A Scalable FPGA Accelerator for Convolutional Neural Networks

Ke Xu[1,2], Xiaoyun Wang[1,2], Shihang Fu[1,2], and Dong Wang[1,2](✉)

[1] Institute of Information Science, Beijing Jiaotong University, Beijing 100044, China
[2] Beijing Key Laboratory of Advanced Information Science and Network Technology, Beijing 100044, China
{17112071,16120304,17125155,wangdong}@bjtu.edu.cn

Abstract. Convolution Neural Networks (CNN) have achieved undisputed success in many practical applications, such as image classification, face detection, and speech recognition. As we all know, FPGA-based CNN prediction is more efficient than GPU-based schemes, especially in terms of power consumption. In addition, OpenCL-based high-level synthesis tools in FPGA is widely utilized due to the fast verification and implementation flows. In this paper, we propose an FPGA accelerator with a scalable architecture of deeply pipelined OpenCL kernels. The design is verified by implementing three representative large-scale CNNs, AlexNet, VGG-16 and ResNet-50 on Altera OpenCL DE5-Net FPGA board. Our design has achieved a peak performance of 141 GOPS for convolution operation, and 103 GOPS for the entire VGG-16 network that performs ImageNet classification on DE5-Net board.

Keywords: FPGA · OpenCL · Convolution Neural Networks Optimization

1 Introduction

Convolutional Neural Network (CNN) is a widely-regarded algorithm in the field of artificial intelligence. It has achieved great success in image classification [1], object detection [2], and speech recognition [3]. In the past decade, CNN has significantly improved the accuracy and performance of image classification. This is mainly due to the continuous improvement of data sets and the successive enhancement of the neural network structure. Being compute-intensive, GPUs are now widely used to train CNN. However, the GPUs with high power dissipations at the deployment level of the CNNs is not the best choice. FPGA based hardware accelerators with provide massive processing elements, reconfigurable interconnections and lower power dissipation are naturally suitable to implement neural network circuits.

The traditional FPGA development method uses hardware description language (HDL). The work of [7,8] propose efficient CNN accelerators on embedded FPGA platforms. However, traditional register-transfer-level (RTL) design

© Springer Nature Singapore Pte Ltd. 2018
C. Li and J. Wu (Eds.): ACA 2018, CCIS 908, pp. 3–14, 2018.
https://doi.org/10.1007/978-981-13-2423-9_1

flows takes a lot of time to simulate and compile before actually running hardware accelerators. With the development of FPGA high-level synthesis tool (HLS), high-level programming language (C/C++) is used to replace low-level HDL, which improves the speed of FPGA implementation and verification flows. Greatly reducing the development cycle, to design FPGA has brought great convenience. In recent yeas, the use of HLS to design CNN architecture has continued to emerge. The work of [9] using the Vivado-HLS tool on a Xilinx VC707 FPGA board. However, only convolution layers are implemented on AlexNet [1]. In [10], author present a systematic methodology for maximizing the throughput of an FPGA-based accelerator. In this work, an entire CNN model is proposed consisting of all CNN layers: convolution, normalization, pooling and classification layers. The scalable of accelerator architecture only use like AlexNet and VGG [4]. The feedforward neural networks with shortcut connections like ResNet [5] dose not work. The main contribution of this work are:

(1) Propose a FPGA accelerator with a scalable architecture of deeply pipelined OpenCL kernels;
(2) The design is verified by implementing three representative large-scale CNNs, AlexNet, VGG-16 and ResNet-50;
(3) The design space of the proposed architecture was fully explored on Stratix-V A7 FPGA.

2 Background

2.1 Classic Convolution Neural Network

AlexNet. AlexNet was able to achieve record breaking object recognition results on the ImageNet challenge in 2012. It consisted of eight layers in total, 5 convolutional and 3 fully connected, as depicted in Fig. 1. The 3-dimensional (3-D) convolution operation can be defined by

$$D_o(f_o, y, x) = \sum_{f_i=1}^{C_l} \sum_{k_y=0}^{K-1} \sum_{k_x=0}^{K-1} W_l(f_o, f_i, k_y, k_x) \cdot D_i(f_i, y + k_y, x + k_x) \quad (1)$$

where $D_i(f_i, y, x)$ and $D_o(f_o, y, x)$ denote the neurons at position (x, y) in the input feature map f_i and output feature map f_o, respectively. $W_l(f_o, f_i, y, x)$ represents the corresponding weights in the l-th layer that gets convolved with f_i. The size of the convolution filters is $K \times K$, while the total number of input feature maps is C_l. In addition to this, AlexNet considered the use of the ReLU nonlinearity instead of the saturating nonlinearites, such as sigmoids; Using dropout in training and Local Response Normalization (LRN) to reduce the problem of overfitting.

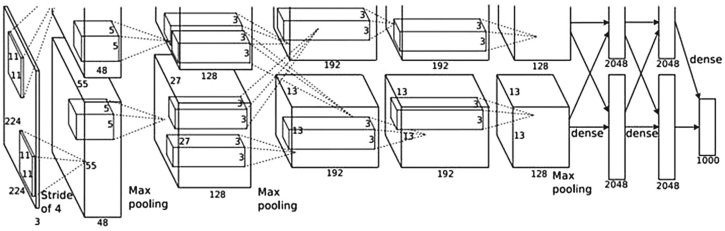


Fig. 1. AlexNet architecture. Figure reproduced from [1]

VGG. VGG achieves its depth by simply stacking more layers while following the standard practices introduced with AlexNet. The size of the convolution kernel is more regular. AlexNet use 11×11, 5×5 and 3×3 filters, but VGG only use 3×3 filters in the entire network. Notably, while using smaller filters, VGG required far more filters per layer. The amount of calculations and parameters of VGG is much larger than AlexNet.

ResNet. Deeper neural networks are more difficult to train, so in [5], author proposed residual learning framework reducing the vanishing gradient problem to ease the training of networks. This residual learning is mainly use of shortcut connections, illustrated in Fig. 2, that connect components of different layers with an identity mapping [6]. In particular, ResNet is built such that each layer learns an incremental transformation, $F(x)$, on top of the input, x, according to

$$H(x) = F(x) - x \tag{2}$$

instead of learning the transformation $H(x)$ directly as done in other standard CNN architectures.

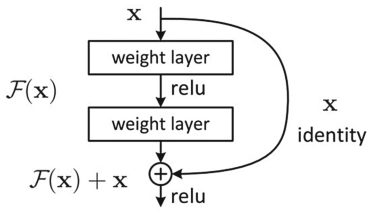


Fig. 2. Schematic of residual learning. Figure reproduced from [5]

2.2 OpenCL Framework on FPGA

OpenCL is an open, cross-platform parallel programming language that can be used in both CPU, DSP, GPU and FPGA developments. Recently, FPGA

vendors such as Xilinx and Intel have released OpenCL SDK for programming FPGAs. The Intel OpenCL environment which can be a mixture of C, C++, and OpenCL, provides a complete CPU/GPU-like development experience and run-time experience on a CPU/FPGA platform, including a complete software workflow spanning multiple target devices and x86 emulation with cycle-accurate FPGA hardware models and cycle-accurate FPGA hardware.

3 Architecture Design and Optimization

3.1 Accelerator Architecture

As shown in Fig. 3, our FPGA design based OpenCL framework consists of a group of OpenCL kernels that are cascaded by using Altera's OpenCL extension Channels. Two data mover kernels, namely MemRD and MemWR, transfer feature map and weight data from/to the global memory feeding other kernel with high throughput data streams. The cascaded kernels form a deep computation pipeline that can implement a serial of basic CNNs operations without the need of storing interlayer data back to global memory. It significantly reduces the bandwidth requirement compared to the work of [10]. The Convolution kernel is designed to implement both the convolution layer and the fully connected layer which are the most compute-intensive operations in CNNs. The Pooling kernel is controlled by the synchronization signal of the MemWR kernel. When the synchronization signal set one, the Pooling kernel operation is performed. This technique is mainly used to achieve overlap between two kernels. The BatchNorm kernel using in [5] loads mean, variance, α and β from global memory and performs the normalization directly on the output data streams of the Convolution kernel. The Local Response Normalization(LRN) kernel using in [1] fetches data from global memory and performs normalization on the feature map of neighboring neurons in deep direction. The Eltwise kernel mapping Eltwise Layer using in [5] loads data from global momory and adds each elements mainly using shortcut connections.

This architecture has the following advances:

(1) The cascaded and overlaped kernels form a deep pipeline architecture.
(2) Using a single hardware kernel to implement both the convolution and fully connected layers.
(3) Scalable hardware structure which implementation many classic CNNs operations, such as LRN kernel to AlexNet, BatchNorm kernel and Eltwise kernel to ResNet.

Convolution Kernel. A single work-item kernel with parallel convolution data paths is designed to implement both the function of the convolution and FC layers. In this paper, we propose to flatten the 3-D convolution operation into

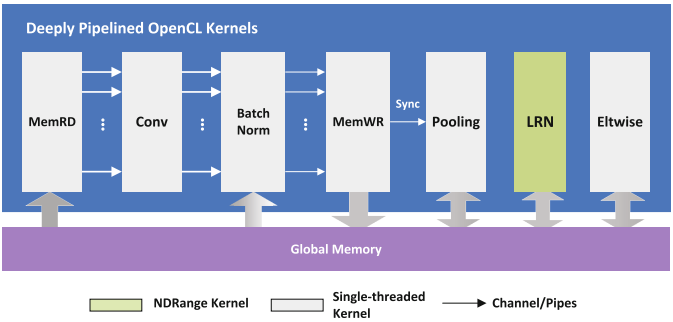


Fig. 3. The top-level architecture of CNN accelerator.

a 1-D convolution operation and integrate it with the full-connect operation as follow:

$$D_o(f_o) = \sum_{f_i'=1}^{C_l \times K \times K} W_l(f_o, f_i') \cdot D_i(f_i') \tag{3}$$

In this way, data vectorization and parallel CU structure are both explored in the design. Vectorized input features D_i and weights W_l are streamed by multiple Channels. A design parameter *VEC_SIZE* determines the degree of data vectorization and controls the input throughput. Another design variable parameter to accelerator the convolution operation *CU_NUM*, represents the parallel factor of weight and reuse factor of data. Due to efficient pipelined by the OpenCL compiler, We propose an efficient convolution pipeline structure consisted of a multiplier-adder tree with a delayed buffer as in Fig. 4.

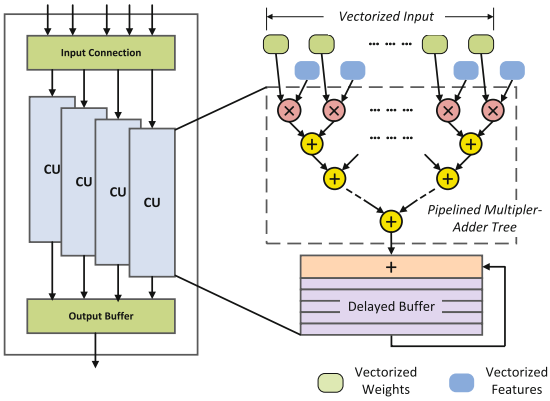
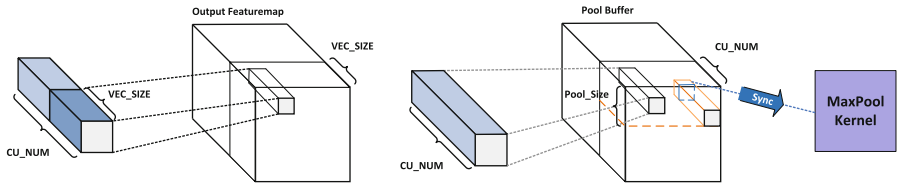


Fig. 4. The hardware architecture of the convolution kernel.

Data Mover Kernel. Two multi-model single work-item kernels are designed to fetch/store data from/to the global memory for the computation pipelines. MemRD kernel detailed schemes in [11] can fetch data from global memory to convolution mode or FC mode. We propose design parameter *FT_NUM* to determine the size of local memory, which further influences the reuse of input data. MemWR kernel is mainly used to receive the output of convolution kernel through the channel and arrange it into the storage structure required for the next convolution or pooling operation. For the convolution mode, the data received from the channel is arranged to have a depth of *CU_NUM*, and MemWR kernel need to divide the depth into *VEC_SIZE* copies and return it to global memory. The pooling mode simply transfer the data received from the channel and directly put back to global memory. In the pooling mode, the MemWR kernel also needs to pipe the synchronization signal to the pooling kernel at the right time for them can overlap work. Note all memory operations should be committed before sending token to the pooling kernel. Detailed MemWR schemes are illustrated in Fig. 5.



(a) convolution mode. (b) pooling mode.

Fig. 5. The hardware architecture of the memWR kernel.

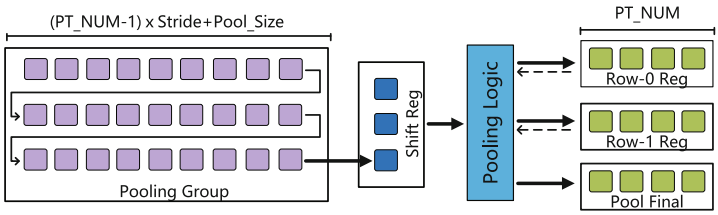


Fig. 6. The hardware architecture of the maxpool kernel.

Pooling Kernel. A shift-register-based hardware structure is proposed for the pooling kernel as shown in Fig. 6. The kernel first fetch the synchronization signal from the blocked channel, only waiting for the synchronization signal from blocked channel to come, the pooling kernel can start working. When the

synchronization signal comes, the pooling kernel read data from global memory to shift register. In the process of data transfer, if the time point of pooling is reached, data will be extracted from the shift register to the pooling logic. Similarly, we designed a parameter *PT_NUM* for adjusting the size of the local memory in pooling kernel to exploiting input data reuse. In the pooling strategy, the first line is processed first, then the second line is compared with the first line, and so on. The final result of the pooling calculation is stored in the ping-pong buffer. During the pooling calculation, the result of the last calculation is also divided into *VEC_SIZE* and returned to global memory for the next convolution calculation. This process is similar to MemWR.

Other Kernel. Besides the most compute-intensive convolution and fully connected kernel, we also designed some common opencl kernels, such as LRN, BatchNorm, Eltwise for the scalability and integrity of the CNN accelerator's overall design. In this architecture, you can choose the basic units used in the network to piece together to implement different network structures. For example, implementation AlexNet just choose convolution kernel, pooling kernel and LRN kernel. Therefore, this scalable architecture can process the complete CNN forword computation flow with little involvement of host CPU.

Table 1. Operations in AlexNet model

Index	Layer	d_x	d_y	d_z	w_x	w_y	w_n	w_m	GOPS
1	Conv1	227	227	3	11	11	3	96	0.281
2	Conv2	55	55	96	5	5	48	256	0.448
3	Conv3	27	27	256	3	3	256	384	0.299
4	Conv4	13	13	384	3	3	192	384	0.224
5	Conv5	13	13	384	3	3	192	256	0.032
6	FC1	6	6	256	6	6	256	4096	0.075
7	FC2	1	1	4096	1	1	4096	4096	0.034
8	FC3	1	1	4096	1	1	4096	1024	0.008
	Output	1	1	1024				Total Ops	1.40

4 Design Space Exploration

In this section, we present an analytical performance model and resource utilization model to choose the best combination of the design parameters (*VEC_SIZE*, *CU_NUM*, *FT_NUM*, *PT_NUM*) that maximizes the performance of the CNN accelerator, while still being able to fit in the limited FPGA resources.

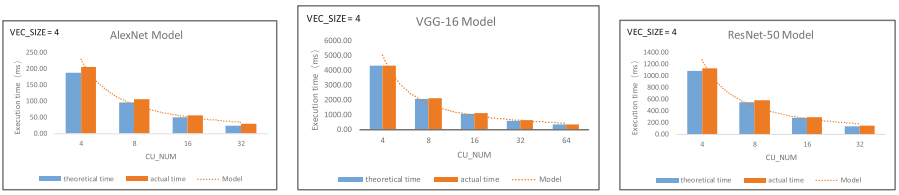
4.1 Performance Model

Convolution and Fully Connected Time. The execution time of convolution and fully connected layer-i is modeled as follow:

$$Convolution\ or\ FC\ Runtime_i = \frac{No.of\ Convolution\ or\ FC\ Ops_i}{VEC_SIZE \times CU_NUM \times Frequency} \quad (4)$$

Table 1 gives a operations summary of each layer in AlexNet model. Note that d_x, d_y and d_z represents the size of the output feature map from the previous layer, not the input size of the current layer. In 3.1 the convolution and fully connected operation have parallelism of two levels, one is the degree of parallelism *VEC_SIZE* based on the depth dimension of the input feature map, and the other is the degree of parallelism *CU_NUM* based on the number of convolution filters. So the speedup ratio for the convolution kernel is $VEC_SIZE \times CU_NUM$. The execution times of AlexNet, VGG-16 and ResNet-50 on *CU_NUM* are shown in Fig. 7.

Other Layers Time. Due to the idea of pipeline and overlap in the overall hardware design, the execution time of other kernels can be basically ignored relative to convolution and fully connected operations.



(a) AlexNet model. (b) VGG-16 model. (c) ResNet-50 model.

Fig. 7. Execution time empirical models for *CU_NUM*.

Memory Bandwidth. In order to reduce the pressure of external memory bandwidth, we use 8-bit fixed point calculations and propose a sliding-window-based data buffering scheme. Using fixed-point instead of floating-point calculations can reduce hardware synthesis costs and memory bandwidth requirements. Fortunately, research shows that using 8-bit fixed-point numbers instead of full-precision floating-point numbers is less than 1% loss in top 1/5 accuratacy for AlexNet/VGG predictions. As shown in Fig. 8, this sliding-window-based data buffering scheme use in MemRD kernel and maxpool kernel to cache data that was fetched from global memory. The filter stride S of this filter window is usually smaller than the filter size K. Therefore, a large portion of data can be reused during the convolution and maxpool computation. To exploiting data

reuse, the MemRD kernel design a *FT_NUM* parameter and maxpool kernel design a *PT_NUM* parameter. These kernel fetches a window of data that covers the area of *FT_NUM* or *PT_NUM* of filters each time, and caches the data in the on-chip buffers or shift register.

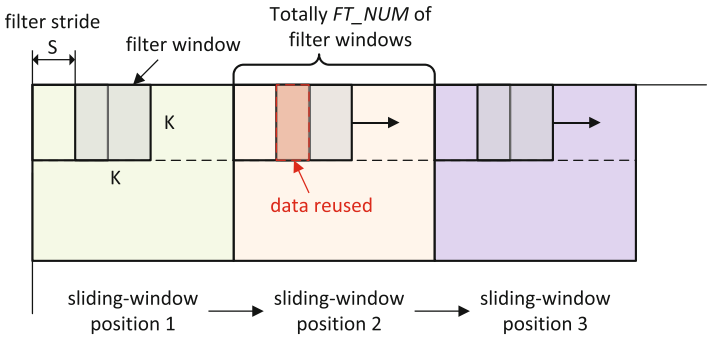
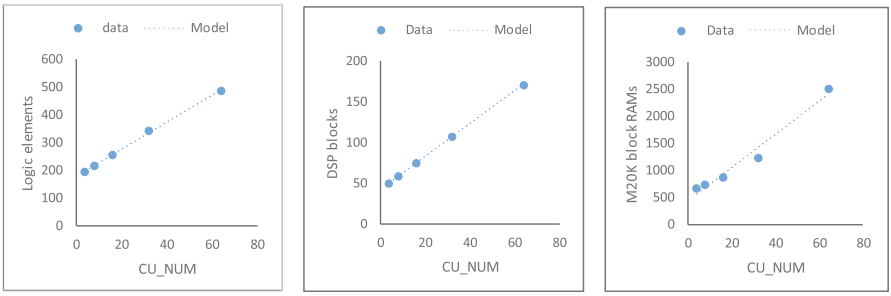


Fig. 8. The hardware architecture of the convolution kernel.



(a) logic elements model. (b) DSP model. (c) on-chip memory model.

Fig. 9. Resource utilization empirical models for *CU_NUM* on VGG-16.

4.2 Resource Utilization Model

In this subsection, we analyze resource utilization model on DE5-Net board. As discussed in 3.1, two design parameters *VEC_SIZE*, *CU_NUM* are used to control the hardware cost of the CNN accelerator. Therefore, we mainly consider the impact of the following two parameters in resource utilization model. Figure 9 shows the model with parameter *CU_NUM* on VGG-16. As the parameter *CU_NUM* gradually increases, both logic elements model and DSP utilization model present a trend of linear increase. However, the on-chip memory utilization model shows small discrepancy due to the complexity of load/store units.

5 Experimental Results

In this section, we present the experimental results to validate the scalable of this CNN accelerator by implementation three large-scale CNN models: AlexNet, VGG-16 and ResNet-50 on DE5-Net platform.

5.1 Experimental Setup

We use DE5-Net FPGA development board from Altera and compare with DE5a-Net listed its specification in Table 2. The OpenCL kernel codes are compiled using Altera OpenCL SDK 16.1, and the Quartus 16.1 is used as the FPGA implementation tool. The host machine is equipped with an Intel i7-5930K CPU and 64 GB memories. The data of images are first loaded from hard disks to the host programs, and then sended to the FPGA accelerators to perform CNN forword computations.

Table 2. Comparision of FPGA accelerator boards.

Specification	DE5-Net	DE5a-Net
FPGA	Stratix-V GXA7	Arria-10 GX1150
Logic elements	622 k	1150 k
DSP blocks	256	1518
M20K RAMs	2560	2560

5.2 Results and Discussion

In this subsection, we first list the best parameter configuration on different networks. Then, we show the benchmark of our CNN accelerator. Finally, we discuss the scalability of this hardware architecture. As discussed in 4, four design parameters *VEC_SIZE*, *CU_NUM*, *FT_NUM*, *PT_NUM* are used to control the hardware cost and throughput of the FPGA accelerator. Therefore, design space exploration can be quantitatively performed by implementing the accelerator with different parameter configuration. The final design variables for three networks optimized on the DE5-Net board are shown in Table 3.

In Table 4, we summarize the resource utilization, execution time and performance of different networks on the best parameters. We can see that different networks have different parameters and achieve different performance on same FPGA board. To prove how fast this accelerator can accelerate CNN computations, we also compare with CPU by using the Caffe deep learning framework. The execution time for AlexNet, VGG-16 and ResNet-50 is 189 ms, 1547 ms and 1238 ms, respectively. We can see that using FPGA-based accelerator can achieve more than 10 times faster on average in implementation CNN-based

Table 3. Optimized parameters.

	AlexNet	VGG-16	ResNet-50
VEC_SIZE	8	8	16
CU_NUM	48	32	16
FT_NUM	7	7	7
PT_NUM	2	4	4

Table 4. Summary of the resource utilization, execution time and throughput on different networks.

	AlexNet	VGG-16	ResNet-50
Logic elements	491.3 k	368.5 k	532.6 K
DSP blocks	236	170	256
M20K RAM	2252	1133	1537
Frequency	197.9 MHz	219.7 MHz	223.6 MHz
Execution time	18.08 ms	355.92 ms	102.97 ms
Throughput	77.5 GOPS	103 GOPS	75.7 GOPS

image classification applications. In future works, we will explore sparse convolution algorithms and using Winograd transformations to reduce the number of computations and to improve the performance of this accelerator.

6 Conclusion

In this work, we implemented a scalable FPGA accelerator for convolutional neural networks using OpenCL framework. An efficient and scalable hardware architecture with deep pipelined kernels was presented. We proposed and explored four design parameters for hardware costs and bandwidth limited, and implemented three large-scale CNNs, AlexNet, VGG-16 and ResNet-50 on DE5-Net FPGA board.

Acknowledgment. This work was supported by NNSF of China Grants NO. 61574013, 61532005.

References

1. Krizhevsky, A., Sutskever, I., Hinton, G.E., et al.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
2. Ren, S., et al.: Faster R-CNN: towards real-time object detection with region proposal networks. In: Advances in Neural Information Processing Systems, pp. 91–99 (2015)

3. Abdel-Hamid, O., et al.: Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition. In: Acoustics, Speech and Signal Processing, pp. 4277–4280 (2012)
4. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
5. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
6. Hadji, I., Wildes, R.P.: What Do We Understand About Convolutional Networks? arXiv preprint arXiv:1803.08834 (2018)
7. Qiu, J., Wang, J., Yao, 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 (2016)
8. Wang, C., Gong, L., Yu, Q., Li, X., Xie, Y., Zhou, X.: DLAU: a scalable deep learning accelerator unit on FPGA. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **36**(3), 513–517 (2017)
9. Zhang, C., et al.: Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 161–170 (2015)
10. Suda, N., et al.: 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 (2016)
11. Wang, D., Xu, K., Jiang, D.: PipeCNN: an OpenCL-based open-source FPGA accelerator for convolution neural networks. In: Field Programmable Technology (ICFPT), pp. 279–282 (2017)