[DL] A Survey of FPGA Based Neural Network Accelerator

KAIYUAN GUO, SHULIN ZENG, JINCHENG YU, YU WANG AND HUAZHONG YANG, Tsinghua University, China

Recent researches on neural network have shown great improvement in computer vision over traditional algorithms based on handcrafted features and models. Neural network is now greatly adopted in regions like image, speech and video recognition. But the great computation and storage complexity of neural network based algorithms poses great difficulty on its application. CPU platforms are hard to offer enough computation capacity. While GPU platforms are highly parallelized, the energy efficieny is low. The high energy cost of GPU causes problems for a wide application of neural network.

To address the above problems, various FPGA based hardware accelerators for neural networks have been proposed. Specialized hardware are designed to achieve high speed and low power neural network process. In this paper, we give an overview of previous work on neural network accelerators based on FPGA and summarize the main techniques used. Investigation from software to hardware, from circuit level to system level is carried out to complete analysis of FPGA based neural network accelerator design and serves as a guide to future work.

Additional Key Words and Phrases: FPGA, Neural Network

ACM Reference Format:
Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang AND Huazhong Yang. 2017. [DL] A Survey of FPGA Based Neural Network Accelerator. *ACM Trans. Reconfig. Technol. Syst.* 9, 4, Article 11 (December 2017), 16 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Recent research on Neural Network (NN) is showing great improvement over traditional algorithms in computer vision. Various network models, like convolutional neural network (CNN), recurrent neural network (RNN), have been proposed for image, video, and speech process. CNN [21] improves the top-5 image classification accuracy on ImageNet [38] dataset from 73.8% to 84.7% and further helps improve object detection [8] with its outstanding ability in feature extraction. RNN [15] achieves state-of-the-art word error rate on speech recognition. In general, NN features a high fitting ability to a wide range of pattern recognition problems. This makes NN a promising candidate to many artificial intelligence applications.

But the computation and storage complexity of NN models are high. The research on NN is also increasing the size of NN models. The largest neural network model for an 224×224 image classification requires upto 39 billion floating point operations (FLOP) and more than 500MB model parameters [42]. As the computation complexity is propotional to the input image size, processing images with higher resolutions may need more than 100 billion operations.

Traditional hardware platforms are not suitable for neural network process. A common CPU can perform 10-100G FLOP per second, and the power efficieny is usually below 1GOPs/W. So CPUs neither meet the high performance requirements in cloud applications nor the low power

Author's address: Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang AND Huazhong Yang, Tsinghua University, Tsinghua University, Beijing, Beijing, 100084, China, gky15@mails.tsinghua.edu.cn,yu-wang@mail.tsinghua.edu.cn.

1936-7406/2017/12-ART11 $15.00
https://doi.org/0000001.0000001

requiremetns in mobile applications. In contrast, GPUs offer upto 10TOP/s peak performance and is a good choice for high performance neural network applications. Development frameworks like Caffe [19] and Tensorflow [3] also offers easy-to-use interfaces which makes GPU the first choice of neural network acceleration. But GPUs are power consuming and thus not suitable for mobile applications.

On the other hand, FPGA is becoming a candidate to implement energy efficient neural network accelerator. With a specific hardware design, FPGAs are able to implement high parallelism and make use of the properties of neural network computation to remove unecessary logic. Therefore FPGAs are possible to achieve higher energy efficieny compared with CPU and GPU.

But FPGA based accelerator designs are still faced with two problems:

- Current FPGAs usually support working frequency at 100-300MHz, which is much less than CPU and GPU. The FPGA's logic overhead for reconfigurability also reduces the overall system performance. Straight forward design on FPGA is hard to achieve high performance and high energy efficiency.
- Implementation of neural networks on FPGAs is much harder than that on CPUs or GPUs. Development framework like Caffe and Tensorflow for CPU and GPU is needed for FPGA.

Many researches on the above two problems have been carried out for energy efficient and flexible FPGA based neural network accelerator. In this paper, we summarize the techniques proposed in these work. Specifically, we will introduce the techiques from the following aspects:

- We investigate current techniques for high performance and energy efficient neural network accelerator designs. Techniques in both software level and hardware level are evaluated.
- We investigate state-of-the-art automatic design methods of FPGA based neural network accelerators.

The rest part of this paper is organized as follows:

2 PRELIMINARY ON NEURAL NETWORK

In this section, we introduce the basic operations included in neural network algorithms. Neural network is a bio-inspired model, which usually includes several layers. Each layer receives input from a set of neurons and output a set of neurons. The synapses connecting input and output neurons are modeled as parameters, which is referred to as weights in this paper. In the rest part of this section, we introduce different types of layers in neural network models.

Fully conencted (FC) layer implements a connection between every input neuron and output neuron with a weight. This type of layer is adopted in both CNN and RNN. The input and output neurons of an FC layer are two vectors $\mathbf{x}$ and $\mathbf{y}$. The weights of this layer can be modeled as a matrix W. A bias vector b is added to each of the output neuron. The computation of this layer is described as equation 1.

$$\mathbf{x} = W\mathbf{y} + \mathbf{b} \tag{1}$$

Convolution (CONV) layer is used for 2-d neuron process. This is commonly adopted in CNN for image process. The input and output neurons of this layer can be described as sets of 2-d feature maps, F_{in} and F_{out}. Each feature map is denoted as a channel. A CONV layer implements a 2-d convolution kernel K_{ij} for each input and output channel pair and a bias scalar b_i for each output channel. The computation of a CONV layer with M input channels and N output channels can be described as equation 2.

$$F_{out}(j) = \sum_{i=0}^{M-1} \text{conv2d}(F_{in}(i), K_{ij}) + b_j,\ j = 0, 1, ..., N-1 \tag{2}$$

There are varieties of 2-d convolutions in CONV layer. Usually standard convolution with padding is used when the kernel size is 3×3. For larger kernels like 5×5 and 7×7, a stride larger than 1 is usually used to reduce the number of operation. Recent work is also using 1×1 convolution kernels [16, 18].

Non-linear layer applies a non-linear function on each of the input neurons. Sigmoid function and tanh function are commonly adopted in early models are are still used in RNN for acoustic or speech recognition. Rectified linear unit (ReLU) [21] is the adopted in many state-of-the-art models. This function maintains the positive neurons and filters negative neurons as zero. Varieties of ReLU are also used, such as PReLU and Leaky ReLU [50].

Pooling layer is also used for 2-d neuron process like CONV layer. A pooling layer downsamples each of the input channel respectively, which helps reduce feature dimension. There are two kinds of down sampling method: average pooling and max pooling. Average pooling splits a feature map into small windows, i.e. 2×2 windows, and finds the average value of each window. Max pooling method finds the maximum value in each window. Common window size includes 2×2, stride=2 and 3×3, stride=2.

Element-wize layer is usually used in RNN and is introduced in ResNet [16]. This layer receives two neuron vectors of the same size and applies element-wise operations on corresponding neurons of the two vectors. In ResNet, this layer is element-wise addition. For RNN, this layer can be element-wise subtraction or multiplication.

Among these types of layers, FC layer and CONV layer contributes to most of the computation and storage in neural networks. In the following sections, both software level and hardware level designs focus on these two types of layers.

3 DESIGN METHODOLOGY

Before going into the details of the techniques used for fast and energy efficient neural network accelerator, we first give an overview of the design methodology. In general, the design target of a neural network processing system includes the following three aspects: high model accuracy, high throughput and high energy efficiency. For certain applications, high flexibility should also be considered.

In general, a larger neural network model usually results in a higher model accuracy. Different network structures like the ones in [16, 21, 42] surely affect model accuracy, but is out of the discussion of this paper. With a same model, applying model compression methods can acheive the trade-off between throughput and model accuracy. Some of the model compression methods even achieves acceleration without model accuracy loss.

The throughput of a neural network processing system can be expressed by equation 3. With model compression methods, we can reduce the workload. With a certain FPGA chip, the on-chip resource is limited. Increasing the peak performance means to reduce the size of each computation unit and increase the working frequency. Reducing the size of computation units usually means to simplify the basic operations in neural network model, which is a hardware-software co-design problem. Increasing working frequency, on the other hand is pure hardware design work. A high utilization ratio is kept by reasonable parallelism implementation and efficient memory system. Most of this part is affected by hardware design. But model compression can also reduce the storage requirment of a neural network model and benefits the memory system.

$$throughput = \frac{peak\_performance \times utilization}{workload} \tag{3}$$

Energy efficiency is evaluated by the number of operations (multiplication or addition in this case) executed with unit energy cost. Given a certain network model, the energy efficiency of a

neural network processing system is inversely proportional to the energy cost, which is expressed in equation 4. The energy cost comes from 2 parts: computation and memory access.

$$E_{total} = N_{effect\_op} \times E_{unit\_op} + N_{mem\_access} \times E_{unit\_mem\_access} \quad (4)$$

The first item in equation 4 is the energy cost for computation. This part is greatly affected by model compression. Model compression methods can reduce the actual number of operations carried out on hardware, N_{effect_op} and simplify the operations to reduce the unit energy cost of a single operation E_{unit_op}. Given an FPGA chip, E_{unit_op} is also affected by its hardware implementation. The second item in equation 4 is the energy cost for memory access. The number of memory access N_{mem_access} is affected by the memory system and scheduling method. The energy for each memory access can be reduced by model compression methods by reducing the bit-width of data.

From the analysis of throughput and energy, we see that neural network accelerator involves software-hardware co-design. In the following sections, we will introduce previous work in both software and hardware level.

4 SOFTWARE DESIGN: MODEL COMPRESSION

As introduced in section 3, the design of energy efficient and high performacne neural network accelerator involves software and hardware co-design. In this section, we investigate the software level network model compression methods. Many researches on this topic have been proposed to reduce the number of weights or reduce the number of bitwidth for the neurons and weights, which helps reudce the computation and storage complexity. But these methods can also sacrifice the model accuracy. The trade-off between model compression and model accuracy loss is discussed in this section.

4.1 Data Quantization

One of the most commonly used method for model compression is the quantization on the weights and neurons. The neurons and weights of a neural network is usually represented by floating point data in common developing frameworks. Recent work try to replace this representation with low-bit fixed-point data or even a small set of trained values. On one hand, using less bits for each neuron or weight helps reduce the bandwidth and storage requirement of the neural network processing system. On the other hand, using a simplified representation reduce the hardware cost for each operation. The benefit on hardware will be discussed in detail in section 5. Two kinds of quantization methods are discussed in this section: linear quantization and non-linear quantization.

4.1.1 Linear Quantization. Linear quantization finds the nearest fixed-point representation of each weight and neuron. The problem of this method is that the dynamic range of floating point data greatly exceeds that for fixed point data. Most of the weights and neurons will suffer from overflow or underflow. Qiu, et al. [37] finds that the dynamic range of the weights and neurons in a single layer is much more limited and differs across different layers. Therefore they assign different fractional bit-widths to the weights and neurons in different layers. To decide the fractional bit-width of a set of data, i.e. the neurons or weights of a layer, the data distribution is first analyzed. A set of possible fractional bit-widths are chosen as candidate solutions. Then the solution with the best model performance on training data set is chosen. In [37], the optimized solution of a network is chosen layer by layer to avoid an exponential design space exploration. Guo, et al. [12] further improves this method by fine tuning the model after the fraction bit-width of all the layers are fixed.

The method of choosing certain fractional bit-width equals to scale the data with a scaling factor of 2^k. Li, et al. [22] scales the weights with trained parameter W^l for each layer and quantize the

weights with 2-bit data, representing W^l, 0 and $-W^l$. The neurons in this work is not quantized. So the the network still implements 32-bit floating point operations. Zhou, et al. [58] further quantize the weights of a layer with only 1 bit to $\pm s$, where $s = E(|w^l|)$ is the expectation of the absolute value of the weights of this layer. Linear quantization is also applied to the neurons in this work.

4.1.2 Non-linear Quantization. Compared with linear quantization, non-linear quantization independently assigns values to different binary code. The translation from a non-linear quantized code to its corresponding value is thus a look-up table. This kind of methods helps further reduce the bit-width used for each neuron or weight. Chen, et al. [6] assign each of the weight to an item in the look-up table by a pre-defined hash function and train the values in look-up table. Han et al. [14] assigns the values in look-up table to the weights by clustering the weights of a trained model. Each look-up table value is set as the cluster center and further fine-tuned with training data set. This method is able to compress the weights of state-of-the-art CNN models to 4-bit without accuracy loss. Zhu, et al. [59] propose the ternary quantized network where all the weights of a layer are quantized to three values: W^n, 0, and W^p. Both the quantized value and the correspondance between weights and look-up table are trained. This method sacrifices less than 2% accuracy loss on ImageNet data set on state-of-the-art network models. The weight bit-width is reduced from 32-bit to 2-bit, which means about 16× model size compression.

4.1.3 Comparison. We compare different quantization methods in Figure 1. The labels describe the experiments as $BW_{weight} \times BW_{neurons}$ and network name. The "(FT)" denotes that the network is fine-tuned after a linear quantization. Comparing different methods on different models is a little bit unfair. But it still gives some insights. For linear quantization, 8-bit is a clear bound to ensure negligible accuracy loss. With 6 or less bits, using fine-tune or even training each weight from the beginning, will cause obvious accuracy degradation.

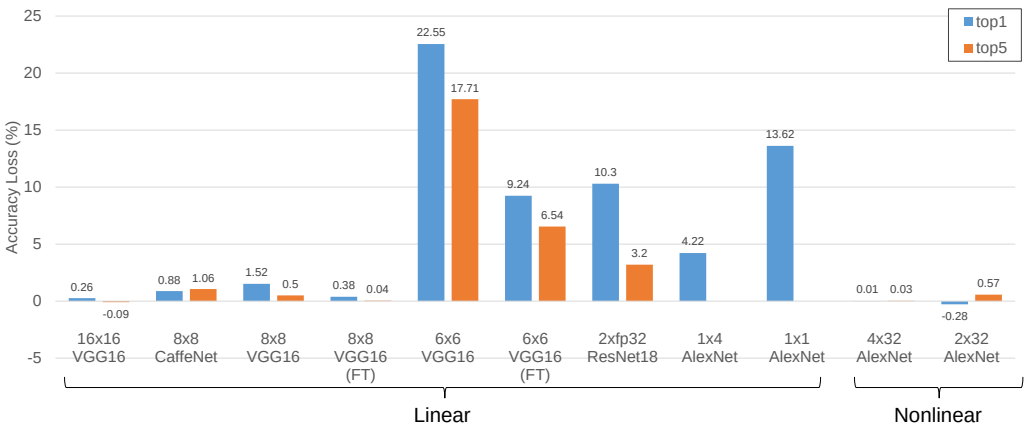

Fig. 1. Comparison between different quantization methods from [12, 14, 22, 37, 58, 59].

4.2 Weight Reduction

Besides narrowing the bit-width of neurons and weights, another method for model compression is to reduce the number of weights. One kind of method is to approximate the weight matrix with a low rank representation. Qiu, et al. [37] compress the weight matrix W of an FC layer with singular value decomposition. An $m \times n$ weight matrix W is replaced by the multiplication of two

matrices $A_{m\times p}B_{p\times n}$. For a sufficiently small p, the total number of weights is reduced. This work compress the largest FC layer of VGG network to 36% of its original size with 0.04% classification accuracy degradation. Zhang, et al. [56] use similar method for convolution layers and takes the effect of the following non-linear layer into the decomposition optimization process. The proposed method achieves 4× speed up on state-of-the-art CNN model targeting at ImageNet, with only 0.9% accuracy loss.

Pruning is another kind of method to reduce the number of weights. This kind of method directly remove the zeros in weights or remove those with small absolute values. The challenge in pruning is how to make more weights zero while keeping the model accuracy. One solution is the application of lasso object function during training. Liu, et al. [25] apply the spase group-lasso object function on the AlexNet [21] model. 90% weights are removed after training with less than 1% accuracy loss. Another solution is to prune the zero weights during training. Han, et al. [14] directly removes the values in network with zero or small absolute value. The left weights are then fine-tuned are training set to recover accuracy. Experimental result on AlexNet show that 89% weights can be removed while keeping the model accuracy.

5 HARDWARE DESIGN: EFFICIENT ARCHITECTURE

In this section, we investigate the hardware level techniques used in state-of-the-art FPGA based neural network accelerator design to achieve high performance and high energy efficiency. We classify the techniques into 3 levels: computation unit level, loop unrolling level, and system level.

5.1 Computation Unit Designs

Computation unit level design affect the peak performance of the neural network accelerator. With a certain FPGA chip, the available resource is limited. A smaller computation unit design means more computation units and higher peak performance. A carefully designed computation unit array can also increase the working frequency of the system and thus improve peak performance.

5.1.1 Low Bit-width Unit. Reduce the number of bit-width for computation is a direct way to reduce the size of computation units. The feasibility of using less bits comes from the quantization methods as introduced in section 4.1. Most of the state-of-the-art FPGA designs replace the 32-bit floating point units with fixed point units. Podili, et al. [35] implements 32-bit fixed point untis for the proposed system. 16-bit fixed point units are widely adopted in [9, 23, 37, 49, 51]. ESE [13] adopts 12-bit fixed-point weight and 16-bit fixed point neurons design. Guo, et al. [12] use 8-bit units for their design on embedded FPGA. Recent work is also focusing on extremely narrow bit-width design. Prost-Boucle, et al. [36] implements 2-bit multiplication with 1 LUT for ternary network. Experiments in [34] shows that FPGA implementation of Binarized Neural Network (BNN) outperforms that on CPU and GPU. Though BNN suffers from accuracy loss, many designs explore the benefit of using 1 bit for computation [20, 24, 29, 31, 32, 44, 57].

The designs mentioned above are focused on computation unit for linear quantization. For non-linear quantization, translating the data back to full precision for computation is still of high cost. Samragh, et al. [39] proposes the factorized coefficients based dot product implementation. As the possible values of weights are quite limited for non-linear quantization, the proposed computation unit accumulates the multiplicator for each possible value and calculate the result as the weighted sum of the values in look-up table. In this way, the multiplication needed for one output neuron is a constant as the number of values in look-up table. The original multiplications are replaced by random addressed accumulations.

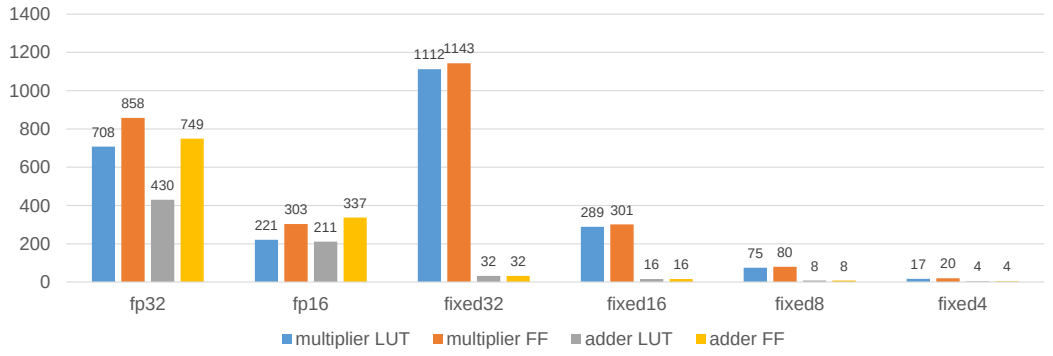

Fig. 2. FPGA resource consumption comparison for multiplier and adder with different types of data.

Most of the designs use a same bit-width through the process of a neural network. Qiu, et al. [37] finds that neurons and weights in FC layers can use less bits compared with CONV layers while the accuracy is maintained. Heterogeneous computation units are used in the designs of [11, 57].

The size of computation units of different bit-width is compared in Figure 2. The resouce consumption is the synthesis result by Vivado 2017.2. All the IPs are required to not use DSP resources. Though we tend to use DSPs in real implementations, this result shows the actual hardware cost. It is also common to implement the computation units in a hybrid way like [37], where some of the operators are implemented by DSP and others by logic.

In neural network,the number of multiplication and addtion is approximately the same. So the sum of resource for multiplier and adder shows the overall cost. Operations with 32-bit fixed point data consumes similar resource as 32-bit floating point operations. For 16-bit operations, using fixed-point format saves about 30% resource. As introduced in section 4.1, 8-bit fixed point is the bound for linear quantization. Comapred with the 32-bit version, this allows 14× more hardware operators within the same area of logic. If further research can utilize 4-bit operations, this advantage becomes 54×.

5.1.2 Fast Convolution Unit. For CONV layers, the convolution operations can be accelerated by special algorithms. Discrete Fourier Transformation (DFT) based fast convolution is widely adopted in digital signal processing. Zhang, et al. [53] propose a 2D DFT based hardware design for efficient CONV layer execution. For an $F \times F$ filter convolved with $K \times K$ filter, DFT converts the $(F-K+1)^2K^2$ multiplications in sapce domain to F^2 complex multiplications in frequency domain. For a CONV layer with M input channel and N output channel, MN times of frequency domain multiplications are needed while only $(M+N)$ times DFT/IDFT are needed. The conversion of convolution kernels is once for all. So the domain conversion process are of low cost for CONV layers. This technique does not work for CONV layers with stride>1 or 1×1 convolution. Ding, et al. [7] suggests that a block-wise circular constraint can be applied on the weight matrix. In this way, the matrix vector multiplication in FC layers are converted to a set of 1D convolutions and can further accelerated in frequency domain. This method can also be applied to CONV layers by treating the $K \times K$ convolution kernels as $K \times K$ matrices and is not limited by K or stride.

Frequency domain methods require complex number multiplication. Another kind of fast convolution involves only real number multiplication [47]. The convolution of a 2D feature map F_{in} with a kernel K using Winograd algorithm is expressed by equation 5.

$$F_{out} = A^T[(GF_{in}G^T) \odot (BF_{in}B^T)]A \tag{5}$$

G, B and A are transformation matrix which only related to the sizes of kernel and feature map. $\odot$ denotes an element-wise multiplication of two matrices. For a 4×4 feature map convolved with 3×3 kernel, the transformation matrices are described as follows:

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 0 & -1 \end{bmatrix}$$

Winograd based methods are also limited by the kernel size and stride as DFT based methods. The most commonly used Winograd transformation is for 3×3 convolution in [27, 49].

5.1.3 DSP Optimization. Recent FPGAs implement hardened DSP units together with the reconfigurable logic to offer a high computation capacity. The basic function of a DSP unit is a multiplication accumulation (MAC). The bit-width for multiplication and addition is fixed. When the bit-width used in neural network does not match that of the DSP units, the FPGA is not fully utilized. The latest DSP units in Altera's FPGA implements 2 18×19 multipliers and can be configured into a 27×27 multiplier or a 32-bit floating point multiplier [1]. That of Xilinx's FPGA implements one 27×18 multiplier [2]. As mentioned in section 5.1.1, many designs adopt multiplication with less or equal than 16 bits, which can cause great DSP under utilization.

Nguyen, et al. [33] propose the design to implement two narrow bit-width fixed-point multiplication with a single wide bit-width fixed-point multiplier. In this design, AB and AC is executed with one multiplication $A(B << k + C)$. If k is sufficiently large, the bits for AB and AC does not overlap in the multiplication result and can be directly seperated. The design in [33] implements two 8-bit multiplications with one 25×18 multiplier, where k is 9. Similar method can be applied on other bit-width and DSPs.

5.1.4 Frequency Optimization Methods. All the above techniques introduced targets at increasing the number of computation units within a certain FPGA. Increasing the working frequency of the computation units also improves the peak performance.

To implement high parallelism, neural network accelerators usually implements matrix-vector multiplication or matrix-matrix multiplications rather than vector inner product as the basic operation. Different computation units share operators. Simply broadcast data to different computation units leads to large fan-out and high routing cost and thus reduce the working frequency. Wei, et al. [46] use the systolic array structure in their design. The shared data are transferred from one computation unit to the next in a chain mode. So the data is not broadcasted and only local connections between different computation units are needed. The drawback is the increase in latency. As the process of a neural network model is determined and the systolic structure is fully pipelined, the latency overhead can be fully covered.

Latest FPGAs support 700-900MHz DSP theoretical peak working frequency. But existing designs usually work at 100-300MHz [12, 28, 37, 51]. As claimed in [48], the working frequency is limited by the routing between on-chip SRAM and DSP units. The design in [48] use different working frequencies for DSP units and surrounding logic. Neighbour slices to each DSP unit are used as local RAMs to separate the clock domain. The prototype design in [48] achieves the peak DSP working frequency at 741MHz and 891MHz on FPGA chips of different speed grades. Yet this method is not adopted by a complete neural network accelerator design.

5.2 Loop Unrolling Strategies

CONV layers and FC layers contribute to most of the computations and storage requirment of a neural network. As introduced in section 2. We express the CONV layer function in equation 2 as

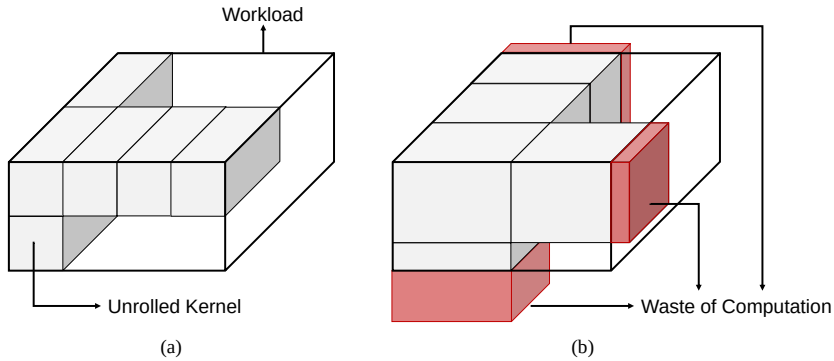

Fig. 3. Comparison between appropriate and inappropriate loop unroll parameters. (a) Appropriate parameters. (b) Inappropriate parameters.

nested loops in Algorithm 1. To make the code clear to read, we merge the loops along x and y directions for feature maps and 2-D convolution kernels respectively. An FC layer can be treated as a CONV layer with feature map and kernel both of size 1×1. Besides the loops in Algorithm 1, we also parallelize the process of multiple inputs as a batch. This forms the batch loop.

Algorithm 1 Convolution Layer

Require: feature map F_{in} of size $M \times Y \times X$; convolution kernel Ker of size $N \times M \times K \times K$; bias vector b of size N
Ensure: feature map F_{out}

```
1: function ConvLayer(F_in, Ker)
2:     Let F_out ← zero array of size N × (Y − K + 1) × (X − K + 1)
3:     for n = 1; n < N; n++ do                                  ▷ Output channel loop
4:         for m = 1; m < M; m++ do                              ▷ Input channel loop
5:             for each (y, x) within (Y − K + 1, X − K + 1) do  ▷ Feature map loop
6:                 for each (ky, kx) within (K, K) do            ▷ Kernel loop
7:                     F_out[n][y][x] += F_in[m][y − ky + 1][x − kx + 1] ∗ K[n][m][ky][kx]
8:         F_out[n] += b[n]
9:     return F_out
```

To parallelize the execution of the loops, we unroll a certain part of the loops and map every operation in this part as a hardware computation unit. An imappropriate set of loop unroll parameter may lead to serious hardware underutilization. We take an example of three nested loops as shown in Figure 3. The big cube denotes all the operations within the loops. The length of each edge denotes the trip count of a loop. The small cube denotes the unrolled kernel, whose edges denote the unrolling parameter. A complete execution of the workload means to fullfill the big cube with small cubes. Figure 3(a) shows an appropriate set of unroll parameters. But for Figure 3(b), the red part of some of the small cubes are out of the big cube, which means the hardware is wasted.

It is obvious from Figure 3 that if the trip count of a loop is too small, the unroll parameter for this loop is limited. For a CNN model, the loop dimension varies greatly among different layeres. For a common network used on ImageNet classification like ResNet [16], the channel numbers vary from 3 to 2048, the feature map sizes vary from 224×224 to 7×7, the convolution kernel

sizes vary from 7×7 to 1×1. Besides the under utilization problem, loop unrolling also affect the datapath and on-chip memory design. Thus loop unrolling strategy is a key feature for a neural network accelerator design.

Various work are proposed focusing on how the unroll parameter should be chosen. Zhang, et al. [52] propose the idea of unrolling the input channel and output channel loops and choose the optimized unroll parameter by design space exploration. Along these two loops, there is no input data cross dependency between neighbouring iterations. So no multiplexer is needed to rounte data from on-chip buffer to computation units. But the parallelism is limited as $7 \times 64 = 448$ multipliers. For larger parallelism, this solution is easy to suffer from the under utilization problem. Ma, et al. [28] further extends the design space by allowing parallelism on the feature map loop. The parallelism reaches $1 \times 16 \times 14 \times 14 = 3136$ multipliers. A shift register structure is used to route feature map pixels to the computation units.

The kernel loop is not chosen in the above work because kernel sizes vary greatly. Motamedi, et al [30] analysis the kernel unrolling on AlexNet. Even with 3×3 unroll for the 11×11 and 5×5 kenrels, the overall system performance still reaches 97.4% of its peak performance for the convolution layers. For certain networks like VGG [42], only 3×3 convolution kernels are used. Qiu, et al. [37] use the line-buffer structure to achieve 3×3 sliding window function and fully parallelize the kernel loop. Another reason to unroll kernel loop is to achieve acceleration with fast convolution algorithms. Design in [53] implements fully parallelized frequency domain multiplication on 4×4 feature map and 3×3 kernel. Lu, et al. [27] implement Winograd algorithm on FPGA with a dedicated pipeline for equation 5. The convolution of 3×3 kernel on 6×6 kernel is fully paralleized.

The above solutions are only for a single layer. But there is hardly a one-size-fits-all solution for a whole network, especially when we need high parallelism. Designs in [23, 26] propose fully pipelined structure with each layer a pipe stage. As each layer is executed with an independent part of hardware and each part is small, loop unrolling method can be easily chosen. But this method is memory consuming because ping-pong buffers are needed between neighbouring layers for the feature maps. Design in [54] is similar but implemented on FPGA clusters to resolve the scalability problem. Shen, et al. [40] group the layers of a CNN by the loops' trip count and map each group onto one hardware module. Actuallly these solutions can be treated as unrolling the batch loop, because different inputs are processed in parallel on different layer pipe stage. The design in [27] implements parallelized batch both within a layer and among different layers. The drawback of batch parallel method is that the latency is higher compared with a $batch = 1$ design of a same parallelism.

Most of the current designs follow one of the above methods for loop unrolling. A special kind of design is for sparse neural network. Han, et al. [13] propose the ESE architecture for sparse LSTM network acceleration. Unlike processing a dense network, all the computation units will not work synchronously. This causes difficulty in sharing data between different computation units. ESE implements only the output channel (the output neurons of the FC layers in LSTM) loop unroll within a layer to simplify hardware design and parallelize batch process.

5.3 System Design

A typical FPGA based neural network accelerator system is shown in Figure 4. The logic part of the whole system are denoted with the blue boxes. The host CPU issues workload or commands to the FPGA logic part and monitors its working status. On the FPGA logic part, a controller is usually implemented to communicate with host and generates control signals to all the other modules on FPGA. The controller can be an FSM or an instrcution decoder. The on the fly logic part is implemented for certain designs if the data loaded from external memory needs preprocess. This

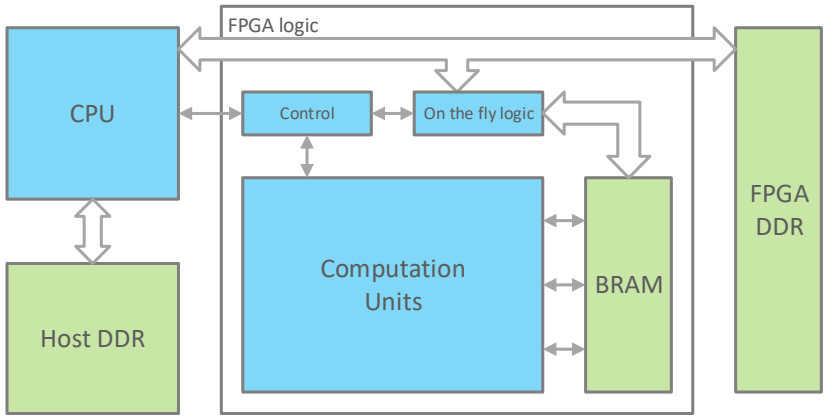

Fig. 4. Block graph of a typical FPGA based neural network accelerator system

module can be data arrangement module, data shifter [37], FFT module [53], etc. The computation units are as discussed in section 5.1 and section 5.2.

The memory hiearchy of the system mainly contains three parts, denoted as the green boxes in Figure 4: Host DDR, FPGA DDR and on-chip block RAM. For state-of-the-art network, the number of weights can reach up to 100M. Using even 8-bit or less bit-width quantization will result in tens of MB storage requirement. Most of the current FPGAs implements less than 10MB on-chip memory and 1-8GB external DDR integrated on board. So for common designs, a two level memory hiearchy is used with DDR and on-chip memory.

From system level, the performance of a neural network accelerator is limited by two factor: the on-chip computation resource and the off-chip memory bandwidth. Various researches have been proposed to achieve the best performance within a certain off-chip memory bandwidth. Zhang, et al. [52] introduce the roofline model in their work to analyze whether a design is memory bounded or computation bounded. An example of a roofline model is shown in Figure 5.

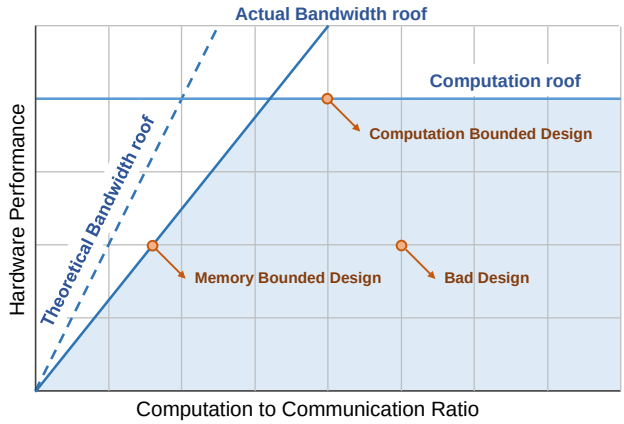

Fig. 5. An example of the roofline model. The shaded part denotes the valid design space given bandwidth and resource limitation.

The figure use the computation to communication (CTC) ratio as x-axis and hardware performance as y-axis. CTC is the number of operations that can be executed with a unit size of memory access. Each hardware design can be treated as a point in the figure. So y/x equals to the bandwidth requirement of the design. Given a certain platform, the availabel bandwidth is limited and can be described as the theoretical bandwidth roof in Figure 5. But the actual bandwidth roof is below the theoretical roof because for DDR access, the achievable bandwidth depends on the data access pattern. Sequential DDR access achieves much higher bandwidth than random access. The other roof is the computation roof, which is limited by the available resource on FPGA.

So a higher CTC ratio means the hardware is more likely to achieve the computation bound. Increasing the CTC ratio also reduce DDR access, which greatly reduce the energy cost according to [17]. In section 5.2, we have discussed the loop unrolling strategies to increase the parallelism while reducing the waste of computation for a certain network. When the loop unrolling strategy is decided, the scheduling of the rest part of the loops decides how the hardware can reuse data with on-chip buffer. This involves loop tiling and loop interchange strategy.

Loop tiling is a higher level of loop unrolling. All the input data of a loop tile will be stored on-chip and the loop unrolling hardware kernel works on these data. A larger loop tile size means that each tile will be loaded from external memory to on-chip memory less times. Loop interchange strategy decides the processing order of the loop tiles. External memory access happens when the hardware is moving from one tile to the next. Neighbouring tile may share a part of data. For example in a CONV layer, neighbouring tile can share input feature map or the weights. This is decided by the execution order of the loops.

In [28, 52], design space exploration is done on all the possible loop tiling sizes and loop orders. Many designs also explores the design space though some of the loop unrolling, tiling and loop order is already decided [30, 37]. Shen, et al. [41] also discuss the affect of batch parallelism over the CTC for different layers. This is a loop dimension not focused on in previous work.

All the above work give one optimized loop unrolling strategy and loop order for a whole network. Guo, et al. [12] implements flexible unrolling and loop order configuration for different layers with an instruction interface. The data arrangement in on-chip buffers are controlled through instructions to fit with different feature map sizes. This means the hardware can always fully utilize the on-chip buffer to use the largest tiling size according to on-chip buffer size. This work also propose the "back and forth" loop execution order to avoid total on-chip data refresh when a innermost loop finishes.

Alwani, et al. [4] address the external memory access problem by fusing two neighbouring layers together to avoid the intermediate result transfer between the two layers. This strategy also helps the CPU based program to increase cache hit rate.

6 PERFORMANCE EVALUTATION

REFERENCES

[1] [n. d.]. https://www.altera.com/products/fpga/stratix-series/stratix-10/features.html. ([n. d.]). Accessed Dec 7, 2017.
[2] [n. d.]. https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf. ([n. d.]). Accessed Dec 7, 2017.
[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).
[4] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.
[5] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C Ling, and Gordon R Chiu. 2017. An OpenCL (TM) Deep Learning Accelerator on Arria 10. *arXiv preprint arXiv:1701.03534* (2017).

Table 1. Performance and resource utilization of state-of-the-art neural network accelerator designs

	Data Format	Perf. (GOP/s)	Power (W)	Efficiency (GOP/J)	Resource(%)			FPGA chip
					DSP	logic	BRAM	
[5]	FP16	1382	45	30.7	97	58	92	Arria 10 GX1150
[13]	INT16/12	2520	41	61.5	54.4	88.6	87.7	XCKU060
[45]	INT16	12.73	1.75	7.27	94.54	66.64	6.07	XC7Z020
[53]	FP32	123.5	13.18	9.37	87.5	85.4	64	Stratix V
[55]	FP32/INT16	1790	37.46	47.8	86	43	46	GX1150
[28]	INT16-8	645.25	21.2	30.43	100	38	70	GX1150
[37]	INT16	136.97	9.63	14.22	89.2	83.5	86.7	XC7Z045
[43]	INT16-8	117.8	19.1	6.2	12.5	22	65.2	5SGSD8
[52]	FP32	61.62	18.61	3.3	80	61.3	50	XC7VX485T
[9]	INT16	364.4	25	14.6	65	25	46	5SGSMD5
[27]	INT16	2940.7	23.6	124.6	-	-	-	ZCU102
[35]	INT32	229	8.04	28.5	100	83.7	17.6	Stratix V
[31]	1bit	329.47	2.3	143.2	0.5	34.4	11.4	Zynq XC7Z020
[20]	2bit	410.22	2.26	181.51	40.5	82.7	37.7	Zynq XC7Z020
[29]	1bit	40770	48	849.38	-	-	-	GX1155
[23]	INT16	565.94	30.2	22.15	59.56	63.21	65.07	XC7VX690T
[26]	INT16-8	222.1	24.8	8.96	39.9	26.6	39.7	XC7VX690T
[54]	INT16	1280.3	160	8	-	-	-	XC7Z020+ XC7VX690T×6
[12]	INT8	84.3	3.5	24.1	87	84	89	XC7Z045
[49]	INT16	229.5	9.4	24.42	91.9	71	83.2	XC7Z045
[10]	FP32	7.26	19.63	0.37	42	65.31	52.04	XC7VX485T
[51]	INT16	354	26	13.6	78	81	42	XC7VX690T

[6] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. 2015. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*. 2285–2294.

[7] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, et al. 2017. CirCNN: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 395–408.

[8] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 580–587.

[9] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. (2017), 152–159.

[10] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. 2017. FPGA-based accelerator for long short-term memory recurrent neural networks. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 629–634.

[11] Jianxin Guo, Shouyi Yin, Peng Ouyang, Leibo Liu, and Shaojun Wei. 2017. Bit-Width Based Resource Partitioning for CNN Acceleration on FPGA. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 31–31.

[12] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2017. Angel-Eye: A Complete Design Flow for Mapping CNN onto Embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017).

[13] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA.. In *FPGA*. 75–84.

[14] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
[15] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. 2014. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (2014).
[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[17] M. Horowitz. [n. d.]. Energy table for 45nm process, Stanford VLSI wiki.[Online]. https://sites.google.com/site/seecproject. ([n. d.]).
[18] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).
[19] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).
[20] Li Jiao, Cheng Luo, Wei Cao, Xuegong Zhou, and Lingli Wang. 2017. Accelerating low bit-width convolutional neural networks with embedded FPGA. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 1–4.
[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
[22] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary weight networks. *arXiv preprint arXiv:1605.04711* (2016).
[23] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 1–9.
[24] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. 2017. A 7.663-TOPS 8.2-W Energy-efficient FPGA Accelerator for Binary Convolutional Neural Networks. *arXiv preprint arXiv:1702.06392* (2017).
[25] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. 2015. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 806–814.
[26] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. 2016. Automatic code generation of convolutional neural networks in FPGA implementation. In *Field-Programmable Technology (FPT), 2016 International Conference on*. IEEE, 61–68.
[27] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. 2017. Evaluating fast algorithms for convolutional neural networks on fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 101–108.
[28] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 45–54.
[29] Duncan JM Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip HW Leong. 2017. High performance binary neural networks on the Xeon+ FPGAâĎć platform. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 1–4.
[30] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. 2016. Design space exploration of fpga-based deep convolutional neural networks. In *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*. IEEE, 575–580.
[31] Hiroki Nakahara, Tomoya Fujii, and Shimpei Sato. 2017. A fully connected layer elimination for a binarizec convolutional neural network on an FPGA. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 1–4.
[32] Hiroki Nakahara, Haruyoshi Yonekawa, Hisashi Iwamoto, and Masato Motomura. 2017. A Batch Normalization Free Binarized Convolutional Deep Neural Network on an FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 290–290.
[33] Dong Nguyen, Daewoo Kim, and Jongeun Lee. 2017. Double MAC: Doubling the performance of convolutional neural networks on modern FPGAs. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 890–893.
[34] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. 2016. Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC. In *Field-Programmable Technology (FPT), 2016 International Conference on*. IEEE, 77–84.
[35] Abhinav Podili, Chi Zhang, and Viktor Prasanna. 2017. Fast and efficient implementation of Convolutional Neural Networks on FPGA. In *Application-specific Systems, Architectures and Processors (ASAP), 2017 IEEE 28th International*

Conference on. IEEE, 11–18.

[36] Adrien Prost-Boucle, Alban Bourge, Frédéric Pétrot, Hande Alemdar, Nicholas Caldwell, and Vincent Leroy. 2017. Scalable high-performance architecture for convolutional ternary neural networks on FPGA. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 1–7.

[37] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. 2016. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 26–35.

[38] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[39] Mohammad Samragh, Mohammad Ghasemzadeh, and Farinaz Koushanfar. 2017. Customizing neural networks for efficient fpga implementation. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 85–92.

[40] Yongming Shen, Michael Ferdman, and Peter Milder. 2016. Overcoming resource underutilization in spatial CNN accelerators. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 1–4.

[41] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Escher: A CNN Accelerator with Flexible Buffering to Minimize Off-Chip Transfer. In *Proceedings of the 25th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCMâĂŹ17). IEEE Computer Society, Los Alamitos, CA, USA*.

[42] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[43] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. 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*. ACM, 16–25.

[44] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 65–74.

[45] Stylianos I Venieris and Christos-Savvas Bouganis. 2017. fpgaConvNet: Automated Mapping of Convolutional Neural Networks on FPGAs. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 291–292.

[46] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 29.

[47] Shmuel Winograd. 1980. *Arithmetic complexity of computations*. Vol. 33. Siam.

[48] Ephrem Wu, Xiaoqian Zhang, David Berman, and Inkeun Cho. 2017. A high-throughput reconfigurable processing array for neural networks. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 1–4.

[49] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. 2017. Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 62.

[50] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. 2015. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853* (2015).

[51] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*. IEEE, 1–8.

[52] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.

[53] Chi Zhang and Viktor Prasanna. 2017. Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 35–44.

[54] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. 2016. Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ACM, 326–331.

[55] Jialiang Zhang and Jing Li. 2017. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network.. In *FPGA*. 25–34.

[56] Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, and Jian Sun. 2015. Efficient and accurate approximations of nonlinear convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1984–1992.

[57] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani B Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs.. In *FPGA*. 15–24.

[58] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).

[59] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. 2016. Trained ternary quantization. *arXiv preprint arXiv:1612.01064* (2016).