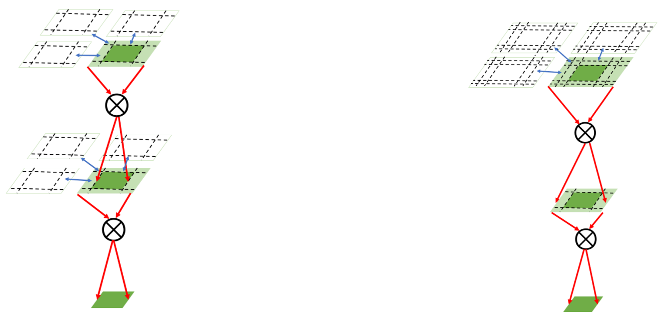
4.1 Single-Layer Tiling

Fig. 2 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 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.



(a) Fusing without grouping (b) Fusing with grouping

Fig. 3: Fusing and grouping illustration.

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. 3 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. 3(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. 3(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. In other words, grouping 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.

Fig. 4 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 at the group input layers 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.

The span of each tile (i, j) 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. 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).

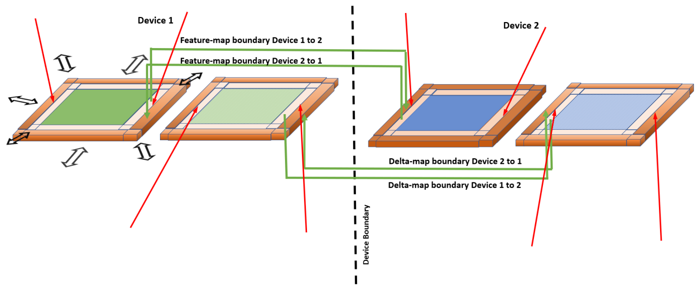

Device 1
Feature-map boundary Device 1 to 2
Feature-map boundary Device 2 to 1
Device 2
Delta-map boundary Device 2 to 1
Delta-map boundary Device 1 to 2
Device Boundary

Fig. 4: Group boundary communication illustration.

Suppose we have a grid of tiles and want to create the grouping profile in the forward pass. To derive required boundary and tile data, we begin at the feature-map output of the last layer of the last group and recursively traverse backward among intermediate layers and groups. For any group, (s, e), the feature map output of the last layer of the group, e, is partitioned length and breadth wise equally among the tiles. Then, 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

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

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

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

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

for convolutional layer $l-1$.

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. 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

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

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

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

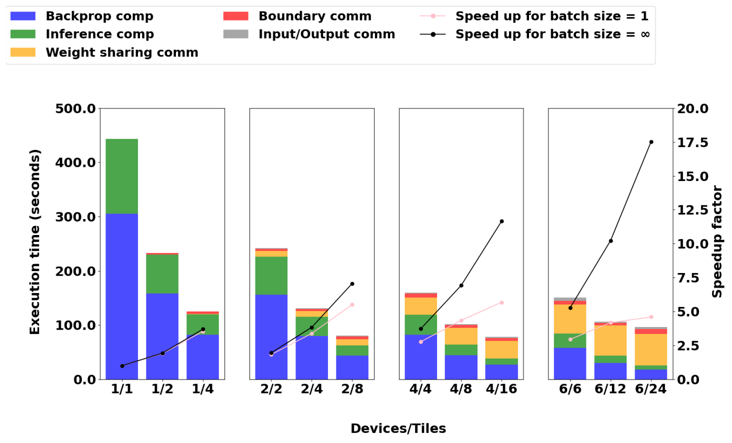


Fig. 5: Execution time split and speedup with number of tiles and devices.

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

for convolutional layer l.

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 Section 4.1. 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 during the forward pass and can be re-used to avoid additional communication for this step.

5 Experiments and Results

We implemented our distributed training approach in C on top of the Darknet framework and validated our model using the first 16 layers of the Yolov2 CNN. A reference implementation is available at [20]. 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. More details and results can be found in [21].

5.1 Speedup

Fig. 5 shows the execution times for a single training sample (batch size of 1) across different combinations of devices and cores ranging from 1 to 6 devices,

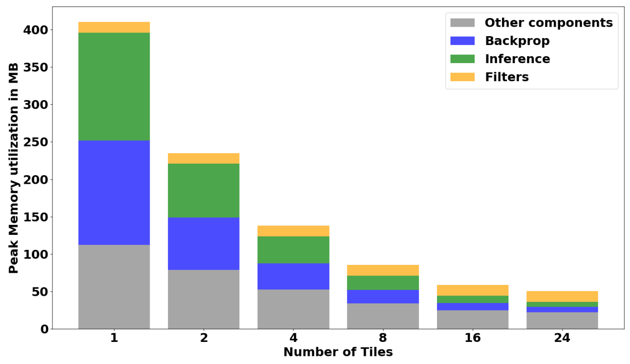


Fig. 6: Memory utilization with number of tiles.

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. Results are broken down into execution times for back-propagation and forward inference computations, communication times for filter weight updates and boundary data exchanges, and input/output communication overheads. A 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 for the different configurations are shown with respect to this baseline. 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). 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.

5.2 Memory

Fig. 6 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

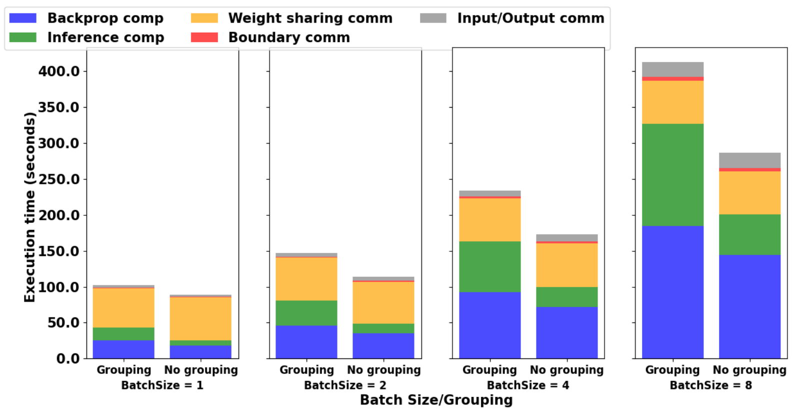


Fig. 7: Comparison with batch size and grouping.

maps, delta maps, filters and other implementation-related components such as a preallocated buffer for intermediate computation, communication buffers and code space. The 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 leading to diminishing returns.

5.3 Batching and Grouping

We further conducted experiments on a batch of samples of various sizes. We also performed a comparison between grouping profiles - with and without grouping. Fig. 7 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 (no grouping) performs significantly better than with grouping across all batch sizes. In case of the Raspberry Pis, total execution time is dominated by computation times or weight updates, and the improvement comes from optimization of computation. 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.

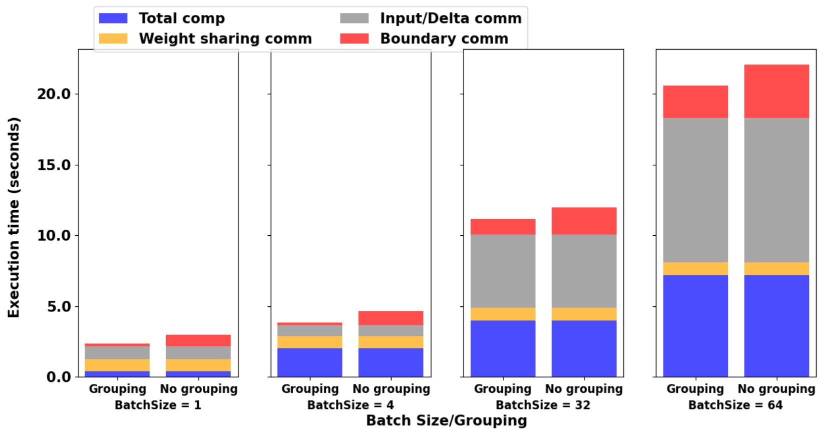


Fig. 8: 2-tile experiment with GPUs.

5.4 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. 8 illustrates the single batch training cycle time for different batch sizes for a 2-tile setup (each board training on the GPU). On the GPUs, the inference plus backprop computation is much faster than on the Pis, thus making communication and synchronization overhead the limiting factor. In this case, the difference in boundary communication overhead among different groupings though small is noticeable. The case of with grouping performs better than without grouping since it synchronizes less frequently. On the GPUs, the extra redundant computation in the grouping 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 grouping is optimal.

6 Summary and Conclusions

In this paper, 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 level within each sample. 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. A grouping optimization algorithm including cost model and additional results are discussed in [21]. A reference implementation of our approach is available at [20]. Future work will explore weight partitioning techniques and how to extend our approach to other weight-dominated layers.

References

1. W. Ren *et al.*, "A Survey on Collaborative DNN Inference for Edge Intelligence," *arXiv preprint arXiv:2207.07812*, 2022.
2. Yousefpour *et al.*, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *JSA*, vol. 98, p. 289–330, Sep. 2019.
3. R. Stahl *et al.*, "DeeperThings: Fully Distributed CNN Inference on Resource-Constrained Edge Devices," *IJPP*, vol. 49, no. 4, pp. 600–624, 2021.
4. L. Zhou *et al.*, "Adaptive Parallel Execution of Deep Neural Networks on Heterogeneous Edge Devices," in *SEC*, 2019.
5. J. Du *et al.*, "A Distributed In-Situ CNN Inference System for IoT Applications," in *ICCD*, 2020.
6. T. Li *et al.*, "Federated Learning: Challenges, Methods, and Future Directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.
7. Z. Zhao *et al.*, "DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters," *IEEE TCAD*, vol. 37, no. 11, pp. 2348–2359, 2018.
8. J. Mao *et al.*, "MoDNN: Local distributed mobile computing system for Deep Neural Network," in *DATE*, 2017.
9. S. Zhang *et al.*, "DeepSlicing: Collaborative and Adaptive CNN Inference With Low Latency," *IEEE TPDS*, vol. 32, no. 9, pp. 2175–2187, 2021.
10. M. Alwani *et al.*, "Fused-layer CNN accelerators," in *MICRO*, 2016.
11. R. Stahl *et al.*, "Fused depthwise tiling for memory optimization in TinyML deep neural network inference," in *TinyML Research Symp.*, 2023.
12. Z. Wang *et al.*, "CoopFL: Accelerating federated learning with dnn partitioning and offloading in heterogeneous edge computing," *Comput. Netw.*, vol. 220, 2023.
13. T. Sen and H. Shen, "Distributed Training for Deep Learning Models On An Edge Computing Network Using Shielded Reinforcement Learning," in *ICDCS*, 2022.
14. P. P. Ray, "A review on TinyML: State-of-the-art and prospects," *JKSUCI*, vol. 34, no. 4, pp. 1595–1623, 2022.
15. M. M. Grau *et al.*, "On-device Training of Machine Learning Models on Microcontrollers With a Look at Federated Learning," in *GoodIT*, 2021.
16. J. Lin *et al.*, "On-device training under 256KB memory," in *NeurIPS*, 2022.
17. H. Cai *et al.*, "TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning," in *NeurIPS*, 2020.
18. H.-Y. Chiang *et al.*, "MobileTL: On-device Transfer Learning with Inverted Residual Blocks," in *AAAI/IAAI/EAAI*, 2023.
19. Y. Chen *et al.*, "Exploring the Use of Synthetic Gradients for Distributed Deep Learning across Cloud and Edge Resources," in *HotEdge*, 2019.
20. http://github.com/SLAM-Lab/Dist-CNN-Training.
21. P. Rama *et al.*, "Distributed convolutional neural network training on resource-constrained mobile and edge clusters," UT Austin, Tech. Rep. UT-CERC-24-02, May 2024.