Deep Fusion: A Software Scheduling Method for Memory Access Optimization

Yimin Zhuang[1,2](✉), Shaohui Peng[1,2], Xiaobing Chen[1,2], Shengyuan Zhou[1,2], Tian Zhi[1,2], Wei Li[1,3], and Shaoli Liu[1,3]

[1] SKL of Computer Architecture, Institute of Computing Technology, CAS, Beijing, China
{zhuangyimin,pengshaohui18z,chenxiaobing,zhousy,zhitian, liwei2017,liushaoli}@ict.ac.cn

[2] University of Chinese Academy of Sciences, Beijing, China

[3] Cambricon Tech. Ltd, Shanghai, China

Abstract. Deep neural networks (DNNs) have been considered to be the state-of-the-art artificial intelligence methods in a very broad range of applications. However, DNNs are compute intensive and memory intensive which are difficult to be employed in practical scenarios. Due to their favorable parallel computing ability, a series of DNN accelerators have been proposed. However, the improvement of on-chip computing capacity and the increasing number of parameters in the neural networks make access to memory a bottleneck. In this paper, we analyze the existing DNN algorithms. We observe that the special structure of neural networks makes it have two useful characteristics, which are unilateral directivity and local independence. Based on these characteristics, we propose a general software scheduling method to reduce memory access cost. Based on the experimental results, our method can reduce 32% memory access cost and achieve a speedup of 1.6x in average on our experiment platform and the best result is in ResNet-50, which is up to 56% and 2.62x.

Keywords: Fusion · Reuse · On-chip Memory

1 Introduction

Deep neural networks (DNNs) are ubiquitous in a very broad range of applications, such as speech recognition [1], object detection [2,3], semantic segmentation [4] and so on. With the continuous development of DNNs both the number of neurons and synapsis increases exponentially. As a result, the operations of computing and memory accessing will grow far beyond the hardware processing capability especially for the embedded systems. A large number of solutions have been proposed by the researchers to address this limitation, such as pruning [5],

© IFIP International Federation for Information Processing 2019
Published by Springer Nature Switzerland AG 2019
X. Tang et al. (Eds.): NPC 2019, LNCS 11783, pp. 277–288, 2019.
https://doi.org/10.1007/978-3-030-30709-7_22

data compressing [6], low-precision quantization [7], etc. However, the existing general processor platforms (such as CPU, FPGA, DSP, etc.) are still difficult to fully meet the requirements of practical applications.

Some researchers considered the general characteristics of DNN algorithms and designed neural network accelerators [11–13]. DianNao [8] is a dedicated accelerator which makes advantages of the data locality and computational properties of DNNs. DaDianNao [9] adopts time-division multiplexing of neurons to acquire high performance. EIE [10] utilizes sparse data to speed up the process of computation. Generally, DNN accelerators prefer to add private on-chip memory for performance improvement. Data is loaded from DRAM to on-chip memory and then the results are stored back to DRAM after computation. However, for most of the neural network accelerators, a large increase in the computational resources will aggravates the shortage of memory bandwidth and resource contention of on-chip network. The data transmission latency between internal and external storage will make up a large portion in the program execution time.

In this work, we propose a general software scheduling method to optimize the memory access by making advantages of both unidirectional data transportation and local data independence. Besides, we propose an on-chip memory reuse method to expand the on-chip memory size.

The paper is organized as follows. In Sect. 2, we show the bottleneck that we face of memory access and the optimization potential of DNNs. In Sect. 3, we introduce the details of our method. The experimental methodology and experimental results are presented in Sect. 4. Section 5 makes a conclusion at last.

2 Motivation

2.1 Memory Access Bottleneck

Most DNN algorithms are computational and memory intensive. A number of accelerators which can offer high compute capability have been proposed to solve the computationally intensive problem. As a matter fact, the current mainstream neural network accelerators have TFLOPS-level operation capability which is far beyond the bandwidth of the current external memory. However, most of these work assume away the question of memory access. To illustrate this problem, we analyze the amount of computation and memory access for all layers in ResNet-18 [21].

As shown in Fig. 1, the ratio of computation to memory access for each layer is different in ResNet-18 which need different requirement for bandwidth and compute capability. Taking the element-wise layer as example, we need a bandwidth of 12 GB/s if our compute capability is 1 GFLOPS. Meanwhile, the requirement of bandwidth is only 10 MB/s for convolution layers with the same compute capability of 1 GFLOPS. Although the hardware architecture of neural network accelerators is well-designed to make a balance between memory bandwidth and computation capability, they will never reach their full potential

without software optimization. We further statistics the proportion of computation and memory access for each layer in the whole ResNet-18. As shown in Table 1, more than 95% of data transmission account is in certain layers including convolution layers, BatchNorm layers, scale layers, ReLU layers and eltwise layers. However, the computation amount of these layers is small except convolution layers, which is less than 1% in the whole network. Thus, the bottleneck of memory access is serious in these layers.

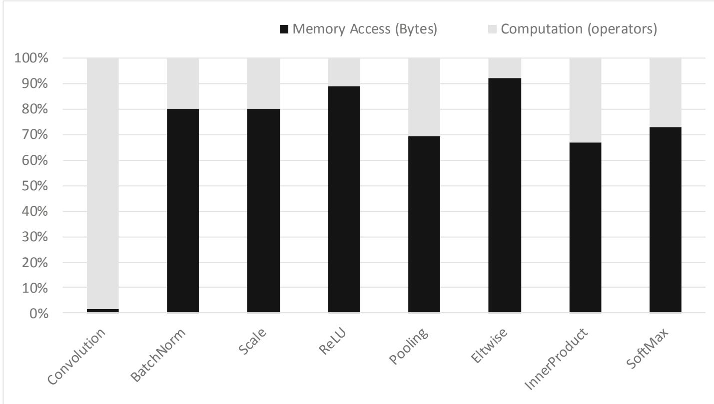

Fig. 1. Compute-to-global-memory-access ratio for each layer in ResNet-18

2.2 Potential of Optimization

To solve this bottleneck, we further analyze the characteristics of DNN algorithms. We can neutralize the inhomogeneity between memory access and computation of different layers by fusing multiple layers together because of the important characteristic of unilateral directivity in DNN algorithms. Since the dataflow of DNNs is unilateral, multiple layers can be computed together. Neurons are stored on-chip and are considered as input neurons for one hidden layer. The output neurons are still stored on-chip and are used as input neurons for next hidden layers. Besides, we can obviate a large number of data transmission from DRAM to on-chip memory or on-chip memory to DRAM. We observe that more than 99.6% data transmission can be reduced in ResNet-18 if all layers can be fused together.

However, it is almost unavailable to fuse all layers in real neural networks. Among the reasons for this state of affairs, one may cite the mismatch between the size of on-chip memory and neurons, the small on-chip memory size limited by the hardware overhead and the vast on-chip memory size needed to cache the intermediate data of the fused layers. Another important characteristic of DNN algorithms to relieve this problem is local independence. Each point of output neurons in one layer only depends on a defined region of input neurons. Hence, the neurons can be tiled into pieces and we can compute each piece separately. Thus,

Table 1. Proportion of computation and memory access for each layer in ResNet-18

Layer	Computation amount	Memory accessing amount
Convolution	99.55%	45.59%
BatchNorm	0.15%	15.27%
Scale	0.15%	15.24%
ReLU	0.06%	13.10%
Pooling	0.05%	2.92%
Eltwise	0.02%	6.41%
InnerProduct	0.03%	1.46%
SoftMax	0.00%	0.01%

we can fuse more layer with the same on-chip memory size. And combining the aforementioned features, we further propose an on-chip memory reuse method. We will present the detail of our method in the following section.

2.3 Existing Works

Some works which fuse the active layers after the convolution layers have been done in some mainstream machine learning frameworks, including MXNet [14] and Tensorflow [15,16]. However, the compute capacity is much higher than bandwidth in most accelerators. The ratio in GPU V100 is 10x and it is much greater in other accelerators. Thus, it is meaningful to fuse more layers. Manoj Alwani et al. [17] proposed a method to fuse multiple convolution layers, but it aims at hardware implementation. As a result, it is not general and flexible enough. Thus, a general software scheduling method with deeper fusion is important to solve the bottleneck of memory access.

3 Optimization Method

In this section, we propose a software scheduling method on neural network accelerators. The method consists of two mainly parts. One is layer fusion by software which can greatly reduce the demand of memory access. The other is on-chip memory reuse method, which can solve the large memory space required by layer fusion and the limitation of on-chip memory size in the accelerators. We will tile the data of each layer into pieces, then for each calculation, we get a piece of output from corresponding pieces of input, as shown in Fig. 2. We will describe our method in detail in this section.

3.1 Layer Fusion

At first, we show the details of the software scheduling method. To make the program of software more flexible, we decoupled the fusion process into two phases. One phase is operational-related shape deduction (SD) and the other phase is operational-independent shape transfer (ST).

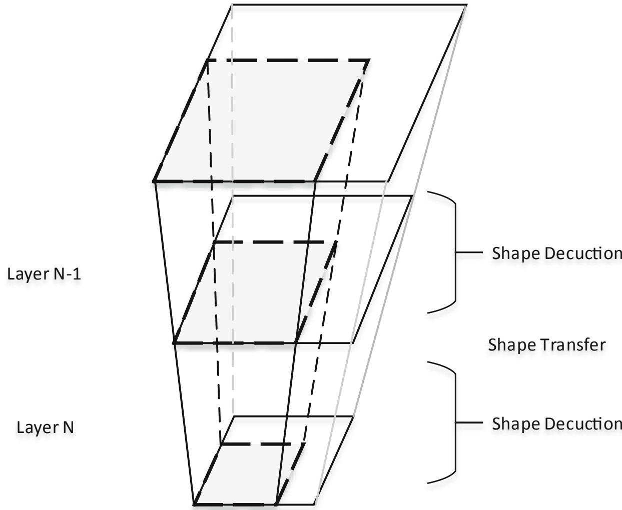

Fig. 2. An example for layer fusion process

SD. Shape deduction is to get the coordinate relationship between input data and output data. Most of layers in DNNs have four dimensions including batch, height, width and channel. In this subsection, for simplicity, we use the shape deduction between height dimension and width dimension as an example. For other dimension or dimensions greater than four, the method is almost the same. Besides, we prefer to infer shape from output shape to input shape, because sometimes there is redundant data for input data which will affect the shape deduction.

We use $Range(W)$ to represent an interval and $W \in [w_b, w_e)$. Similarly, we can use $Range(X, Y)$ to represent a range on a two-dimensional plane. Hence, we can use the following expressions to represent the process of shape deduction.

$$Range(W_i, H_i) = kernel(W_o, H_o)$$

Where $kernel(\cdot)$ is the function of shape deduction, $Range(X_i, Y_i)$ is coordinate range of input data and $Range(X_o, Y_o)$ is the coordinate range of output data. The shape deduction is related to the operators of each layer. For different layers, the deduction formulas are different. We take some typical layers as examples and we use xxx_kernel to distinct different kernel functions. Here xxx is usually an abbreviation of layer name.

Convolution/Pool. Convolution and pool are the most typical layers in DNNs. There are some basic parameters in these operators, such as kernel size, stride, etc. We use kh, kw, sh and sw as abbreviations.

$$
\begin{aligned}
&Range(W_i, H_i) = cvpl_kernel(Range(W_o, H_o)):\\
&(W_{i_b}, W_{i_e}) = (W_{o_b} * sw, W_{o_e} * sw + kw)\\
&(H_{i_b}, H_{i_e}) = (H_{o_b} * sh, H_{o_e} * sh + kh)
\end{aligned}
$$

Pad. Pad operator generally occurs in convolution or pooling layers. However, pad operator will change the shape of data, thus we make it as a separate layer.

$$
\begin{aligned}
&Range(W_i, H_i) = pad_kernel(Range(W_o, H_o)):\\
&W_{i_b}(W_{i_e}) = \begin{cases} 0 & if \quad W_{o_b}(W_{o_e}) < pad_left \\ W_{o_b}(W_{o_e}) - pad_left & if \quad pad_left \leq W_{o_b}(W_{o_e}) \leq W + pad_left \\ W & if \quad W_{o_b}(W_{o_e}) > W + pad_left \end{cases}\\
&H_{i_b}(H_{i_e}) = \begin{cases} 0 & if \quad H_{o_b}(H_{o_e}) < pad_up \\ H_{o_b}(H_{o_e}) - pad_up & if \quad pdf_up \leq H_{o_b}(H_{o_e}) \leq H + pad_up \\ H & if \quad H_{o_b}(H_{o_e}) > H + pad_up \end{cases}
\end{aligned}
$$

BatchNorm/Scale/Active. For these layers, they will not change the shape of data, thus it makes shape deduction directly.

$$
\begin{aligned}
&Range(W_i, H_i) = elt_kernel(Range(W_o, H_o)):\\
&(W_{i_b}, W_{i_e}) = (W_{o_b}, W_{o_e})\\
&(H_{i_b}, H_{i_e}) = (H_{o_b}, H_{o_e})
\end{aligned}
$$

ST. In shape deduction phase, each layer only focuses on the coordinate of output data and returns the coordinate of the input data. In shape transfer phase, it will call the kernel function defined in shape deduction phase. The coordinate of output data will be set as input to the kernel function of current layer and the result of kernel function will be passed to the kernel function of the previous layer. Thus, we will get all coordinate information of all layer be fused after we go through all these layers. The pseudocode is shown in Fig. 3.

```
ShapeTransfer(Range(Xo,Yo)):
Coordinates[FusionLayerNum] = Range(xo,Yo)
For (LayerIndex=FusionLayerNum-1; LayerIndex>=0; LayerIndex--):
    Coordinates[LayerIndex]=kernel(Coordinates[LayerIndex+1])
Return coordinates;
```

Fig. 3. Pseudocode for shape transfer

3.2 On-Chip Memory Reuse

Although layer fusion can greatly reduce the requirements of memory access, it is limited by the size of on-chip memory. In this part, we analyze the characteristics of DNN at first, and then introduce the on-chip memory reuse method which makes use of the characteristics to break the memory limitation.

Base on the characteristic of unilateral directivity for DNNs, once the input data of one layer has been used and this data does not need by other layers, the memory space of this data can be reused. Besides, the shape of data can be gotten in advance in most of inference phase, which makes data reused on-chip is available.

The process of data distribution can be represented in a simplified sequence. Here, we only care about the point when the memory usage status changes, and we define the equivalent life time of each data from the allocate point to the free point. To illustrate this process more clearly, an example sequence is shown in left side of Fig. 4.

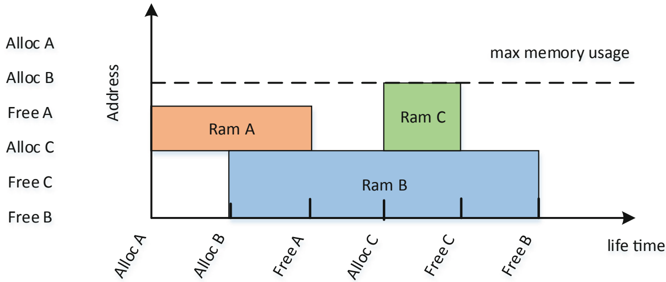

Fig. 4. The left side shows an example of memory distribution sequence. The right side shows a possible memory distribution result.

Figure 4 also shows an intuitive memory reuse method. We consider two data dependent if there is overlap between the life times of these two data. Otherwise, the memory space can be shared by these two data. As what we show in Fig. 4, ram A and ram B are dependent, and ram B and ram C are dependent too. But for ram A and ram C, they are independent, thus they share the same memory space.

The intuitive memory reuse method can save a large number of space, but it is limited by the data size. Once the size of input data or the size of output data is larger than on-chip memory size, we cannot fuse more layers.

To make these cases can be fused, we propose a deep memory reuse method. Base on the characteristics of local independence, when a local part of input data has been used to get a local part of output data, the memory space of this local part can reused. Thus, even if the life time of input data and output data has overlap, the output data can reuse part of memory space of the input space. The most special case is some element wise layers, such as add, BatchNorm,

scale, etc. The input data and output data of these layers have the same size and can share the same memory space.

To illustrate the point more clearly for general cases, we take a concrete example of convolution layer which is shown in Fig. 5.

The horizontal axis in Fig. 5 is the growth of output data in H and W dimension. Here, we consider the multiplication of height and width as one dimension. For the calculation of each point, we get the address of first point in the piece of data we need and the address of current output point. Then, we join these points into two lines, as shown in Fig. 5. For each point, the memory space for those input data whose address is below the input address line can be reused, because these data have been used to calculate the output data before current point.

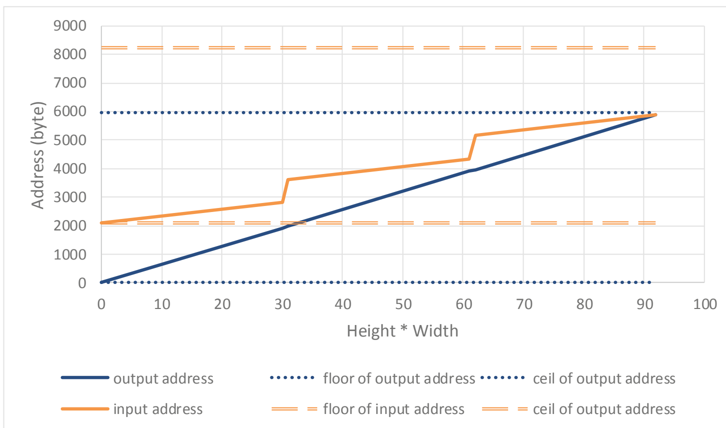

Fig. 5. The height, width and channel of input data and output data for this convolution is [8, 64, 3] and [3, 31, 16] and the sizes of kernel and stride are 4×4 and 2×2.

Even the life time of output data and input data is overlapped, they can still share a part of memory space. As shown in Fig. 5, the address range of input data is from 2096 (bytes) to 8240 (byte) and that of output data is from 0 (byte) to 5952 (byte). Thus, we reduce 31% memory usage in this case.

3.3 Fusion Method

Combining the methods above, we will show the implementation of our fusion method in this part. Figure 6 is an executive flow chart of our method. For each fusion, we first tile output data of the last layer into pieces. Then we do shape transfer for each piece and in the phase of shape transfer, it will call the kernel functions defined by each fused layer to do shape deduction. We then allocate memory space for all data. If the memory distribution is well-done, we can try

to fuse the next layer. Otherwise, we shrink the tiled size and try to do shape transfer and memory distribution again. If the tiled size is the smallest tiled size, it means the current layer cannot be fused and returns the already fused layer list.

This process can be done before networks execution and we can get the coordinate and memory address for each piece of data. According to these information, we can execute the entire network through the specific instruction set or opcodes provided by the accelerators. The pseudocode is shown in Fig. 7. The in and out are the first input data and last output data for current fused list. The compute function is defined by each layer according to their own algorithms. The input data is loaded from DRAM and all middle data is stored on-chip. The output data is stored back to DRAM when we get a piece of final results.

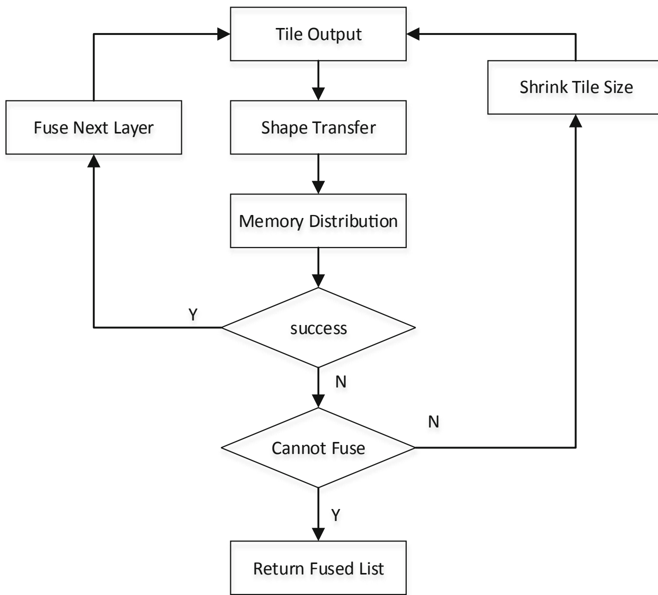

Fig. 6. Flowchart of layer fusion method

```
Execution (in, out):
  For (i = 0; i < PiecesNum; i++):
    mid_data = Load(in[i])
    For (j = 0; j < FusionLayerNum; j++):
        mid_data = compute<i>(mid_data)
        out[i] = mid_data
    store(out[i])
```

Fig. 7. Pseudocode for execution

4 Experiment

4.1 Experiment Methodology

We design a prototype accelerator as our experiment platform. The structure of the prototype refers to the design of DaDianNao [9]. In our experiment platform, we limit the bandwidth between DRAM and on-chip memory to 1.5 GB/s. the compute capability of the prototype accelerator is 200 GFLOPS and we set 768 KB size of on-chip memory.

We choose five typical NN models as the benchmarks to evaluate our method, i.e. VGG-19 [18], GoogLeNet [19], InceptionV3 [20], ResNet-18 [21] and ResNet-50 [21]. Besides, we evaluate our optimization in the prototype accelerator, and compare the result of the memory access reduction and execution time improvement between the method without optimization, with only layer fusion and with both layer fusion and on-chip memory reuse.

4.2 Layer Fusion Result

We take the results of the method without optimization as the baseline. Then we test two methods with only layer fusion and with both layer fusion and on-chip memory reuse. The results are presented in Figs. 8 and 9.

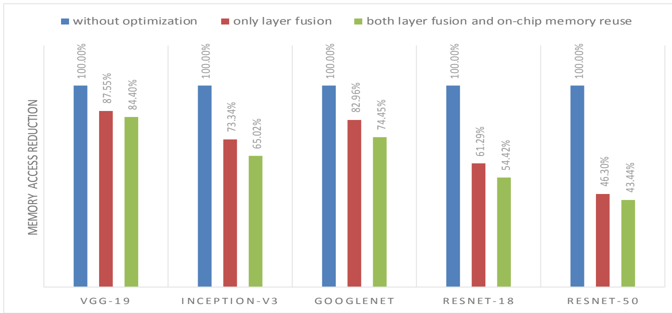

Fig. 8. Ratio of memory access reduction w.r.t the baseline.

Figure 8 shows the result of memory access reduction rate compared with baseline. We get more than 15% reduction of memory access in our benchmarks, especially for ResNet-50 which acquires 56% reduction. The performance improvement of execution time has similar tendency as shown in Fig. 9. We get at least 1.26x performance improvement in VGG-19 and at most 2.62x performance improvement in ResNet-50. Besides, we can get a better effect which is more than 5% improvement in average for both memory access reduction and performance by using on-chip memory reuse in addition.

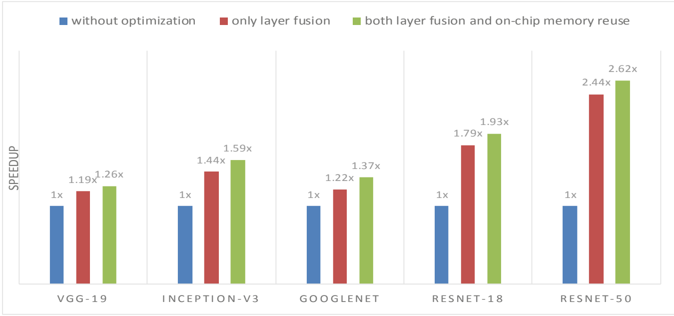

Fig. 9. Speedup w.r.t. the baseline (execution time).

We find that the result in VGG-19 is not much better than other networks. Then we further analyze the fusion status in VGG-19. We observe that the most layers in VGG-19 is convolution layers and the kernel size of all these layers is 3x3 which result in a large synapse data size. However, synapse is shared by all input data in convolution layers. Thus, if we tile input data into pieces, each piece of data need the same synapse data and these synapse data will take up an independent memory space. The memory space is run out of soon if we fuse more convolution layers. If we have much more on-chip memory space, we can layer out more synapse data or we can reuse the space of synapse data if the input data is not tiled into pieces. However, it is a tradeoff between performance and the area of accelerators.

5 Conclusion

The development of neural network accelerators makes DNNs run faster and faster, but the slowly development of bandwidth for DRAM makes accelerators stuck in a memory access bottleneck. It is very important to solve this problem to utilize accelerators more effective.

In this paper, we propose a new software scheduling method to optimize memory access. It mainly consists of two parts, one is layer fusion and the other is on-chip memory reuse. They utilize the properties of DNNs. That is unilateral directivity and local independence. Based on the experimental results, our method achieves 32% memory access reduction and 1.6x speedup in average. To get a better performance, we can expand the amount of space on chip, however, it is a tradeoff between performance and the area of accelerators.

Acknowledgment. This work is partially supported by the National Key Research and Development Program of China (under Grant 2017YFB1003104), the NSF of China (under Grants 61432016, 61532016, 61672491, 61602441, 61602446, 61732002, 61702478, 61732007 and 61732020), Beijing Natural Science Foundation (JQ18013), the 973 Program of China (under Grant 2015CB358800), National Science and Technology Major Project (2018ZX01031102), the Transformation and Transfer of Scien-

tific and Technological Achievements of Chinese Academy of Sciences (KFJ-HGZX-013), Key Research Projects in Frontier Science of Chinese Academy of Sciences (QYZDB-SSW-JSC001) , Strategic Priority Research Program of Chinese Academy of Science (XDB32050200, XDC01020000) and Standardization Research Project of Chinese Academy of Sciences (BZ201800001).

References

1. Xiong, W., et al.: Achieving human parity in conversational speech recognition. In: IEEE/ACM Transactions on Audio, Speech, and Language Processing, p. 99 (2016)
2. Ren, S., et al.: Faster R-CNN: towards real-time object detection with region proposal networks. IEEE Trans. Pattern Anal. Mach. Intell. **39**(6), 1137–1149 (2017)
3. Redmon, J., Farhadi, A.: YOLOv3: An Incremental Improvement (2018)
4. Noh, H., Hong, S., Han, B.: Learning Deconvolution Network for Semantic Segmentation (2015)
5. Han, S., et al.: Learning both Weights and Connections for Efficient Neural Networks (2015)
6. Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding. Fiber **56**(4), 3–7 (2015)
7. Jacob, B., et al.: Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference (2017)
8. Chen, T., et al.: DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. ACM Sigplan Not. **49**(4), 269–284 (2014)
9. Chen, Y., et al.: DaDianNao: A Machine-Learning Supercomputer (2014)
10. Han, S., et al.: EIE: efficient inference engine on compressed deep neural network. ACM Sigarch Comput. Archit. News **44**(3), 243–254 (2016)
11. Shen, Y., Ferdman, M., Milder, P.: Escher: a CNN accelerator with flexible buffering to minimize off-chip transfer. In: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) IEEE Computer Society (2017)
12. Chen, Y.-H., et al.: Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE J. Solid-State Circuits **52**(1), 127–138 (2017)
13. Liu, S., et al.: Cambricon: an instruction set architecture for neural networks. In: ACM/IEEE International Symposium on Computer Architecture (2016)
14. Chen, T., et al.: MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. Statistics (2015)
15. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning (2016)
16. Abadi, M., et al.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems (2016)
17. Alwani, M., et al.: Fused-Layer CNN Accelerators. In: IEEE/ACM International Symposium on Microarchitecture (2016)
18. Simonyan, K., Andrew Z.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
19. Szegedy, C., et al.: Going Deeper with Convolutions (2014)
20. Xia, X., Cui, X., Bing, N.: Inception-v3 for flower classification. In: International Conference on Image (2017)
21. He, K., et al.: Deep Residual Learning for Image Recognition (2015)