Model Parallelism Optimization for Distributed Inference Via Decoupled CNN Structure

Jiangsu Du, Xin Zhu, Minghua Shen, *Member, IEEE*, Yunfei Du, Yutong Lu, *Member, IEEE*, Nong Xiao, *Senior Member, IEEE*, and Xiangke Liao, *Member, IEEE*

Abstract—It is promising to deploy CNN inference on local end-user devices for high-accuracy and time-sensitive applications. Model parallelism has the potential to provide high throughput and low latency in distributed CNN inference. However, it is non-trivial to use model parallelism as the original CNN model is inherently tightly-coupled structure. In this article, we propose DeCNN, a more effective inference approach that uses decoupled CNN structure to optimize model parallelism for distributed inference on end-user devices. DeCNN is novel consisting of three schemes. Scheme-1 is structure-level optimization. It exploits group convolution and channel shuffle to decouple the original CNN structure for model parallelism. Scheme-2 is partition-level optimization. It is based on channel group to partition the convolutional layers, and then leverages input-based method to partition the fully connected layers, further exposing high degree of parallelism. Scheme-3 is communication-level optimization. It uses inter-sample parallelism to hide communications for better performance and robustness, especially in the weak network connections. We use ImageNet classification task to evaluate the effectiveness of DeCNN on a distributed multi-ARM platform. Notably, when using the number of devices from 1 to 4, DeCNN can accelerate the inference of large-scale ResNet-50 by 3.21$\times$, and reduce 65.3 percent memory footprint, with 1.29 percent accuracy improvement.

Index Terms—Intelligent applications, distributed deep learning, distributed inference, model parallelism, decoupled CNN structure

1 Introduction

Convolutional neural network (CNN) is widely used in end-user IoT and mobile devices for intelligent applications [1]. When deploying CNN inference on the local end-user devices, it not only removes the concern about data privacy but also reduces the burden on external networks and servers. Moreover, running the local CNN inference is more stable and can eliminate the latency from the external communications. Typically, CNN inference is a resource-hungry task as intermediate results and weights take a lot of memory footprint, and various operations require massive computation. In comparison, end-user device is resource-constrained and running local CNN inference is difficult to satisfy the real-time or time-sensitive requirements of applications in the practical deployment. For example, a single Raspberry Pi 4B takes about 4800 ms to infer the VGG-16 in practice. In addition, the memory capacity is relatively small in a single end-user device, further posing challenges to deploy large-scale CNN inference.

To mitigate the gap between large workloads of CNN inferences and limited resources of end-user devices, many model compression methods have been explored to reduce the computing workloads of CNNs. In [2], they prune useless weights of convolutional kernels to reduce the resources required by CNN inference. In [3], they study the distribution of weights and introduce a weight quantization method. It is a pity that model compression methods often fail to provide great performance and are sometimes at the expense of accuracy.

In practical applications, a lot of end-user devices are available and connected with each other via a local area network. When inputs arrive at a single device, we can distribute CNN inference workloads over several idle local devices to access more resources. For example, the privacy concern, e.g., the embarrassing situation, can be a major obstacle to the deployment of smart home systems [4]. When some tasks, e.g., image classification and behavior detection, require CNN inferences, the workloads can be distributed into multiple end-user devices, e.g., washing machine, fridge, and sweeping robot, via wireless local area network (WLAN) for instant response, limiting data to the internal network. For distributing the workload, there are three paradigms [5]: data parallelism, pipeline parallelism, and model parallelism. Among these paradigms, model parallelism is very attractive, as it can simultaneously optimize latency, throughput, and memory footprint of CNN inferences.

However, it is non-trivial to use model parallelism to distribute the inference of existing CNN models on the local end-user devices [6]. This is because existing CNN models are inherently tightly-coupled structure, and it always takes frequent communications to remove data dependencies. Further, network connections between end-user devices are usually weak. With frequent communications and weak connections, it takes large communication

• *The authors are with the School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510275, China, and also with the National Supercomputer Center in Guangzhou, Guangzhou 510006, China. E-mail: {dujs, zhux25}@mail2.sysu.edu.cn, {shenmh6, duyunfei, xiaon6}@mail.sysu.edu.cn, yutong.lu@nscc-gz.cn, xkliao@nudt.edu.cn.*

Manuscript received 24 June 2020; revised 21 Aug. 2020; accepted 6 Sept. 2020. Date of publication 1 Dec. 2020; date of current version 11 Feb. 2021.
(Corresponding authors: Minghua Shen and Yunfei Du.)
Recommended for acceptance by P. Balaji, J. Zhai, and M. Si.
Digital Object Identifier no. 10.1109/TPDS.2020.3041474

overheads, further resulting in the inefficiency of existing model parallelism in distributed CNN inference. Thus, we are encouraged to focus on the CNN structure feature to explore model parallelism optimization for distributed CNN inference.

In this paper, we propose DeCNN, an effective CNN inference approach that uses multi-level model parallelism optimization for distributed inference on the local end-user devices. The basic optimization is to decouple the convolutional layers of CNN structure. With decoupled CNN structure, we eliminate data dependencies between channel groups in convolutional layers. Further, the partitioning and communication optimizations are explored to enable the more effective model parallelism for distributed inference. The contributions of this paper are summarized as follows.

- We propose a DeCNN approach that leverages multi-level optimization to provide high throughput, low latency, and small memory footprint for distributed CNN inference on local end-user devices.
- We propose Scheme-1 that exploits group convolution and channel shuffle to decouple the convolutional layers of CNN structure for layer partitioning.
- We propose Scheme-2 that uses inter-channel partitioning of convolutional layers and input-based partitioning of fully connected layers for model parallelism.
- We propose Scheme-3 that exploits inter-sample parallelism to hide the communication of current inference to the computation of next inference for robustness in the weak network connections.

We demonstrate the effectiveness of the DeCNN inference approach on three representative CNN models, VGG-16 [7], ResNet-34 [8], and ResNet-50 [8]. They are used to process ImageNet classification task on a distributed multi-ARM platform. Results show that DeCNN outperforms existing approaches, especially in the weak network connections. When deploying DeCNN on the number of devices from 1 to 4, in terms of large-scale ResNet-50, it provides 3.81× performance improvement and 70.9 percent memory footprint reduction, with 0.34 percent lower Top-1 accuracy. Further with another setup, it achieves 3.21× performance improvement and 65.3 percent memory footprint reduction, with 1.29 percent higher Top-1 accuracy. We believe that our DeCNN is a very attractive inference approach in the practical intelligent applications. The DeCNN code is available at https://github.com/sysu-eda/Distributed-CNN-Inference.

2 Background and Motivation

2.1 Layer Types in CNN Model

A CNN model is composed of multiple layers, and an input sample flows through these layers to get the predicted result. Generally, the CNN model includes convolutional layer (CL), fully connected layer (FC), pooling layer, batch-norm layer, and activation layer. In the inference phase, the batch-norm layer normalizes input x by

$$y = \gamma \frac{x - mean}{\sqrt{var + \epsilon}} + \beta = \left(\frac{\gamma}{var + \epsilon}\right) x + \left(\beta - \frac{mean}{\sqrt{var + \epsilon}}\right). \qquad (1)$$

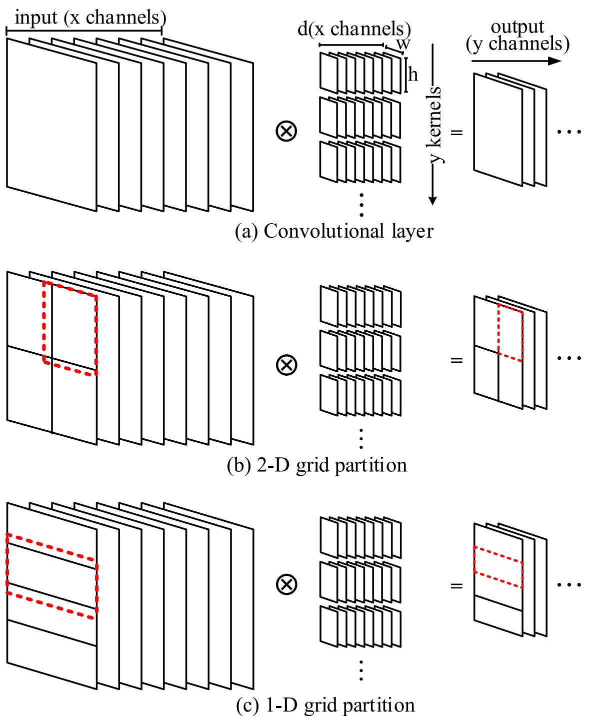


Fig. 1. Convolutional layer and intra-channel partitioning.

All inputs except for x are constants. The activation layer is also an unary function. Thus, there is no data dependency when partitioning batch-norm and activation layers into different devices. In contrast, data dependency happens when partitioning CL, FC and the pooling layer. Since the computation features of pooling layer is similar to the CL, we only consider the CL and take the pooling layer as a special case of the CL. Thus we focus on CL and FC in this paper.

In general, CL and FC are the most resource-demanding layers. Fig. 1a shows the CL structure. In practical CNN models, a CL contains hundreds of convolution kernels (y). A convolution kernel is a 3-D (width, height and depth) tensor, and each kernel operates on all the input channels (x). When performing a CL, every kernel slides on the input through width and height dimensions, and then a new channel is produced as output. In this way, hundreds of kernels produce the corresponding number of output channels (y). FC is a simple structure where all the input neurons are connected to all the output neurons, and each connection represents a weight, which is a multiplication operation.

2.2 Group Convolution and Channel Shuffle

Besides the normal convolution (Conv) used in above CLs, there are some convolution variants. Group convolution (GConv) is a representative variant shown in Fig. 2. In GConv, input channels are evenly grouped at first and each kernel only operates on input channels of the same group. The channel number of each kernel in GConv is reduced. Thus, a new hyperparameter called group number is introduced and it is typically set at 2, 3, 4, and 8. GConv can reduce both the computing complexity and weight amount compared with Conv. Note that, in Fig. 2, input, kernels, and output in the same color are completely independent with that of other colors.

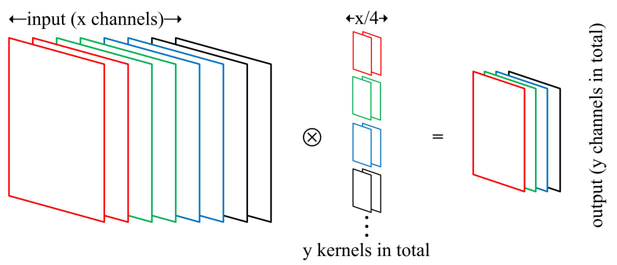


Fig. 2. Group convolution.

ShuffleNet [9] and Condensenet [10] adopts the idea of GConv to construct their models. However, as shown in Fig. 3a, as each output channel only uses the input channels within the same group, there is no information exchange among groups, resulting in accuracy decrease. To solve the problem, both of them adopts the similar structure, called channel shuffle in ShuffleNet and permute in Condensenet. As shown in Fig. 3b, the input is first divided into subgroups by channel, then each group in the layer is fed with subgroups from different groups. Thus, each convolution kernel can extract feature information in a global way.

2.3 Motivation

Here we analyze the structure features of existing CNN models, and then we point out that it has a significant impact on the performance of model parallelism for distributed CNN inference on end-user devices.

For the FC of CNN models, the amount of input and output data is generally much less than that of weights. Employing model parallelism to FC is to partition its weights into different devices and the input data flows to weights. Thus, there is at least one synchronization point for a FC. For the CL of CNN models, two existing partitioning methods are shown in Figs. 1b, 1c. When distributing a CL, each device processes a part of data of each input channel and generates the related output. In this way, we regard these partitions as intra-channel partitioning method.

When the width and height of a convolution kernel exceed 1, the computation of a kernel requires data from neighborhood devices and communication between devices happens. As shown, the area framed by the red dotted line is larger than the partitioned area since its computation depends on the neighborhood data. Also, in intra-channel partitioning methods, weights of kernels would not be divided and each device needs to keep all weights, making the reduction on memory footprint modest. For recent CNN models like ResNet-34 and ResNet-50, they have only 1 FC and their tens of CLs dominate almost all the computation and memory footprint. For previous CNN models like AlexNet and VGG, they have 3 FCs dominating most of the

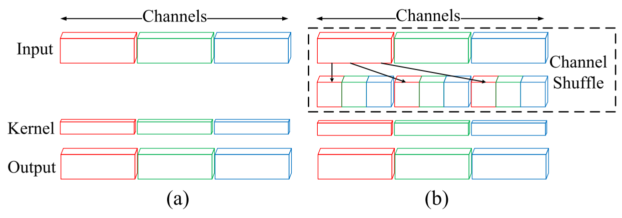


Fig. 3. Channel shuffle.

TABLE 1
Communications of Two Intra-Channel Partitioning Methods in Model Parallelism

Name	#Sync Point	#Comm	Volume(FP32)	overheads
2-D Grid	17	204	1.2MB	210ms
1-D Grid	17	102	1.8MB	230ms

The communication overheads are estimated in theory under 240Mpbs/20ms network connection.

memory footprint and their CLs still dominate most of the computation. We adopt large-scale ResNet-50 as an example to demonstrate that the inherent structures of CNN models results in significant communication overheads in model parallelism, impacting the overall performance in distributed CNN inference.

We first focus on the number of communications in model parallelism. As the number of CLs largely exceeds that of FCs, we only analyze the CLs of CNN models. When adopting intra-channel partitioning methods, 17 CLs of ResNet-50 require communications to remove data dependencies. It means that there are 17 synchronization points when distributing the inference of ResNet-50. Fig. 1b shows 2-D grid partitioning method, a representative intra-channel partitioning method. When deploying onto 4 devices, it takes 12 communications for a single synchronization point. A complete inference of ResNet-50 takes 204 times of communications and the transferred data is 1.2 MB. Fig. 1c shows 1-D grid partitioning method, an intra-channel partitioning method used in MoDNN [6]. It takes 102 times of communications and 1.8 MB data to be transferred in total. Table 1 shows the details of communications of two intra-channel partitioning methods in model parallelism. Thus, the CNN structure results in the large number of communications in model parallelism.

We next focus on the latency of communications in model parallelism. Typically, the network connection between distributed end-user devices is relatively weak. For example, ideally, 2.4 GHZ WIFI should provide 600 Mbps bandwidth and actually, the practical bandwidth is much smaller than the theoretical value. In addition, the communication latency between devices is very high due to network latency and wake-up time of radio modules. In terms of network latency, end-user devices are usually connected by wireless networks, like WLAN, and the communication latency can range from 1 ms to 20 ms [11]. In terms of wake-up time, in order to save energy, end-user devices may turn off its radio modules and wake up from sleep mode when receiving new information, taking a period of time [12]. We assume that the roundtrip latency is 20 ms and bandwidth is 240 Mbps and all communications of a synchronization point start asynchronously, both partitioning methods take at least 200 ms latency for communications. In practice, a Raspberry Pi 4B takes about 1400 ms to complete an inference of ResNet-50. Thus, communication overhead largely impacts the performance in model parallelism. Notably, the time for communication latency dominates the communication overhead.

Thus in this paper, we are motivated to decouple the CNN structure to reduce the expensive communication overheads in model parallelism for better performance in distributed inference.

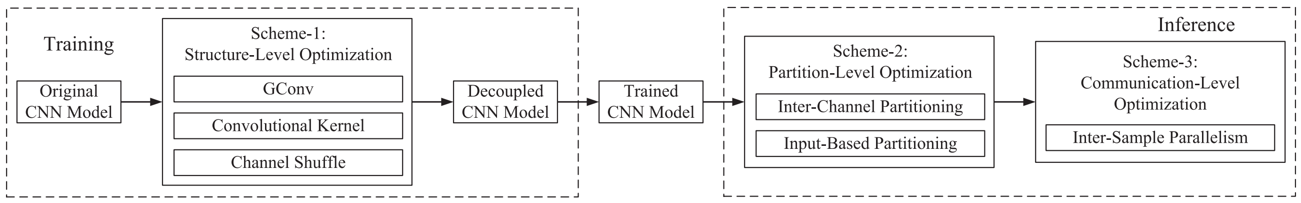


Fig. 4. DeCNN overview.

3 The DeCNN Inference Approach

Fig. 4 shows the overview of the proposed DeCNN inference approach. Scheme-1 is structure-level optimization and it employs GConv, channel shuffle, and more convolutional kernels to decouple the existing CNN models. As Scheme-1 decouples the original structure of existing CNN models, this enables us to train the decoupled CNN models for inference. Here we train the decoupled CNN model on high-performance GPU clusters. Next Scheme-2 is partition-level optimization and it explores the partitioning of decoupled CNN models for model parallelism. Notably, CLs and FCs have different partitioning methods. The inter-channel partitioning method is used for CLs while the input-based partitioning method is used for FCs. At last, Scheme-3 is communication-level optimization and it exploits inter-sample parallelism to overlap the inference of two consecutive samples to hide communications. In detail, Scheme-3 hides the communication of current inference to the computation of the next inference.

3.1 Scheme-1: Structure-Level Optimization

The Scheme-1 decouples the original structure of CNN models so that the decoupled CNN model can be partitioned in a more effective way for better model parallelism in the distributed inference. Note that this Scheme-1 employs three strategies, which is related to GConv, channel shuffle, and the number of convolution kernels, respectively.

As there is no data exchange between channel groups in GConv, we use GConv to replace the Conv of CNN model as shown in Fig. 5b. Here we select large-scale CNN model ResNet-50 to demonstrate the effectiveness of this strategy. We use 4-group GConv to replace the Conv of ResNet-50 and we do not replace the Conv in the first layer. This is because the input of the first layer is the image which only has 3 channels and cannot be evenly divided by the group number. With this strategy, each kernel only operates on the input within the corresponding group. Each 4-group GConv has 4 kernels to process all input, and we keep the number of kernels unchanged. This enables the theoretical

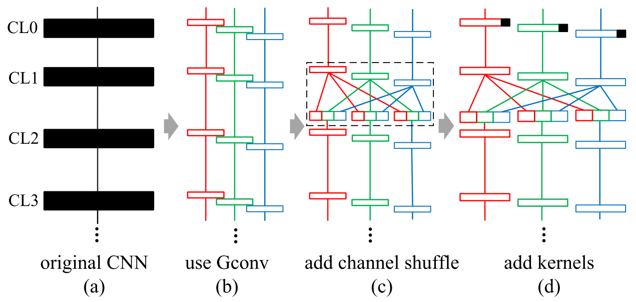


Fig. 5. The decoupled CNN structure scheme. The model here is abstracted and only CLs are shown.

computation amount of CLs and the memory footprint of weights to reduce 4 times, with the same memory footprint of intermediate results. It is a pity that this strategy has a slight impact on accuracy, and the results shows that there is about 9 percent degradation in accuracy.

To mitigate the problem, we then employ channel shuffles to improve the accuracy of the CNN model as shown in Fig. 5c. As the channel shuffle needs to exchange data between channel groups, it imposes new synchronization point when distributing the CNN model. Here we only adopt the small number of channel shuffles to obtain the similar accuracy, when comparing with the large number of channel shuffles. For example in ResNet-50, the accuracy of using 3 channel shuffles is almost the same with using 17 channel shuffles. This basic strategy is to evenly place the channel shuffles and avoid using such layers which have large amounts of output.

In order to further improve the accuracy of the CNN model, we increase the number of convolution kernels in each CL as shown in Fig. 5d. This is because this strategy can increase the number of weights to improve the CNN model accuracy. Typically, the number of kernels increases $1.5\times$ to $2\times$ for the same accuracy as the original CNN model. With this strategy, the theoretical computation amount of CLs and memory footprint of weights are increased but they are still smaller than the original CNN model.

In Scheme-1, we coordinate GConv, channel shuffle, and the number of convolution kernels to decouple the original structure of existing CNN models. Then, the original CNN model only has very limited data dependencies between channel groups, which forms loosely-coupled CNN model. Further, we explore the partitioning of decoupled CNN structure for effective model parallelism in the distributed CNN inference on local end-user devices.

3.2 Scheme-2: Partition-Level Optimization

Scheme-2 explores model parallelism based on decoupled CNN structure. In Scheme-2, an inter-channel partitioning method is proposed for CLs while an input-based partitioning method is proposed for FCs, both of which can expose the high degree of parallelism for distributed CNN inference.

3.2.1 *Inter-Channel Partitioning*

Instead of intra-channel partitioning method, we propose a more effective inter-channel partitioning method for CLs in decoupled CNN model. As the basic input, kernel, and output within a group are completely independent with that of other groups, we partition the decoupled CNN model by channel group for model parallelism. All components within a channel group should be allocated into the same

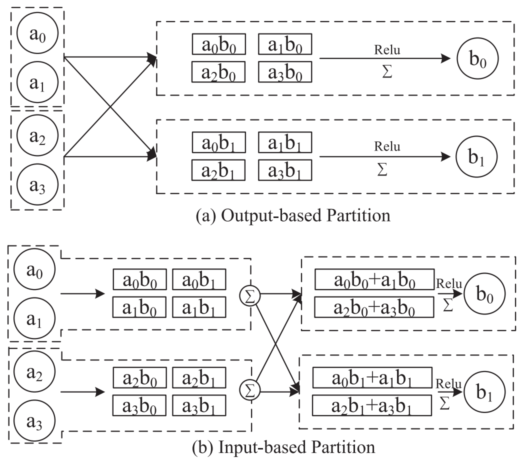


Fig. 6. The output- and input-based partitioning pethods. a_i represents the input neuron and b_i represents the output neuron. a_ib_i represents the multiplication operation which is related to a_i and b_i.

device. In details, we partition different input channels and different kernels into different devices. This is different from the previous intra-channel partitioning methods, which breaks each input channel and puts fragments into different devices. It makes each device to retain all kernels, resulting in the inefficiency of model parallelism. Notably, in our inter-channel partitioning method, each device only needs to retain kernels of the allocated channel group and the number of kernels in each channel group is the same. If the number of devices is the divisor of the number of groups, the memory footprint required by kernels is split evenly. Further, the channel shuffle only needs a synchronization point in model parallelism, and we transfer part of the output data of the previous layer to the corresponding device for performing the next layer.

3.2.2 Input-Based Partitioning

We explore the partitioning of FCs and in FCs, the number of weights is the product of the number of input neurons ($inum$) and the number of output neurons ($onum$). This encourages us to focus on the input- and output-based partitioning methods as shown in Fig. 6.

In output-based partitioning method, the weights having the same output neurons are assigned to the same device, and the intermediate results from the previous layer are transferred to all devices. The computation then starts and gets the intermediate results of each output neuron by accumulative operations. In this way, the communication amount is equal to $(d-1)\times inum$, where d represents the number of devices. In input-based partitioning method, the weights having the same output neurons are assigned to the same device as well. After obtaining intermediate results from the previous layer, the computation starts directly and accumulates these results once. Then the results by the first accumulative operation are transferred to corresponding devices and the second accumulative operation calculates the intermediate results of each output neuron. In this way, the communication amount is equal to $(d-1)\times onum$.

Typically, since the number of input neurons $inum$ is larger than the number of output neurons $onum$, the input-based partitioning method takes less communication overheads than the output-based partitioning method. In addition, the output-based method takes an extra gather operation to get the final result in the last FC of the decoupled CNN model. Thus, we adopt the input-based partitioning method for model parallelism.

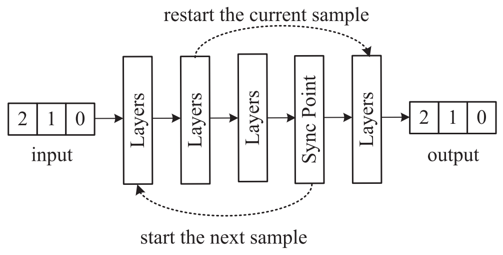


Fig. 7. Inter-sample parallelism.

3.3 Scheme-3: Communication-Level Optimization

The computation of inferences has to interrupt when meeting a synchronization point, during which the computing power is idle. Since the synchronization time is considerable and longer than the execution time of most layers, we propose the Scheme-3, inter-sample parallelism, which overlaps the inference of two consecutive samples and hides the communication of current sample to the computation of next sample.

Algorithm 1. Inter-sample parallelism

```
$layers[];$
for $i = 0$ to $i = (\#Layers - 1)$ do
  $layers[i].run();$
  if $sync\,point$ then
    $start\ async\ data\ communications;$
    for $j = 0$ to $j = (\#Layers - 1)$ do
      $layers[j].run();$
      if $communication\ complete$ or $i < j + 1$ then
        $break;$
      end if
    end for
  end if
end for
```

As shown in Fig. 7, the inference of the first sample starts executing and it flows through the decoupled CNN model until meeting a synchronization point. In the synchronization point, communications start asynchronously and the computing power is allocated to perform the inference of the next sample. Every time completing the execution of a layer in the next inference, the program checks whether the data required for the previous inference is ready in the buffer. If all communications have finished and the essential data has been in the buffer, the computing power turns back to the inference of the current sample. Notably, the progress of executing the inference of next sample cannot exceeds that of the current sample. Moreover, the inter-sample parallelism does not require extra memory footprint. Since weights never change in the inference and the intermediate results become meaningless right after the results of the next layer are created, the inference of the next sample can reuse the memory footprint created by the inference of the

TABLE 2
Impacts of Scheme-1 on Model Accuracy

CNN Model	Scheme-1 Structure-Level Exploration			Training Epochs	Theoretical Memroy (MB)			Theoretical Complexity (GFLOPs)	Top-1 Acc	Top-5 Acc
	group number	kernel number	channel shuffle		inter	weights	sum			
ResNet-50	None	None	None	90	45.87	102.01	147.88	4.091	72.72	90.90
	4	×1.5	+3	90	68.50	65.06	133.56	2.343	72.384	90.682
	4	×1.75	+3	90	73.80	86.20	160.00	3.252	73.79	91.56
ResNet-34	None	None	None	90	16.36	87.12	103.48	3.67	70.26	89.678
	4	×1.5	+3	90	24.24	50.96	75.20	2.17	69.214	88.698
	4	×1.75	+3	90	28.08	68.75	96.83	2.92	70.91	89.656
VGG-16	None	None	None	90	57.74	553.34	611.12	15.353	73.124	91.354
	4	×1.75	+3	90	102.60	536.48	639.07	11.470	72.557	90.918

"inter" is the memory footprint required by intermediate results. "Complexity" is the theoretical computation amount, and it is based on the number of multiply-accumulate operations [13].

current sample. Thus, with the utilization of idle computing resources, the inter-sample parallelism improves the overall throughput for better performance in model parallelism. Algorithm 1 shows the implementation details of inter-sample parallelism.

4 Evaluation

In this section, we evaluate the effectiveness of the DeCNN inference approach on a distributed multi-ARM platform. As DeCNN adopts decoupled CNN structure, we study the impacts of Scheme-1 on the accuracy of original CNN models at first. Further, we study the impacts of Scheme-2 and Scheme-3 on the overall performance of DeCNN in model parallelism for distributed inference.

4.1 Experiment Setup

To demonstrate the effectiveness of Scheme-1, we select three representative CNN models, VGG-16 [7], ResNet-34 [8], and ResNet-50 [8] for comparisons. We use 4-group GConv here to decouple these models and Pytorch [14] is used to implement the original CNN models and the decoupled CNN models, respectively. We train these models for ImageNet classification task [15] on a high-performance GPU cluster and evaluate their accuracy-resource efficiency, which is widely used for evaluating a new CNN model. For fair comparisons, both original and decoupled CNN models adopt the same settings, including the hyperparameters. We train 90 epochs for each model. To demonstrate the effectiveness of DeCNN, we compare the proposed Scheme-2 and Scheme-3 with the state-of-the-art partitioning methods in model parallelism. Here we select the 1-D partitioning method from MoDNN [6] and the 2-D partitioning method from Coates *et al.* [16] for CLs, both of them use the proposed input-based partitioning method for FCs. In addition, we perform the inference of the original models on a single device as the baseline.

The experiments are performed on the multi-ARM platform and it is equipped with four Raspberry Pi 4B devices, each of which consists of a ARM Cortex-A72 SoC operating at 1.5 GHz and 4 GB memory. We use ARM Compute Library 19.08 [17] to implement single-image inference of these CNN models in FP32. Note that these four devices are connected via a gigabyte TL-SG5218 switch which can provide up to 880 Mbps bandwidth and 0.15 ms roundtrip latency. To evaluate the performance of DeCNN under different network connections, we use traffic control software to generate another 6 network environments of bandwidth and roundtrip latency: 480 Mbps/1 ms, 480 Mbps/5 ms, 240 Mbps/5 ms, 240 Mbps/10 ms, 240 Mbps/20 ms and 240 Mbps/40 ms. In addition, the communication is through messages implemented by OpenMPI [18]. To ensure validity and reliability, we use the average result of executing 100 samples, and leave devices idle for cooling down after each run [35].

4.2 Model Accuracy Analysis

In this subsection, we provide a comparison between decoupled CNN models and the original CNN models to show the effectiveness of Scheme-1 employed in the DeCNN approach. The results are shown in Table 2. The baseline is the original CNN models, including VGG-16, ResNet-34, and ResNet-50. The second column shows parameters used in Scheme-1, and the remaining columns show training epochs, theoretical resource demand, and model accuracy.

In terms of ResNet-34, when scaling the number of kernels by 1.5×, the decoupled ResNet-34 is about 1 percent lower Top-1 accuracy than the original ResNet-34. Notably, when scaling the number of kernels by 1.75×, the decoupled ResNet-34 is about 0.65 percent higher Top-1 accuracy than the original ResNet-34. In addition, the decoupled ResNet-34 has a slight improvement on overall memory footprint and computation amount. Scheme-1 works better for ResNet-50 than ResNet-34. When scaling the number of kernels by 1.5×, the decoupled ResNet-50 is about 0.34 percent lower Top-1 accuracy than the original ResNet-50. Further, when scaling the number of kernels by 1.75×, the decoupled ResNet-50 is about 1.41 percent higher Top-1 accuracy than the original ResNet-50. Also, the decoupled ResNet-50 has a slight improvement on overall memory footprint and computation amount. In terms of VGG-16, when scaling the number of kernels by 1.75×, the decoupled VGG-16 is about 0.57 percent lower Top-1 accuracy than the original VGG-16. In addition, the decoupled VGG-16 has a slight improvement on computation amount.

Benefiting from structure-level optimization, the Scheme-1 has the potential to enable decoupled CNN models to provide better accuracy, especially with large-scale CNN models. Thus, we believe that Scheme-1 is an effective method in the model parallelism for distributed CNN inference.

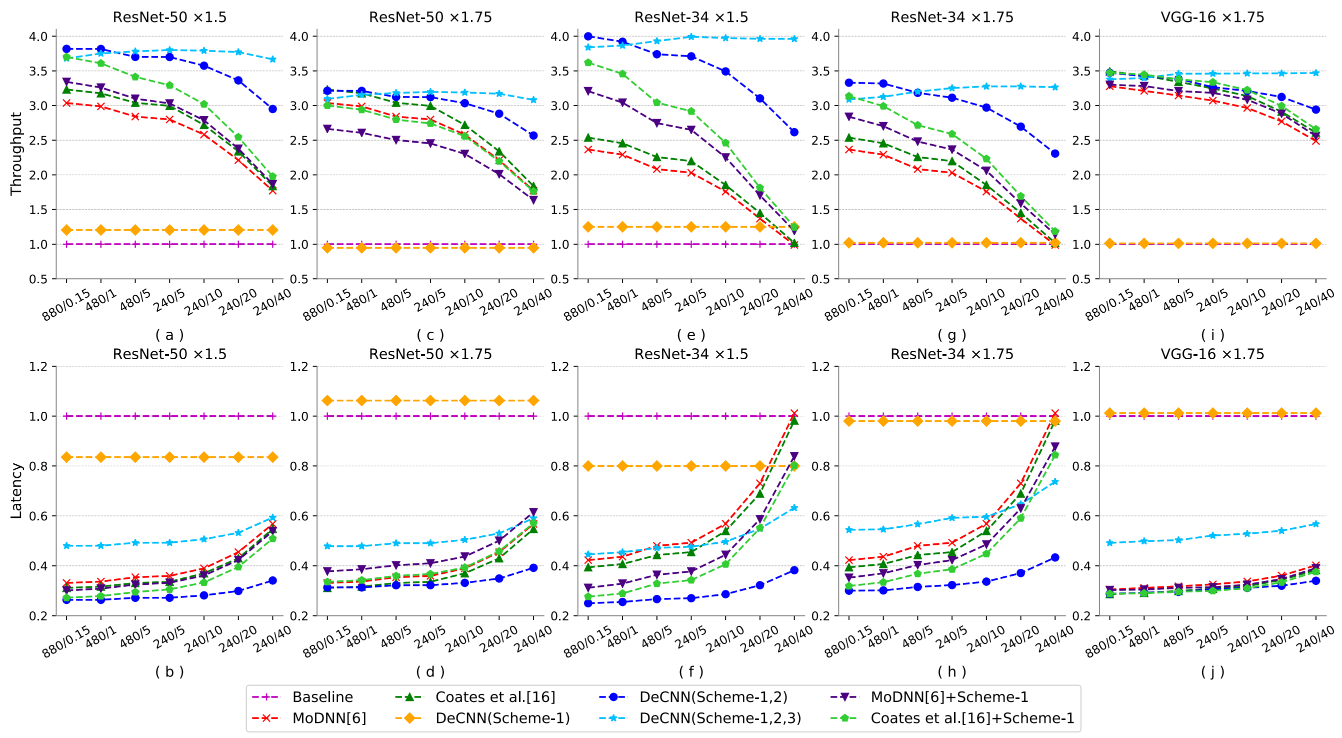


Fig. 8. Comparisons of throughput and latency. Distributed experiments use 4 cevices.

4.3 Throughput and Latency Analysis

In this subsection, we evaluate the impacts of Scheme-1, Scheme-2, and Scheme-3, respectively, on the performance of the proposed DeCNN approach. We also provide a comparison of DeCNN with MoDNN [6] and Coates *et al.* [16] using original models to demonstrate the overall effectiveness of DeCNN in the distributed CNN inference. To show how much DeCNN truly benefits from more parallelism, we compare it with MoDNN [6] and Coates *et al.* [16] using decoupled models. These experiments are mainly distributed over 4 devices under different network connections, and the 2-device distribution is only under 240 Mbps/20 ms network connection for evaluating the scaling ability of DeCNN. Fig. 8 shows the throughput and latency results using 4 devices under different network connections, and Fig. 10 displays the throughput results scaling from 1 to 4 devices with speedup over baseline under 240 Mbps/20 ms network connections. The baseline is original CNN model running on a single device. The MoDNN [6] is original CNN model based on 1-D partitioning method, running on multiple devices. The Coates *et al.* [16] is original CNN model based on 2-D partitioning method, running on multiple devices. The MoDNN [6] + Scheme-1 is decoupled CNN model based on 1-D partitioning method, running on multiple devices. The Coates *et al.* [16] + Scheme-1 is decoupled CNN model based on 2-D partitioning method, running on multiple devices. The DeCNN(Scheme-1) is decoupled CNN model running on a single device. The DeCNN(Scheme-1,2) is decoupled CNN model with partition-level optimization, running on multiple devices. The DeCNN(Scheme-1,2,3) is decoupled CNN model with partition- and communication-level optimizations, running on multiple devices.

4.3.1 *Scheme-1 and Scheme-2*

First, we evaluate the impacts of Scheme-1 on the performance of DeCNN by comparing DeCNN(Scheme-1) with baseline. When scaling the kernel number by 1.5×, the decoupled ResNet-50 takes less inference time than the original ResNet-50. When scaling by 1.75×, the decoupled ResNet-50 has a slight longer inference time compared with the original ResNet-50. This is because GConv reduces the channel number processed by a single kernel, further reducing the degree

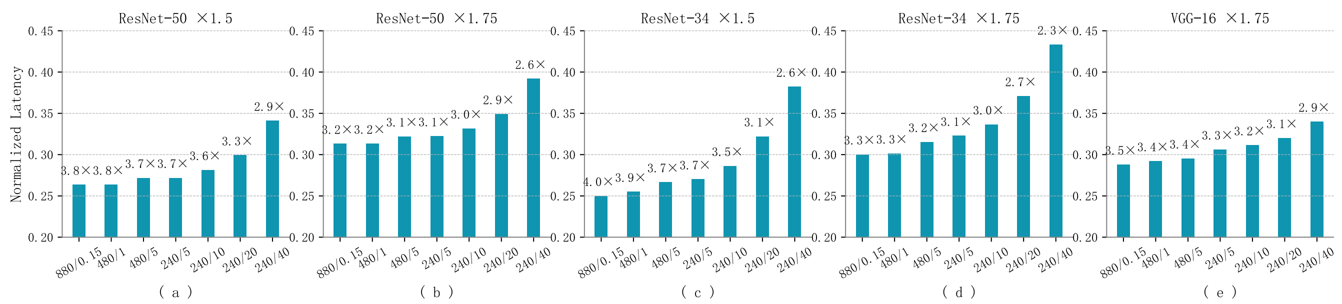


Fig. 9. Latency of DeCNN(Scheme-1,2) using 4 devices normalized to baseline. The baseline is the original CNN model running on a single device. The speedup of DeCNN(Scheme-1,2) is shown at the top of each bar.

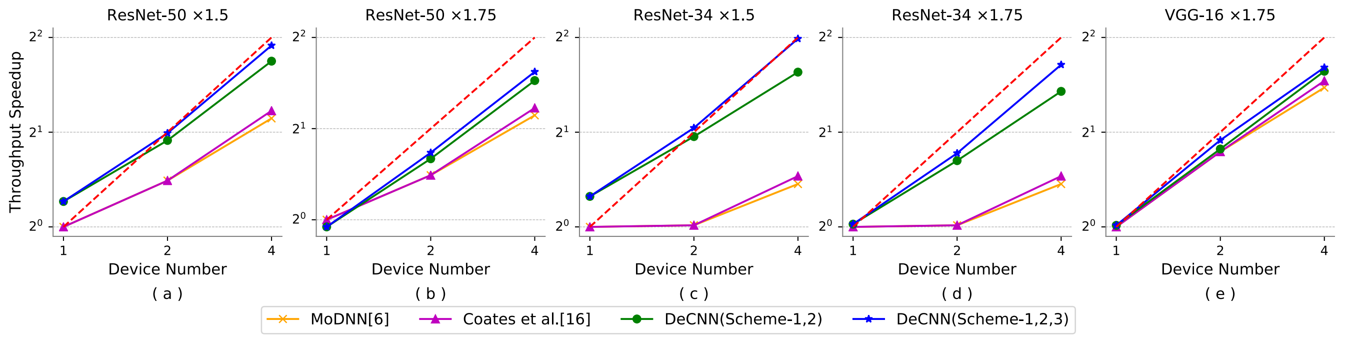


Fig. 10. Throughput scaling results with speedup over baseline, under 240Mpbs/20ms network connection. The red dotted line represents the linear scaling.

of parallelism in ResNet-50 [19], [20]. When scaling the kernel number by 1.75×, the decoupled ResNet-34 takes less inference time than the original ResNet-34. This is because the structure of ResNet-34 is different from the ResNet-50. ResNet-34 uses BasicBlock and ResNet-50 uses Bottleneck as building blocks as shown in Fig. 11. All kernels in ResNet-34 are 3×3 and most input has more channels than the input in ResNet-50. As a result, the decoupled ResNet-34 has a higher degree of parallelism, further accelerating the inference time. In addition, the decoupled VGG-16 has the similar performance as the original VGG-16, when scaling kernel numbers by 1.75×. This is because all its kernel sizes are more than 3 × 3 and input includes more channels than ResNet-50 as well.

Next, we demonstrate the effectiveness of model parallelism based inference over the single-device execution. We compare DeCNN(Scheme-1,2) using 4 devices with the baseline. The baseline is the original CNN model running on a single device. We display the normalized latency, and the results of DeCNN(Scheme-1,2) are shown in Fig. 9. Latency is the execution time of a single-image inference of a target model. Under the ideal 880 Mbps/0.15 ms network connection, DeCNN(Scheme-1,2), our model parallelism based inference, achieves a significant speedup at up to 4.0× for ResNet-34 from about 1250 ms to 310 ms in Fig. 9c. For other models, DeCNN(Scheme-1,2) reduces the latency of VGG16 from about 4800 ms to 1400 ms (3.45× speedup), and for ResNet-50, it can reduce the latency from 1440 ms to 380 ms (3.81× speedup). Even in extremely weak 240 Mbps/40 ms network connections, DeCNN(Scheme-1,2) still achieves about 2.5× speedup over the baseline. Thus, the strong scaling results from 1 to 4 devices illustrate that the model parallelism based inference is very effective in reducing the execution time of the CNN inference.

Then, we evaluate the impacts of Scheme-2 on the performance of DeCNN. Here Scheme-2 is based on Scheme-1 to provide the model parallelism for distributed inference. To eliminate the impact of Scheme-1, we provide a fair comparison of DeCNN(Scheme-1,2) with other two distributed approaches such as MoDNN [6] and Coates *et al.* [16] using decoupled models. As shown in Fig. 8, under all network connections, DecNN(Scheme-1,2) has better performance compared with MoDNN and Coates et al using decoupled models. Notably in the weak network connections, the advantage of Scheme-2 becomes more obvious. Benefiting from Scheme-1, it decouples data dependencies between channels, further reducing communication overheads in Scheme-2. Benefiting from Scheme-2, it maintains the better load balanced partitioning in model parallelism.

From the perspective of communication overheads, we demonstrate the combined effectiveness of DeCNN(Scheme-1,2) by comparing it with MoDNN [6] and Coates *et al.* [16] using original models. We select ResNet-50 and consider the range from 880 Mbps/0.15 ms to 240 Mbps/40 ms in Fig. 8a, DeCNN(Scheme-1,2) accelerates throughput from 3.81 to 2.95×. MoDNN [6] is from 3.03 to 1.77× and Coates *et al.* is from 3.23 to 1.84×. DeCNN(Scheme-1,2) provides better throughput than the MoDNN and Coates *et al.* methods. Further in Fig. 8b, DeCNN(Scheme-1,2) provides much better latency than the MoDNN and Coates *et al.* methods. This benefit is originated from both the decoupling method in computation amount and the partitioning method in model parallelism. The ×1.5 ResNet-50 has less computation amount than the original ResNet-50. More importantly, DeCNN(Scheme-1,2) adopts inter-channel partitioning method enabling the number of synchronization points is only 3. MoDNN and Coates *et al.* adopt intra-channel partitioning method and they require 17 synchronization points to remove data dependencies. Also, the total communication amount of DeCNN(Scheme-1,2) is 1.58 MB, which is between MoDNN (1.8 MB) and Coates *et al.* (1.2 MB). These results are summarized at Table 1 and Table 3. Actually, with the increasingly weak network connections, the synchronization point takes more communication time, resulting in the degradation in performance. Thus in ResNet-50, DeCNN(Scheme-1,2) outperforms the previous MoDNN and Coates *et al.* methods. As shown in Fig. 10, ResNet-50 using DeCNN(Scheme-1,2) also has the performance advantage

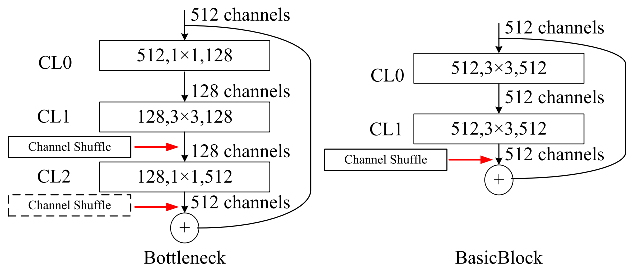


Fig. 11. Building block in ResNet-50.

TABLE 3
Communications of ×1.5 Decoupled ResNet-50 (CLs)

Name	#Sync Point	#Comm	Data Amounts(FP32)
DeCNN	3	36	1.58MB

compared with previous methods when executing on 2 devices under the weak network connection. We further observe the results of ResNet-34 and our DeCNN(Scheme-1,2) has still better performance. Specially when the network connection is extremely weak such as 240 Mbps/40 ms, our DeCNN (Scheme-1,2) has about 2.5× performance improvement compared with baseline, and the previous MoDNN and Coates *et al.* methods have the similar performance as the baseline. We point out that in original ResNet-34, all kernels are 3 × 3 and every CL requires communications to remove the data dependency. As a result, MoDNN and Coates *et al.* require 36 synchronization points, and our DeCNN(Scheme-1,2) is still 3 synchronization points. We continue to observe the results of VGG-16 and our DeCNN(Scheme-1,2) still outperforms the previous MoDNN and Coates *et al.* methods.

From the perspective of load balance, we demonstrate the combined effectiveness of DeCNN(Scheme-1,2). Our DeCNN(Scheme-1,2) adopts inter-channel partitioning method to provide better load balance in model parallelism compared with the previous MoDNN and Coates *et al.* methods. When a sample flows through a CNN model, the height and width of intermediate results gradually become small and extract features of the sample. In the last few CLs of ResNet-34 and ResNet-50, the height and width of intermediate results reduce to 14, and further reduce to 7, which cannot be evenly split by the previous intra-channel partitioning methods in MoDNN and Coates *et al.* This enables different devices to have different amounts of workloads, resulting in load imbalance problem in model parallelism. Benefiting from inter-channel partitioning method, our DeCNN(Scheme-1,2) partitions channel groups into different devices. This enables each device to have the same amounts of workloads, further maintaining better load balance in model parallelism. In addition in VGG-16, our DeCNN(Scheme-1,2) has still good load balance in model parallelism, comparing with the previous MoDNN and Coates *et al.* methods.

Overall, the combination of Scheme-1 and Scheme-2 enables DeCNN to provide better performance than the previous MoDNN and Coates *et al.* methods, especially in the extremely weak network connections.

4.3.2 *Scheme-3*

At last, we evaluate the impacts of Scheme-3 on the performance of DeCNN. Here Scheme-3 is based on the combination of Scheme-1 and Scheme-2 to further improve the DeCNN performance in the model parallelism for distributed inference. Thus we compare DeCNN(Scheme-1,2,3) with DeCNN(Scheme-1,2). When the network connections become weak increasingly, DeCNN(Scheme-1,2,3) provide better throughput than DeCNN(Scheme-1,2) as shown in Fig. 8. Fig. 10 also shows that the 2-device execution using DeCNN(Scheme-1,2,3) largely outperforms previous methods under 240 Mpbs/20 ms network connection. This is because Scheme-3 almost hides all communication overheads, further improving the throughput in model parallelism. In addition, Scheme-3 splits the inference of a sample and overlaps it with operations from inferences of neighborhood samples. This makes DeCNN(Scheme-1,2,3) impact the latency, when comparing to DeCNN(Scheme-1,2). In the experiment with 240Mbps/40ms network connection, our DeCNN (Scheme-1,2,3) has better latency than the previous MoDNN and Coates et al methods only for ResNet-34. With network connection goes worse, our DeCNN (Scheme-1,2,3) provides a better latency than the previous methods for other models such as ResNet-50 and VGG-16. Overall, the combination of Scheme-1 and Scheme-2 and Scheme-3 enables our DeCNN approach to provide a higher throughput with little loss of latency, compared with previous MoDNN and Coates et al methods.

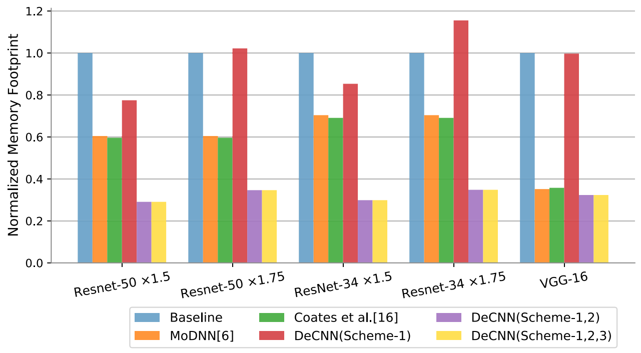


Fig. 12. Memory footprint comparison.

4.4 Memory Footprint Analysis

In this subsection, we provide a memory comparison of DeCNN with two representative methods in MoDNN [6] and Coates *et al.* [16] to show that the benefits of Scheme-2 and Scheme-3 carry over to our distributed inference approach. Fig. 12 shows the memory footprint results of each approach, normalized to the baseline.

We first compare DeCNN(Scheme-1) with baseline, because they are performed on a single device. When scaling the number of kernels by 1.5× in ResNet-50, the Scheme-1 enables DeCNN to generate less memory footprint than baseline. When scaling the number of kernels by 1.75× in ResNet-50, the DeCNN(Scheme-1) has a slight impact on the memory footprint, comparing to baseline. For ResNet-34 and VGG-16, they have the similar memory footprint results as the ResNet-50.

To illustrate the effectiveness of model parallelism based inference over single-device execution in reducing memory footprint, we compare DeCNN(Scheme-1,2) using 4 devices with the baseline. The baseline is the original CNN model running on a single device. As shown in Fig. 12, DeCNN(Scheme-1,2) only takes below 40 percent memory footprint in each device compared with the baseline, greatly reducing the required memory size. For example, compared with 485MB memory footprint in the baseline, ResNet50 ×1.5 only takes 145 MB memory footprint (70.1 percent reduction) in each device when distributing over 4 devices. More typically, when increasing from 1 to 4 devices, DeCNN(Scheme-1,2) reduces the memory footprint of VGG-16 from 1.6 GB to 517 MB (67.7 percent reduction) in each device. The significant memory footprint reduction in each device can enable the high-accuracy CNN inference in end-user devices with less memory capacity, e.g., IoT-oriented Orange Pi [21] which is generally with 256 MB/512 MB/1 GB RAM choices.

To demonstrate the advancement of DeCNN in memory footprint, we next compare DeCNN(Scheme-1,2) with MoDNN [6] and Coates *et al.* [16]. In terms of ResNet-50 and ResNet-34, the combination of Scheme-1 and Scheme-2 enables DeCNN to produce much better memory footprint results than the previous MoDNN and Coates *et al.* methods. As ResNet-50 and ResNet-34 are CL-dominant CNN models, almost all of memory footprint comes from CLs within which kernel weights take a large proportion. Further, Scheme-2 allocates different kernels to different devices to reduce intermediate results and weights, further improving the memory footprint results significantly. In addition, as VGG-16 is a FC-dominant CNN model, DeCNN(Scheme-1,2) has a small improvement on memory footprint, compared with the previous MoDNN and Coates *et al.* methods.

Finally, as Scheme-3 reuses the memory footprint for different samples, the combination of Scheme-1 and Scheme-2 and Scheme-3 enables DeCNN to have the same memory footprint results as the DeCNN(Scheme-1,2). Overall, while Scheme-1 has a slight impact on memory footprint, the combination of Scheme-1 and Scheme-2 and Scheme-3 enables DeCNN to generate the better memory footprint results, especially in the CL-dominant CNN models.

4.5 Summary

Benefiting from multi-level optimization, our DeCNN approach provides higher throughput, lower latency, and smaller memory footprint than the previous MoDNN and Coates *et al.* methods, especially in the extremely weak network connections. When deploying the large-scale ResNet-50 CNN model on the distributed multi-ARM platform to infer image classification on ImageNet, our DeCNN approach can achieve 3.21× performance improvement, 65.3 percent memory footprint reduction, and 1.29 percent accuracy improvement, comparing to the original ResNet-50 model running on a single device. Thus, we believe that our DeCNN is an effective inference approach in the practical intelligent applications.

5 Related Work

There are a large number of works aiming to accelerate deep learning models for intelligent applications. In this section, we mention only those most closely related to our work.

Model compression has two representative methods, one is pruning and the other is quantization. The former is to remove useless weights to reduce the resources required by NN inference [2], [22]. The latter is to employ low-precision value to replace high-precision value for reducing the resource requirements [3], [22], [23]. Hardware acceleration is to explore the mapping of NN models onto specific architecture for performance improvement. The works [24], [25], [26] optimize the CNN inference on CPUs, GPUs, and ARMs, respectively. The TVM work [27] provides a compilation framework to automatically optimize the NN models on various hardware platforms. Typically, model compression and hardware acceleration provide good performance at the cost of accuracy. In addition, these two single-node optimizations can be integrated into the distributed multi-node optimizations for further performance improvement.

Data parallelism is based on sample to explore the partitioning of NN workloads. The work [28] explores distributed DNN training for high throughput. The work [29] optimizes asynchronous SGD to reduce the communication overheads while the work [30] exploits large batch size for synchronous SGD optimization. In [31], they explore heterogeneity-aware decentralized training for improving data parallelism. The author [32] from AWS IoT Greengrass explores the deep learning inference at the edge.

Pipeline parallelism is based on layer to explore the partitioning of NN workloads. In [33], they optimize the pipeline parallelism to train specific model structures such as FCs and recurrent layers. In [34], they explore the overlaps between computation and communication, and leverages a automatic scheme to generalize pipeline parallelism in DNN training. In [35], they explore pipeline parallelism for DNN inference on the ARM platform. Specially, they allocate different cores to different layers and balance the workloads of each core for pipeline parallelism.

Model parallelism is based on structure to explore the partitioning of NN workloads. The work [36] explores model parallelism for only FCs in the CNN training. The work [16] explores model parallelism for distributed CNN training on 4 devices with high performance interconnections. The work [6] explores model parallelism for CLs in the distributed CNN inference in the context of mobile platform. Our DeCNN work are the first to explore structure-level optimization which forms the base of model parallelism in our distributed CNN inference. We also explore partition-level optimization and specially, we explore the partitioning of CLs and FCs, respectively. At last, we explore communication-level optimization and exploit inter-sample parallelism to hide the communications for better performance and robustness, especially in the weak network connections.

6 Conclusion and Future Work

Due to tightly-coupled structure, existing model parallelism methods are difficult to expose a high degree of parallelism in the distributed inference. In this paper, we propose an effective inference approach, named DeCNN, to provide high throughput, low latency, and small memory footprint for distributed CNN inference. The DeCNN approach consists of three-level optimization on model parallelism. The first is to decouple the original CNN models, the second is to explore the partitioning of CLs and FCs, and the third is to exploit inter-sample parallelism to hide communications. We select three representative CNN models to evaluate the effectiveness of DeCNN on the distributed multi-ARM platform. Notably when deploying the large-scale ResNet-50 on four devices, our DeCNN approach can achieve 3.21× performance improvement, 65.3 percent memory footprint reduction, and 1.29 percent accuracy improvement, comparing to the original ResNet-50 running on a single device. We believe that our DeCNN is a very attractive inference approach for intelligent applications.

In the next work, we deploy the DeCNN approach on the large-scale distributed IoT devices to provide inference service in high-accuracy and time-sensitive intelligent applications.

ACKNOWLEDGMENTS

This research was supported by the National Key R&D Program of China 2018YFB0204303, in part by the Natural Science Foundation of China under Grant No. U1811464, 62072479, 61433019, and 61802446, in part by the Projects for Guangdong Introducing Innovative and Enterpreneurial Teams under Grant No. 2016ZT06D211, in part by the Guangdong Basic and Application Basic Research Teams under Grant No. 2018B030312002, and in part by the Fundamental Research Funds for the Central Universities under Grant No. 19lgpy215. The authors would like to appreciate the insightful comments and feedbacks from anonymous reviewers.

REFERENCES

[1] N. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, and F. Kawsar, “An early resource characterization of deep learning on wearables, smartphones and Internet-of-Things devices, ” in *Proc. Int. Workshop Internet Things Towards Appl.*, 2015, pp. 7–12.

[2] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, “Pruning convolutional neural networks for resource efficient inference,” in *Proc. 5th Int. Conf. Learn. Representations*, 2017, pp. 1–17.

[3] R. Zhao *et al.*, “Improving neural network quantization without retraining using outlier channel splitting,” in *Proc. 36th Int. Conf. Mach. Learn.*, 2019, pp. 7543–7552.

[4] G. Yang *et al.*, “Convolutional neural network-based embarrassing situation detection under camera for social robot in smart homes,” *Sensors*, vol. 18, no. 5, 2018, Art. no. 1530.

[5] B. Tal and T. Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *ACM Comput. Surv.*, vol. 52, no. 4, pp. 1–43, 2018.

[6] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, “MoDNN: Local distributed mobile computing system for deep neural network,” in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2017, pp. 1396–1401.

[7] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *Proc. 3rd Int. Conf. Learn. Representations*, 2014, pp. 1–14.

[8] K. He, X. ZhangS. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Int. Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[9] X. Zhang, X. Zhou, M. Lin, and J. Sun, “ShuffleNet: An extremely efficient convolutional neural network for mobile devices,” in *Proc. Int. Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 6848–6856.

[10] G. Huang, S. Liu, L. Maaten, and K. Weinberger, “CondenseNet: An efficient DenseNet using learned group convolutions,” in *Proc. IEEE Int. Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 2752–2761.

[11] K. Sui *et al.*, “Characterizing and improving WiFi latency in large-scale operational networks,” in *Proc. 14th Int. Conf. Mobile Syst. Appl. Serv.*, 2016, pp. 347–360.

[12] H. Zhou, “Modeling of node energy consumption for wireless sensor networks,” *Wireless Sens. Netw.*, vol. 3, no. 1, pp. 18–23, 2011.

[13] M. Song *et al.*, “In-situ AI: Towards autonomous and incremental deep learning for IoT systems,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 92–103.

[14] A. Paszke *et al.*, “PyTorch: An imperative style, high-performance deep learning library,” in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035.

[15] J. Deng *et al.*, “ImageNet: A large-scale hierarchical image database,” in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.

[16] A. Coates *et al.*, “Deep learning with COTS HPC systems,” in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 1337–1345.

[17] Arm Official Website, ”Arm compute library,” 2020. [Online]. Available: https://www.arm.com/whyarm/techn-ologies/compute-library

[18] Open MPI Team, ”Open MPI: Open source high performance computing,” 2020. [Online]. Available: https://www.open-mpi.org/

[19] N. Ma, X. Zhang, H. Zheng and J. Sun, “ShuffleNet V2: Practical guidelines for efficient CNN architecture design,” in *Proc. 15th Eur. Conf. Comput. Vis.*, 2018, pp. 122–138.

[20] Y. He, X. Zhang, and J. Sun, “Channel pruning for accelerating very deep neural networks,” in *Proc. IEEE Int. Conf. Comput. Vis.*, 2017, pp. 1398–1406.

[21] Orange Pi Homepage, ”Orange Pi,” 2020. [Online]. Available: http://www.orangepi.org/

[22] F. Tung and G. Mori, “CLIP-Q: Deep network compression learning by in-parallel pruning-quantization,” in *Proc. Int. Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 7873–7882.

[23] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim, “uLayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization,” in *Proc. 14th EuroSys Conf.*, 2019, pp. 45:1–45:15.

[24] Y. Liu *et al.*, “Optimizing CNN Model Inference on CPUs,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 1025–1040.

[25] L. Wang *et al.*, “A unified optimization approach for CNN model inference on integrated GPUs,” in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 99:1–99:10.

[26] H. Lan *et al.*, “FeatherCNN: Fast inference computation with TensorGEMM on ARM architectures,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 3, pp. 580–594, Mar. 2020.

[27] T. Chen *et al.*, “TVM: An automated end-to-end optimizing compiler for deep learning,” in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 578–594.

[28] Y. You, A. Bulu, and J. Demmel, “Scaling deep learning on GPU and knights landing clusters,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, pp. 9:1–9:12.

[29] G. Cong *et al.*, “Fast neural network training on a cluster of GPUs for action recognition with high accuracy,” in *J. Parallel Distrib. Comput.*, vol. 134, pp. 153–165, 2019.

[30] Y. You, Z. Zhang, C. Hsieh, J. Demmel, and K. Keutzer, “ImageNet training in minutes,” in *Proc. 47th Int. Conf. Parallel Process.*, 2018, pp. 1:1–1:10.

[31] Q. Luo, J. Lin, Y. Zhuo, and X. Qian, “Hop: Heterogeneity-aware decentralized training,” in *Proc. 24th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2019, pp. 893–907.

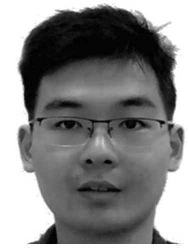
[32] A. Wang, “Parallelizing across multiple CPU/GPUs to speed up deep learning inference at the edge,” 2019. [Online]. Available: https://aws.amazon.com/cn/blogs/machine-learning/parallelizing-across-multiple-cpu-gpus-to-speed-up-deep-learning-inference-at-the-edge/

[33] Y. Huang *et al.*, “GPipe: Efficient training of giant neural networks using pipeline parallelism,” in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2019, pp. 103–112.

[34] D. Narayanan *et al.*, “PipeDream: Generalized pipeline parallelism for DNN training,” in *Proc. 27th Symp. Operating Syst. Princ.*, 2019, pp. 1–15.

[35] S. Wang *et al.*, “High-throughput CNN inference on embedded ARM big.LITTLE multi-core processors,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2254–2267, 2019.

[36] A. Krizhevsky, “One weird trick for parallelizing convolutional neural networks,” 2014, *arXiv: abs/1404.5997*.



Jiangsu Du received the BSc degree from the Wuhan University, in 2016, and the MSc degree from the Edinburgh Parallel Computing Center, University of Edinburgh, in 2017. He is currently working toward the PhD degree with the School of Computer Science and Engineering, Sun Yat-sen University. His research interests focus on parallel and distributed system, and AI technology.

Xin Zhu received the BS degree from the Sun Yat-sen University, Guangzhou, China. She is currently working toward the master’s degree with the School of Computer Science and Engineering, Sun Yat-sen University. Her research interests focus on parallel and distributed computing of deep-learning-based models.

Minghua Shen (Member, IEEE) received the PhD degree in computer science from the Peking University, in 2017. He is currently an associate researcher with the School of Computer Science and Engineering, Sun Yat-sen University, China. His research interests include design automation, programming system, and hardware accelerator. He is a member of the ACM.

Yunfei Du received the BS degree from the Beijing Institute of Technology, and the MS and PhD degrees from the National University of Defense Technology, Changsha, China. He is currently a chief engineer with the National Supercomputer Center in Guangzhou. He is also a professor with the School of Computer Science and Engineering, Sun Yat-sen University. His research interests focus on parallel and distributed systems, fault tolerance, and scientific computing.

Yutong Lu (Member, IEEE) received the MSc and PhD degrees in computer science from the National University of Defense Technology (NUDT), Changsha, China. She is currently a professor with the School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China. She is also the director of National Supercomputer Center in Guangzhou. Her research interests include parallel system management, high-speed communication, distributed file systems, and advanced programming environments with the MPI.

Nong Xiao (Senior Member, IEEE) received the BS and PhD degrees in computer science from the College of Computer, National University of Defense Technology (NUDT), China, in 1990 and 1996, respectively. He is currently a professor with the School of Computer Science and Engineering, Sun Yat-sen University. His current research interests include parallel and distributed computing, computer storage system, and computer architecture. He is a member of ACM.

Xiangke Liao (Member, IEEE) received the BS degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 1985, and the MS degree from the National University of Defense Technology, Changsha, China, in 1988. He is an academician with the Chinese Academy of Engineering. His research interests include parallel and distributed computing, high-performance computer systems, operating systems, cloud computing, and networked embedded systems.