HLS-Based Acceleration Framework for Deep Convolutional Neural Networks

Ashish Misra and Volodymyr Kindratenko(✉)

University of Illinois, Urbana, IL 61801, USA
{ashishm,kindrtnk}@illinois.edu

Abstract. Deep Neural Networks (DNNs) have been successfully applied in many fields. Considering performance, flexibility, and energy efficiency, Field Programmable Gate Array (FPGA) based accelerator for DNNs is a promising solution. The existing frameworks however lack the possibility of reusability and friendliness to design a new network with minimum efforts. Modern high-level synthesis (HLS) tools greatly reduce the turnaround time of designing and implementing complex FPGA-based accelerators. This paper presents a framework for hardware accelerator for DNNs using high level specification. A novel architecture is introduced that maximizes data reuse and external memory bandwidth. This framework allows to generate a scalable HLS code for a given pre-trained model that can be mapped to different FPGA platforms. Various HLS compiler optimizations have been applied to the code to produce efficient implementation and high resource utilization. The framework achieves a peak performance of 23 frames per second for SqueezeNet on Xilinx Alveo u250 board.

Keywords: Accelerator design · High level synthesis · FPGA

1 Introduction

Deep Neural Networks (DNNs) have made a profound impact on applications such as image classification [1, 2] and speech recognition [3, 4]. However, they demand extensive computations and impose extreme timing constraints because of their deep topological structures, complicated cross-layer connections, and massive amounts of data to process. As a result, it becomes challenging to achieve high performance and good energy efficiency when mapping DNNs onto generic computing systems. To mitigate this problem, many hardware (HW) accelerators for DNN inference have been explored. Among these designs, Field-Programmable Gate Array (FPGA) based accelerators have gained great popularity due to their reconfigurability, massive fine-grained parallelism, and performance per watt advantage.

The extreme scale integration of modern system on-chip (SoC) and burgeoning design complexity of emerging applications has made it imperative to design at a higher level of abstraction in order to achieve high productivity. To manage this issue, high-level synthesis (HLS) tools have emerged to allow application developers to describe the hardware accelerator using common software (SW) programming languages, such as C/C++, by automatically generating RTL from behavioral descriptions [5, 7].

F. Rincón et al. (Eds.): ARC 2020, LNCS 12083, pp. 221–231, 2020.
https://doi.org/10.1007/978-3-030-44534-8_17

A typical DNN architecture has multiple layers that extract features from the input data. Convolution is the most computationally expensive function which requires millions of floating-point operations (FLOPs) in these networks. Thus, it needs a good accelerator architecture which should balance maximum memory access and computation and software linkage to DNN frameworks. Currently, many open-source software frameworks have been released for DNN research but most of them suffer from scalability problems. Existing FPGA-based convolutional neural network (CNN) accelerator designs primarily focus on optimizing the computational resources without considering the impact of the external memory transfers or optimizing the external memory transfers through data reuse [5], or on optimizing only the convolution layers [8].

To address the above-mentioned problems, we present a systematic methodology for maximizing the throughput of an FPGA-based accelerator for an entire DNN model consisting of convolution, pooling and certain layers executed in software. In this paper, we describe a framework, which starts from a trained network (Caffe/TensorFlow) and generates a deployable accelerator for image classification. The entire compilation procedure is end-to-end and automated, which makes it possible for DNN researchers and users to use FPGA as a powerful device to perform model inference. In this paper, we introduce this novel architecture and provide the following contributions:

1. A configurable streaming framework for DNN accelerators that exploits operator level, loop level, input channel and output channel parallelism.
2. Automatically generated verification network in C++ that allows users to test the correctness of the design. The framework can be exploited either as individual kernels or as a set of layers scheduled on the hardware.
3. The framework allows to use certain layers as HW and certain layers as SW, providing the designer with a choice of possible configurations.

2 Related Work

The analysis of a good accelerator design in the context of DNNs should be based on three factors: (i) Number of frames per second (FPS) achieved at run-time; (ii) Flexibility of the design to handle many classes of DNNs; and (iii) Minimum loss of accuracy for classification of an image with dataset with hundreds of classes. We only focus on the related work that have demonstrated at least two of these requirements.

Shawahna et al. [9] present an extensive comparison of accelerator designs for FPGAs; but they do not consider first and third factor in their comparison. Guan et al. [8] present an extensive framework showing VGG-19, ResNet-152 and LSTM-LM; but they do not report latency or FPS obtained after deployment. Qiu et al. [10] report the comparison of latency for VGG16-SVD network for FPGA, CPU and GPU. The latency reported is 224.60 ms and total operations is 30.76 giga operations (GOPs), hence the overall performance is 136.97 GOPs/second (GOPS). The FPS reported is 4.45 for 16-bit quantized weights and 5.88 for 8-bit. While this work demonstrates the best possible frame rate for a large network, it still does not report the accuracy of the network after quantizing the weights. Also, the authors modified VGG-16 to VGG16-SVD, hence

the actual 30.76 GOPs parameter have been downsized. Zhang et al. [11] also report 158.8 ms for VGG16-SVD network. This is a significant improvement as compared to previous results, but the authors also do not report the FPS. Suda et al. [12] also present the latency of VGG-16 as 651.2 ms. This work reports all three factors and can be used for comparison.

Using the fixed-point data types requires that trained weights must be quantized as reported by Song H. et al. [13]. This work however shows that with appropriate quantization of the weights, acceleration can be achieved at the expense of accuracy. Xilinx ml-suite [14] reports the best possible performance of 4127 images/sec on `int8` data type with GoogleNet.

Other optimizations that can be applied include: (1) Algorithmic optimizations for convolution operations, for example, the core of computation engine can be designed using a Wallace tree [6] or a systolic array. Our work focuses on balanced tree for multiply and accumulate (MAC) `int16` operations. (2) Dataflow optimizations for maximum memory bandwidth. The dataflow model requires that there is a non-stop dataflow from memory-in to memory-out with maximum data transfer in each cycle. The work in [6] introduces a roof-line model for analysis of the design for memory throughput. Authors in [15] show how efficiently streaming can be applied in the design.

3 Proposed Design

3.1 Architecture Design

Here we describe the proposed design that exploits the concurrency features intrinsic in convolution function. Figure 1 shows each unit and function in the proposed architecture. The design works on 128 input channels, 64 output channels and kernel size, w_{size}, either 3 or 1. The complete module as shown in Fig. 1 is instantiated four times, allowing to compute 256 output channels concurrently. The weight file obtained after quantization is stored as `int16` and each weight is multiplied by 64. The value of 64 is chosen based on that no layer in SW shows underflow or overflow. No change is made to the input image. Multiplication is carried out in `int16` but results are maintained as `int` for bias addition and finally divided by 64 to scale down the values to `int16`. Next, we describe the HW and SW units that are used to implement the network.

Funtion_0: The first layer of each network is different, for example, the kernel size can vary from 1 to 11, the number of input channels is usually 1 or 3 and the input dimension can be arbitrary high as 448. Hence this layer can be executed in the SW or HW framework, depending on its complexity. The first layer requires significant amount of MAC operations; hence it is better to design a new kernel customized for parameters.

HW_Interface_1: The proposed architecture has been built on the dataflow model (recognized by `#pragma HLS Dataflow`), which allows to concurrently access non-overlapping data stored in different memory banks. Hence input data and weight data for a layer have been placed in two DDR memory banks and concurrently accessed using AXI4 interface as shown in Fig. 1.

HW_Unit_1: The first two functions in the design are accessing data from each DDR bank and convert it to 128 input streams and 128 weight streams. The weight streamer saves the data in on-chip buffer (*wgt_buff[128][64][9]*) and then streams are created in a different loop structure. The aim of using the *wgt_buff* is to initialize the streams to zero if the number of input channels is less than 128. Since the size of the input data is larger than the weight data, more cycles can be spend checking the number of channels and creating a second loop structure in the *stream_weight()* function. Each of the streams is mapped to FIFO which can be mapped to either LUTs or BRAMs present on the chip.

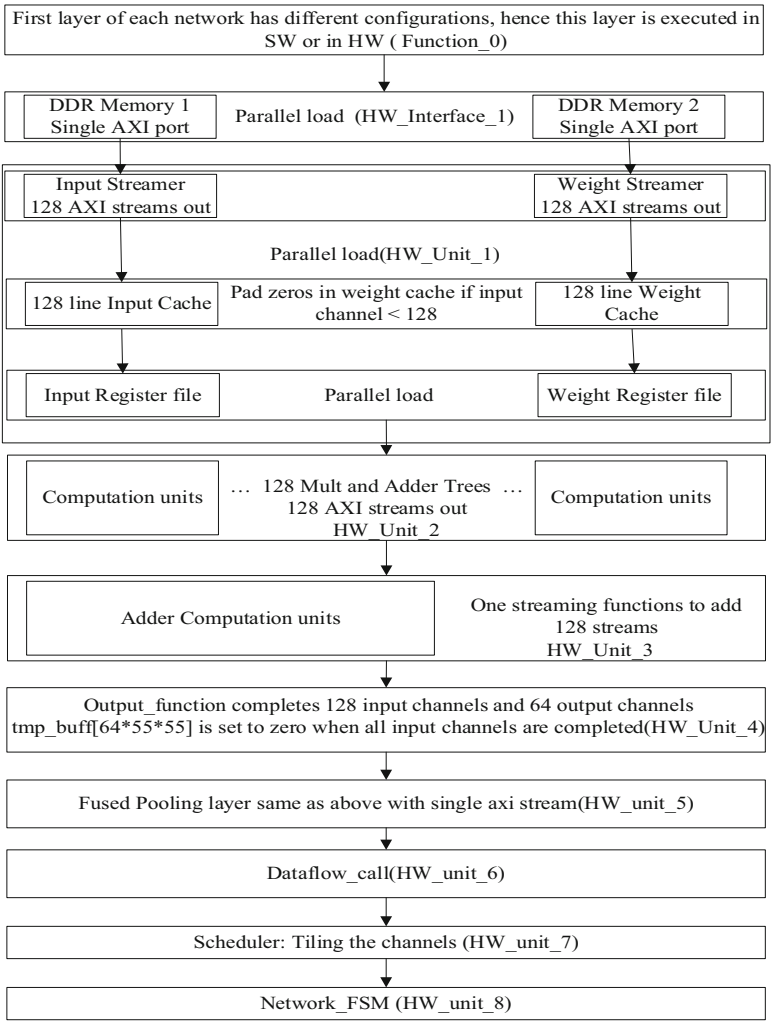

Fig. 1. Dataflow representation for convolution and pooling layer.

HW_Unit_2: This unit contains 128 identical convolutional functions, each with a different stream interface. This is because streams are static in high level synthesis flow. Hence 256 streams (one input and one weight) from HW_function_1 reach to 128 concurrent computation units.

There are primarily three objectives to be achieved in this unit. The line buffer receives the data from a stream which should not be stalled as shown in Algorithm 1. This is achieved by overlapping the computation and stream access (as multiplication and loading are parallel). The loading of *wgt_mac_0* for computation; streaming out the data; and loading new line in the line buffer is done by pipelining. The kernel weights should not be loaded again and again for the output channel. This is achieved by looping for 64 or less output channels.

Algorithm 1. Convolution

```
1)  Input: Two axi streams for each convo() function
2)  Given: osize, stride, padding, wsize, ochan, ichan .
3)  Output:  one output stream
4)    for yy  = 0 to osize:
5)      for ochan_no = 0 to ochan:
6)        load wgt_mac register
7)        for xx = 0 to osize:
8)          if wsize == 1:
9)            call convo_1d()
10)         else:
11)           load wgt_mac_0 from line buffer
12)         stream_convo_out_1_0  <<  call convo_2d()
13)         if  ochan_no == 0:
14)           load line buffer next line
15)   call rotate line buffer
```

Once the line buffer is loaded, instead of completing an input channel frame, first line of 64 output channels is computed and then the line buffer is updated. If there are more than 128 input channels, a temporary buffer is used to store the data and this data is accessed again for computing all the input channels.

The third objective in the computation algorithm is to achieve a pipelined MAC tree for sum of product operation. The HLS tool can produce a balanced tree if integer operations are performed, hence objective is achieved in the synthesis process. The delay of the tree is given as worst delay of an operator. Though the latency of the tree may be high, the initiation interval is one, which means next input can be taken after one cycle.

HW_Unit_3: This unit contains one streaming function to add data coming from 128 computation units and produces one output stream.

HW_Interface_2: The two DDR memories discussed in HW_Interface_1 were used for getting input data. Similarly, remaining two DDR memories are used for storing output and network parameters data. Temporary buffer can also be utilized if enough BRAMs are present in the chip and one DDR bank can be eliminated. Since this design computes 64 output channels and 128 input channels, this means if 1024 input channels are present, then intermediate tile data of size $64*i_{size}*i_{size}$ has to be stored in a DDR bank (*temp_buff*). This DDR bank should be accessed when adding the values to the

next tile computed. All the parameters of the network such as weight offset and bias offset are precomputed and copied to DDR 4.

HW_Unit_4: This unit receives one stream from HW_Unit_3 and contains one function to read/write data from two DDR banks or temporary buffer (*para_buff)*. The bias is added at this stage and data is stored back in one of the DDR banks or *para_buff*. Algorithm 2 demonstrates this process. This unit also checks whether pooling is required, if yes then stream data goes to a pooling function else the polling unit is bypassed.

Algorithm 2. DDR Access

```
1)  Input: One axi stream. Given: osize, ochan,
          bias buffer, bb, ichan_en, aa, out_offset.
2)  Output: Data written to DDR banks
          declare out_buff for burst use
3)    for yy = 0 to osize:
4)      for ochan_no = 0 to ochan:
5)        for xx = 0 to osize:
6)          datatype_inh sum = 0;
7)          datatype_inh sum1 = 0;
8)          datatype_inh dp_0;
9)          stream_adder_out_0 >> dp_0;
10)         if bb > 0 :
11)           sum1 = dp_0 + para;
12)         else:
13)           sum1 = dp_0;
14)         para_buff[xx] = sum1;
15)         sum = sum1 + bias[ochan + aa*ochan_fac] ;
16)         if  sum > 0 :
17)           out_buff[xx] = sum >> 64;
18)         else:
19)           out_buff[xx] = 0;
20)         if bb == ichan_index-1:
21)           if (pool_on ==1):
22)             Stream_out_pool_fused << out_buff[xx];
23)           else:
24)           write_ddr3_with_burst();
25)           write_ddr4_with_zero();
26)           para_buff_with_zero();
27)         else:
28)           write_ddr4with data();
```

Algorithm 2 works in conjunction with Algorithm 3. It takes bias buffer which is pre-loaded, *bb* variable which is dependent on the number of input channels ($bb > 0$ if $i_{chan} > 128$), *aa* variable which is dependent on the number of output channels ($aa > 0$ if $o_{chan} > 64$), and i_{chan_en} variable that defines the number of iteration for all the input channels. The streams bring the data in, which is then summed up with temporary data from previous iteration, bias is added, relu activation is applied, last iteration is checked, data is written to DDR3 bank and *para_buff* is initialized to zero again.

HW_Unit_5: This unit contains three functions to complete pooling in a fused manner. First function receives one stream from previous unit and caches it in a small memory. Second function does the pooling operation and third function stores the data in the DDR. If a convolution unit with stride of 2 is required, this unit is enabled as well.

HW_Unit_6: This unit defines one function that calls all the above units in a dataflow model. Total of (128 × 4) 512 input streams, (128 × 4) 512 weight streams, 4 adder streams, 4 output streams, 12 pooling streams, are defined in this function. This function connects all the streams with one input function and one output function, 512 convolution functions 4 adder functions, 4 pooling functions. All the units are instantiated in this function.

HW_Unit_7: This unit, calls the HW_unit_6 in a sequential way for completing one layer. Since the architecture works on input and output of tile size 128 × 256, the scheduler calls HW_unit_6. Suppose the output channels are less than 256, then o_{chan_index}=1, if $o_{chan} > 256$, then $o_{chan\_index} = o_{chan}/256$ (Algorithm 3).

If $i_{chan} < 128$, then i_{chan_index}=1, else it is shifted by 7. The call function (line 8) takes weight offset ($aa * i_{chan} * out_{chan} * w_{size} * w_{size} + bb * 128 * w_{size} * w_{size} + wgt\_offset$) and output offset ($aa * out_{chan} * o_{size} * o_{size} + out_{offset}$) as arguments.

Algorithm 3. Scheduler (layer_128ic_256oc())

```
1)  Input: DDR pointers.
2)  Given: o_size, stride, padding, w_size, o_chan, out_offset, wgt_offset.
    Output:  one output streams
3)  Delare o_chan_index, out_chan, i_chan_index
4)    if o_chan <= 256:
5)      o_chan_index = 1;
6)      o_ut_chan = o_chan;
7)    else:
8)      o_chan_index = o_chan >> 8;
9)      out_chan = 256;
10)   if i_chan <= 128:
11)     i_chan_index = 1;
12)   else:
13)     i_chan_index = i_chan >> 7;
14)   if (ochan <= 256):
15)     ochan_fac = ochan >> 2;
16)   else:
17)     ochan_fac = 64;
18)   for aa = 0 to o_chan_index:
19)     for bb = 0 to i_chan_index:
20)       call hw_unit_6
```

HW_Unit_8: All the layers are completely scheduled by an FSM designed in HW. A python script generates this FSM along with weight offset, bias offset and output offset for each layer. The generated HW creates FIFO channels for each parameter in HW_unit_7. Such an FSM helps with DDR bank swapping and no host intervention is required (Algorithm 4).

Vivado HLS schedule reports the initiation interval for each function, which determines how well the dataflow is pipelined. The initiation intervals for each function reported are *convo* (1 cycle), *addstreams* (1 cycle), *stream_in* (1 cycles), *stream_out* (1 cycle), *stream_weight* (1 cycles) and *pooling* (1 cycle).

Algorithm 4. Layer Scheduler

```
1)  Input: Input data, complete weights,
2)  Given: externally generated FSM with weight and output offsets
3)  Output:  data for last layer
4)  Restart FSM,
5)  Load bias in bias buffer,
6)  Call HW_unit_6
7)  If last state is reached, go to 10
8)  Swap the DDR pointers
9)  Go to 6 for next layer,
10) For next image wait for signal to toggle and go to 4
```

From a given trained Caffe-based network, weight and network parameters are extracted to generate the complete network scheduler. A python program has been written to generate all the linear weights and output offset. The weights are then quantized using scripts and standard deviation process. This process is discussed in Algorithm 5.

Algorithm 5. Network Scheduler

```
1)  Input: Input data, complete weights,
2)  Given: extracted weight file, paramters and network from
           caffe model/tensorflow
3)  Perform offline quantization to generate new weight file.
4)  Output:  image class
5)  Load new image
6)  Multiply each weight each 64 and store in DDR banks as int16.
7)  Call layer scheduler, Copy the output for last layer
8)  Call the tensorflow function for last layer or output the results
9)  Go to 5 for a new image
```

Table 1 shows that 29% FF, 78% LUT, 52% DSP, 54% BRAM and 44% URAM are utilized in the design. The entire design is set to synthesize at 200 MHz, but the functions report a frequency of 300 MHz. There is still possibility that more computation can be done, however maintaining the LUTs resources utilization at this level becomes difficult.

3.2 Verification Setup and Executable Setup

Firstly, the implemented DNN is verified in Caffe/TensorFlow and tested for ten images from a trained data set (trained.caffemodel) and a DNN architecture file (deploy.prototxt). The complete network is then rebuild in C++ using a python script from the parameters extracted from deploy.prototxt file. The SW emulation of written HLS code is tested in this C++ network. Each layer testing can also be done using the C++ layer data. In this work, the reported results are on Vivado SDx with all layers in HW.

Table 1. Resource consumption of each function in Vivado HLS.

	FFs	LUTs	DSPs	BRAMs	URAM
Stream_in x1	14555	38191	128	0	0
Stream_wgt	13682	38191	38	0	512
Mac tree/Convo_0 x512	609/1571	105/1971	9/11	0/5	0
Stream_adder 0 x4	1610	5465	1	0	0
Stream_out x4	346	917	2	2	0
Stream_out_pool_fused x4	1231	2741	0	16	0
Layer_128ic_64oc	896837	1249546	5845	2648	512
Kernel_7_layer (first layer)	107284	91492	648	280	0
CNN (top)	1018689 (29%)	1350874 (78%)	6501 (52%)	2945 (54%)	704 (55%)
Total on Alveo u250	3456000	1728000	12288	5376(18 Kb)	1280

4 Results

The design has been implemented with Xilinx SDx 2019.1 on Xilinx Alveo u250 board. We present the results of SqueezeNet tested on Alveo u250. Table 3 shows the FPGA execution time for all layers executed in HW. The total number of MAC operations in SqueezeNet is 861.34 M [16] and the total convolution operations comprising of MACs result in 861.34 * 2=1722.68 M. The total comparators are 9.67 M and the additions in other layers are 226 K. This yields to total operations as 1732.546 MFLOPs. For calculating the GFLOPS, we first calculate the total number of operations in network and then divide this by the execution time 1732.546/0.043 = 40.291 GFLOPS.

Four processing units use 5845 DSP, out of which 5333 can be active at any time with kernel size 3 is running. `Int16` takes one DSP slice hence the peak performance achieved is $5333 \times 200 \times 10^6 = 1066.6$ GFLOPS. Similarly, when kernel size 1, peak performance is $128 \times 4 \times 200 \times 10^6 = 102$ GFLOPS. The layer 1 has kernel size 7 and is designed separately for achieving better performance. The same dataflow model architecture has been used with a configuration of three input channels and 24 output channels. Similarly, when kernel size 7 is running, peak performance is $648 \times 200 \times 10^6 = 129$ GFLOPS.

The verification results from the execution on Vivado SDx [17] shows top-1 accuracy for the SqueezeNet reported in Caffe framework as 57.5%. Our framework shows an additional loss of 1.2% due to the quantization process applied and `int16` used as the base data type.

Table 3. Comparison of HW and SW based on SqueezeNet

Platform	Type	Latency (sec)	FPS
Caffe CPU	Intel i7-6700 K	0.1701	5
FPGA	Alveo u250	0.043	23

5 Conclusion

In this work we have successfully tested SqueezeNet in our framework with a frequency of 200 MHz. We have achieved a frame rate of 23 frames/second. The accelerator and the verification setup have been generated using python scripts which allow user the configurability and scalability of input and output channels with kernel size of 3 or 1.

Acknowledgments. This work is funded by the National Science Foundation's Major Research Instrumentation program, grant #1725729. We thank Yuan Ma for his help in setting up the simulation using Caffe and Tanitpong Lawphongpanich for his contribution with TensorFlow testing.

References

1. 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)
2. Russakovsky, O., et al.: ImageNet large scale visual recognition challenge. Int. J. Comput. Vision **115**(3), 211–252 (2015)
3. Graves, A., Mohamed, A.-R., Hinton, G.: Speech recognition with deep recurrent neural networks. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 6645–6649. IEEE (2013)
4. Abdel-Hamid, O., Mohamed, A.-R., Jiang, H., Deng, L., Penn, G., Yu, D.: Convolutional neural networks for speech recognition. IEEE/ACM Trans. Audio Speech Lang. Process. **22**(10), 1533–1545 (2014)
5. Sankaradas, M., et al: A massively parallel coprocessor for convolutional neural networks. In: 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2009, pp. 53–60. IEEE (2009)
6. 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)
7. Ovtcharov, K., Ruwase, O., Kim, J.-Y., Fowers, J., Strauss, K., Chung, E.S.: Accelerating deep convolutional neural networks using specialized hardware. Microsoft Research Whitepaper **2**(11), 1–4 (2015)
8. Guan, Y., et al.: FP-DNN: an automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 152–159. IEEE (2017)

9. Shawahna, A., Sait, S.M., El-Maleh, A.: FPGA-based accelerators of deep learning networks for learning and classification: a review. IEEE Access **7**, 7823–7859 (2018)
10. Qiu, J., 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)
11. Zhang, J., Li, J.: Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 25–34. ACM (2017)
12. 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. ACM (2016)
13. 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)
14. https://www.xilinx.com/products/acceleration-solutions/xilinx-machine-learning-suite.html. Accessed 21 Aug 2019
15. Aydonat, U., O'Connell, S., Capalija, D., Ling, A.C., Chiu, G.R.: An OpenCL™ deep learning accelerator on arria 10. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 55–64. ACM (2017)
16. https://dgschwend.github.io/netscope/#/preset/squeezenet. Accessed 21 Aug 2019
17. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1027-sdsoc-user-guide.pdf. Accessed 21 Aug 2019