# [DL] A Survey of FPGA-based Neural Network Inference Accelerators

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

Recent research on neural networks has shown a significant advantage in machine learning over traditional algorithms based on handcrafted features and models. Neural networks are now widely adopted in regions like image, speech, and video recognition. But the high computation and storage complexity of neural network inference poses great difficulty on its application. It is difficult for CPU platforms to offer enough computation capacity. GPU platforms are the first choice for neural network processes because of its high computation capacity and easy-to-use development frameworks.

However, FPGA-based neural network inference accelerator is becoming a research topic. With specifically designed hardware, FPGA is the next possible solution to surpass GPU in speed and energy efficiency. Various FPGA-based accelerator designs have been proposed with software and hardware optimization techniques to achieve high speed and energy efficiency. In this article, we give an overview of previous work on neural network inference accelerators based on FPGA and summarize the main techniques used. An investigation from software to hardware, from circuit level to system level is carried out to complete analysis of FPGA-based neural network inference accelerator design and serves as a guide to future work.

CCS Concepts: • **General and reference → Surveys and overviews**; • **Computer systems organization** → *Parallel architectures*;

Additional Key Words and Phrases: FPGA architecture, neural network, parallel processing

**ACM Reference format:**
Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. 2019. [DL] A Survey of FPGA-based Neural Network Inference Accelerators. *ACM Trans. Reconfigurable Technol. Syst.* 12, 1, Article 2 (March 2019), 26 pages.
https://doi.org/10.1145/3289185

## 1 INTRODUCTION

Recent research on Neural Networks (NN) is showing great improvement over traditional algorithms in machine learning. Various network models, like convolutional neural network (CNN) and recurrent neural network (RNN), have been proposed for image, video, and speech processes. CNN [28] improved the top-5 image classification accuracy on the ImageNet [50] dataset from 73.8% to 84.7% in 2012 and further helps improve object detection [13] with its outstanding ability

This work was supported by National Key R&D Program of China (2018YFB0105005, 2017YFA0207600), National Natural Science Foundation of China (No. 61622403, 61621091), DeePhi Technology and Xilinx.

Authors' addresses: K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, Tsinghua University, 30 Shuangqing Rd, Haidian District, Beijing, Beijing, 100084, China; emails: {gky15, zengsl18, yjc16}@mails.tsinghua.edu.cn, yu-wang@mail.tsinghua.edu.cn, yanghz@tsinghua.edu.cn.

1936-7406/2019/03-ART2 $15.00
https://doi.org/10.1145/3289185

Table 1. Performance and Resource Utilization of State-of-the-Art Neural Network Accelerator Designs

| | AlexNet [28] | VGG19 [56] | ResNet152 [22] | MobileNet [24] | ShuffleNet [79] |
|---|---|---|---|---|---|
| Year | 2012 | 2014 | 2016 | 2017 | 2017 |
| # Param | 60M | 144M | 57M | 4.2M | 2.36M |
| # Operation | 1.4G | 39G | 22.6G | 1.1G | 0.27G |
| Top-1 Accuracy | 61.0% | 74.5% | 79.3% | 70.6% | 67.6% |

in feature extraction. RNN [21] 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 ability makes NN a promising candidate for many artificial intelligence applications.

But the computation and storage complexity of NN models are high. In Table 1, we list the number of operations, number of parameters (add or multiplication), and top-1 accuracy on the ImageNet dataset [50] of state-of-the-art CNN models. Take CNN as an example. The largest CNN model for a 224 × 224 image classification requires up to 39 billion floating point operations (FLOP) and more than 500MB model parameters [56]. As the computation complexity is proportional to the input image size, processing images with higher resolutions may need more than 100 billion operations. Latest work like MobileNet [24] and ShuffleNet [79] are trying to reduce the network size with advanced network structures but with obvious accuracy loss. The balance between the size of NN models and accuracy is still an open question today. In some cases, the large model size hinders the application of NN, especially in power-limited or latency critical scenarios.

Therefore, choosing a proper computation platform for neural-network-based applications is essential. A typical CPU can perform 10-100GFLOP per second, and the power efficiency is usually below 1GOP/J. So it is difficult for CPUs to meet the high performance requirements in cloud applications or the low power requirements in mobile applications. In contrast, GPUs offer up to 10TOP/s peak performance and are good choices for high-performance neural network applications. Development frameworks like Caffe [26] and Tensorflow [4] also offer easy-to-use interfaces, which makes GPU the first choice of neural network acceleration.

Besides CPUs and GPUs, FPGAs are becoming a platform candidate to achieve energy-efficient neural network processing. With a neural network-oriented hardware design, FPGAs can implement high parallelism and make use of the properties of neural network computation to remove additional logic. Algorithm researches also show that an NN model can be simplified in a hardware-friendly way while not hurting the model accuracy. Therefore FPGAs are possible to achieve higher energy efficiency compared with CPU and GPU.

FPGA-based accelerator designs are faced with two challenges in performance and flexibility:

- 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. It is difficult to achieve high performance and high energy efficiency with a straightforward design on FPGA.
- Implementations of neural networks on FPGAs is much harder than on CPUs or GPUs. Development frameworks like Caffe and Tensorflow for CPU and GPU are absent for FPGA.

Many designs addressing the above two problems have been carried out to implement energy efficient and flexible FPGA-based neural network accelerators. In this article, we summarize the techniques proposed in these work from the following aspects:

- We first give a simple model of FPGA-based neural network accelerator performance to analyze the methodology in energy efficient design.

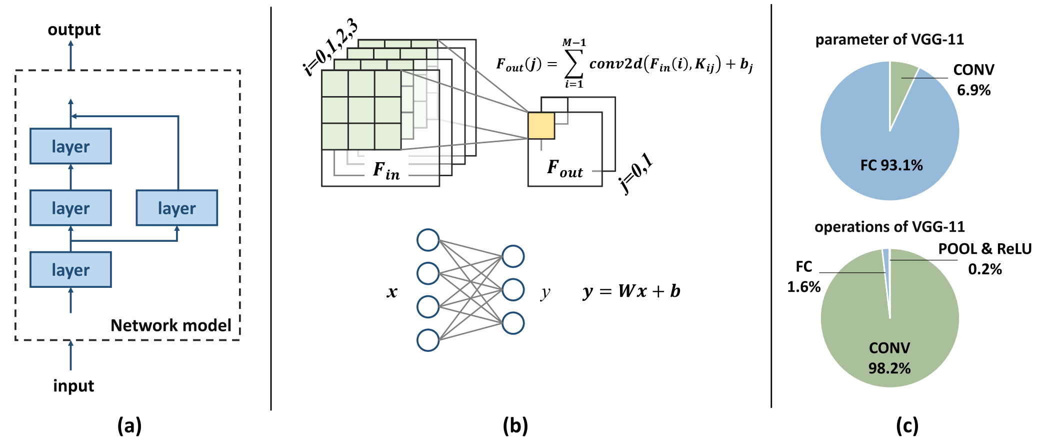

Fig. 1. (a) Computation graph of a neural network model. (b) CONV and FC layers in an NN model. (c) CONV and FC layers dominate the computation and parameter of a typical NN model: VGG11.

- We investigate current technologies for high performance and energy-efficient neural network accelerator designs. We introduce the techniques on both the software and hardware levels and estimate the effect of these techniques.
- We compare state-of-the-art neural network accelerator designs to evaluate the techniques introduced and estimate the achievable performance of FPGA-based accelerator design, which is at least 10× better energy efficient than current GPUs.
- We investigate state-of-the-art automatic design methods of FPGA-based neural network accelerators.

The rest part of this article is organized as follows: Section 2 introduces the basic operations of neural networks and the background of FPGA-based NN accelerator. In Section 3, we analyze the design target of NN accelerators and corresponding methods. Section 4 and Section 5 review the techniques in NN model compression and accelerator design, respectively. Section 6 compares existing designs and evaluates the techniques. Section 8 introduces the methods for a flexible accelerator design. Section 9 concludes this article.

## 2 PRELIMINARY

Before discussing the system design for neural network acceleration, we first introduce the basic concepts of neural networks and the typical structure of FPGA-based NN accelerator design.

### 2.1 Neural Network

In this section, we introduce the basic functions in a neural network. In this article, we only focus on the inference of NN, which means using a trained model to predict or classify new data. The training process of NN is not discussed in this article. A neural network model can be expressed as a directed graph shown in Figure 1(a). Each vertex of the graph denotes a layer that conducts operations on data from a previous layer or input and generates results to the next layer or output. We refer the parameter of each layer as weights and the input/output of each layer as activations through this article.

Convolution (CONV) layers and fully connected (FC) layers are two common types of layers in NN models. The functions of these two layers are shown in Figure 1(b). CONV layers conduct two-dimensional (2D) convolutions on a set of input feature maps $F_{in}$ and add the results to get

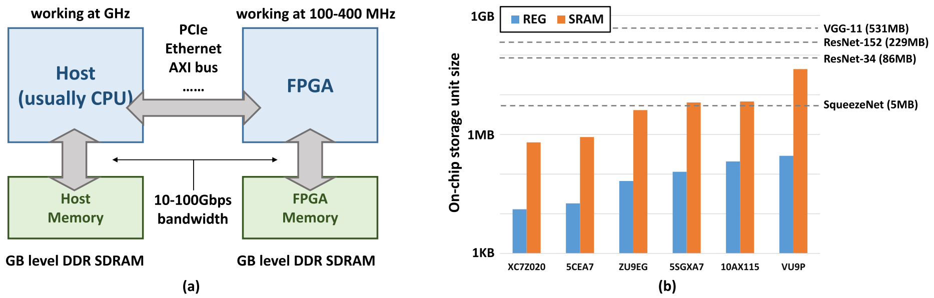

Fig. 2. (a) A typical structure of an FPGA-based NN accelerator. (b) Gap between NN model size and the storage unit size on FPGAs. The bar chart compares the register and SRAM sizes on FPGA chips in different scales. The dotted line denotes the parameter sizes of different NN models with 32-bit floating point parameters.

output feature maps $F_{out}$. FC layers receive a feature vector as input and conduct matrix-vector multiplications.

Besides CONV and FC layers, NN layers also have pooling, ReLU [28], concat [58], element-wise [22], and other types of layers. But these layers contributes little to the computation and storage requirement of a neural network model. Figure 1(c) shows the distribution of weights and operations in the VGG-11 model [56]. In this model, CONV and FC layers together contribute more than 99% of the network's weights and operations, which is similar to most of the CNN models. Compared with CNN, RNN models [6, 21] usually have no CONV layers and only FC layers contribute to most of the computation and storage. So most of the neural network acceleration systems focus on these two types of layers.

## 2.2 FPGA-based Accelerator

In recent years, FPGA has become a promising solution for algorithm acceleration. Compared with CPU, GPU, and DSP platforms, for which the software and hardware are designed independently, FPGA enables the developers to implement only the necessary logic in hardware according to the target algorithm. By eliminating the redundancy in general hardware platforms, FPGAs can achieve higher efficiency. Application-specific integrated circuit– (ASICs) based solutions achieve even higher efficiency but require much longer development cycle and higher cost.

For FPGA-based neural network accelerator, a typical architecture of the system is shown in Figure 2(a). The system usually consists of a CPU host and an FPGA part. A pure FPGA chip usually works with a host PC/server through PCIe connections. SoC platforms (like the Xilinx Zynq Series) and Intel HARPv2 [18] platform integrate the host and the FPGA in the same chip or package. Both the host and the FPGA can work with their own external memory and access each others' memory through the connection. Most of the designs implement NN accelerator on the FPGA part and control the accelerator with the software on the host.

Typical FPGA chips consist of large on-chip storage units like registers and Static Random-Access Memory (SRAM) but are still too small compared with NN models as shown in Figure 2(b). Common models implement 100–1000MB parameters while the largest available FPGA chip implements <50MB on-chip SRAM. This gap requires that external memory like DDR SDRAM is needed. The bandwidth and power consumption of DDR limits the system performance.

The computation capacity of FPGA is relatively higher. Common FPGAs implement hundreds to thousands of DSP units, each of which can compute 18 × 27 or 18 × 19, achieving up to 10TFLOP/s (floating point operations per second) on the largest FPGAs. But for low-end FPGAs like Xilinx

Table 2. List of Symbols

| Symbol | Description | Unit |
|---|---|---|
| $IPS$ | Throughput of the system, measured by the number of inference processed each second | $s^{-1}$ |
| $W$ | Workload for each inference, measured by the number of operations* in the network, mainly addition and multiplication for neural network. | GOP |
| $OPS_{peak}$ | Peak performance of the accelerator, measured by the maximum number of operations can be processed each second. | GOP/s |
| $OPS_{act}$ | Run-time performance of the accelerator, measured by the number of operations processed each second. | GOP/s |
| $\eta$ | Utilization ratio of the computation units, measured by the average ratio of working computation units in all the computation units during each inference. | — |
| $f$ | Working frequency of the computation units. | GHz |
| $P$ | Number of computation units in the hardware design. | — |
| $L$ | Latency for processing each inference | s |
| $C$ | Concurrency of the accelerator, measured by the number of inference processed in parallel | — |
| $Eff$ | Energy efficiency of the system, measured by the number of operations can be processed within unit energy. | GOP/J |
| $E_{total}$ | Total system energy cost for each inference. | J |
| $E_{static}$ | Static energy cost of the system for each inference. | J |
| $E_{op}$ | Average energy cost for each operation in each inference. | J |
| $N_{x\_acc}$ | Number of bytes accessed from memory ($x$ can be SRAM or DRAM). | byte |
| $E_{x\_acc}$ | Energy for accessing each byte from memory($x$ can be SRAM or DRAM). | J/byte |

*Each addition or multiplication is counted as 1 operation.

XC7Z020, this number is reduced to 20GFLOP/s, which is hard to support real-time video processing for applications on mobile platforms.

Even faced with the above challenges, researchers have proposed a series of optimization methods from algorithm to architecture to design high-performance NN accelerators on FPGA, which will be discussed in the following sections of this article.

## 3 DESIGN METHODOLOGY AND CRITERIA

Before going into the details of the techniques used for neural network accelerators, we first give an overview of the design methodology. In general, the design target of a neural network inference accelerator includes the following two aspects: high speed (high throughput and low latency) and high energy efficiency. The symbols used in this section are listed in Table 2.

**Speed.** The throughput of an NN accelerator can be expressed by Equation (1). The on-chip resource for a certain FPGA chip is limited. We can increase the peak performance by (1) increasing the number of computation units $P$ by reducing the size of each computation unit and (2) increasing the working frequency $f$. Reducing the size of computation units can be achieved by sacrificing data precision, which may hurt the model accuracy and requires hardware-software co-design. However, increasing working frequency is pure hardware design work. Corresponding techniques on software models and hardware are introduced in Sections 4 and 5, respectively. A high utilization ratio $\eta$ is ensured by reasonable parallelism implementation and efficient memory system. The property of the target model, i.e., the data access pattern or data-computation ratio, also affect if the hardware can be fully utilized at runtime. But most of the previous work targeting higher utilization ratio focus on the hardware side.

$$IPS = \frac{OPS_{act}}{W} = \frac{OPS_{peak} \times \eta}{W} = \frac{fP \times \eta}{W} \tag{1}$$

Most of the FPGA-based NN accelerators compute different inputs one by one. Some designs process different inputs in parallel. So the latency of the accelerator is expressed as Equation (2). Common concurrent design includes layer pipeline and batch processing. This is usually considered together with loop unrolling and will be introduced in Section 5.2. In this article, we focus more on optimizing the throughput. As different accelerators may be evaluated on different NN models, a common criterion of speed is the $OPS_{act}$, which eliminates the effect of different network models to some extent.

$$L = \frac{C}{IPS} \tag{2}$$

**Energy Efficiency.** Energy efficiency (*Eff*) is another critical criterion to computing systems. For neural network inference accelerators, energy efficiency is defined as Equation (3). Like throughput, we count the number of operations rather than the number of inference to eliminate the difference of workload $W$. If the workload for the target network is fixed, then increasing the energy efficiency of a neural network accelerator means to reduce the total energy cost, $E_{total}$ to process each input.

$$Eff = \frac{W}{E_{total}} \tag{3}$$

$$E_{total} \approx W \times E_{op} + N_{SRAM\_acc} \times E_{SRAM\_acc} + N_{DRAM\_acc} \times E_{DRAM\_acc} + E_{static} \tag{4}$$

The total energy cost mainly comes from two parts: computation and memory access, which is expressed in Equation (4). The first item in Equation (4) is the dynamic energy cost for computation. Given a certain network, the workload $W$ is fixed. Researchers have been focusing on optimizing the NN models by quantization (narrowing the bit-width used for computation) to reduce $E_{op}$ or sparsification (setting more weights to zeros) to skip the multiplications with these zeros to reduce $N_{op}$, which follows similar rules as for throughput optimization.

The second and third items in Equation (4) are the dynamic energy cost for memory access. As shown in Section 2.2, an FPGA-based NN accelerator usually works with an external DRAM. We separate the memory access energy into the DRAM part and the SRAM part. $N_{x\_acc}$ can be reduced by quantization, sparsification, efficient on-chip memory system, and scheduling method. Thus these methods help reduce dynamic memory energy. Corresponding methods will be introduced in Section 5.3. The unit energy $E_{x\_acc}$ can hardly be reduced given a certain FPGA platform.

The fourth item $E_{static}$ denotes the static energy cost of the system. This energy cost can hardly be improved given the FPGA chip and the scale of the design.

From the analysis of speed and energy, we see that neural network accelerator involves both optimizations on NN models and hardware. In the following sections, we will introduce previous work in these two aspects, respectively.

## 4 HARDWARE ORIENTED MODEL COMPRESSION

As introduced in Section 3, the design of energy efficient and fast neural network accelerator can benefit from the optimization of NN models. A larger NN model usually results in higher model accuracy. This means it is possible to trade the model accuracy for the hardware speed or energy cost. Neural network researchers are designing more efficient network models from AlexNet [28] to ResNet [22], SqueezeNet [25], and MobileNet [24]. The latest work tries to directly optimize the processing latency by searching a good network structure [59] or skipping some layers at runtime to save computation [65]. Within these methods, the main differences between the handcrafted/generated networks are the size of and the connections between each layer. The basic operations are the same and the differences hardly affect the hardware design. For this reason, we will not focus on these techniques in this article. But designers should consider using these techniques to optimize the target network.

Other methods try to achieve the tradeoff by compressing existing NN models. They try to reduce the number of weights or reduce the number of bits used for each activation or weight, which helps lower the computation and storage complexity. Corresponding hardware designs can benefit from these NN model compression methods. In this section, we investigate these hardware oriented network model compression methods.

## 4.1 Data Quantization

One of the most commonly used methods for model compression is the quantization of the weights and activations. The activations and weights of a neural network are usually represented by floating point data in common developing frameworks. Recent work tries to replace this representation with low-bit fixed-point data or even a small set of trained values. On the one hand, using fewer bits for each activation 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 of 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 activation. The problem with this method is that the dynamic range of floating-point data greatly exceeds that for fixed-point data. Most of the weights and activations will suffer from overflow or underflow. Qiu et al. [49] finds that the dynamic range of the weights and activations in a single layer is much more limited and differs across different layers. Therefore they assign different fractional bit-widths to the weights and activations in different layers. To decide the fractional bit-width of a set of data, i.e., the activations 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 Ref. [49], the optimized solution of a network is chosen layer by layer to avoid an exponential design space exploration. Wang et al. [64] try to use large bit-width for only the first and last layer and quantize the middle layers to ternary or binary. The method needs to increase the network size to keep high accuracy but still brings hardware performance improvement. Guo et al. [17] choose to fine-tune the model after the fractional bit-width of all the layers are fixed.

The method of choosing a fractional bit-width equals to scale the data with a scaling factor of $2^k$. Li et al. [29] scales the weights with trained parameter $W^l$ for each layer and quantizes the weights with 2-bit data, representing $W^l$, 0, and $-W^l$. The activations in this work are not quantized. So the network still implements 32-bit floating point operations. Zhou et al. [82] 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 activations in this work.

*4.1.2 Non-linear Quantization.* Compared with linear quantization, non-linear quantization independently assigns values to different binary codes. 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 activation or weight. Chen et al. [9] 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 tables. Han

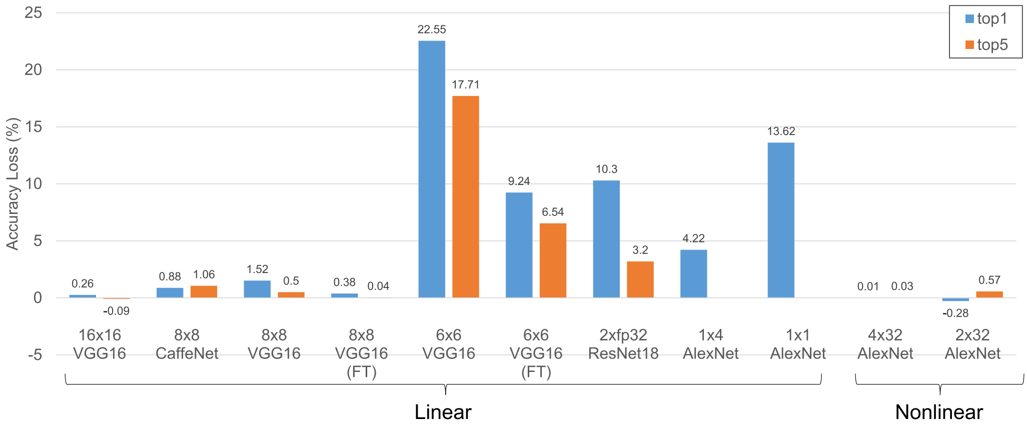

Fig. 3. Comparison between different quantization methods from References [17, 20, 29, 49, 82, 83]. The quantization configuration is expressed as (weight bit-width)×(activation bit-width). The "(FT)" denotes that the network is fine-tuned after a linear quantization.

et al. [20] assign the values in look-up tables to the weights by clustering the weights of a trained model. Each look-up table value is set as the cluster centre and further fine-tuned with training data set. This method can compress the weights of state-of-the-art CNN models to 4-bit without accuracy loss. Zhu et al. [83] 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 correspondence between weights and look-up table are trained. This method sacrifices less than 2% accuracy loss on ImageNet dataset on state-of-the-art network models. The weight bit-width is reduced from 32 bits to 2 bits, which means about 16× model size compression.

*4.1.3 Comparison.* We compare some typical quantization methods from References [17, 20, 29, 49, 82, 83] in Figure 3. All the quantization results are tested on ImageNet data set and the absolute accuracy loss compared with corresponding baseline floating point models is recorded. 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 fewer bits, using fine-tune or even training each weight from the beginning will cause noticeable accuracy degradation. If we require that 1% accuracy loss is within the acceptable range, then linear quantization with at least $8 \times 8$ configuration and the listed non-linear quantization are available. We will further discuss the performance gain of quantization in Section 5.

### 4.2 Weight Reduction

Besides narrowing the bit-width of activations 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. [49] 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 compresses the largest FC layer of VGG network to 36% of its original size with 0.04% classification accuracy degradation. Zhang et al. [80] use a 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 removes the zeros in weights or removes those with small absolute values. The challenge in pruning is the tradeoff between the ratio of zero weights and the model accuracy. One solution is application of the lasso method, which applies L1 normalization to the weights during training. Liu et al. [33] apply the sparse group-lasso method on the AlexNet [28] 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. [20] directly remove the weights of a network that are zero or have small absolute value. The left weights are then fine-tuned with the training dataset to recover accuracy. Experimental results on AlexNet show that 89% weights can be removed while keeping the model accuracy.

The hardware gain from weight reduction is the reciprocal of the compression ratio. According to the above results, the potential speed improvement from weight reduction is up to 10×.

## 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 three levels: computation unit level, loop unrolling level, and system level.

### 5.1 Computation Unit Designs

Computation unit level design affects the peak performance of the neural network accelerator. The available resource of an FPGA chip 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 Computation Unit.* Reduce the number of bit-width for computation is a direct way to reduce the size of computation units. The feasibility of using fewer 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. [47] implement 32-bit fixed-point units for the proposed system. 16-bit fixed-point units are widely adopted in References [14, 30, 49, 70, 73]. ESE [19] adopts 12-bit fixed-point weight and 16-bit fixed-point neurons design. Guo et al. [17] 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. [48] implements 2-bit multiplication with 1 LUT for ternary networks. Experiments in References [46] show 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 data for computation [12, 27, 31, 41, 43, 44, 60, 71, 81].

The designs mentioned above focus on computation units for linear quantization. For non-linear quantization, translating the data back to full precision for computation still costs many resources. Samragh et al. [51] propose 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 multipliers for each possible weight value and calculate the result as the weighted sum of the values in look-up tables. In this way, the multiplication needed for one output neuron equals to the number of values in look-up table. The multiplications are replaced by random-addressed accumulations.

Most of the designs use one bit-width through the process of a neural network. Qiu et al. [49] finds that neurons and weights in FC layers can use fewer bits compared with CONV layers while

Table 3. FPGA Resource Consumption Comparison for Multiplier and Adder with Different Types of Data

| | Xilinx Logic | | | | Xilinx DSP | | | Altera DSP | |
|---|---|---|---|---|---|---|---|---|---|
| | Multiplier | | Adder | | Multiply and add | | | Multiply and add | |
| | LUT | FF | LUT | FF | LUT | FF | DSP | ALM | DSP |
| fp32 | 708 | 858 | 430 | 749 | 800 | 1284 | 2 | 1 | 1 |
| fp16 | 221 | 303 | 211 | 337 | 451 | 686 | 1 | 213 | 1 |
| fixed32 | 1112 | 1143 | 32 | 32 | 111 | 64 | 4 | 64 | 3 |
| fixed16 | 289 | 301 | 16 | 16 | 0 | 0 | 1 | 0 | 1 |
| fixed8 | 75 | 80 | 8 | 8 | 0 | 0 | 1 | 0 | 1 |
| fixed4 | 17 | 20 | 4 | 4 | 0 | 0 | 1 | 0 | 1 |

the accuracy is maintained. Heterogeneous computation units are used in the designs of References [16, 81].

The size of computation units of different bit-widths is compared in Table 3. Three kinds of implementations are tested: separate multiplier and adder with logic resource on Xilinx FPGA, multiply-add function with DSP units on Xilinx FPGA, and multiply-add function with DSP units on Altera FPGA. The resource consumption is the synthesis result by Vivado 2018.1 targeting Xilinx XCKU060 FPGA and Quartus Prime 16.0 targeting Altera Arria 10 GX1150 FPGA. The pure logic modules and the floating-point multiply and add modules are generated with IP core. The fixed-point multiply and add modules are implemented with $A * B + C$ in Verilog and automatically mapped to DSP by Vivado/Quartus.

We first give an overview of the size of the computation units by logic-only implementations. By compressing the weights and activations from 32-bit floating-point number to 8-bit fixed-point number, the multiplier and the adder are scaled down to about 1/10 and 1/50, respectively. Using 4-bit or smaller operators can bring further advantage but also incur significant accuracy loss as introduced in Section 4.1.

Recent FPGAs consist of a large number of DSP units, each of which implements hard multiplier, pre-adder and accumulator core. The basic pattern of NN computation, multiplication and sum, also fits into this design. So we also test the multiply and add function implemented with DSP units. Because of the different DSP architectures, we test on both Xilinx and Altera platforms. Compared with the 32-bit floating-point function, fixed-point functions with narrow bit-width still shows an advantage in resource consumption. But for Altera FPGA, this advantage is not obvious, because the DSP units natively support floating-point operations.

Fixed-point functions with 16-or-less-bit fixed-point data are well fit into 1 DSP unit on either Xilinx or Altera FPGA. This shows that quantization hardly benefits the hardware if we use narrower bit-width like 8 or 4 in the aspect of computation. The problem is that the wide multipliers and adders in DSP units are underutilized in these cases. Nguyen et al. [45] propose the design to implement two narrow bit-width fixed-point multiplication with a single wide bit-width fixed-point multiplier. In this design, two multiplications, $AB$ and $AC$, are executed in the form of $A(B \ll k + C)$. If $k$ is sufficiently large, then the bits for $AB$ and $AC$ do not overlap in the multiplication result and can be directly separated. The design in Reference [45] implements two 8-bit multiplications with one $25 \times 18$ multiplier, where $k$ is 9. Similar methods can be applied to other bit-width and DSPs.

5.1.2 *Fast Convolution Method.* For CONV layers, the convolution operations can be accelerated by alternative algorithms. Discrete Fourier Transformation– (DFT) based fast convolution is widely adopted in digital signal processing. Zhang et al. [75] 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 the space domain to $F^2$ complex multiplications in the frequency domain. For a CONV layer with $M$ input channel and $N$ output channel, $MN$ times of frequency domain multiplications and $(M+N)$ times DFT/IDFT are needed. The conversion of convolution kernels is once for all. So the domain conversion process is of low cost for CONV layers. This technique does not work for CONV layers with stride>1 or $1 \times 1$ convolution. Ding et al. [11] suggest that a blockwise circular constraint can be applied to the weight matrix. In this way, the matrix-vector multiplication in FC layers are converted to a set of 1D convolutions and can be accelerated in the 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 [68]. 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 that only related to the sizes of kernel and feature map. $\odot$ denotes an element-wise multiplication of two matrices. For a $4 \times 4$ feature map convolved with a $3 \times 3$ kernel, the transformation matrices are described as follows:

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

Multiplication with transformation matrices $A$, $B$, and $G$ induce only a small number of shift and addition because of the special matrix entries. In this case, the number of multiplication is reduced from 36 to 16. The most commonly used Winograd transformation is for $3 \times 3$ convolutions in [36, 70].

The theoretical performance gain from fast convolution depends on the convolution size. Limited by the on-chip resource and the consideration of flexibility, current designs are not choosing large convolution sizes. Existing work point out that up to 4× theoretical performance gain can be achieved by fast convolution with FFT [75] or Winograd [36] with reasonable kernel sizes. Zhuge et al. [84] even try to use both FFT and Winograd methods in their design to fit different kernel sizes in different layers.

*5.1.3 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.

Latest FPGAs support 700–900MHz DSP theoretical peak working frequency. But existing designs usually work at 100–400MHz [17, 38, 49, 73, 77]. As claimed in Reference [69], the working frequency is limited by the routing between on-chip SRAM and DSP units. The design in Reference [69] uses different working frequencies for DSP units and surrounding logic. Neighbor slices to each DSP unit are used as local RAMs to separate the clock domain. The prototype design in Reference [69] achieves the peak DSP working frequency at 741MHz and 891MHz on FPGA chips of different speed grades. Xilinx has also proposed the CHaiDNN-v2 [1] and xfDNN [2] with this technique and achieves up to 700MHz DSP working frequency. Compared with existing designs for which the frequency is within 300MHz, this technique brings at least 2× peak performance gain.

### 5.2 Loop Unrolling Strategies

CONV layers and FC layers contribute to most of the computations and storage requirement of a neural network as introduced in Section 2. We express the CONV layer function in Figure 1(b) as 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 2D convolution kernels, respectively. An FC layer can be expressed as a CONV layer with feature map and kernel both of size $1 \times 1$. Besides the loops in Algorithm 1, we also call the parallelism of the process of multiple inputs as a batch. As we treat FC layers and CONV layers all as nested loops, the loop unrolling strategy can be applied both in CNN accelerators and RNN accelerators. But as the case for FC layers are rather simple, we tend to use CNN as examples in this section.

**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 $F_{out}$
function ConvLayer($F_{in}, Ker$)
    Let $F_{out} \leftarrow$ zero array of size $N \times (Y - K + 1) \times (X - K + 1)$
    for $n = 1; n < N; n++$ do                                      ▷ Output channel loop
        for $m = 1; m < M; m++$ do                                  ▷ Input channel loop
            for each $(y, x)$ within $(Y - K + 1, X - K + 1)$ do    ▷ Feature map loop
                for each $(ky, kx)$ within $(K, K)$ do               ▷ Kernel loop
                    $F_{out}[n][y][x] += F_{in}[m][y - ky + 1][x - kx + 1] * K[n][m][ky][kx]$
        $F_{out}[n] += b[n]$
    return $F_{out}$
```

*5.2.1 Choosing Unroll Parameters.* To parallelize the execution of the loops, we unroll the loops and parallelize the process of a certain number of iterations on hardware. The number of the parallelized iterations on hardware is called the unroll parameter. Inappropriate unroll parameter selection may lead to serious hardware underutilization. Take a single loop as an example. Suppose the trip count of the loop is $M$ and the parallelism is $m$. The utilization ratio of the hardware is limited by $m/M\lceil M/m \rceil$. If $M$ is not divisible by $m$, then the utilization ratio is less than 1. For processing an NN layer, the total utilization ratio will be the product of the utilization ratio on each of the loops.

For a CNN model, the loop dimension varies greatly among different layers. For a typical network used on ImageNet classification like ResNet [22], the channel numbers vary from 3 to 2048; the feature map sizes vary from $224 \times 224$ to $7 \times 7$, the convolution kernel sizes vary from $7 \times 7$ to $1 \times 1$. Besides the underutilization 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 to choose the unroll parameters. Zhang et al. [74] 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 neighboring iterations. So no multiplexer is needed to route data from the 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 underutilization problem. Ma et al. [38] 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. [42] use kernel unrolling on AlexNet. Even with $3 \times 3$ unrolling for the $11 \times 11$ and $5 \times 5$ kernels, the overall system performance still reaches 97.4% of its peak performance for the convolution layers. For certain networks like VGG [56], only $3 \times 3$ convolution kernels are used. Another reason to unroll kernel loop is to achieve acceleration with fast convolution algorithms. The design in Reference [75] implements fully parallelized frequency domain multiplication on $4 \times 4$ feature map and $3 \times 3$ kernel. Lu et al. [36] implement a Winograd algorithm on FPGA with a dedicated pipeline for Equation (5). The convolution of a $6 \times 6$ feature map with a $3 \times 3$ kernel is fully parallelized.

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. The designs in References [30, 35, 78] propose fully pipelined structures with each layer a pipe stage. As each layer is executed with an independent part of the hardware and each part is small, a loop unrolling method can be easily chosen. This method is memory consuming, because ping-pong buffers are needed between adjacent layers for the feature maps. Agressive design with binarized weights [71] can fit into FPGA better. Design in Reference [76] is similar but implemented on FPGA clusters to resolve the scalability problem. Shen et al. [54] and Lin et al. [32] group the layers of a CNN by the loops' trip count and map each group onto one hardware module. These solutions can be treated as unrolling the batch loop, because different inputs are processed in parallel on different layer pipeline stages. The design in Reference [36] implements parallelized batch both within a layer and among different layers.

Most of the current designs follow one of the above methods for loop unrolling. A special kind of design is for sparse neural networks. Han et al. [19] 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 unrolling within a layer to simplify hardware design and parallelize batch process.

*5.2.2 Data Transfer and On-chip Memory Design.* Besides the high parallelism, the on-chip memory system should efficiently offer the necessary data to each computation units every cycle. To implement high parallelism, neural network accelerators usually reuse data among a large number of computation units. Simply broadcasting data to different computation units leads to large fan-out and high routing cost and thus reduce the working frequency. Wei et al. [67] 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 are not broadcast, and only local connections between different computation units are needed. The drawback is the increase in latency. The loop execution order is scheduled accordingly to cover the latency. Similar designs are adopted in References [7, 38].

For software implementation on GPU, the im2col function is commonly used to map 2D convolution as a matrix-vector multiplication. This method incurs considerable data redundancy and can hardly be applied to the limited on-chip memory of FPGAs. Qiu et al. [49] uses the line buffer design to achieve the $3 \times 3$ sliding window function for 2-d convolution with only two lines of duplicated pixels.

### 5.3 System Design

A typical FPGA-based neural network accelerator system is shown in Figure 4. The logic part of the whole system is denoted by the blue boxes. The host CPU issues workload or commands

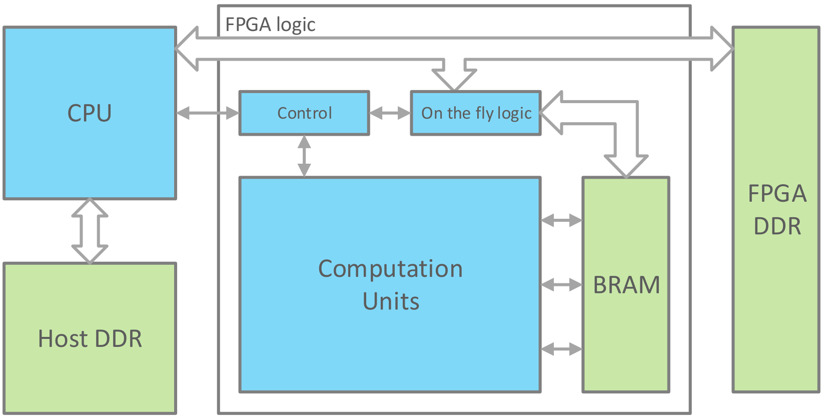

Fig. 4. Block graph of a typical FPGA-based neural network accelerator system.

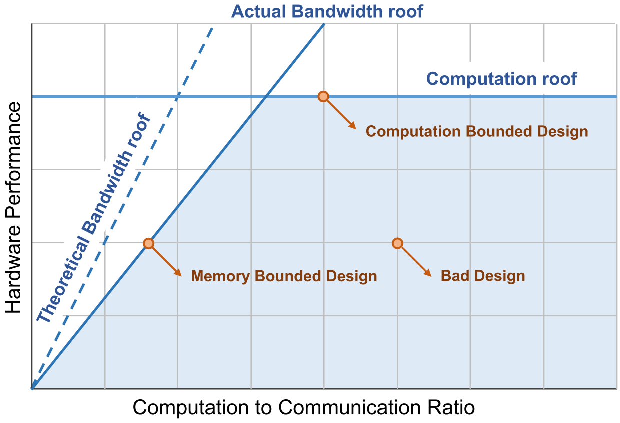

Fig. 5. An example of the roofline model. The shaded part denotes the valid design space given bandwidth and resource limitation.

to the FPGA logic part and monitors its working status. On the FPGA logic part, a controller is usually implemented to communicate with the host and generates control signals to all the other modules on FPGA. The controller can be an FSM or an instruction decoder. The on the fly logic part is implemented for certain designs if the data loaded from external memory needs preprocess. This module can be data arrangement module, data shifter [49], FFT module [75], and so on. The computation units are as discussed in Section 5.1 and Section 5.2. As introduced in Section 2.2, on-chip SRAM of an FPGA chip is too limited compared with the large NN models. So for common designs, a two-level memory hierarchy is used with DDR and on-chip memory.

*5.3.1 Roofline Model.* From the system level, the performance of a neural network accelerator is limited by two factors: 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. [74] 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.

The figure uses the computation to communication (CTC) ratio as the $x$-axis and hardware performance as the $y$-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. The available bandwidth of a target platform 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 the achievable bandwidth of DDR 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.

*5.3.2 Loop Tiling and Interchange.* A higher CTC ratio means the hardware is more likely to achieve the computation bound. Increasing the CTC ratio also reduce DDR access, which significantly saves energy according to Reference [23]. 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 fewer 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. Neighboring tiles may share a part of the data. For example, in a CONV layer, neighboring tiles can share the input feature map or the weights. This is decided by the execution order of the loops.

In References [38, 74], design space exploration is done on all the possible loop tiling sizes and loop orders. Many designs also explore the design space with some of the loop unrolling, tiling, and loop order already decided [42, 49]. Shen et al. [55] also discuss the effect 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. [17] implements flexible unrolling and loop order configuration for different layers with an instruction interface. The data arrangement in on-chip buffers is 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 proposes the "back-and-forth" loop execution order to avoid total on-chip data refresh when an innermost loop finishes.

*5.3.3 Cross-Layer Scheduling.* Alwani et al. [5] address the external memory access problem by fusing two neighboring layers together to avoid the intermediate result transfer between the two layers. This strategy helps reduce 95% off-chip data transfer with extra 20% on-chip memory cost. Even software program gains 2× speedup with this scheduling strategy. Yu et al. [72] realize this idea on a single-layer accelerator design by modifying the order of execution through an instruction interface.

*5.3.4 Regularize Data Access Pattern.* Besides increasing CTC, increasing the actual bandwidth roof helps improve the achievable performance with a certain CTC ratio. This is achieved by regularizing the DDR access pattern. The common feature map formats in the external memory include $NCHW$ or $CHWN$, where $N$ means the batch dimension, $C$ means the channel dimension, $H$ and $W$ means the feature map $y$ and $x$ dimension. Using any of these formats, a feature map tile may be cut into small data blocks stored in discontinuous addresses. Guan [14] suggest that a channel-major storage format should be used for their design. This format avoids data duplication while long DDR access burst is ensured. Qiu et al. [49] propose a feature map storage format that arranges the $H \times W$ feature map into $(HW/rc)$ tile blocks of size $r \times c$. So the write burst size can be increased from $c/2$ to $rc/2$.

## 6 EVALUATION

In this section, we compare the performance of state-of-the-art neural network accelerator designs and try to evaluate the techniques mentioned in Section 4 and Section 5. We mainly reviewed the FPGA-based designs published in the top FPGA conferences (FPGA, FCCM, FPL, FPT), EDA conferences (DAC, ASPDAC, DATE, ICCAD), and architecture conferences (MICRO, HPCA, ISCA, ASPLOS) since 2015. Because of the diversity in the adopted techniques, target FPGA chips, and experiments, we need a tradeoff between the fairness of comparison and the number of designs we can use. In this article, we pick the designs with (1) whole system implementation and (2) experiments on real NN models with reported speed, power, and energy efficiency.

The designs used for comparison are listed in Table 4. For data format, the "INT A/B" means that activations are A-bit fixed-point data and weights are B-bit fixed-point data. We also investigate the resource utilization and draw advice to both accelerator designers and FPGA manufacturers.

Each of the designs in Table 4 drawn as a point in Figure 6, using $log_{10}(power)$ as the $x$ coordinate and $log_{10}(speed)$ as the $y$-axis. Therefore, $y - x = log_{10}(energy\_efficiency)$. Besides the FPGA-based designs, we also plot the GPU experimental results used in References [17, 19] as standards to measure the FPGA designs' performance.

***Bit-width Reduction.*** Among all the designs, 1- to 2-bit-based designs [27, 41, 43] show outstanding speed and energy efficiency. This shows that extremely low bit-width is a promising solution for high-performance design. As introduced in Section 4.1, linear quantized 1-2 bit network models suffer from great accuracy loss. Further developing related accelerator will be of little use. More effort should be put on the models. Even trading speed with accuracy can be acceptable considering the current hardware performance.

Besides the 1/2bit designs, the rest of the designs adopts 32-bit floating-point data or linear quantization with 8 or more bits. According to the results in Section 4.1, within 1% accuracy loss can be achieved. So we think the comparison between these designs is fair in accuracy. INT16/8 and INT16 are commonly adopted. But the difference between these designs is not obvious. This is because the underutilization of DSPs discussed in Section 5.1.1. The advantage of INT16 over FP32 is obvious except for Reference [77], where the hard-core floating-point DSPs are utilized. To a certain extent, this shows the importance of fully utilizing the DSPs on-chip.

***Fast Convolution Algorithm.*** Among all the 16-bit designs, Reference [36] achieves the best energy efficiency and the highest speed with the help of the $6 \times 6$ Winograd fast convolution, which is 1.7× faster and 2.6× energy efficient than the 16-bit design in Reference [77]. The design in Reference [75] achieves 2× speedup and 3× energy efficiency compared with Reference [74] where both designs use 32-bit floating-point data and FPGA with 28nm technology node. Compare with the theoretical 4× performance gain introduced in Section 5.1.2, there is still a 1.3× to 1.5× gap. Not all the layers can use the most optimized fast convolution method because of kernel size limitation.

***System Level Optimization.*** The overall system optimization is not well addressed in most of the work. As this is also related to the HDL design quality, we can roughly evaluate the effect. Here we compare three designs [30, 35, 73] on the same XC7VX690T platform and try to evaluate the

Table 4. Performance and Resource Utilization of State-of-the-Art Neural Network Accelerator Designs

| | Data | Speed | Power | Eff. | Resource(%) | | | |
|---|---|---|---|---|---|---|---|---|
| | Format | (GOP/s) | (W) | (GOP/J) | DSP | logic | BRAM | FPGA chip |
| [43] | 1 bit | 329.47 | 2.3 | 143.2 | 1 | 34 | 11 | Zynq XC7Z020 |
| [41] | 1 bit | 40770 | 48 | 849.38 | — | — | — | GX1155 |
| [27] | 2 bit | 410.22 | 2.26 | 181.51 | 41 | 83 | 38 | Zynq XC7Z020 |
| [17] | INT8 | 84.3 | 3.5 | 24.1 | 87 | 84 | 89 | XC7Z020 |
| [57] | INT16/8 | 117.8 | 19.1 | 6.2 | 13 | 22 | 65 | 5SGSD8 |
| [35] | INT16/8 | 222.1 | 24.8 | 8.96 | 40 | 27 | 40 | XC7VX690T |
| [38] | INT16/8 | 645.25 | 21.2 | 30.43 | 100 | 38 | 70 | GX1150 |
| [19] | INT16/12 | 2520 | 41 | 61.5 | 54 | 89 | 88 | XCKU060 |
| [61] | INT16 | 12.73 | 1.75 | 7.27 | 95 | 67 | 6 | XC7Z020 |
| [49] | INT16 | 136.97 | 9.63 | 14.22 | 89 | 84 | 87 | XC7Z045 |
| [70] | INT16 | 229.5 | 9.4 | 24.42 | 92 | 71 | 83 | XC7Z045 |
| [73] | INT16 | 354 | 26 | 13.6 | 78 | 81 | 42 | XC7VX690T |
| [14] | INT16 | 364.4 | 25 | 14.6 | 65 | 25 | 46 | 5SGSMD5 |
| [30] | INT16 | 565.94 | 30.2 | 22.15 | 60 | 63 | 65 | XC7VX690T |
| [53] | INT16 | 431 | 25 | 17.1 | 42 | 56 | 52 | XC7VX690T |
| | | 785 | 26 | 30.2 | 53 | 8.3 | 30 | XCVU440 |
| [76] | INT16 | 1280.3 | 160 | 8 | — | — | — | XC7Z020+<br>XC7VX690T×6 |
| [77] | INT16 | 1790 | 37.46 | 47.8 | 91 | 43 | 53 | GX1150 |
| [36] | INT16 | 2940.7 | 23.6 | 124.6 | — | — | — | ZCU102 |
| [7] | FP16 | 1382 | 45 | 30.7 | 97 | 58 | 92 | GX1150 |
| [47] | INT32 | 229 | 8.04 | 28.5 | 100 | 84 | 18 | Stratix V |
| [15] | FP32 | 7.26 | 19.63 | 0.37 | 42 | 65 | 52 | XC7VX485T |
| [74] | FP32 | 61.62 | 18.61 | 3.3 | 80 | 61 | 50 | XC7VX485T |
| [75] | FP32 | 123.5 | 13.18 | 9.37 | 88 | 85 | 64 | Stratix V |
| [77] | FP32 | 866 | 41.73 | 20.75 | 87 | — | 46 | GX1150 |

effect. All the three designs implement 16-bit fixed-point data format except that Reference [35] uses 8 bits for weights. No fast convolution or sparsity is utilized in any of the work, even though Reference [30] achieves 2.5× the energy efficiency of Reference [35]. It shows that a system level optimization has a strong effect even comparable to the use of fast convolution algorithm.

We also investigate the resource utilization of the designs in Table 4. Three kinds of resources (DSP, BRAM, and logic) are considered. We plot the designs in Figure 7 using two of the utilization ratio as x and y coordinate. We draw the diagonal line of each figure to show the designs' preference on hardware resource. The BRAM-DSP figure shows an obvious preference on DSP over BRAM. A similar preference appears on DSP over logic. This indicates that current FPGA designs are more likely computation bounded. FPGA manufacturers targeting neural network applications can adjust the resource allocation accordingly. Compared with that, the preference on logic and BRAM seems to be random. A possible explanation is that some of the designers use both logic and DSPs to implement high parallelism, while some prefers to use only DSPs to achieve high working frequency.

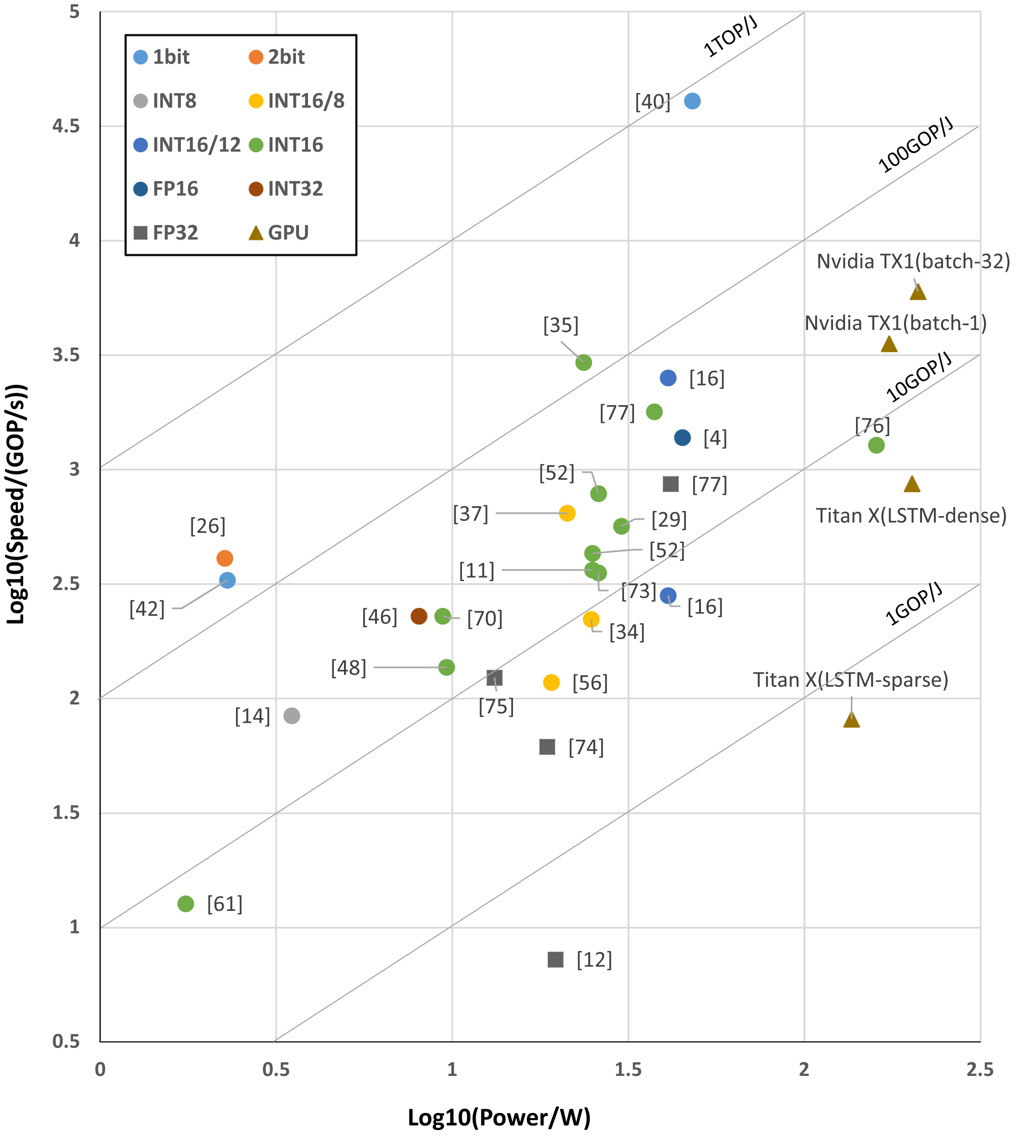

Fig. 6. A comparison between different designs on a logarithm coordinate of power and performance.

***Comparision with GPU.*** In general, FPGA-based designs have achieved comparable energy efficiency to GPU with 10-100GOP/J. But the speed of GPUs still surpasses FPGAs. Scaling up the FPGA-based design is still a problem. Zhang et al. [76] propose the FPGA-cluster-based solution using 16-bit fixed-point computation. But the energy efficiency is worse than the other 16-bit fixed-point designs.

Here we estimate the achievable speed of an ideal design. We use the 16-bit fixed-point design in Reference [36] as a baseline, which is the best 16-bit fixed-point design with both the highest speed and energy efficiency. Eight-bit linear quantization can be adopted according to the analysis in Section 4.1, which achieves another 2× speedup and better energy efficiency by utilizing 1 DSP

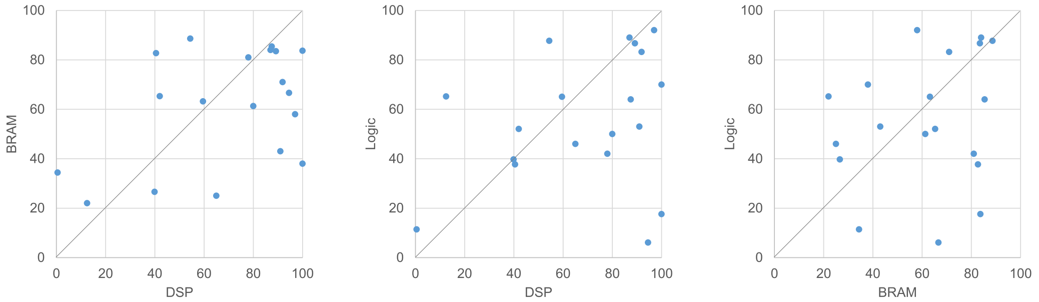

Fig. 7. Resource utilization ratio of different accelerator designs.

as 2 multipliers. The double frequency optimization further improves the system speed by 2×. Consider a sparse model that is similar to the one in Reference [19] with 10% non-zero values. We can estimate a similar 6× improvement as in Reference [19]. In general about 24× speedup and 12× better energy efficiency can be achieved, which means 72TOP/s speed with about 50W. This shows that it is possible to achieve over 10× higher energy efficiency on FPGA over 32-bit floating-point process on GPU.

The left problem is as follows: Do all the techniques (double MAC, sparsification, quantization, fast convolution) and the double frequency design work well together? Pruning a single element in a 2D convolution kernel is of no use for fast convolution, because the 2D kernel is always processed as a whole. Directly pruning 2D kernels as a whole may help. But the reported accuracy of this method is lower [39] than a fine-grained pruning. The irregular data access pattern for processing sparse network and the increase in parallelism also brings challenges to the design of memory system and scheduling strategy.

## 7 TECHNIQUE DISCUSSION

To give a better overview of all the techniques introduced in Sections 4 and 5, we give a brief summary in this section to see how these techniques contribute to FPGA-based NN accelerator designs. Each technique is judged from two aspects: how it affects hardware design and to which level it relates to NN models. Figure 8 shows the summary.

A hardware design basically consists of three parts: datapath, memory, and scheduling. For the design target of high speed, datapath decides the $OPC_{peak}$ while the memory system and scheduling strategy decides $\eta$. For the design target of energy efficiency, datapath decides $E_{op}$ while the memory system decides $N_{SRAM_{acc}}$ and $N_{DRAM_{acc}}$. We can see that existing researches are approaching the design target from every aspect by utilizing the neural network model features from single neuron level to the whole network level.

What is the future of FPGA-based neural network inference accelerator? Currently, much of the techniques lie in the neuron level and the convolution level. There are two reasons for this. The first reason is that few feature can be utilized in the layer level and network level. Most of the existing NN models introduce a simple structure with cascaded layers [28, 56] or simply adding a by-path [22]. New features like depthwise convolution [24] and the complex branch in SSD [34] may brings more design opportunities. But few work focuses on these models. The second reason is that the scale of an FPGA chip is limited. An FPGA chip usually consists of hundreds to thousands of DSPs. This number is still too small compared with a single neural network layer with more than 100M operations.

So further opportunities may come from two aspects. The first is the evolution of network structure. The second is the scaling up of FPGA-based system, with larger chips or multiple chips.

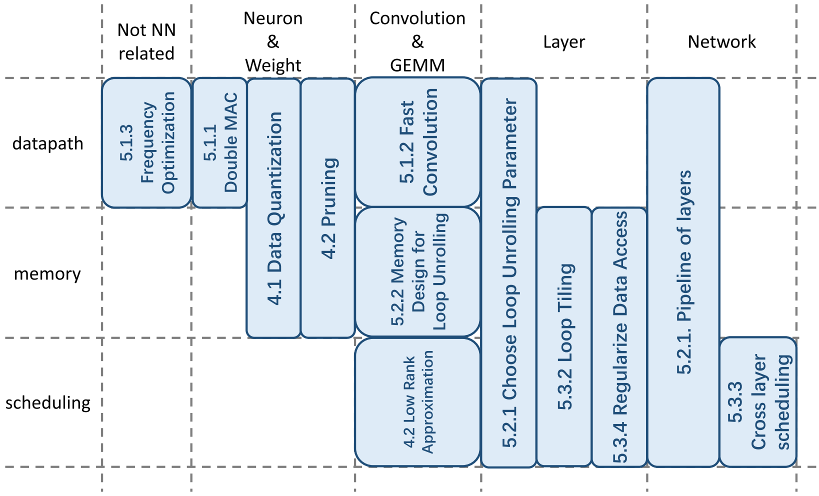
Not NN related
Neuron & Weight
Convolution & GEMM
Layer
Network
datapath
memory
scheduling
5.1.3 Frequency Optimization
5.1.1 Double MAC
4.1 Data Quantization
4.2 Pruning
5.1.2 Fast Convolution
5.2.2 Memory Design for Loop Unrolling
4.2 Low Rank Approximation
5.2.1 Choose Loop Unrolling Parameter
5.3.2 Loop Tiling
5.3.4 Regularize Data Access
5.2.1. Pipeline of layers
5.3.3 Cross layer scheduling

Fig. 8. A brief summary of both the software and hardware techniques in Sections 4 and 5.

Existing designs using small models with binary weights are making their FPGAs relatively larger. These designs already introduce some subversive ideas like mapping the whole networks spatially onto hardware [71]. Besides the opportunities, designers are also faced with the scaling up challenges, from the limitation of loop unrolling, bandwidth, and so on.

## 8 DESIGN AUTOMATION AND FLEXIBILITY

Mapping a certain CNN model onto an FPGA accelerator still requires much heavier work than developing with existing deep learning frameworks. In some application scenarios, various NN models are to be supported with the FPGA accelerator. Thus the design automation of CNN accelerators is also important. Various researches have been focusing on CNN accelerator design toolflows. Venieris et al. [63] give a detailed discussion on different toolflows in supported models, interface, hardware architecture, design space exploration, and arithmetic precision. In this chapter, as we have been focusing on detailed techniques used in model optimization and hardware design, we only classify the toolflows into two categories: hardware design automation and software design automation. Hardware design automation generates different hardware designs according to different NN models. Software design automation keeps the same accelerator and generates different inputs to the accelerator. The discussion in this section can serve as a supplementary to Reference [63].

### 8.1 Hardware Design Automation

Hardware design automation is widely adopted in FPGA-based accelerators because of the reconfigurability of FPGAs [10, 37, 40, 52, 61, 62, 66]. These proposed techniques focus on automatically generate the HDL design based on the network parameter. Difference between these methods is the selection of an intermediate level description of the network to cover the gap between high-level network description and low-level hardware design.

A straightforward way is no intermediate description. The design flow in Reference [37] searches the optimized parameter for a handcrafted Verilog template with the input network description and platform constraint. This method is similar to the optimization methods mentioned in Section 5. DiCecco et al. [10] use a similar idea based on OpenCL model. This enables that the development tool be integrated with Caffe and one network can be executed on different platforms.

Venireis et al. [62] describes the network model as a DFG in their design tool. Then the network computaion process is translated to hardware design with DFG mapping method.

DnnWeaver [52] use a virtual instruction set to describe a network. The network model is first translated into an instruction sequence. Then the sequence is mapped as hardware FSM states but not executed like traditional CPU instructions.

Hardware design automation directly modifies the hardware design to support different networks. This means the hardware can always achieve the best performance on the target platform. This is suitable for FPGA because of its reconfigurability. It works in situations where network switching is not frequent and the reconfiguration overhead does not care. For example, for a large-scale cloud service, the change in network models can be covered by switching between different FPGA chips. So the FPGAs do not need to be reconfigured frequently.

### 8.2 Software Design Automation

Software design automation tries to run different networks on the same hardware accelerator by simply changing the input, in most cases, an instruction sequence. The difference between these work is the granularity of instruction. At a lower level, Guo et al. [17] propose the instruction set with only three kinds of instructions: LOAD, CALC, and SAVE. The granularity of the LOAD and SAVE instructions are the same as the data tiling size. Each CONV executes a set of 2-D convolutions given the feature map size encoded in the instruction. The channel number is fixed as the hardware unrolling parameter. At this level, the software compiler can carry out static scheduling and dynamic data reuse strategy accordingly for each layer. DNNDK [3] uses similar ideas but with more functions in the instructions to support various networks.

Zhang et al. [73] use a layer level instruction set. The control of a CNN layer is designed as a configurable hardware FSM. Compared with Reference [17], this reduces the memory access for instruction while increasing the hardware cost on the configurable FSM.

TVM [8] implements a uniform mapping optimization framework for different kinds of platforms including CPU, GPU, FPGA, and ASIC. The framework allows developers to define customized parallel primitive to support customized hardware, including FPGA accelerators. This means the scheduling granularity is more flexible.

Instruction based methods do not modify hardware and thus enables that the accelerator can switch between networks at runtime. An example of the application scenario is the real-time video processing system on a mobile platform. The process of a single frame can involve different networks if the task is complex enough. Reconfigure the hardware causes unacceptable overhead while instruction based methods can solve the problem if all the instructions of all the networks are prepared in memory.

### 8.3 Mixed Method

Wang et al. [66] propose a design automation framework mixing the above two by both optimizing hardware design and compile software instructions. The hardware is first assembled with pre-defined HDL templates using the optimized hardware parameter. The data control flow of the computation process is controlled by software binaries, which is compiled according to the network description. It is possible that the hardware can be used for a new network by simply changing the software binaries.

## 9 CONCLUSION

In this article, we review state-of-the-art neural network accelerator designs and summarize the techniques used. According to the evaluation result in Section 6, with software hardware co-design, FPGA can achieve more than 10× better speed and energy efficiency than state-of-the-art GPU. This shows that FPGA is a promising candidate for neural network acceleration. We also review the methods used for accelerator design automation, which shows that current development flow can achieve both high performance and runtime network switch.

But there is still a gap between current designs and the estimation. On the one hand, quantization with extremely narrow bit-width is limited by the model accuracy, which needs further algorithm research. On the other hand, combining all the techniques needs more research in both software and hardware to make them work well together. Commercial tools including DNNDK [3] is taking a first step but still has a lone way to go. Scaling up the design is also a problem. Future work should focus on solving these challenges.

## REFERENCES

[1] Xilinx Inc. 2018. *CHaiDNN*. Retrieved August 23, 2018 from https://github.com/Xilinx/chaidnn.

[2] Xilinx Inc. 2018. *xfDNN*. Retrieved December 3, 2018 from https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf.

[3] DeePhi Tech. 2017. *DNNDK*. Retrieved December 3, 2018 from http://www.deephi.com/technology/dnndk.

[4] 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: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 265–283.

[5] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–12.

[6] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. 2016. Deep speech 2: End-to-end speech recognition in English and Mandarin. In *Proceedings of the International Conference on Machine Learning*. 173–182.

[7] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL (TM) deep learning accelerator on arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 55–64.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 578–594.

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

[10] Roberto DiCecco, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Graham Taylor, and Shawki Areibi. 2016. Caffeinated FPGAs: FPGA framework for convolutional neural networks. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'16)*. IEEE, 265–268.

[11] 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.

[12] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. 2018. ReBNet: Residual binarized neural network. In *Proceedings of the 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'18)*. IEEE, 57–64.

[13] 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.

[14] 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. In *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*. IEEE, 152–159.

[15] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. 2017. FPGA-based accelerator for long short-term memory recurrent neural networks. In *Proceedings of the 22nd Asia and South Pacific Design Automation Conference (ASP-DAC'17)*. IEEE, 629–634.

[16] Jianxin Guo, Shouyi Yin, Peng Ouyang, Leibo Liu, and Shaojun Wei. 2017. Bit-width based resource partitioning for CNN acceleration on FPGA. In *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17).* IEEE, 31–31.

[17] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2018. Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 37, 1 (2018), 35–47.

[18] P. K. Gupta. 2016. Accelerating datacenter workloads. In *Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL'16).*

[19] 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 *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17).* 75–84.

[20] 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).

[21] 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).

[22] 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.

[23] M. Horowitz. [n.d.]. Energy table for 45nm process, Stanford VLSI wiki. Retrieved from https://sites.google.com/site/seecproject.

[24] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[25] 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).

[26] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2017. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia.* ACM, 675–678.

[27] Li Jiao, Cheng Luo, Wei Cao, Xuegong Zhou, and Lingli Wang. 2017. Accelerating low bit-width convolutional neural networks with embedded FPGA. In *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL'17).* IEEE, 1–4.

[28] 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.

[29] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary weight networks. *arXiv preprint arXiv:1605.04711* (2016).

[30] 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 *Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL'16).* IEEE, 1–9.

[31] 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. In *FPGA.* 290–291.

[32] Xinhan Lin, Shouyi Yin, Fengbin Tu, Leibo Liu, Xiangyu Li, and Shaojun Wei. 2018. LCP: A layer clusters paralleling mapping method for accelerating inception and residual networks on FPGA. In *Proceedings of the 55th Annual Design Automation Conference.* ACM, 16.

[33] 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.

[34] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. Ssd: Single shot multibox detector. In *Proceedings of the European Conference on Computer Vision.* Springer, Berlin, 21–37.

[35] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. 2016. Automatic code generation of convolutional neural networks in FPGA implementation. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'16).* IEEE, 61–68.

[36] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. 2017. Evaluating fast algorithms for convolutional neural networks on FPGAs. In *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17).* IEEE, 101–108.

[37] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL'17).* IEEE, 1–8.

[38] 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.

[39] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. 2017. Exploring the granularity of sparsity in convolutional neural networks. In *Proceedings of the Computer Vision and Pattern Recognition Workshops*. 1927–1934.

[40] Raghid Morcel, Haitham Akkary, Hazem Hajj, Mazen Saghir, Anil Keshavamurthy, Rahul Khanna, and Hassan Artail. 2017. Minimalist design for accelerating convolutional neural networks for low-end FPGA platforms. In *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*. IEEE, 196–196.

[41] Duncan J. M. Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H. W. Leong. 2017. High performance binary neural networks on the Xeon+FPGA$^{TM}$ platform. In *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. IEEE, 1–4.

[42] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. 2016. Design space exploration of FPGA-based deep convolutional neural networks. In *Proceedings of the 21st Asia and South Pacific Design Automation Conference (ASP-DAC'16)*. IEEE, 575–580.

[43] Hiroki Nakahara, Tomoya Fujii, and Shimpei Sato. 2017. A fully connected layer elimination for a binarizec convolutional neural network on an FPGA. In *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. IEEE, 1–4.

[44] 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.

[45] Dong Nguyen, Daewoo Kim, and Jongeun Lee. 2017. Double MAC: Doubling the performance of convolutional neural networks on modern FPGAs. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*. IEEE, 890–893.

[46] 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 *Proceedings of the International Conference on Field-Programmable Technology (FPT'16)*. IEEE, 77–84.

[47] Abhinav Podili, Chi Zhang, and Viktor Prasanna. 2017. Fast and efficient implementation of convolutional neural networks on FPGA. In *Proceedings of the IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP'17)*. IEEE, 11–18.

[48] 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 *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. IEEE, 1–7.

[49] 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.

[50] 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. *Int. J. Comput. Vis.* 115, 3 (2015), 211–252. DOI: https://doi.org/10.1007/s11263-015-0816-y

[51] Mohammad Samragh, Mohammad Ghasemzadeh, and Farinaz Koushanfar. 2017. Customizing neural networks for efficient FPGA implementation. In *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*. IEEE, 85–92.

[52] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–12.

[53] Junzhong Shen, You Huang, Zelong Wang, Yuran Qiao, Mei Wen, and Chunyuan Zhang. 2018. Towards a uniform template-based architecture for accelerating 2D and 3D CNNs on FPGA. In *Proceedings of the ACM/SIGDA International Symposium*. 97–106.

[54] Yongming Shen, Michael Ferdman, and Peter Milder. 2016. Overcoming resource underutilization in spatial CNN accelerators. In *Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL'16)*. IEEE, 1–4.

[55] 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 (FCCM'17)*. IEEE Computer Society, Los Alamitos, CA.

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

[57] 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, New York, NY, 16–25.

[58] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'15).*

[59] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. 2018. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626* (2018).

[60] 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.

[61] 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.

[62] Stylianos I. Venieris and Christos-Savvas Bouganis. 2017. Latency-driven design for FPGA-based convolutional neural networks. In *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL'17).* IEEE, 1–8.

[63] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. 2018. Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions. *ACM Comput. Surv.* 51, 3 (2018), 56.

[64] Junsong Wang, Qiuwen Lou, Xiaofan Zhang, Chao Zhu, Yonghua Lin, and Deming Chen. 2018. Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA. *arXiv preprint arXiv:1808.04311* (2018).

[65] Xin Wang, Fisher Yu, Zi-Yi Dou, and Joseph E. Gonzalez. 2017. Skipnet: Learning dynamic routing in convolutional networks. *arXiv preprint arXiv:1711.09485* (2017).

[66] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. 2016. DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family. In *Proceedings of the 53nd ACM/EDAC/IEEE Design Automation Conference (DAC'16).* IEEE, 1–6.

[67] 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.

[68] Shmuel Winograd. 1980. *Arithmetic Complexity of Computations.* Vol. 33. SIAM, Philadelphia, PA.

[69] Ephrem Wu, Xiaoqian Zhang, David Berman, and Inkeun Cho. 2017. A high-throughput reconfigurable processing array for neural networks. In *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL'17).* IEEE, 1–4.

[70] 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.

[71] Li Yang, Zhezhi He, and Deliang Fan. 2018. A fully onchip binarized convolutional neural network FPGA impelmentation with accurate inference. In *Proceedings of the International Symposium on Low Power Electronics and Design.* ACM, 50.

[72] Jincheng Yu, Yiming Hu, Xuefei Ning, Jiantao Qiu, Kaiyuan Guo, Yu Wang, and Huazhong Yang. 2017. Instruction driven cross-layer CNN accelerator with winograd transformation on FPGA. In *Proceedings of the International Conference on Field Programmable Technology.* 227–230.

[73] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'16).* IEEE, 1–8.

[74] 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.

[75] 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.

[76] 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.

[77] Jialiang Zhang and Jing Li. 2017. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. 25–34.

[78] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, New York, NY, 56.

[79] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2017. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6848–6856.

[80] 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.

[81] 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 *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17*). 15–24.

[82] 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).

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

[84] Chuanhao Zhuge, Xinheng Liu, Xiaofan Zhang, Sudeep Gummadi, Jinjun Xiong, and Deming Chen. 2018. Face recognition with hybrid efficient convolution algorithms on FPGAs. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. ACM, 123–128.

Received December 2017; revised August 2018; accepted October 2018