IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 25, 2019, accepted November 13, 2019, date of publication November 26, 2019,
date of current version December 11, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2956080

An Embedded Inference Framework for Convolutional Neural Network Applications

SHENG BI[1,3], YINGJIE ZHANG[1], MIN DONG[1], AND HUAQING MIN[2]
[1]School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China
[2]School of Software Engineering, South China University of Technology, Guangzhou 510006, China
[3]Shenzhen Academy of Robotics, Shenzhen 518000, China

Corresponding authors: Min Dong (hollymin@scut.edu.cn) and Huaqing Min (hqmin@scut.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61703168 Grant 61703284, in part by the Guangdong Province Science and Technology Plan Projects under Grant 2017A030310163 and Grant 2017A010101031, in part by the Guangdong Science and Technology Innovation Strategy Project under Grant 2018B010108002, in part by the Shenzhen Basic Research Project under JCYJ20160429161539298, and in part by the Shenzhen Peacock Project under Grant KQTD20140630154026047.

ABSTRACT With the rapid development of deep convolutional neural networks, more and more computer vision tasks have been well resolved. These convolutional neural network solutions rely heavily on the performance of the hardware. However, due to privacy issues or the network instability, we need to run convolutional neural networks on embedded platforms. Critical challenges will be raised by limited hardware resources on the embedded platform. In this paper, we design and implement an embedded inference framework to accelerate the inference of the convolutional neural network on the embedded platform. For this, we first analyzed the time-consuming layers in the inference process of the network, and then we design optimization methods for these layers. Also, we design a memory pool specifically for neural networks. Our experimental results show that our embedded inference framework can run a classification model MobileNet in 80ms and a detection model MobileNet-SSD in 155ms on Firefly-RK3399 development board.

INDEX TERMS Deep learning, embedded system, mobile computing, mobile sensing.

I. INTRODUCTION

Convolutional neural network (CNN) plays a very important role in the field of computer vision. Deep Convolutional neural network has greatly promoted the development of computer vision, especially in object recognition, object detection and semantic segmentation. Since AlexNet [1] won the ImageNet Challenge: ILSVRC 2012 [2], in order to get higher accuracy, the CNN has become deeper and more complex, which has become the trend of designing network [3]–[5]. However, in many real word applications such as self-driving car, robotics and augmented reality, convolutional neural networks need to be deployed on an embedded platform with limited computing resources.

Many embedded applications often rely on a cloud-based approach [6]–[12]. In cloud-based approach, embedded platform is only used to capture data, and the inference process is completed on the server. A cloud-based approach enables the user of embedded devices to enjoy the huge benefits of convolutional neural networks. However, a cloud-based approach has its disadvantages. First, due to the communication costs, the cloud-based applications depend heavily on network quality. Therefore, in order to ensure the practicability of the application, we need to limit the amount of data sent by the embedded platform. Second, cloud-based approaches may involve private data, and sending personal data to the cloud is a challenge [13]. With the rapid development of 5G, there will be a very attractive solution. However, uploading the data from embedded platforms to cloud can cause privacy problems.

Another way to use CNN with embedded platform is on-device approach. In on-device approach, CNN runs on embedded platform directly. In recent years, there are many lightweight convolutional neural networks that have been designed specifically for embedded platforms such as MobileNet [14] SqueezeNet [15], ShuffleNet [16]. So, in on-device approach, deploying these lightweight CNNs on embedded platform is a common feature. A typical framework of on-device approach is TensorFlow-Lite [17].

In this paper, we present our embedded inference framework that uses the CPU cores on embedded platforms to execute the lightweight CNNs required by applications. Our inference framework can process lightweight MobileNet [14] on commodity embedded platform at the

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaodong Xu.

speed of 12 FPS. This greatly reduces privacy issues due to cloud participation.

Before implementing our inference framework with any optimization techniques, we use MobileNet [14] as an example to analyze the time-consuming layers in the inference process of CNN. Then we design optimization methods for these time-consuming network layers, greatly improving the inference speed of lightweight convolutional neural networks. Our optimization methods include optimization of 1×1 convolutional layer and 3×3 depthwise-separable convolutional layer, and also some optimizations for memory.

Our contributions of this work are summarized as follows:

- Design and implement an efficient embedded inference framework for CNN, by optimizing time-consuming part in the inference process.
- Design an efficient optimization method for 1×1 convolutional layer by memory packing.
- Design an efficient optimization method for 3×3 depthwise-separable convolutional layer.
- Design a memory pool specifically for deep neural networks to further improve the inference performance.

The rest of this paper is organized as follows. Section II describes the work related to the embedded platform convolutional neural network methods. In Section III, we identify the time-consuming part of the lightweight MobileNet. The design considerations of our embedded inference framework are presented in Section IV. In Section V, the implementation of our inference framework is presented. Finally, we reported the results of our experiment in Section VI and concludes this paper and elaborates on the future work in Section VII.

II. RELATED WORK

Running convolutional neural network with an embedded platform continues to attract researchers' attention. There are mainly two types of approaches: cloud-based approach and on-device approach. In this paper, we only focus on the on-device approach due to the limitations of cloud-based approach which we have already mentioned in the Section I. Next, we provide the overview of modern on-device approaches. We divide on-device approaches into two categories: hardware designing and lightweight convolutional neural network designing.

Due to the intensive computing of convolutional neural network, there has been a lot of work related to specific hardware design [18]–[22] for CNN and CNN-based computer vision tasks. Zhang et al. proposed a CNN accelerator on Field-Programmable Gate Array (FPGA) platform using many optimizing techniques, such as loop tiling and transformation in [19]. In [18], an efficient CNN and an efficient CNN-based face detection system are implemented using a single FPGA by Farabet et al. In [21], Chen et al. proposed the DianNao, which achieved high throughput and low energy consumption when processing CNNs, by carefully optimizing the impact of memory on accelerator design, performance and energy consumption. For some low-cost applications, adding a specific hardware may not be cost effective.

On the other hand, designing a lightweight convolutional neural network and deploying it on an embedded platform has been considered as an alternative approach. In spite of the fact that a lot of convolutional neural network are designed for desktop platform, there are many lightweight convolutional neural network [14]–[16], [23], [24], specially designed for embedded platform. Forrest N. Iandola et al. used a bottleneck approach to design a very small but efficient network SqueezeNet, which achieves AlexNet-level accuracy on ImageNet with much fewer parameters even ban be compressed to less than 0.5MB, in [15]. MobileNet in [14] is efficient for embedded vision applications by using depthwise-seprable convolution and it has two simple hyper-parameters that can efficiently trade off between inference efficiency and accuracy. In [24], MobileNetV2 has been designed and it has some performance improvements on both inference efficiency and accuracy adopting inverted residual structure with linear bottleneck. In [16], ShuffleNet uses channel shuffle operation and pointwise group convolution to reduce computation cost and achieve higher accuracy than MobileNet, but from the implementation point of view, channel shuffle operation is not friendly to the cache.

When it comes to how to deploy these lightweight networks to embedded platforms, there are some deploying frameworks such as TensorFlow-Lite [17] or Caffe [25] compiled with OpenBLAS [26]. As far as we known, these frameworks convert convolution layer into matrix multiplication through 'im2col' method, and then improve inference performance by optimizing matrix multiplication. Although this is effective for convolution layer of any parameters, it will increase memory footprint. Therefore, in this paper, we designed and implemented a more efficient embedded inference framework. We optimized some time-consuming network layers individually, which not only improves the inference performance, but also reduces the memory footprint. Moreover, we designed a memory pool specifically for deep neural networks to further improve the inference performance. We also compared our framework with the popular frameworks in the open source community NCNN [27] and MNN [28].

III. WORKLOAD CHARACTERIZATION

In this section we first show a common optimization technique: merge the batch normalization layer with pervious layer. And then, we show the effect of merge the ReLU layer with previous layer. Finally, we identify the computationally intensive parts of the MobileNet model through experiments.

A. MERGE BATCH NORMALIZATION

In modern networks, it's common to include a batch normalization layer after every convolutional or depthwise-separable convolutional layer. Batch normalization layer takes the output of the pervious layer and applies the following operations to every single output value:

$$z = \gamma * \frac{y - \mu}{\sqrt{V + \epsilon}} + \beta.$$

Here, y is an element in the output feature map from the previous layer. γ and β are the parameters of this batch normalization layer. μ and V are mean and variance of that output channel, respectively. Since both convolution (or depthwise-separable convolution) layer and batch normalization layer are linear transformation, we can combine these two layers into a single layer.

The math for combine the batch normalization layer into convolution or depthwise-separable convolution layer is fairly straightforward. Expand y in the formula:

$$z = \gamma * \frac{\sum_{i=1}^{n} x_i \cdot w_i + b - \mu}{\sqrt{V + \epsilon}} + \beta.$$

After rewriting this formula so that γ, β, μ, V only apply to w_i and b, it gives us:

$$z = \sum_{i=1}^{n} x_i \cdot (w_i \cdot \frac{\gamma}{\sqrt{V + \epsilon}}) + (b - \mu) \cdot \frac{\gamma}{\sqrt{V + \epsilon}} + \beta.$$

So, as a common optimization trick, we can just update the parameters of the convolution layer and completely remove the batch normalization layer without affecting the inference results.

B. MERGE ACTIVATION LAYER

Fusing the activation layer with the previous layer is a common way to reduce the number of memory accesses. It is often used for inference acceleration on desktop platforms. Usually a network layer is followed by a non-linear activation layer. ReLU is the most commonly used activation layer. ReLU layer takes the output of previous layer and applies a maximize operation with zero on each element. So, ReLU layer can be applied on the input itself without allocate memory for output. What's more, ReLU layer is simple enough to be fused with previous layer: we only need to add a maximize operation before we store the result of previous layer into memory. This will reduce the number of data flows between the CPU register and the memory. In Figure 1, we demonstrate how does this trick reduce the number of data flows. Table 1 shows the influence that fusing the ReLU layer does to the time of inference. The more complex the model is, the more activation layers are, and the more obvious the acceleration effect of this method is.

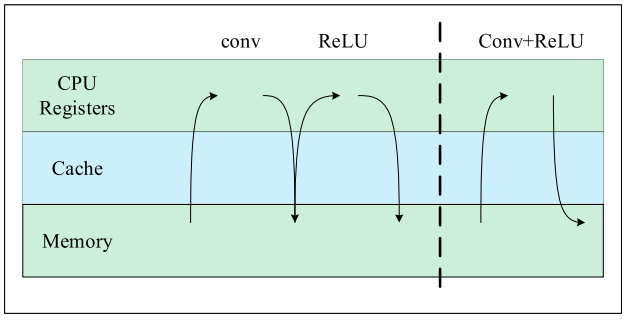


FIGURE 1. Data flow before fuse ReLU (left part) and after fused ReLU (right part).

TABLE 1. Fuse ReLU performance gain.

Inference time (ms)	not fuse ReLU	fuse ReLU	Gain
MobileNet	289.20	276.53	4.4%
MobileNet-SSD	607.32	571.34	5.9%

C. LATENCY BREAKDOWN

In order to identify the computationally intensive parts of the MobileNet model, we implemented a multi-cores CNN inference framework. This framework doesn't employ optimization techniques such as ARM NEON instruction and memory optimization.

For the sake of simplicity, we use Caffe's pretrained MobileNet model directly, and perform batch normalization layer fusion under the Caffe framework. Thus, our framework doesn't need to implement the batch normalization layer.

In Figure 2, we show the inference time broken down per layer type. It indicates that the convolutional layer which kernel size is 1 dominates the inference time. So, we focus on the optimization of 1×1 convolutional layer. On the other hand, since most lightweight networks currently use depthwise-separable convolutional layer and 1×1 convolutional layer to substitute 3×3 convolutional layer to reduce the amount of computation [14], we also focus on optimizing depthwise-separable convolutional (kernel size is 3).

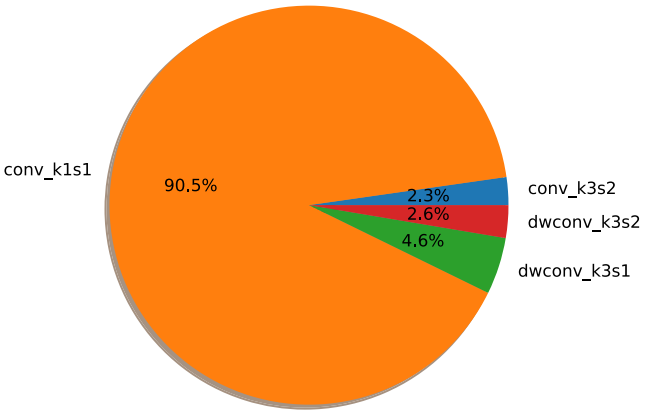


FIGURE 2. Inference time break down per layer type of MobileNet model. We ignore the softmax and pooling layer in this figure because these layers are very simple and have little impact.

IV. DESIGN CONSIDERATIONS

We design the framework with following goals:

1) **No dependency on cloud.** Our main goal, for this paper, is to use the resources of embedded platform only without any dependency on cloud to process CNNs. There are many compelling applications in this area, such as processing privacy data or processing with poor or expensive network service. In these applications, cloud is either unallowed (due to privacy concerns) or unstable (due to network issues).
2) **Maximize CPU utilization.** Embedded platforms typically have multiple CPU cores and run multiple applications simultaneously. However, due to process scheduling and CPU affinity issues, one application

may use different CPU core at different time, and one CPU core may run multiple applications. This problem will lead to cache miss and affect the running efficiency of applications. So, we assume that some CPU cores will be specifically allocated to the CNN model, and other applications on the embedded platform will use the remaining CPU cores. The framework will maximize the utilization of the allocated CPU cores.

3) **Minimize memory footprint.** Multiple CPU cores share the memory of the embedded platform, which means that all applications on the embedded platform share the same memory. If the CNN model is running with a lot of memory, it will cause other applications to run slowly or even stop running. So, the framework will minimize the memory footprint.
4) **Focus on the computationally intensive part.** We notice that it is more productive to focus on the computationally intensive part than on the other parts. So, we measured the running time of MobileNet model on a RK3399 system and broken down the processing latency per layer type. To do this, we implemented a multi-cores CNN execution framework without any other optimization techniques applied such as NEON and memory optimization, as shown in Section III.
5) **No accuracy loss.** In this paper, we focus on deploying the original CNN model on embedded platform. We didn't use the optimization methods that will reduce the model accuracy such as half floating point precision and convolutional layer decomposition [29]–[31].

V. IMPLEMENTATION

In this section, we first show the entire architecture of our inference framework. And then we describe, in detail, the optimization techniques we adopted to optimize the inference of CNNs.

A. FRAMEWORK OVERVIEW

The overall architecture of our inference framework is shown in Figure 3. Different deep learning framework use their own model format. In order to use many pretrained models from different framework, we design our own model format. So pretrained models need to be converted to our framework format. We didn't adopt optimization techniques which will reduce the model accuracy such as layer decomposition [29]–[31]. Therefore, our model conversion is very simple, just serialize the trained parameters directly into our model format.

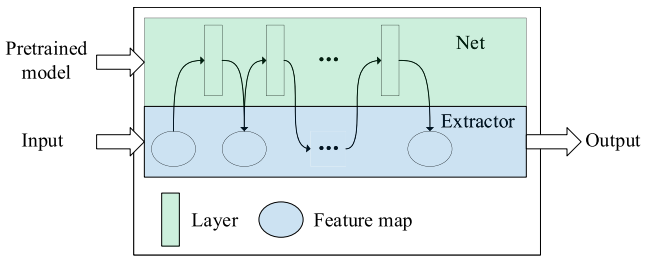


FIGURE 3. Our inference framework overview. Net part and Extractor part are managed separately in memory.

Our inference framework currently supports the models from two deep learning frameworks, namely Caffe [25] and PyTorch [32].

Our inference framework can be roughly divided into two parts: Net and Extractor. The Net part includes the structure of the CNN and the trained parameters of each network layer. The Extractor part includes the input and output of the CNN and the intermediate results of the network.

During the inference phase, the Extractor gets its input and sends the input to the Net part, and the intermediate results of the network are stored in the Extractor. Finally, the output of the network is obtained from the Extractor.

B. 1 × 1 CONVOLUTION OPTIMIZATION

As shown in Section III, the 1×1 convolutional layer is the main performance bottleneck. We employ memory packing to accelerate the inference of 1×1 convolutional layer. Our key observation is that the feature maps of CNN are stored in memory, one channel by one channel. Therefore, the data in the same channel is continuous in memory, as shown in Figure 4. However, the computation of 1×1 convolution will access data across input channels to compute one output value, which is a bad access pattern. This access pattern doesn't take full advantage of the memory locality principle. So, memory packing is adopted to solve this problem.

Moreover, we employ OpenMP [33] to utilize multi-cores, and we also use ARM NEON instruction to maximize the CPU utilization.

In the optimization of 1×1 convolutional layer, we use OpenMP [33] to assign each CPU core 8 output channels at a time. And further, 2 NEON registers are used to represent a 1×8 block, which means that we need 16 NEON registers per CPU for output channels. At each iteration, we compute 2 NEON registers per channel, as shown in Figure 4.

Then, we do memory packing for input feature map every 4 input channels. Note that memory packing for kernel can be done accordingly in the process of constructing Net part. We design Alg.1 Memory packing that packs the memory of input feature map according to the access pattern of 1×1 convolution to take full advantage of the cache locality.

As shown in Figure 4, the outermost loop in Alg.1 walks through input feature map channels with steps of 4. The second loop in Alg.1 walks inside the input channel with steps of 8 (the size of 2 NEON registers). The innermost loop iterates 4 input channels and assigns 8 values of input per channel to *Out* continuously in Line 4-11. To ensure continuity, Line 12 moves the *Out* forward by 8.

During 1×1 convolution, we use 8 NEON registers to load the input data, 2 registers per input channel, and 8 NEON registers to load the convolution kernel, 1 register per output channel, as shown in Figure 4.

Now, we are ready to describe our Alg.2 the optimization algorithm for 1×1 convolutional layer. The input feature map

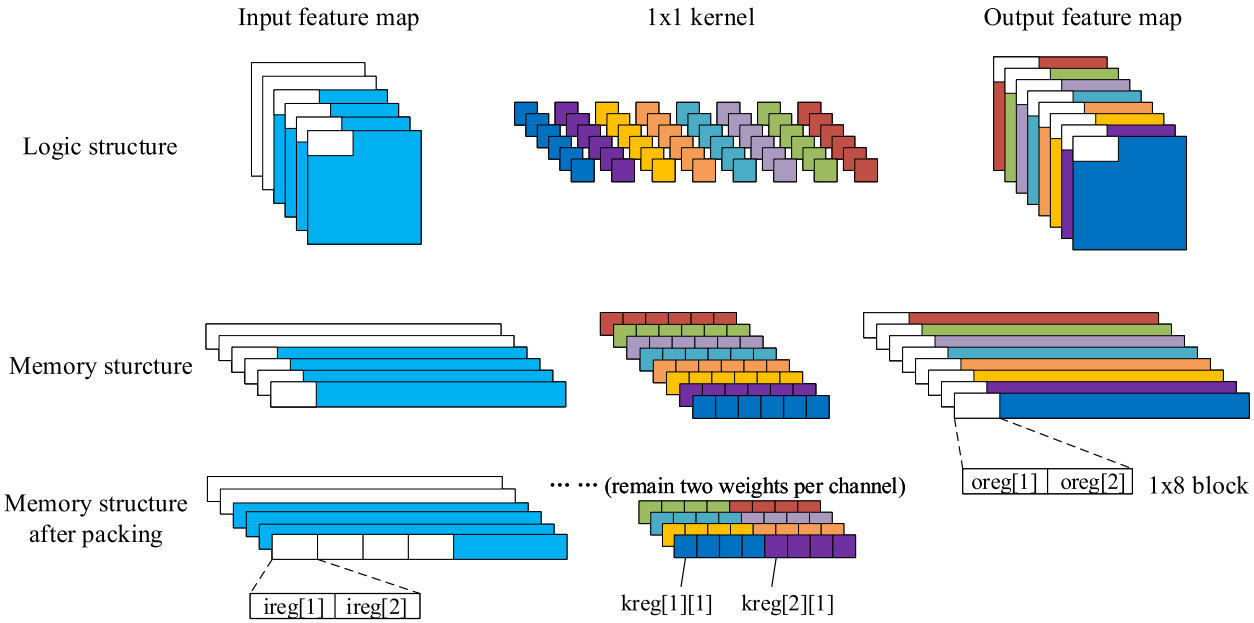


FIGURE 4. 1 × 1 convolution optimization demonstration.

Algorithm 1 Memory Packing

Input: Input feature map *In*
Output: Packed input feature map *Out*

```
1:  for ic = 1 : In.channels steps of 4 do
2:    for k = 1 : In.channelSize steps of 8 do
3:      for i = 0 : 3 do
4:        Out[1] ← In[ic + i][k]
5:        Out[2] ← In[ic + i][k + 1]
6:        Out[3] ← In[ic + i][k + 2]
7:        Out[4] ← In[ic + i][k + 3]
8:        Out[5] ← In[ic + i][k + 4]
9:        Out[6] ← In[ic + i][k + 5]
10:       Out[7] ← In[ic + i][k + 6]
11:       Out[8] ← In[ic + i][k + 7]
12:       Out ← Out + 8
13:     end for
14:   end for
15: end for
16: return Out
```

Algorithm 2 1 × 1 Convolution

Input: Packed input feature map *In*, packed kernel weights *kernel*
Output: Output feature map *Out*

```
1:  for oc = 1 : Out.channels steps of 8 do
2:    load output into oreg[1 . . . 16] (according to Figure 4)
3:    for ic = 1 : In.size steps of 4 do
4:      load input into ireg[1 . . . 8] (according to Figure 4)
5:      load kernel into kreg[1 . . . 8] (according to Figure 4)
6:      // inner loop will be unrolled for performance
7:      for i = 1 : 8 do
8:        // first part of one block
9:        oreg[i] ← oreg[i] + ireg[1] · kreg[i][1]
10:       oreg[i] ← oreg[i] + ireg[2] · kreg[i][2]
11:       oreg[i] ← oreg[i] + ireg[3] · kreg[i][3]
12:       oreg[i] ← oreg[i] + ireg[4] · kreg[i][4]
13:       // second part of one block
14:       oreg[i + 8] ← oreg[i + 8] + ireg[5] · kreg[i][1]
15:       oreg[i + 8] ← oreg[i + 8] + ireg[6] · kreg[i][2]
16:       oreg[i + 8] ← oreg[i + 8] + ireg[7] · kreg[i][3]
17:       oreg[i + 8] ← oreg[i + 8] + ireg[8] · kreg[i][4]
18:     end for
19:   end for
20:   store oreg[1 . . . 16] into Out
21: end for
22: return Out
```

of Alg.2 is the feature map after memory packing obtained from Alg.1. The outermost loop in Alg.2 walks through output feature map channels with steps of 8 and uses 16 NEON registers *oreg* to store the computation results, 2 registers per channel. And in the second, we use 8 NEON registers *ireg* to load 4 input channel data, 2 registers per channel, and load 1 × 1 convolution kernel of size *inputChannels* × *OutputChannels* = 4 × 8 = 32 into another 8 registers *kerg*.

The innermost loop of Alg.2 iterates 8 output channels. In Line 8-17, for each output channel, it computes the results *oreg* from input feature map *ireg* and kernel weights *kreg*, according to the computing rule of 1 × 1 convolution. Finally, results in *oreg* are store in memory in Line 20.

C. 3 × 3 DEPTHWISE-SEPARABLE CONVOLUTION OPTIMIZATION

Although 3 × 3 depthwise-separable convolution does not account for a large proportion of inference time, we often use 3 × 3 depthwise-separable convolution combined with 1 × 1 convolution to replace 3 × 3 convolution for performance.

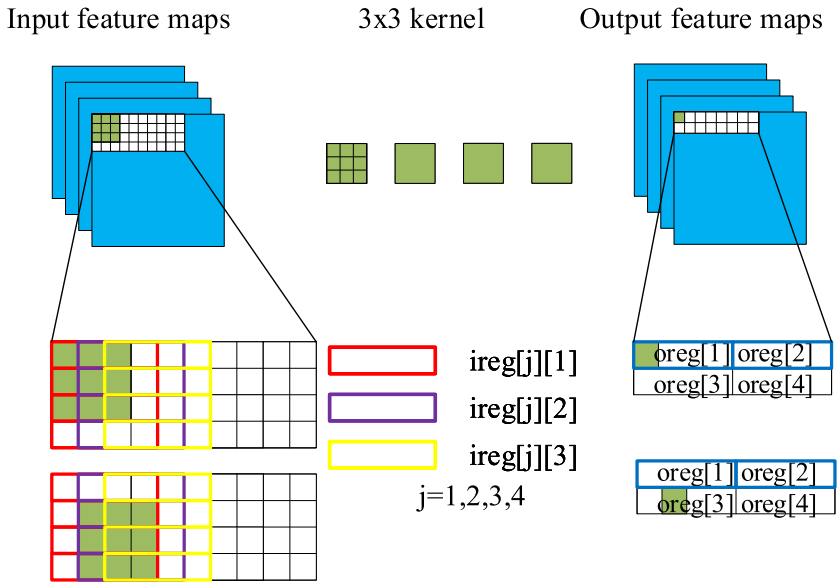


FIGURE 5. 3 × 3 depthwise-separable convolution optimization demonstration. We only show the optimization approach with stride 1, but our approach is suitable for the other strides.

Algorithm 3 3 × 3 Depthwise-Separable Convolution With Stride 1

Input: Input feature map *In*, kernel weights *kernel*
Output: Output feature map *Out*

1: **for** $oc = 1 : Out.channels$ **do**
2: **for** $i = 1 : Out.height$ **steps of** 2 **do**
3: **for** $j = 1 : Out.width$ **steps of** 8 **do**
4: load input into $ireg[1 \ldots 4][1 \ldots 3]$ (according to Figure 5)
5: load kernel into $ker[1 \ldots 3]$ (according to Figure 5)
6: // first part of one block
7: $oreg[1] \leftarrow \sum_{x=1}^{3}\sum_{y=1}^{3} ireg[x][y] \cdot ker[x][y]$
8: $oreg[3] \leftarrow \sum_{x=1}^{3}\sum_{y=1}^{3} ireg[x+1][y] \cdot ker[x][y]$
9: load next input registers (according to Figure 5)
10: $oreg[2] \leftarrow \sum_{x=1}^{3}\sum_{y=1}^{3} ireg[x][y] \cdot ker[x][y]$
11: $oreg[4] \leftarrow \sum_{x=1}^{3}\sum_{y=1}^{3} ireg[x+1][y] \cdot ker[x][y]$
12: store $oreg[1 \ldots 4]$ into *Out*
13: **end for**
14: **end for**
15: **end for**
16: **return** *Out*

Therefore, we also optimized 3 × 3 depthwise-separable convolution using ARM NEON instruction and then distributed the computation to every CPU cores by channel using OpenMP [33].

In Figure 5, we demonstrate the optimization of 3 × 3 depthwise-separable convolution with stride 1 using ARM NEON instruction. We will calculate 4 NEON registers for output in each iteration. We load the corresponding part of input feature map into NEON registers, and compute the output according to the rule of 3 × 3 depthwise-separable convolution, as shown in Alg.3 Line 7-11.

Here, we describe how to calculate *Oreg*[1]'s result in Figure 5 detailly. In Figure 5, we use different colors to distinguish registers that load data from different columns. The red registers contain data from first column to the fourth column. The purple and yellow registers contain data from second column to fifth column and third column to sixth column, separately. Moreover, we use a variable *j* to represent registers in different rows. We denote the red register in the *j*th row by $ireg[j][1]$. The purple and yellow register in *j*th row by $ireg[j][2]$ and $ireg[j][3]$ separably. According to the 3 × 3 depthwish-separable convolution rules, we can get the following result:

$$\begin{aligned} &Oreg[1] \\ &= ireg_{1,1} \cdot ker_{1,1} + ireg_{1,2} \cdot ker_{1,2} + ireg_{1,3} \cdot ker_{1,3} \\ &\quad + ireg_{2,1} \cdot ker_{2,1} + ireg_{2,2} \cdot ker_{2,2} + ireg_{2,3} \cdot ker_{2,3} \\ &\quad + ireg_{3,1} \cdot ker_{3,1} + ireg_{3,2} \cdot ker_{3,2} + ireg_{3,3} \cdot ker_{3,3} \\ &= \sum_{x=1}^{3}\sum_{y=1}^{3} ireg_{x,y} \cdot ker_{x,y}. \end{aligned}$$

Here we denote $ireg[j][i]$ by $ireg_{j,i}$ for simplicity, and $ker_{j,i}$ is the parameter in the *j*th row and *i*th column of the kernel. So, the calculations of *Oreg*[2], *Oreg*[3] and *Oreg*[4] are similar to *Oreg*[1] as shown in Alg.3.

Note that we only show the optimization of 3 × 3 depthwise-separable convolution with stride 1, however our method is also applicable to other strides. We will show the performance of both stride 1 and 2 in Section VI.

D. MEMORY OPTIMIZATION

1) LIGHT MODE

Convolutional neural networks have to keep track of the intermediate features maps that are extracted during inference phase. Once a given layer's inference computation is

complete, however, the input feature map(s) of this layer will be not reused if there is no layer use the input feature map(s) later. So an optimization trick called light mode will release the memory of input feature maps which will not be reused later, after the computation of current network layer. The light mode will greatly reduce the memory footprint during inference phase. Experiments in section shows the impact of light mode.

2) MEMORY POOL

We employ light mode to reduce the memory footprint during inference of networks. However, light mode introduces a lot of memory allocation and deallocation operations which will degrade performance. Therefore, we design a memory pool specifically for deep neural networks, to improve performance by sacrificing some memory. The normal memory pool operates on some specific size of memory. Our memory pool uses feature map as the basic unit for memory allocation and deallocation, which is different from the normal memory pool. Only the features maps use our memory pool, the weights of the network do not use, because the weights of the network will be allocated at the beginning of the application and be deallocated when the application stop.

We demonstrate the workflow of the memory pool in Figure 6 for a 4-layer network. First, our memory pool allocate memory for the input and output of the first layer in the network. After the computation of the first layer, the input feature maps of the first layer may not be used anymore. So we can selectively use the memory of first layer input feature maps to store the output feature maps of the second layer. And, if the size of first layer input feature maps is less than the size of second layer output feature maps, our memory pool will allocate new memory for the second layer output feature maps, as shown in Figure 6.

Note that, memory pool will increase the maximum memory footprint during the inference process as shown in Figure 6. However, combined with light mode, memory pool will contribute to inference performance. We will show it in Section VI.

VI. EXPERIMENTS

This section evaluates the performance gains of the optimization methods we employed in Section V. We first introduce the experimental environment, including the hardware platform and the model we selected. Next we verify the performance improvement of various methods through experiments. Finally, we compare our system with other methods.

A. EXPERIMENTAL SETUP

We used a Firefly-RK3399 development board as our experiment platform. It integrates quad-core Cortex-A53 and dual-core Cortex-A72 with separate NEON coprocessor. Its memory size is 2GB. In this paper, we did not design a load balancing scheduling algorithm between CPUs with different computing powers. Therefore, unless mentioned, we used two Cortex-A72 cores as default.

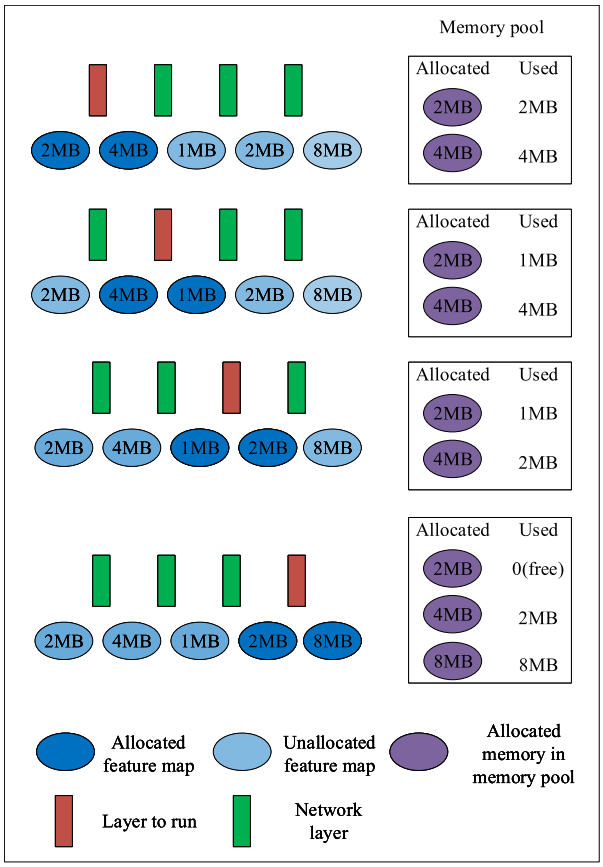


FIGURE 6. Memory pool and light mode demonstration. For brevity, we use an 4-layer linear network to show the workflow of memory pool. The left part shows the inference process of the 4-layer linear network, and the right part shows the evolution of the memory pool during the inference process.

As mentioned in Section III, for the sake of simplicity, we used Caffe's pretrained models for both MobileNet (trained on ImageNet) and MobileNet-SSD (trained on PASCAL VOC0712). The network structures of MobileNet and MobileNet-SSD have been described in [14]. We used some layer specifications in the MobileNet model to verify the performance gain of our optimization methods for 1×1 convolution and 3×3 depthwise-separable convolution. Also, we used MobileNet and MobileNet-SSD model to test the effect of our memory pool.

B. 1×1 CONVOLUTION PERFORMANCE

Table 2 summarizes the performance of 1×1 convolution compared to the non-optimized method. Our experiment tests several specifications of 1×1 convolution in MobileNet under various CPU core configurations. Comparing to the non-optimized method, our optimization method for 1×1 convolution greatly improves the inference efficiency of 1×1 convolution under almost all CPU core configurations. Our inference framework maximizes CPU utilization and works well when using multi-cores. When the number of CPU cores doubles, our inference framework performance is nearly doubled, as shown in Table 2.

TABLE 2. 1 × 1 convolution performance gain (in ms).

Input Size	Output Channel	1xA53		2xA53		4xA53		1xA72		2xA72	
		non	optimized	non	optimized	non	optimized	non	optimized	non	optimized
112x112x32	64	153.94	110.10	57.94	63.01	28.54	33.99	37.86	20.25	26.41	18.21
56x56x64	128	121.94	12.63	51.32	6.46	22.00	3.52	18,00	5.96	12.74	3.07
56x56x128	128	283.30	25.79	105.26	13.15	43.78	7.12	101.14	12.17	49.59	6.35
28x28x128	256	55.14	10.20	28.09	5.24	15.04	2.71	14.86	5.02	7.90	2.65
28x28x256	256	229.92	21.04	101.47	10.98	47.79	5.89	65.27	10.55	52.64	5.29
14x14x256	512	55.23	9.29	27.64	4.75	14.44	2.58	16.60	4.62	8.66	2.38
14x14x512	512	110.52	19.62	61.40	9.95	29.20	5.21	33.81	9.41	19.46	4.91
7x7x512	1024	55.96	9.51	27.95	4.80	14.24	2.53	17.73	4.64	9.84	2.52
7x7x1024	1024	110.42	18.85	55.64	9.60	28.12	5.17	36.12	9.15	18.49	5.05

TABLE 3. 3 × 3 depthwise-separable convolution with stride 1 performance gain (in ms).

Input Size	1xA53		2xA53		4xA53		1xA72		2xA72	
	non	optimized	non	optimized	non	optimized	non	optimized	non	optimized
112x112x32	13.40	6.14	6.98	3.57	4.17	2.59	5.60	4.03	3.47	2.74
56x56x128	11.53	4.93	6.06	2.76	3.55	2.14	4.59	2.85	2.81	2.04
28x28x256	5.51	2.14	2.92	1.18	1.70	0.85	1.77	1.06	1.15	0.94
14x14x512	3.70	1.96	1.97	1.09	1.11	0.78	1.28	0.92	0.79	0.56
7x7x1024	2.23	1.67	1.19	0.98	0.68	0.61	0.82	0.76	0.64	0.43

TABLE 4. 3 × 3 depthwise-separable convolution with stride 2 performance gain (in ms).

Input Size	1xA53		2xA53		4xA53		1xA72		2xA72	
	non	optimized	non	optimized	non	optimized	non	optimized	non	optimized
112x112x64	9.18	8.04	5.35	4.53	3.70	3.38	4.65	4.62	3.32	3.40
56x56x128	4.61	2.90	2.62	1.67	1.83	1.40	2.42	1.48	1.80	1.31
28x28x256	2.82	1.76	1.57	1.07	1.05	0.84	1.55	1.00	1.14	0.81
14x14x512	1.84	1.64	1.06	0.90	0.73	0.60	0.98	0.56	0.47	0.37

The irregular results do exist in Table 2, which we identified by underline. Also, the fact exists in Table 2 that the performance of non-optimized method will be more than double when the number of CPU cores doubles. It should be the cause of compiler optimization causing these irregular results.

The reason why our optimization method can improve performance is that it enhances memory access locality and makes cache hit rate higher. The performance gain with input size of $112 \times 112 \times 32$ is at most 30%, whereas the gain of the other input size is about $70\% - 90\%$. This is because without optimization, the number of input channels directly affects the hit rate of the cache. The fewer the number of input channels, the better the locality of cross-channel memory access due to the calculation of output data. So, the performance gain with input channel of 32 is relatively small since it has better locality of memory access.

C. 3 × 3 DEPTHWISE-SEPARABLE CONVOLUTION PERFORMANCE

The computational intensity of 3 × 3 depthwise-separable convolution is not particularly high compared to the 1×1 convolution. However, optimizing the 3×3 depthwise-separable convolution still has considerable benefits, since the number of 3 × 3 depthwise-separable convolution is large in many lightweight CNN models.

We summarizes the performance of 3 × 3 depthwise-separable convolution with stride 1 in Table 3, and the performance of 3 × 3 depthwise-separable convolution with stride 2 in Table 4. The specifications of 3 × 3 depthwise-separable convolution are also coming from MobileNet model, like 1 × 1 convolution.

The performance gain of 3 × 3 depthwise-separable convolution comes from the usage of ARM NEON instruction. The gain of stride 1 is larger than stride 2 since the computational cost of stride 2 is much smaller than stride 1 and compiler can optimizes stride 2 better. Although the performance improvement is not very high, the number of 3 × 3 depthwise-separable convolution is very large. So, the optimization of 3 × 3 depthwise-separable convolution is valuable (suppose each layer is reduced by 1 second, then n layers are reduced by n seconds).

D. MEMORY POOL EFFECT

Our memory pool adopts a layer-wise memory allocation policy. We use MobileNet and MobileNet-SSD to illustrate the impact of memory pool on memory footprint and inference performance with convolutional optimiztions or without.

In Figure 7, we show the changes in memory usage during the three inferences of MobileNet, under four different memory strategies, namely no light mode without memory pool, no light mode with memory pool, light mode without memory pool and light mode with memory pool. In terms of maximum memory footprint, light mode reduces the memory usage, comparing (c) with (a)

FIGURE 7. Memory usage during three inferences of MobileNet. (a) no light mode without memory pool, (b) no light mode with memory pool, (c) light mode without memory pool (d) light mode with memory pool.

(without memory pool) and (d) with (b) (with memory pool) in Figure 7. By comparing (b) with (a) and (d) with (c) in Figure 7, our memory pool slightly increases some memory footprint.

However, if we consider from the performance perspective, our memory pool improves the performance of our inference framework, regardless of whether there are convolutional optimizations or not, as shown in Table 5. The performance improvement of memory pool for MobileNet is not ideal, so we added the experiment of MobileNet-SSD to show this.

TABLE 5. Performance under different memory strategies (in ms).

With convolutionl optimiztions	Light mode	Memory pool	MobileNet	MobileNet-SSD
No	No	No	276.53	578.38
	No	Yes	227.38	518.56
	Yes	No	292.64	592.57
	Yes	Yes	**198.73**	**493.46**
Yes	No	No	90.11	228.44
	No	Yes	81.30	181.45
	Yes	No	95.17	238.38
	Yes	Yes	**80.10**	**154.68**

In order to reduce the maximum memory footprint, we design light mode. But light mode will increase the number of system calls, because it will allocate and free memory frequently. So, our memory pool can solve this problem. This is why our memory pool can enhance the performance. As described in Section V, our memory pool will slightly increase the maximum memory footprint during the inference process. But, our memory pool will contribute to inference performance when combined with light mode.

E. COMPARISON WITH OTHER APPROACHES

We now compare the performance of our inference framework with TensorFlow-Lite, the open source deep learning framework for on-device inference developed by Google. We also compare our framework with Caffe, a very influential framework, but we compile Caffe with OpenBLAS on the Firefly-RK3399 development board. Moreover, we compare our framework with NCNN [27] and MNN [28]. These two frameworks are popular and with high performance in the open source community.

As shown in Table 6, our inference framework outperforms TensorFlow-Lite, Caffe-OpenBLAS and MNN. In the case of MobileNet, our framework can be comparable to NCNN. But our framework performs better than NCNN in terms of MobileNet-SSD. Although our inference framework has been specifically optimized for very few network layers, we have optimized those layers that are very time consuming. Therefore, as long as the network structure uses the network layer we have optimized, our framework can produce some acceleration effects.

TABLE 6. Compare with other approaches.

Frameworks	MobileNet	MobileNet-SSD
TensorFlow-Lite	96ms	195ms
Caffe-OpenBLAS	122ms	235ms
MNN	89ms	182ms
NCNN	**79ms**	174ms
Ours	80ms	**155ms**

VII. CONCLUSION AND FUTURE WORK

In this paper, we design and implement an embedded inference framework to accelerate the inference process of

convolutional neural networks on embedded platform. Our framework specifically optimizes 1×1 convolutional layer and 3×3 convolutional layer, greatly improving the inference performance of our framework. We also design a memory pool specifically for deep neural network to achieve significant speedups. We implement our inference framework with OpenMP to use multi-cores and with ARM NEON to maximize the use of a single CPU core resources. The experimental results on Firefly-RK3399 show that our inference framework outperforms other approaches.

Another interesting aspect of this work that can be explored in the future would be optimizing general 3×3 convolutional layer with winograd algorithm to further improve the inference performance. We may support our framework with more convolutional neural network models in the future.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, vol. 141, no. 5, pp. 1097–1105.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Represent.*, 2015, pp. 1–14.

[4] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 2818–2826.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.

[6] S. Abolfazli, Z. Sanaei, E. Ahmed, A. Gani, and R. Buyya, "Cloud-based augmentation for mobile devices: Motivation, taxonomies, and open challenges," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, pp. 337–368, 1st Quart., 2014.

[7] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making smartphones last longer with code offload," in *Proc. 8th Int. Conf. Mobile Syst., Appl., Services*, 2010, pp. 49–62.

[8] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 301–314.

[9] X. Zhang, A. Kunjithapatham, S. Jeong, and S. Gibbs, "Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing," *Mobile Netw. Appl.*, vol. 16, no. 3, pp. 270–284, 2011.

[10] O. Valery, W.-S. Hung, J.-C. Chou, P. Liu, and J.-J. Wu, "Adaptive OpenCL computation offloading framework on mobile device," in *Proc. ICS*, 2014, pp. 1335–1344.

[11] O. Valery, J.-C. Chou, Y. Tsao, P. Liu, and J.-J. Wu, "A partial workload offloading framework in a mobile cloud computing context," in *Proc. IEEE 8th Int. Conf. Service-Oriented Comput. Appl.*, Oct. 2015, pp. 43–50.

[12] H. Eom, P. S. Juste, R. Figueiredo, O. Tickoo, R. Illikkal, and R. Iyer, "OpenCL-based remote offloading framework for trusted mobile cloud computing," in *Proc. Int. Conf. Parallel Distrib. Syst.*, Dec. 2013, pp. 240–248.

[13] W. Kang and J. Chung, "Power- and time-aware deep learning inference for mobile embedded devices," *IEEE Access*, vol. 7, pp. 3778–3789, 2018.

[14] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*. [Online]. Available: https://arxiv.org/abs/1704.04861

[15] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size," 2017, *arXiv:1602.07360*. [Online]. Available: https://arxiv.org/abs/1602.07360

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

[17] *TensorFlow-Lite*. Accessed: Jun. 13, 2019. [Online]. Available: https://www.tensorflow.org/lite/

[18] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for convolutional networks," in *Proc. Int. Conf. Field Program. Logic Appl. (FPL)*, Aug./Sep. 2009, pp. 32–37.

[19] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. 2015 ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2015, pp. 161–170.

[20] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Proc. IEEE Comput. Vis. Pattern Recognit. Work. (CVPRW)*, Jun. 2011, pp. 109–116.

[21] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[22] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2014, pp. 609–622.

[23] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "ShuffleNet V2: Practical guidelines for efficient CNN architecture design," in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 116–131.

[24] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetv2: Inverted residuals and linear bottlenecks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Apr. 2018, pp. 4510–4520.

[25] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," 2014, *arXiv1408.5093*. [Online]. Available: https://arxiv.org/abs/1408.5093

[26] *OpenBLAS*. Accessed: Jun. 13, 2019. [Online]. Available: http://www.openblas.net/

[27] *NCNN*. Accessed: Jun. 13, 2019. [Online]. Available: https://github.com/Tencent/ncnn

[28] *MNN*. Accessed: Jun. 13, 2019. [Online]. Available: https://github.com/alibaba/MNN

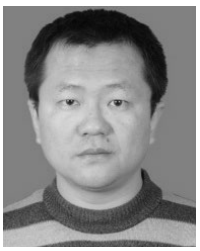
[29] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," 2015, *arXiv:1511.06530*. [Online]. Available: https://arxiv.org/abs/1511.06530

[30] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned CP-decomposition," 2014, *arXiv:1412.6553*. [Online]. Available: https://arxiv.org/abs/1412.6553

[31] L. N. Huynh, Y. Lee, and R. K. Balan, "DeepMon: Mobile GPU-based deep learning framework for continuous vision applications," in *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Services*, 2017, pp. 82–95.

[32] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *Proc. NIPS Autodiff Workshop*, 2017.

[33] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Mar. 1998.



SHENG BI received the M.Sc. and Ph.D. degrees from the South China University of Technology, Guangzhou, China, in 2003 and 2010, respectively. He has been working as a Fellow with the School of Computer Science and Engineering, South China University of Technology, since July 2003, where he was appointed as an Associate Professor, in September 2014. He is also a Research Fellow with the Shenzhen Academy of Robotics. His research interests are intelligent robots, FPGA fast processing algorithms, embedded intelligent terminal, and smartphone development.

YINGJIE ZHANG is currently pursuing the degree with the School of Computer Science and Engineering, South China University of Technology. His research interests include machine learning and deep learning on embedded platform.

MIN DONG received the Ph.D. degree from the University of Science and Technology of China, Hefei, China, in 2005. She has been working as a Fellow with the School of Computer Science and Engineering, South China University of Technology, since July 2003, where she was appointed as an Associate Professor, in September 2009. Her research interests include the Internet of Things (IoT), software architecture, and data mining and analysis.

HUAQING MIN was born in Nanxian, Hunan, China, in 1955. He received the Ph.D. degree in computer software and theory from the Huazhong University of Science and Technology, Wuhan, China, in 1998. He is currently a Professor with the School of Software Engineering, South China University of Technology, Guangzhou, China. His current research interests focus on robotics, intelligent software, and automated systems. He also serves as a Committee Member of the RoboCup in the China Chapter, and makes active contributions to the robot soccer community.