An Efficient Accelerating Method for Large-scale Sparse Neural Networks

Yuntao Lu[1], Chao Wang[2], Lei Gong[3], Xuehai Zhou[4]

University of Science and Technology of China, Hefei, Anhui, China
{luyuntao[1], leigong0203[3]}@mail.ustc.edu.cn
{cswang[2], xhzhou[4]}@ustc.edu.cn

Abstract. Neural networks have been widely used as a powerful representation in lots of research domains, such as Computer Vision, Natural Language Processing, and Artificial Intelligence, etc. In state-of-the-art researches intend to increase the number of neurons and synapses, which makes it both computationally and memory intensive and difficult to deploy on resource-limited platforms. Sparse methods are raised to reduce redundant synapses of neural networks, but conventional accelerators cannot benefit from the sparsity .
In this paper, we propose an efficient accelerating method for sparse neural networks, which compresses sparse synapse weights and processes the compressed structure by an FPGA accelerator. Our compression method will achieve 50% and 10% compression ratio of synapse weights in convolutional and full-connected layers. The experiment results demonstrate that our accelerating method can boost an FPGA accelerator to achieve 3x speedup over a conventional one.

Keywords: Sparse neural networks, Compressed formats, FPGA accelerators

1 Introduction

Neural networks (NNs) are widely used for a large range of applications, such as Computer vision [12], Natural Language Processing [16], and Artificial Intelligence [23], etc. In order to achieve better performance, lots of researches make NN more complicated and large-scale. Besides CPUs and GPUs, NN-specific hardware accelerators perform better energy and performance efficiency [7, 22], thus researches have proposed a variety of accelerators [3, 6, 15, 19, 24].

To achieve higher accuracy, advanced NNs, such as AlexNet [12] and VGG [21], have large-scale size with millions of neurons and synapse weights. For example, AlexNet with a size of 6.5×10^5 neurons and 6×10^7 synapse weights in 2012, and increases to 1×10^6 neurons and 1×10^9 synapse weights [13], or even more [4]. Lots of synapse weights result in intensive computations and memory accesses, hence, it is a challenge to process large-scale NNs in various platforms with different hardware resources. To address the challenge, amounts of training methods, such as sparse coding [17] and automatic encoding/decoding [18], etc,

eliminates a large number of redundant neurons and synapses without accuracy loss. The novel method proposed by Han et al. [10], which utilizes pruning and retraining method, achieves 10% sparsity ratio (the fraction of remaining synapse weights over the originals) with no accuracy loss.

However, the reduction of neurons and synapses does not significantly improve the performance of conventional accelerators, for example, AlexNet with the 11% weights reduction achieves 1.78 speedup over GPU with cuSPARSE library [25]. Additionally, specific NN accelerators, for instance, DianNao [3], fill zero values in the location of pruned synapse weights, that takes unnecessary multiplications with zero.

In this paper, we focus on sparse synapse weights compression which can alleviate storage space and propose an FPGA accelerator to process the compressed data efficiently. Compression formats of weights are coordinate (COO) and compressed sparse row (CSR), which are used in convolutional (CONV) and full-connected (FC) layers differently. Our accelerator consists of processing elements and Buffer Units to process the compressed weights. After receiving data from memory, each processing element works independently with partial weights of a layer. In Section 2, we elaborate the sparse neural networks (SpNNs), existing accelerators and the motivation for construct a SpNN accelerator. In Section 3, we introduce design methods of this work including the compression methods and accelerator architecture. The next section, we describe the performance evaluation method and demonstrate the experimental results. Conclusions are presented in Section 5. Mainly contributions of this paper are as follows:

1. We put forward a method to find the most suitable compressed format sparse weights, which is chosen from COO and CSR.f The compression method will achieve 50% and 90% storage reduction of weights in CONV and FC layer respectively.
2. We employ a PE based FPGA accelerator, which can process compressed weights directly both in CONV and FC layers. Also, our accelerator optimizes computations in a pipeline to output a result every cycle.

2 Background and Motivation

In this section, we first introduce SpNNs, a sparse method, and the state-of-art accelerators. Then, we describe the motivation for constructing a new accelerator for SpNNs.

2.1 Background

Sparse Neural Networks. To address the intensive computations and memory accesses of conventional NNs, numbers of training methods to alleviate the neurons and synapses with no accuracy loss.

A novel method proposed by Han et al. [10] utilizes pruning and retraining techniques to reduce most of redundant synapses with the same accuracy. The

Table 1. The accuracy, weights and sparsity ratio of sparse neural networks using [] method compared with the originals.

Neural Network	Error Ratio (original/pruned)	Weight (original/pruned)	Sparsity Ratio
Lenet-5 [14]	0.80%/0.77%	431K/36K	8.35%
AlexNet [4]	19.73%/19.67%	61M/6.7M	10.98%
VGG-16 [21]	11.32%/10.88%	138M/10.3M	7.46%

method prunes synapses whose values are lower than a customized threshold and retrains the SpNN by adjusting remaining weights. Table 1 presents the remaining synapses of SpNN over conventional NNs on average.

CPU and GPU NN accelerating frameworks. The state-of-the-art open-source neural network frameworks, such as tensorflow [1], caffe [11], torch [5], theano [2], have added functions to train and process SpNNs using CPU and GPU sparse libraries.

Specific SpNN accelerators. S. Han et al. proposed a compressed NN inference engine [8] in 2016, the engine uses compression methods from previous works [9] to reduce computations and storage of FC layers. Furthermore, this work exploits the dynamic sparsity of activation inputs and the load balance among multiple computing units.

Not long after, on the basis of [3,15], Shijin Zhang et al. [25] proposed a SpNN accelerator. This work comes up with a 1-D sparse array format for weights to jump zero values. As a result, the accelerator can process NNs both dense and sparse with 10x speedup and 3.4% energy over mainstream top-of-line GPUs.

2.2 Motivation

Due to the performance of conventional accelerators do not benefit much from synapse reduction, Fig. 1 [25] shows the speedup of CPUs, GPUs and DianNao processing SpNNs. The execution time evaluates the performance of accelerating platforms to process SPNNs (i.e., LeNet-5 [14], AlexNet, and VGG) with the sparsity ratio of lower 10%. The performance of CPUs, except for LeNet-5, is 210% is worse than the dense ones. Reasons for the phenomenon are irregularly memory access and non-computational overheads [20].

Hardware accelerators, i.e., DianNao, for dense NNs can not process sparse weights directly, which fills zero values into the pruned locations. In this way, both computations and memory storage do not reduce actually. With the purpose of overcoming the redundant operations by filling zeros and reducing memory storage and access, [25] and [8] first propose SpNN accelerators in 2016. However, [25], to easily implement and process NNs both dense and sparse, uses a simple 1-D array step index format to skip zero values, which does not reduce the storage of weights. [8] mainly focuses on the compression method of weights in FC layers and ignores the computation intensive CONVs. Hence, the survey above motivates us to find a method to take ful advantages of the sparsity of CONV and FC layers, and deploy the accelerator on FPGA platforms to evaluate the performance.

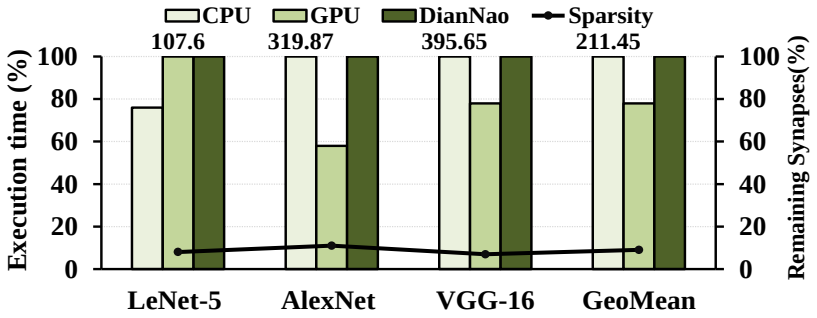

Fig. 1. The speedup of SpNNs over dense NNs on CPUs, GPUs and DianNao.

3 Design of Accelerator

In this section, we introduce our compression method and accelerator architecture in detail.

3.1 Overview Architecture

Fig. 2 illustrates the overview architecture of our accelerator, which consists of a control processor (CP), a compression unit (CU), a direct memory access (DMA) and several processing elements (PEs) containing buffer unit (BU) and function unit (FU). CP controls DMA to transfer compressed weights, which are converted from sparse matrices to COO or CSR formats by CU, from memory (i.e., DDR memory) to BU. Each PE processes core computations of each layer and FU employs multiplications and additions with input data and weights. Ultimately, accumulates results kept in BU will be transferred back to memory through DMA.

In our design, we use 16 bits fixed-point data type for weights instead of 32 bits float-point. Because of minimal accuracy loss, 16 bits fixed-point multipliers take significant lower hardware area and power shown in Table 2 [3].

Table 2. The error rate (for MNIST) and hardware cost between 16 bits fixed-point and 32 bits float-point multipliers.

Data Type	Error Ratio	Area (μm^2)	Power (μW)
16 bits fixed-point	3.4%	1309.32	576.90
32 bits float-point	3.1%	7997.76	14229.60

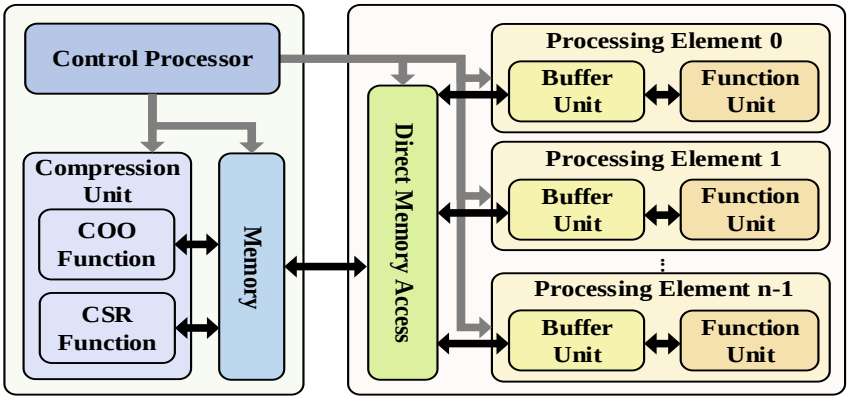

Fig. 2. The architecture of our accelerator.

3.2 Compression Method

CU employs compressed functions including COO and CSR to convert sparse weight matrices of each layer to compressed formats. The compressed formats store non-zero values, the corresponding row and column locations in original sparse matrices. COO and CSR format use 1-D data and col_id arrays to keep non-zero values and its column index. The difference between COO and CSR is values of row_id arrays. In CSR format, the 1-D row_id array stores the pointer, which is its location in the data array, of the first non-zero element in each row rather than row index, i.e., Fig. 3 (a) presents 4×4 matrix with 5 non-zero values, Fig. 3 (b) and (c) illustrate the data formats of COO and CSR.

Synapse weights in CONV and FC layers are both 32 bits float-point type, as we introduce before, we use the 16 bits fixed-point represents non-zero values. However, CONV and FC synapse weights have different organized forms and sparsity ratio, which are 4-D kernel arrays in CONV layers and 2-D weight arrays in FC layers. Therefore, we customize different data type and choose the better for storing.

Customized COO format. COO format stores sparse matrix using three 1-D arrays, which are data array, col_id array, and row_id array. To achieve higher use ratio of storage, we mingle row_id and col_id array which keeps row index in high bit and column index in low bit. CONV layers have large numbers of kernels with few columns, i.e., AlexNet-3 layer has 384×256 kernels with3×3 size. To make the row_id array have the ability to represent large-scale kernels, we assign 20 bits unsigned integer for a row index and 4 bits for a column index. In FC layers, large-scale weight matrices need for more bits, hence, we assign totally 32 bits for a row and column index.

Customized CSR format. Like COO, CSR format also use three 1-D arrays whose row_ptr array is different. The data type of row_ptr array depends on

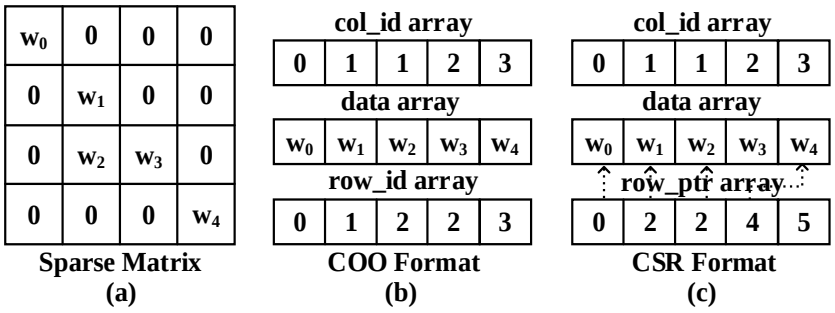

Fig. 3. COO and CSR compressed formats.

the sparsity of each layer, which means remaining number of weights, i.e., the remaining weights of AlexNet-3 and AlexNet-7 are 310K and 1.5G. For CONV layers, we allocate 4 bits for a column index and 32 bits for a row index. For FC layers, the bits of row and column index will be 16 and 32.

The storage usage of two formats in CONV and FC layers are shown in Table 3, which N, C, K represent the number of input feature maps, kernel channels and $K \times K$ kernel size in CONV layers, R_{out} and C_{in} represent the row and column size of weight matrices in FC layers, and α is the sparsity ratio of each layer. Through analyzing the storage utilization in different type layers, we conclude as follows:

- In CONV layers, when $\alpha K < \frac{4}{3}$, the storage space used by COO is lower than CSR, or vice verse.
- In FC layers, when $\alpha C_{in} < 2$, the storage space used by COO is lower than CSR, or vice verse.

Table 3. The storage utilization with COO and CSR of weights in CONV and FC layers.

Layer Type	COO Format (Bytes)	CSR Format (Bytes)
CONV	$6\alpha NKK$	$3\alpha NKK + 4NK$
FC	$6\alpha R_{out}C_{in}$	$4\alpha R_{out}C_{in} + 4R_{out}$

According to the conclusion above, the sparsity α of most CONV is 40% on average, and the kernel size K is lower than 3, thus we use COO. In FC layers, the sparsity α is around 10%, and the column size C_{in} is larger than 1000, as a result, we use CSR.

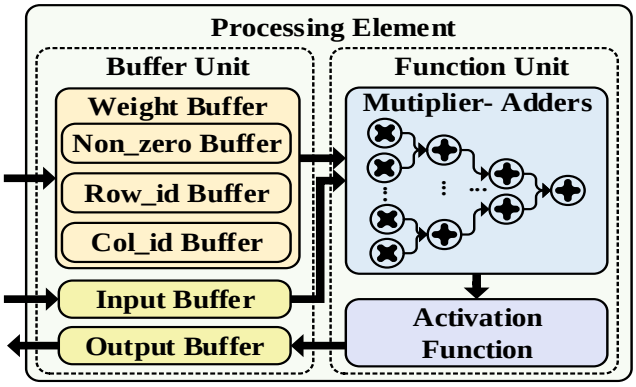

Fig. 4. Processing element architecture.

3.3 Processing Element

PE process computations of SpNNs, i.e, matrix-vector (MV) and vector-vector (VV) multiplications with compressed weights. Fig. 4 illustrates the architecture of PE, which contains BU and FU. FU takes compressed weights and inputs (i.e., feature maps in CONV layers and activations in FC layers) from the memory, produces outputs buffered in BU and transmits back to memory when BU is full.

BU. An weight buffer (WB), input buffer (IB) and output buffer (OB) compose BU, which keeps input data, compressed weights and output intermediate results. WB contains data buffer, row_id buffer and col_id buffer corresponding compressed formats. We did not keep total weights in BU, because of weight size of large-scale SpNNs, i.e., AlexNet and VGG-16, is still range from 6M to 10M. For a memory-limited hardware platform, we can deploy most optimal 2 KB WB in each according to [25]. Assuming that a WB can hold M weights to an FU, for N PEs, the total size of WB is $2 \times N$ KB buffer space. As IB and OB keep inputs and outputs of FU, their size depends on the multiplier number, which is $M \times M \times 32$ bits. Due to the limited buffers in each PE, memory access from PE to memory will occur when inner buffers are full. Hence, an optimal buffer size can hide memory access overhead without waiting for computation data for FU.

FU. FU processes multiplications, accumulations and activation operations for each layer. FU is composed of three components, that are $M \times M$ inputs multipliers, M inputs adders organized in an adder tree structure and an activation function (i.e., ReLU or sigmoid function). To achieve an efficient performance, we pipeline the function operations by five stages: data fetching, multiplying, adding, activation and outputting. Thus, FU calculates $M \times M$ multiplication and output an intermediate result ever cycle.

Algorithm 1 The convolutinal multiplication calculation model

```
1: function CONVOLUTIONAL MULTIPLICATION CORE(in, w, out)
2:     for each row ∈ R do
3:         for each col ∈ C do
4:             for each ch_in ∈ N do
5:                 for each ch_out ∈ M do
6:                     for each nz ∈ NZ do
7:                         i = row_id[nz]%K;
8:                         j = col_id[nz]
9:                         out[ch_out][row][col] += w[k] * in[ch_in][S*row + i][S*col + j];
10:                    end for
11:                end for
12:            end for
13:        end for
14:    end for
15: end function
```

Algorithm 2 A matrix-vector multiplication calculation model

```
function MATRIX-VECTOR MULTIPLICATION CORE(in, w, out)
    for each o ∈ M do
        start = row_ptr[o];
        end = row_ptr[o+1];
        for each i ∈ start to end do
            id = col_id[i];
            out[o] += w[i] * in[id];
        end for
    end for
end function
```

Both COO and CSR formats should store row and column index of non-zero elements into two 1-D arrays. Instead of obtaining the index by an iteration number directly, compressed formats incur index conversion and read overhead.Therefore, calculation cores of CONV and FC layers are different from the conventional patterns. Alg. 1 and 2 present the CONV multiplication calculation model, which use compression method. In CONV layers, we convert CONV and FC multiplications to MV multiplications, which take a non-zero weight vector and a feature matrix as inputs, then produce a new matrix. In FC layers, MV multiplications change to VV multiplications. Inputs are a weight vector and an activation vector, and the output is a vector for next iterations.

3.4 Control Processor

CP controls the execution using customized functions. Functions are used to run the compression stage, data organization, PE execution, etc. To simplify the usage of our accelerator, we provide modules in the operating system to control the execution stages.

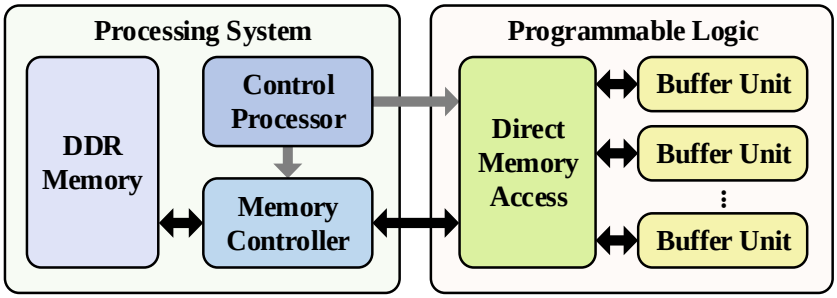

Fig. 5. Prototype of DMA data transfer.

3.5 Direct Memory Access

DMA is designed for transferring computation data from memory to PEs. Fig. 5 shows the data flow through DMA. The DMA and BU are implemented in the accelerator. The control bus (colored gray) allows the processor to communicate with DMA to setup, initiate and control data transfers. The data bus (colored black between memory and DMA) provides the DMA access to memory and BU.

4 Performance Evaluation

4.1 Experimental Methodology

Benchmarks. We use two commonly used neural networks LetNet-5 and AlexNet, characteristics in detail of these NNs are shown in Table. 4(a) and 4(b), including the number of synapse weights and sparsity of main layers.

Table 4. Benchmarks with reaming weights in each layer.

(a) LeNet-5

Layer	Weight	Sparsity
CONV-1	0.5K	66%
CONV-2	25K	12%
FC-3	400K	8%
FC-4	5K	19%

(b) AlexNet

Layer	Weight	Sparsity
CONV-1	35K	84%
CONV-2	307K	38%
CONV-3	885K	35%
CONV-4	664K	37%
CONV-5	442K	37%
FC-6	38M	9%
FC-7	17M	9%
FC-8	4M	26%

Measurements. We implement our accelerator using Vivado High-Level Synthesis to simulate RTL behaviors on Xilinx FPGA xc7z020 with the frequency of 100MHz.

Baselines. We compare our accelerator with a previous FPGA accelerator [24], which process dense NNs. We employ the method proposed to optimize computations in order to get a fair comparison. The accelerator we implement contains accelerate cores for each NN layer, on-chip buffers with a size of kernel size or tile size in CONV or FC layers.

4.2 Experimental Results

COO and CSR compressed formats can reduce almost 50% and 90% storage space of CONV and FC layers in SpNNs, as shown in Fig. 6. Fig. 7 illustrates the comparison of our design with a conventional FPGA accelerator we employed. We use execution cycles to evaluate the speedup and normalized the are results by the conventional FPGA accelerator. Processing each layer of our benchmarks, the results demonstrate that our accelerator achieves 2x speedup in CONV layers and 4x speedup in FC layers over an optimal conventional FPGA accelerator for NNs. This work does not achieve the best performance in theory. The reason is two aspects. One is COO and CSR incur overhead for getting the index of non-zero values. Another is placeholders of a row in CSR row_ptr array of CSR, where all elements are zero, involve in additional operations.

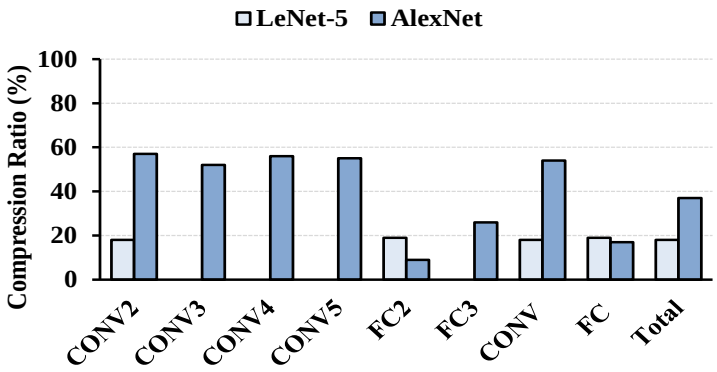

Fig. 6. Compression results of main layers in LeNet-5 and AlexNet.

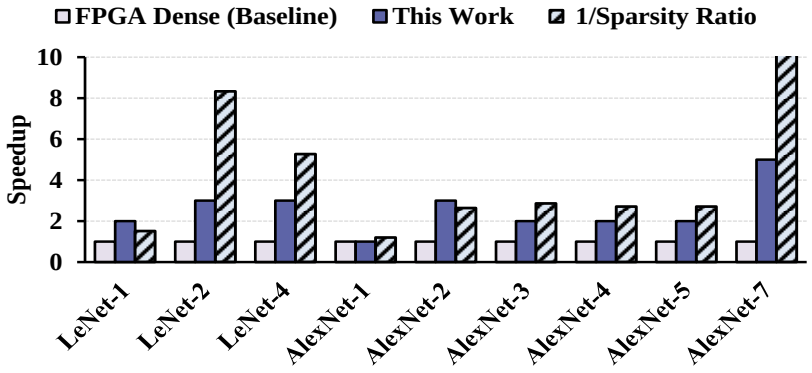

Fig. 7. Speedup of main layers in LeNet-5 and AlexNet.

5 Conclusions

In this work, we propose an accelerating method for sparse neural networks. We analyze two widely used compressed formats of sparse weights in each layer and provide a method to a choose suitable one. Otherwise, we implement an optimal FPGA accelerator to process compressed weights directly. Experiment results by simulation present that this work achieves a higher performance than conventional accelerators. However, our accelerating method can not achieve the theoretical performance, due to compressed formats and optimizations.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)
2. Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., Bengio, Y.: Theano: A cpu and gpu math compiler in python. In: EuroSciPy. pp. 1–7 (2010)
3. Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., Temam, O.: Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In: ACM Sigplan Notices. vol. 49, pp. 269–284 (2014)
4. Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., Andrew, N.: Deep learning with cots hpc systems. In: ICML. pp. 1337–1345 (2013)
5. Collobert, R., Bengio, S., Mariéthoz, J.: Torch: a modular machine learning software library. Tech. rep. (2002)
6. Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L., Luo, T., Feng, X., Chen, Y., Temam, O.: Shidiannao: Shifting vision processing closer to the sensor. In: SCAN. vol. 43, pp. 92–104 (2015)

7. Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B.C., Richardson, S., Kozyrakis, C., Horowitz, M.: Understanding sources of inefficiency in general-purpose chips. In: SCAN. vol. 38, pp. 37–47 (2010)
8. Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M.A., Dally, W.J.: Eie: efficient inference engine on compressed deep neural network. In: ISCA. pp. 243–254 (2016)
9. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149 (2015)
10. Han, S., Pool, J., Tran, J., Dally, W.: Learning both weights and connections for efficient neural network. In: NIPS. pp. 1135–1143 (2016)
11. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: ICM. pp. 675–678 (2014)
12. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: NIPS. pp. 1097–1105 (2012)
13. Le, Q.V.: Building high-level features using large scale unsupervised learning. In: ICASSP. pp. 8595–8598 (2013)
14. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE 86(11), 2278–2324
15. Luo, T., Liu, S., Li, L., Wang, Y., Zhang, S., Chen, T., Xu, Z., Temam, O., Chen, Y.: Dadiannao: A neural network supercomputer. TC 66(1), 73–88 (2017)
16. Mikolov, T., Karafiát, M., Burget, L., Cernockỳ, J., Khudanpur, S.: Recurrent neural network based language model. In: Interspeech. vol. 2, p. 3 (2010)
17. Olshausen, B.A., Field, D.J.: Emergence of simple-cell receptive field properties by learning a sparse code for natural images. Nature 381(6583), 607 (1996)
18. Poultney, C., Chopra, S., Cun, Y.L., et al.: Efficient learning of sparse representations with an energy-based model. In: ANIPS. pp. 1137–1144 (2007)
19. Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., et al.: Going deeper with embedded fpga platform for convolutional neural network. In: FPGA. pp. 26–35 (2016)
20. Rafique, A., Constantinides, G.A., Kapre, N.: Communication optimization of iterative sparse matrix-vector multiply on gpus and fpgas. TPDS 26(1), 24–34 (2015)
21. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2015)
22. Temam, O.: A defect-tolerant accelerator for emerging high-performance applications. In: ISCA'2012. pp. 356–367
23. Wang, F.Y., Zhang, J.J., Zheng, X., Wang, X., Yuan, Y., Dai, X., Zhang, J., Yang, L.: Where does alphago go: from church-turing thesis to alphago thesis and beyond. JAS 3(2), 113–120 (2016)
24. Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J.: Optimizing fpga-based accelerator design for deep convolutional neural networks. In: FPGA. pp. 161–170 (2015)
25. Zhang, S., Du, Z., Zhang, L., Lan, H., Liu, S., Li, L., Guo, Q., Chen, T., Chen, Y.: Cambricon-x: An accelerator for sparse neural networks. In: MICRO. pp. 1–12 (2016)