A Pipeline Energy-efficient Accelerator for Convolutional Neural Networks

Fan Sun[1], Chao Wang[2], Lei Gong[3], Yiwei Zhang[4], Xi Li[5] and Xuehai Zhou[6]

University of Science and Technology of China, Hefei, Anhui, China
{sunfan[1], leigong0203[3], zhyiwei[4]}@mail.ustc.edu.cn
{cswang[2], llxx[5], xhzhou[6]}@ustc.edu.cn

Abstract. Convolutional neural networks (CNNs) have been widely applied for image recognition, face detection, and video analysis because of their ability to achieve accuracy close to or even better than human level perception. However, for large-scale CNNs, the computation-intensive convolution layers and memory-intensive convolution layers have brought many challenges to the implementation of CNN. In order to overcome this problem, this work proposes a pipeline energy-efficient accelerator for convolutional neural networks, and different methods are applied to optimize the convolution layers and convolution layers. For the convolution layer, the accelerator first rearranges the input features into matrix on-the-fly when storing them to the FPGA on-chip buffers, thus the computation of convolution layer can be completed through matrix multiplication. For the fully connected layer, the batch-based method is used to reduce the required memory bandwidth, which also can be completed through matrix multiplication. Then a two-layer pipeline computation method for matrix multiplication is proposed to increase the throughput and also reduce the buffer requirement. The experiment results show that the proposed accelerator can reduce the energy consumptions of CPU and GPU by 333.48x and 17.18x respectively, which demonstrates that the proposed accelerator outperforms CPU and GPU at the aspect of energy efficiency. The proposed accelerator runs under a frequency of 100 MHz which can achieve the throughput of 49.31 GFLOPS. This is done using only 198 DSP48 modules, which shows significant resource utilization improvement compared to the state-of-the-art.

1 Introduction

Convolutional neural networks (CNNs), a category of feed-forward artificial neural networks, have been widely applied in various applications, such as character recognition image classification [7] [15] [18], face detection [17], and nature language analysis [3], because of their ability to achieve accuracy close to or even better than human level perception.

In practice, GPUs are widely used to accelerate the training and classification tasks of CNNs. However, their energy-efficiency is lower [10]. Instead of GPUs, various CNN accelerators have been proposed recently based on FPGAs and

ASICs. Among these platforms, FPGA-based accelerators have become increasingly popular by the virtue of their high reconfigurability, fast turn-around time and better energy efficiency [12].

In [20], a roofline model is proposed, where they evaluate all the combinations of parameters for each convolution layer to find the optimum solution based on their model. In [14], a 3D computation tile for the convolution layer is proposed, and to achieve high performance, they apply an input reuse network composed of a 2D array of registers. Both of them only implement convolution layers, which is part of the whole CNN model. The work in [13] transforms a regular matrix multiplication into a convolution and implements the accelerator for both convolution and fully connected layers. While this study makes a good start on accelerating the entire CNN on an FPGA, the straightforward transformation does not consider potential optimization. In this work, we present a pipeline energy-efficient accelerator for an entire CNN model consisting of convolution layer and fully connected layers. The key contributions of this work are summarized as follows:

1. We apply a method to convert convolution computation to matrix multiplication and propose a scalable architecture to perform the computation in parallelism for convolution layers, which are computation-intensive.

2. We use a batch-based computing method to reduce the required memory bandwidth for weights in fully connected layers, which are memory-intensive.

3. As both the convolution layers and fully connected layers are converted to matrix multiplication, we propose an accelerating method for the two-layer pipeline computation.

4. As a case study, we implement a practical accelerator and make evaluations from computing resource utilization, performance, and power, which demonstrates energy efficiency and high hardware efficiency of the proposed accelerator.

The rest of this paper is organized as follows. Section 2 provided a brief background to the CNN operations. Section 3 presents the methods for optimizing convolution layers and fully connected layers. Section 4 presents the experiment and evaluations, and the paper is concluded in Section 5.

2 Background

Convolutional neural network (CNN) belongs to the classical supervised learning algorithm, it adopts a backward path for training and a feedforward path for prediction. In practice, CNN model is trained off-line and used to execute time-sensitive jobs. So the speed of feedforward process is what matters [20]. In this work, we focus on speeding up the feedforward process.

A typical CNN is composed of multiple computation layers, which can be classified into two categories: a feature extractor and a classifier. The feature extractor, consisting of several convolution layers, optional pooling layers, and optional activation layers, is used to filter input images into feature maps that can be used to replace the original input images. Then the output of feature

extractor is fed into the classifier, which may consist of several fully connected layers. The classifier is used to determine which categories the input images belong to.

2.1 Convolution Layer

A convolution operation is composed of a set of 3D arrays called kernels where each kernel performs the multiply-and-accumulate (MAC) operation and a sliding mechanism to get one output feature map as shown in Equation (1).

$$out(f_o, x, y) = \sum_{f_i=0}^{N_{if}} \sum_{k_x=0}^{K} \sum_{k_x=0}^{K} wt(f_o, f_i, k_x, k_y) * in(f_i, S * x + k_x, S * y + k_y) \quad (1)$$

where out(f_o,x,y) represents the neuron at location (x,y) in the feature maps f_o, in(f_i,x,y) represents the neuron at location (x,y) in the feature maps f_i, wt(f_o,f_i,k_x,k_y) represents the weight at position (k_x,k_y) that performs the convolution with input feature map f_i to get the output feature map f_o. The total number of MAC operations in the convolution layer is N_{of}*X*Y*N_{if}*K*K.

2.2 Pooling Layer

Pooling layer or sub-sampling layer is used to reduce the dimensions of the feature maps. The benefits are that it not only preserves the important information of lower-level features but also helps to abstract the high-level features without redundancy. As shown in Equation (2), pooling computes the maximum or average of the neighboring K*K neurons in the same feature map, where $0 \leq (k_x, k_y) < K$.

$$out(f_o, x, y) = max/average(in(f_o, x + k_x, y + k_y)) \quad (2)$$

2.3 Activation Function

Traditional neural networks use tanh or sigmoid as the activation function, However, in CNN model, Rectified Linear Unit (ReLU) is becoming more and more popular as it converges faster in training compared to sigmoid and tanh. As shown in Equation (3), ReLU has less computational complexity compared to tanh and sigmoid, which also helps the design of the accelerator.

$$y = max(x, 0) \quad (3)$$

2.4 Fully Connected Layer

Fully connected layer or inner product layer is the classification where all the input feature maps are connected to all the output feature maps through the synaptic weights. As shown in Equation (4), each output neuron is the summation of all the input neurons multiplied by the corresponding synaptic weight.

Generally, the fully connected layer is followed by ReLU if it is not the last layer, and the last layer is followed by a Softmax function that converts the data to probability in the range (0,1), based on which to determine which categories the input images belong to and give the accuracy of the CNN model.

$$out(f_o) = \sum_{f_i=0}^{N_{if}} wt(f_o, f_i) * in(f_i) \tag{4}$$

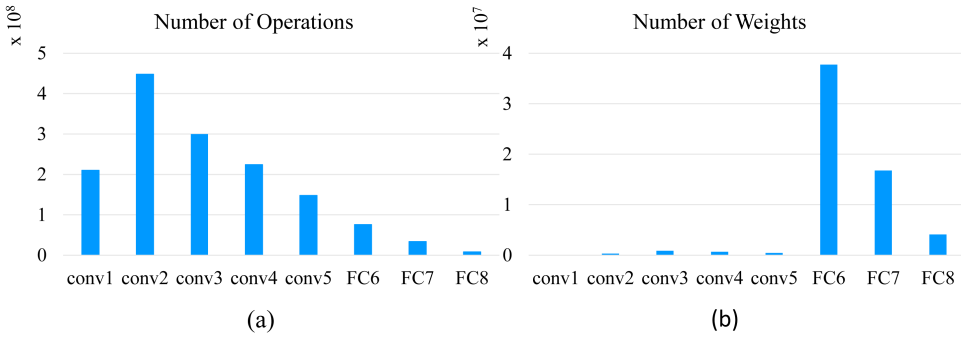

Fig. 1. (a) Number of operations for different layers in AlexNet. (b) Number of weights for different layers in AlexNet.[9]

CNNs are computation-intensive and memory-intensive. To illustrate this problem, we use AlexNet [7] model as an example. The AlexNet model consists of 5 convolution layers and 3 fully connected layers, where each layer has a different number of features and dimensions. Fig.1 has shown the number of operations and number of weights for each layer in the AlexNet model. We can see that convolution operation accounts for 92% of the total operations and weights in fully connected layer accounts for 96% of the total weights, which reflects the computation-intensive feature of convolution layers and the memory-intensive feature of the fully connected layer.

On this account, different methods are used to implement convolution layers and fully connected layers. For convolution layers, we focus on computing parallel and apply a method to convert convolution computation to matrix multiplication, and propose a scalable architecture to perform the computation in parallelism. For fully connected layers, we focus on reducing the required memory bandwidth and apply a batch-based computing method. Moreover, we proposed an accelerating method for a two-layer pipeline computation.

The performance limitation due to the external memory bandwidth can be alleviated by using reduced precision model weights [16]. Using reduced precision data allows us to buffer more data on-chip, which alleviates the influence caused by external memory bandwidth and it consumes less FPGA logic resources compared to float point precision, thus improving the hardware efficiency.

We have conducted the precision experiment using Caffe tool framework [1]. Firstly, we obtain the pre-trained model from Caffe, then we reduce the data precision in the convolution layer and fully connected layer respectively, and test the final accuracy. Finally, we compare this accuracy with the accuracy obtained by using 32-bit floating point precision. We found that it is enough to use 16-bit fixed point in the data of convolution layer and fully connected layer, which degrades the accuracy by only <1% compared to 32-bit floating point precision during the prediction phase. In the design of accelerator system, we use 16-bit fixed point to representation data instead of the 32-bit floating point.

3 Accelerator Design and Optimizing

3.1 Optimizing for Convolution Layer

A convolution operation is composed of a set of 3D arrays called kernels where each kernel performs the MAC operation and a sliding mechanism to get one output feature map. To achieve high performance and also make the design suitable to other CNN model, a scalable convolution core is needed.

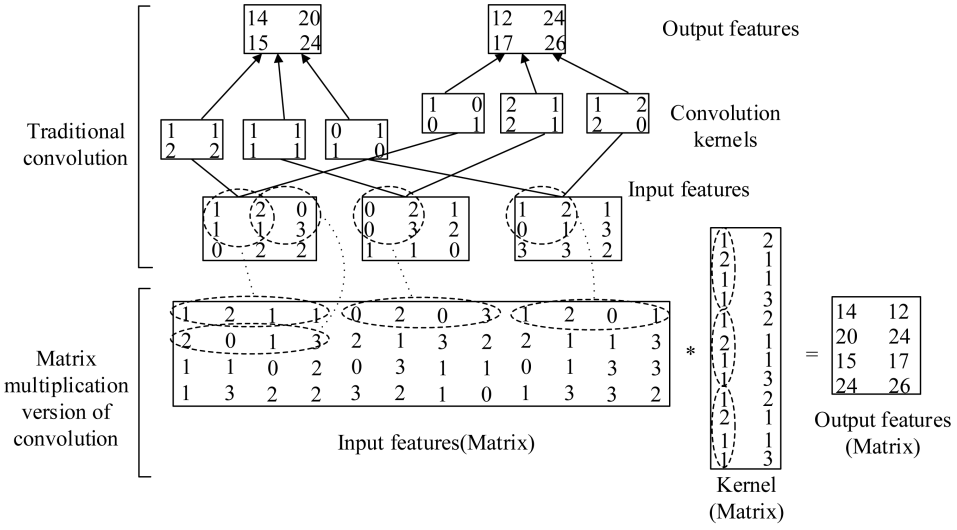

Fig. 2. Matrix multiplication version of convolution.

We implemented the scalable convolution core by mapping the 3D convolutions to matrix multiplication operations similar to that in [2] [16] by flattening and rearranging the input features. As an example, Fig.2 illustrates how traditional convolution is converted to matrix multiplication. The 3 input features with dimensions 3*3 are rearranged to a matrix with dimensions

Algorithm 1 Pseudo Code for the Matrix Multiplication using tile technique

```
1: N_i:the number of input neurons
2: N_o:the number of output neurons
3: TileSize:the tile size of the input data
4: batchsize:the batch size of the input data
5: for n = 0; n < batchsize; n++ do
6:     for k = 0; k < N_i; k+=TileSize do
7:         for j = 0; j < N_o; j++ do
8:             #pragma HLS pipeline
9:             for i = k; i < k+TileSize && i< N_i; i++ do
10:                #pragma HLS UNROLL
11:                out[n][j] += in[n]][i] * wt[i][j];
12:            end for
13:        end for
14:    end for
15: end for
```

of (2*2)*(3*2*2). The input features from the first convolution windows of 2*2 are flattened and arranged horizontally as shown in Fig.2. The entire rearranged input features matrix can be generated by sliding the 2*2 convolution window across all 3 input features. The 3 convolution kernels with dimensions 2*2 are rearranged to a matrix with dimensions of (3*2*2)*(2). The convolution kernels from the first convolution windows of 2*2 are flattened and arranged vertically as shown in Fig.2. The entire rearranged kernel matrix can be generated by sliding the 2*2 convolution window across all 3 convolution kernels. Then the traditional convolution is transformed to matrix multiplication. Note that the rearrangement for the input features is on-the-fly when we store them to the FPGA on-chip buffers, by doing so, we can reduce the external memory requirement because we do not need to store the entire rearranged input features matrix.

Algorithm 1 illustrates the Pseudo Code for the Matrix Multiplication using tile technique. The input data are partitioned into tiles and perform the multiplication with the corresponding weights, so the hardware cost of the accelerator only depends on the tile size, which can save a large number of hardware resources. The tile technique allows us to solve the large computation task with limited hardware resource [19].

Fig.3 describes the architecture for the computation which contains an embedded processor, a DDR3 memory controller, a DMA module, and the computation core. The embedded processor is responsible for providing the programming interface to the users. In particular, it also transfers the input data and the weight data to internal BRAM, activates the accelerator and returns the classification results to the user after execution. The computation core is integrated as a standalone unit which is flexible and adaptive to accommodate different applications with configurations.

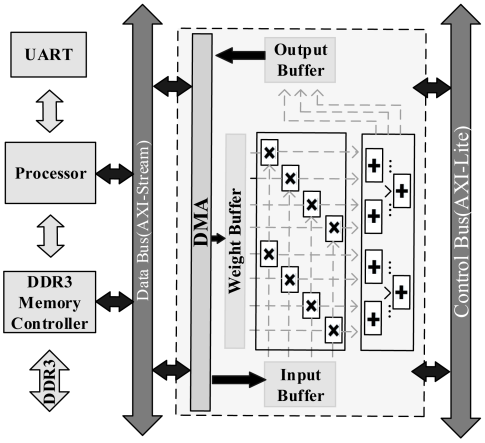

Fig. 3. Accelerator architecture for matrix multiplication.

The total number of multipliers is determined by the tile size. To achieve the goal of each multiplier computes one input data every cycle, we use the loop pipelining technique and loop unrolling technical. We also achieve the double buffering skill to pre-fetch the data for each multiplier so that the computation time and data transfer time can be overlapped.

3.2 Optimizing for fully Connected Layer

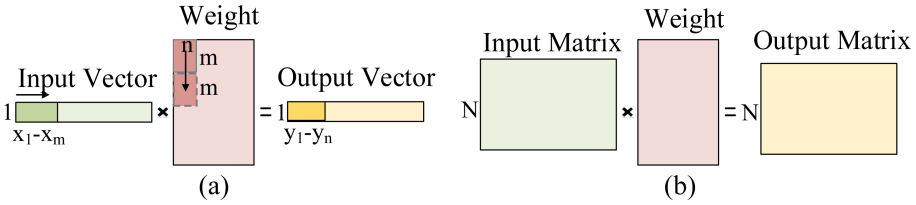

Fig. 4. (a) Original computation method for the fully connected layer. (b) Batch-based computation method for fully connected layer.

The fully connected layer can be regarded as matrix multiplication, and we can use the tile technical to complete the computation as shown in Fig.4 (a). The tile size is x_m, and we first perform the computation of input x_1 to x_m (1*m) with the m*n weights to get the temporary results of y_1 to y_n(1*n)), then we take x_{m+1} to x_{2m} as the input and perform the computation with another m*n weights to update the temporary results of y_1 to y_n. After all the input data and

the first n column in the weight matrix has been traversed, the final results of y_1 to y_n are generated. The other results can be computed in the same way.

Fig.1 has shown that fully connected layer contribute most of the data access, therefore, we use the batch-based method to optimize the memory access. As shown in Fig.4 (b), the input matrix is composed of N input vector, N can be supposed to the batch size. After using the batch-based method, the operation amount is increased by N times without increasing the number of data access for weights, thus reducing the required memory bandwidth. Because it takes N cycles to complete the N*m*n multiplication, during which the accelerator should prepare the data for the next round computation, so N should be no less than the clock cycle used to read the m*n weights.

Pooling layers and activation function are incorporated at the output of convolution layers and fully connected layers with a flag to enable or disable it.

3.3 Two-layer Pipeline Computation Method

Now both the convolution layer and fully connected layer are converted to matrix multiplication, and both of them apply the batch-based method. However, the introduction of batch-based method expands the on-chip buffer N times. To address the problem and also improve the performance, the computation pattern of the matrix multiplication is reordered to reduce the buffer requirement.

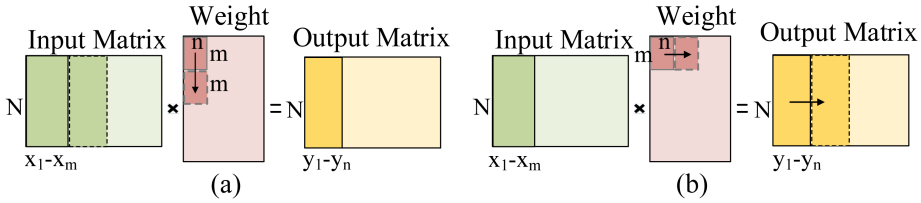

Fig. 5. (a) First computing pattern for matrix multiplication. (b) Second computing pattern for matrix multiplication. The two patterns are used alternately.

In Fig.5 (a), the computation pattern is similar to that in Fig.4 (a). In Fig.5 (b), the computation pattern is different. In this pattern, the temporary results of all the output matrix are generated by only using x_1 to x_m as the input. Then, the temporary results of all the output matrix are updated by taking x_{m+1} to x_{2m} as the input. The window in the weight matrix with size m*n shifts horizontally while the window in Fig.5 (a) shifts vertically.

In this work, the two pattern are applied alternately to implement the matrix multiplication. For the first layer of matrix multiplication, the vertically shifting pattern in Fig.5 (a) is used to produce y_1 to y_n. Thus the second layer of matrix multiplication can start to work with y_1 to y_n as the input by using horizontally shifting pattern in Fig.5 (b). Because the second layer starts before the whole

input matrix is produced from the first layer, so we only need a few buffers to store the N*n results for the first layer of matrix multiplication computation. For the subsequent third layers, the vertically shifting pattern is used similarly to the first layer, and for the fourth layers, the horizontally shifting pattern is used similarly to the second layer. By this way, the required buffers can be reduced significantly and the two-layer pipeline can move on smoothly.

4 Experiment and Evaluations

4.1 Experiment Methodology

1) Platform: We evaluate the performance of the accelerator with the other implementations in CPU and GPU.

Proposed accelerator: We implement the proposed accelerator on Xilinx ZedBoard xc7z020clg484-1 FPGA similar to [21], which consists of two parts: processing system (PS) and programmable logic (PL). PS integrates two ARM Cortex-A9 cores and PL is full of reconfigurable logic. In our experiment, the frequency of PS is 667MHz, and the frequency of PL is 100MHz. Vivado 2016.2 was used as synthesis and physical design tool.

GPU: GPUs have been widely applied to the application of neural network. Therefore, we realize a GPU solution using the Caffe model. The GPU card information is listed in Table 1. We use the function cuEventElapsedTime() to measure computing time and nvprof command to measure the computing power.

CPU: We also accomplish a software program running on CPU, the CPU information is listed in Table 1. For measurements of computing time and power consumption, we use the function gettimeofday() and the PAPI tool [4] respectively.

Table 1. Evaluated CPU and GPU

Platform	Cores	Clock frep(GHz)	Memory(GB)	TDP(W)	Tech(nm)	Cost($)
i7-4790K CPU	4	4	8	88	28	366
Tesla K40 GPU	2880	0.875	12	235	28	2599

2) Benchmarks: We select three datasets MNIST [8], CalTech 101 [6] and CIFAR-10 [5] as the benchmark. MNIST is a handwritten digits database, with 60,000 training examples and 10,000 test examples. CalTech 101 is a data set including high-quality polygon outlines of the primary objects. The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 classes.

4.2 Power and Energy

We first trained the convolution neural networks with the three training datasets respectively. Then we compute the inference process with the test datasets on CPU, GPU, and proposed accelerator.

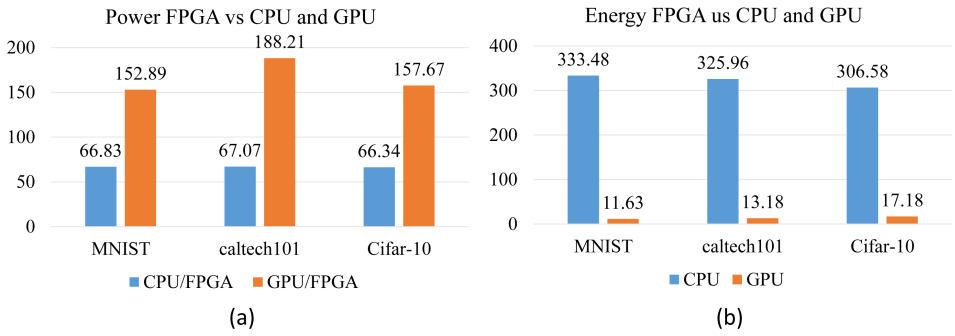

Fig. 6. (a) Power ratios vs CPU and GPU. (b)Energy ratios vs CPU and GPU.

Fig.6 (a) reports the power ratios of our accelerator over CPU and GPU. Compared to CPU, our accelerator can achieve a power reduction of 66.74 on average. Compared to GPU, our accelerator can achieve a power reduction of 166.26 on average.

Fig.6 (b) reports the energy ratios of our accelerator over CPU and GPU. Compared to CPU, our accelerator can achieve an energy reduction of 322 on average. Compared to GPU, our accelerator can achieve an energy reduction of 14 on average. Results show that our accelerator outperforms CPU and GPU at the aspect of energy efficiency.

4.3 Performance

Table 2. Performance comparison with previous works

Architecture	FPGA2015 [20]	DATE2016 [14]	TCSVT2017 [11]	Our Work
precision	32bit float	32bit fixed point	Q15	16bit fixed point
Frequency	100MHz	160MHz	150MHz	100MHz
FPGA Chip	VX485T	VX485T	XC7Z045	XC7Z020
CNN Size	1.33GFLOPS	1.33GFLOPS	1.33GFLOPS	0.82GFLOPS
Performance	61.62GFLOPS	80.78GFLOPS	38.4GFLOPS	49.31GFLOPS
DSP Util	2240	2688	391	198
Perf/DSP	0.027	0.03	0.098	0.249

Table 2 has listed the performance comparison with previous works, we can see that the proposed accelerator can achieve the throughput of 49.31 GFLOPS on the given FPGA board. While there are other architectures achieve better performance, but they consume more DSP blocks. In order to have a fair comparison between the proposed accelerator and existing ones, we use Perf/DSP as the mean of each DSP block in the total performance calculation. We can

see that the proposed accelerator can achieve higher resource utilization than previous works.

5 Conclusions

In this work, a pipeline energy-efficient accelerator for convolutional neural networks is proposed. For the convolution layer computation, the accelerator first rearranges the input features into matrix on-the-fly when storing them to the FPGA on-chip buffers, thus the computation of convolution layer can be completed through matrix multiplication. For the fully connected layer computation, the batch-based method is used to reduce the required memory bandwidth, which also can be completed through matrix multiplication. Then a two-layer pipeline computation method for matrix multiplication is proposed to increase the throughput. The experiment results demonstrate that the proposed accelerator outperforms CPU and GPU at the aspect of energy efficiency. The proposed accelerator runs under a frequency of 100 MHz which can achieve the throughput of 49.31 GFLOPS. This is done using only 198 DSP48 modules, which shows significant resource utilization improvement compared to the state-of-the-art.

In the future work, we will consider pruning the redundant connections and neurons to alleviate the computation and memory pressure and design an accelerator to process the pruned convolutional neural networks efficiently.

References

1. Caffe, J.Y.: An open source convolutional architecture for fast feature embedding. In: ACM International Conference on Multimedia. ACM (2014)
2. Chellapilla, K., Puri, S., Simard, P.: High performance convolutional neural networks for document processing. In: Tenth International Workshop on Frontiers in Handwriting Recognition. Suvisoft (2006)
3. Collobert, R., Weston, J.: A unified architecture for natural language processing: Deep neural networks with multitask learning. In: Proceedings of the 25th international conference on Machine learning. pp. 160–167. ACM (2008)
4. Dongarra, J., Moore, S., Mucci, P., Seymour, K., Terpstra, D., You, H.: Performance application programming interface (papi) (2007)
5. Fei-Fei, L., Fergus, R., Perona, P.: Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. Computer vision and Image understanding 106(1), 59–70 (2007)
6. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images (2009)
7. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. pp. 1097–1105 (2012)
8. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE 86(11), 2278–2324 (1998)
9. Li, H., Fan, X., Jiao, L., Cao, W., Zhou, X., Wang, L.: A high performance fpga-based accelerator for large-scale convolutional neural networks. In: Field Programmable Logic and Applications (FPL), 2016 26th International Conference on. pp. 1–9. IEEE (2016)

10. Ma, Y., Suda, N., Cao, Y., Seo, J.s., Vrudhula, S.: Scalable and modularized rtl compilation of convolutional neural networks onto fpga. In: Field Programmable Logic and Applications (FPL), 2016 26th International Conference on. pp. 1–8. IEEE (2016)
11. Moini, S., Alizadeh, B., Emad, M., Ebrahimpour, R.: A resource-limited hardware accelerator for convolutional neural networks in embedded vision applications. IEEE Transactions on Circuits and Systems II: Express Briefs (2017)
12. Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., et al.: A reconfigurable fabric for accelerating large-scale datacenter services. In: Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on. pp. 13–24. IEEE (2014)
13. 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: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. pp. 26–35. ACM (2016)
14. Rahman, A., Lee, J., Choi, K.: Efficient fpga acceleration of convolutional neural networks using logical-3d compute array. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016. pp. 1393–1398. IEEE (2016)
15. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al.: Imagenet large scale visual recognition challenge. International Journal of Computer Vision 115(3), 211–252 (2015)
16. Suda, N., Chandra, V., Dasika, G., Mohanty, A., Ma, Y., Vrudhula, S., Seo, J.s., Cao, Y.: Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. pp. 16–25. ACM (2016)
17. Sun, Y., Chen, Y., Wang, X., Tang, X.: Deep learning face representation by joint identification-verification. In: Advances in neural information processing systems. pp. 1988–1996 (2014)
18. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 1–9 (2015)
19. Wang, C., Gong, L., Yu, Q., Li, X., Xie, Y., Zhou, X.: Dlau: A scalable deep learning accelerator unit on fpga. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 36(3), 513–517 (2017)
20. Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J.: Optimizing fpga-based accelerator design for deep convolutional neural networks. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. pp. 161–170. ACM (2015)
21. Zhao, Y., Yu, Q., Zhou, X., Zhou, X., Li, X., Wang, C.: Pie: A pipeline energy-efficient accelerator for inference process in deep neural networks. In: Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on. pp. 1067–1074. IEEE (2016)