High-Throughput Convolutional Neural Network on an FPGA by Customized JPEG Compression

Hiroki Nakahara*, Zhiqiang Que†, Wayne Luk†
*Tokyo Institute of Technology, Japan, nakahara.h.ad@m.titech.ac.jp
†Imperial College London, UK, {z.que, w.luk}@imperial.ac.uk

***Abstract*—The growing interest in using FPGAs to accelerate convolutional neural network (CNN) workloads is driving the deployment of FPGAs on cloud services such as Amazon AWS and Microsoft Azure. Such current cloud-based FPGAs have serious problems concerning data transfer bandwidth. In this paper, we compress a transfer image using customized JPEG coding and implement a customized image decoder architecture. We analyze the trade-off between data transfer speed-up and recognition accuracy drop. Based on this compression scheme, we design a high-throughput CNN inference engine. Almost all existing FPGA-based CNN accelerators are based with the same idea as their GPU counterparts, where operations from different network layers are mapped onto the same hardware units working in a multiplexed way. Our fully pipelined architecture maps all the network layers on-chip and transfers the computation from different layers to their unit with independent optimization. We apply two CNN optimization techniques to a residual network, one is a channel shift and point-wise approximation, and the other is a binary weight quantization. We implement the proposed CNN inference accelerator on the Xilinx Virtex UltraScale+ XCVU9P FPGA. Our system peak-performance achieves 2.41 TOPS. Our compressed JPEG image transfer only consumes 4% of the system resource, drops 0.3 points of accuracy and achieves 81,120 FPS which is 65.27 times faster than the conventional straightforward RGB data transfer. Thus, our proposed data transfer architecture is sufficient to increase system performance. As for the system throughput, our system is 3.84-34.41 times higher than existing FPGA implementations. Compared with the Xeon CPU, it achieves 138.38 times higher throughput, and it dissipates 1.2 times lower power, so its efficiency is 177.12 times better. Compared with the Tesla V100 GPU, it achieves 9.48 times higher throughput, dissipates 3.9 times lower power, and its efficiency is 37.52 times better. Thus, our parallel architecture on an FPGA provides superior throughput for the acceleration of a CNN.**

I. Introduction

A. FPGA-based CNN Accelerator on the Cloud

Convolutional neural networks (CNNs) are widely used for computer vision applications, such as segmentation [4], [10], [29]; object detection [28], [35], [36]; and pose estimation [9], [32], [41]. Further, they are used for different domains, such as natural language processing [3], [23], [49]; acoustic signal processing [34]; and AI for games [38]. These applications require high accuracy; thus, modern CNNs contain millions of floating-point parameters, and they require billions of floating-point operations to recognize a single image. Furthermore, recent CNNs tend to be large, as demonstrated by AI researchers. Consequently, the computation of CNNs is almost exclusively done on large clusters of GPUs. However, GPU platforms consume more power than CPU and FPGA platforms. Moreover, modern CNNs involve many operations on an image and thus CPUs are too slow.

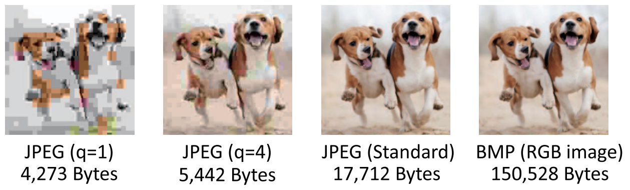


Fig. 1. Image quality versus file size. q denotes the number of bits for a constant JPEG quantization value (2^q). We used 224×224 pixel image.

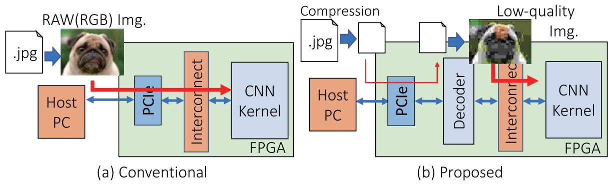


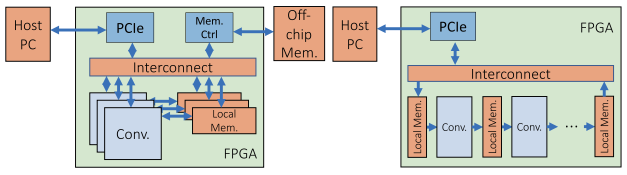
Fig. 2. High-throughput inference system with a customized JPEG coding.

The growing interest in using FPGAs to accelerate CNN provides the driving force behind the deployment of FPGAs on cloud services, such as Amazon AWS and Microsoft Azure. The availability and flexibility of FPGAs in the cloud raise new challenges in the design and implementation of deep learning models on these platforms. The recent adoption of FPGA demonstrates its great ability to run CNN-related applications in both of the above cloud servers. This combination of programmable hardware and DNNs has enabled many possibilities to reshape the landscape of deep learning applications for high throughput and high energy efficiency.

B. Customized JPEG Coding for a High-Speed Data Transfer

Current cloud-based FPGAs have a strict data transfer bandwidth. In particular, communication between the accelerator card and the host is a bottleneck, and transfer technology (especially, the PCI express protocol) development is slow, because this distance is the farthest than other communications (e.g., AWS F1 provides overall read/write at 6.5GB/s from host CPU to FPGA [11]). In this paper, we compress a transfer image using a constant quantization value 2^q of customized JPEG coding protocol. As shown in Fig. 1, there

2576-2621/20/$31.00 ©2020 IEEE
DOI 10.1109/FCCM48280.2020.00010



(a) Stage Pipelining (b) Fully Pipelining

Fig. 3. Comparison of architectures.

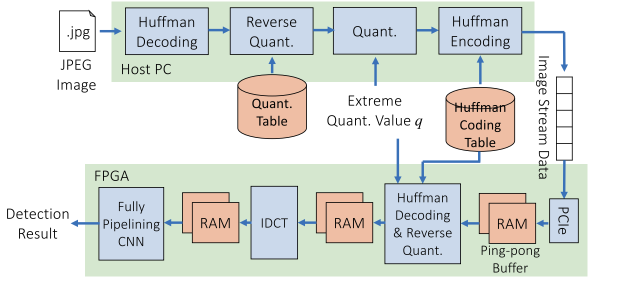


Fig. 4. Proposed JPEG coding system.

is a trade-off between compression ratio (in other words, the JPEG quantization (compression) value) and recognition accuracy. Data transfer speed increases for a constant low quantization value 2^q, while recognition accuracy drops. To recover the situation, we consider a scenario as shown in Fig. 2. For both training and inference, we assume that the same compression quality images are used. Therefore, such accuracy decreases would be relaxed. Our proposed system sends such a compressed data stream to the FPGA card which has a fully pipelined architecture including a customized JPEG-decoding component. We show less hardware overhead and accuracy dropping for our designed JPEG coding process, and achieves throughput improvement by considering the JPEG quantization value for the ImageNet standard dataset.

C. Fully Pipelined CNN Architecture

Almost all of the existing FPGA accelerators are designed with the same idea as their GPU counterparts. As shown in Fig. 3(a), for achieving more generality, all operations from different network layers are mapped onto the same hardware units and working in a multiplexed manner. The result of this is that different layers must be implemented with the same parallelism, which is not flexible enough to take full advantage of the customizability of FPGAs. It leads to a series of conflicts with the inherent computing features of CNNs. Meanwhile, as shown in Fig. 3(b), another implementation of CNNs on an FPGA is a fully pipelined style, which maps all the network layers on-chip, and the computation from different layers is mapped to their hardware unit with independent optimization. Previous work has tried this idea with extremely low-precision (binary) CNNs [12], which use only a one- or two-bit quantization strategy to reduce hardware size. It is suitable for high-throughput system because FPGAs can realize such low-precision design efficiently, whereas GPUs cannot. Also, we propose several techniques to realize a fully pipelined CNN inference engine including a binary weight quantization, approximation by channel shift and point-wise convolution operations, and their pipelined circuit.

The contributions of this study are as follows:

1. We propose an customized JPEG compression data transfer for the CNN inference system to improve a known bottleneck. Our compressed JPEG image transfer only consumed 4% of the system resource, dropped 0.3 point of accuracy and achieved 81,120 FPS which which is 65.27 times faster than the conventional straightforward RGB data transfer. Thus, our proposed decoder architecture is sufficient to increase the system performance.
2. We propose a mixed-precision CNN to fit a fully pipelined architecture that takes full advantage of the customizability of FPGAs. We also introduce a mixed precision scheme and approximation by a channel shift and a point-wise convolution operation.
3. We implement our system on the Xilinx VCU1525 acceleration card and compared it with existing FPGA implementations. As for the system throughput, our system 3.84-34.41 times higher than existing FPGA implementations. Compared with the Xeon CPU, it achieves 138.38 times higher throughput, and it dissipates 1.2 times lower power, so its efficiency is 177.12 times better. Compared with the Tesla V100 GPU, it achieves 9.48 times higher throughput, dissipates 3.9 times lower power, and its efficiency is 37.52 times better.

II. High Throughput Inference System By Customized JPEG Compression

Fig. 4 shows the proposed JPEG coding system. With a standard JPEG incoming image, customized encoding is performed on the host PC, and the compressed data stream is transferred to the accelerator. We implement a dedicated JPEG decoding circuit on the FPGA and convert the transferred stream to a YCrCb image [1]. The proposed method reduces the quantization value of all frequency components of the JPEG image to 2^q, and shares the same Huffman code table to both the host PC and the accelerator. Thus, no header information is required for the JPEG stream and there is no need to decode the header information. There is a trade-off between recognition accuracy and data transfer speed for the quantization scheme. It is evident in the experimental results. Here, details of each process will be described.

A. Pre-processing on a Host PC

We modify the *libjpeg* library to customize a JPEG image compression. It extracts a Huffman coding table and a quantization table from an original JPEG image. Then, it decodes the JPEG stream into an original RGB pixel stream and performs

[1]The standard JPEG decoder convert to YCrCb to RGB. From our experiment, such conversion did not affect recognition accuracy. Thus, we use YCrCb image for the CNN inference.

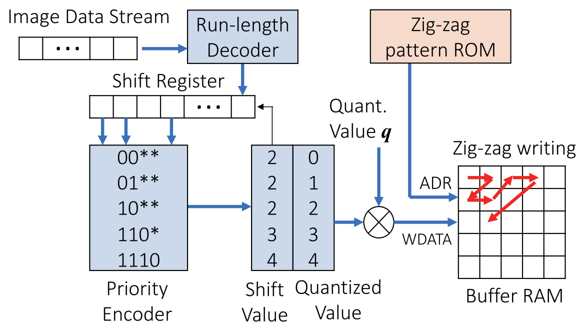


Fig. 5. Huffman decoding and reverse quantization unit.

customization quantization with a 2^q value. Next, it encodes the compressed JPEG data stream by Huffman and run-length codings for customized quantization.

B. Run-length and Huffman decoding circuit

Fig. 5 shows a run-length, Huffman decoding, and reverse quantization circuit. First, it reads incoming compressed JPEG streaming data, then stores into a shift register with run-length decoding. Then, it converts a Huffman code to a quantized value and a shift value which is sent to the shift register. Next, it performs a reverse quantization and writes to the buffer RAM in a zig-zag manner with a zig-zag patterm address from a zig-zag pattern ROM.

C. 2D-IDCT (Inverse Discrete Cosine Transfer) Circuit

The following expression shows a 2D-IDCT.

$$\begin{aligned} f(y,x) &= \sum_{v=0}^{7} \frac{C(v)}{2} \sum_{u=0}^{7} \frac{C(u)}{2} F(v,u) \\ &\quad \times (cos\frac{(2x+1)u\pi}{16} cos\frac{(2y+1)v\pi}{16}), \qquad (1) \end{aligned}$$

where $C(i) = 1/\sqrt{(2)}$ $(u = 0)$ and $C(i) = 1$ $(u > 0)$, and $F(v,u)$ denotes a DCT coefficient. It processes $8 \times 8 = 64$ pixel values to generate 64 coefficients. Transforming an 8×8 pixel block would require 4,096 multiplications and 4,032 additions. To reduce the number of operations, we replace the 2D-IDCT with 16 1D-DCTs (eight 1D-DCTs with eight rows and eight 1D-DCTs with eight columns) as follows:

$$\begin{aligned} F'(u,y) &= \sum_{y=0}^{7} C(v,y)F(y,v) \\ f(x,y) &= \sum_{u=0}^{7} C(u,x)F'(u,y), \end{aligned}$$

where $C(i,j) = C(i)cos\frac{2\pi i(2j+1)}{16}$. Therefore, it includes only 1,024 multiplications and 896 additions to convert an 8×8 pixel block. Various methods for further reducing the number of operations have been proposed. In this paper, we modify the IDCT implementation method introduced in AP-922 [2]. Let γ_k be $cos(\frac{2\pi k}{16})$. Then, we have,

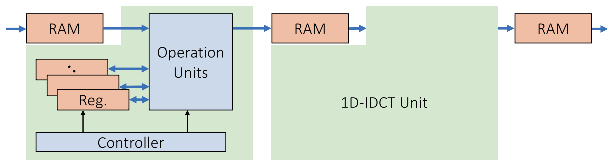


Fig. 6. 2D-IDCT unit.

$$C = \frac{1}{2}\begin{pmatrix} \gamma_4 & \gamma_4 & \gamma_4 & \gamma_4 & \gamma_4 & \gamma_4 & \gamma_4 & \gamma_4 \\ \gamma_1 & \gamma_3 & \gamma_5 & \gamma_7 & -\gamma_7 & -\gamma_5 & -\gamma_3 & -\gamma_1 \\ \gamma_2 & \gamma_6 & -\gamma_6 & -\gamma_2 & -\gamma_2 & -\gamma_6 & \gamma_6 & \gamma_2 \\ \gamma_3 & -\gamma_7 & -\gamma_1 & -\gamma_5 & \gamma_5 & \gamma_1 & \gamma_7 & -\gamma_3 \\ \gamma_4 & -\gamma_4 & -\gamma_4 & \gamma_4 & \gamma_4 & -\gamma_4 & -\gamma_4 & \gamma_4 \\ \gamma_5 & -\gamma_1 & \gamma_7 & \gamma_3 & -\gamma_3 & -\gamma_7 & \gamma_1 & \gamma_5 \\ \gamma_6 & -\gamma_2 & \gamma_2 & -\gamma_6 & -\gamma_6 & \gamma_2 & -\gamma_2 & \gamma_6 \\ \gamma_7 & -\gamma_5 & \gamma_3 & -\gamma_1 & \gamma_1 & -\gamma_3 & \gamma_5 & -\gamma_7 \end{pmatrix}$$

The IDCT multiples by C^T for each column and row of $F(u,v)$. We decompose C^T as follows:

$$C^T = \frac{1}{2}A^T M^T P^T,$$

where

$$P^T = \frac{1}{2}\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

$$M^T = \frac{1}{2}\begin{pmatrix} \gamma_4 & \gamma_2 & \gamma_4 & \gamma_6 & & & & \\ \gamma_4 & \gamma_6 & -\gamma_4 & -\gamma_2 & & & & \\ \gamma_4 & -\gamma_6 & -\gamma_4 & \gamma_2 & & & & \\ \gamma_4 & -\gamma_2 & \gamma_4 & -\gamma_6 & & & & \\ & & & & \gamma_1 & \gamma_3 & \gamma_5 & \gamma_7 \\ & & & & \gamma_3 & -\gamma_7 & -\gamma_1 & -\gamma_5 \\ & & & & \gamma_5 & -\gamma_1 & \gamma_7 & \gamma_3 \\ & & & & \gamma_7 & -\gamma_5 & \gamma_3 & -\gamma_1 \end{pmatrix},$$

and

$$A^T = \frac{1}{2}\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{pmatrix}.$$

We implement P^T by just memory accessing of incoming data F, M^T by $4 \times 8 = 32$ multiplications and $3 \times 8 = 24$ additions, and A^T by eight additions. Compared to directly calculating Expr. (1), there are several advantages. In the implementation, there is no need to transpose the columns for pre-processing. Using two different one-dimensional IDCTs eliminates the need for transposition. Fig. 6 shows a 2D-IDCT unit consisting of two 1D-IDCT units. Since the CNN computation is slower than the IDCT one, we implement a custom processor that includes a register file, operators, and a controller. It computes IDCT with ping-pong buffer RAMs. Note that, as shown in the implementation result, since the IDCT circuit is not a dominant for both hardware resource consumption and performance, we use a half (16-bit) precision for the implementation.

III. Definition of CNN

A typical CNN consists of a group of L sequential layers including a convolutional, a pooling, and a fully-connected (FC) layers. We assume that a pre-trained CNN is available, and the goal is to realize only inferences with high performance and small hardware. In the following section, we present a conventional technique used in the study.

A. Convolutional Operation

The forward two-dimensional (2D) convolution layer computes the output Y by applying a set of m 2D kernels $k \times k$ to c input feature maps X as follows:

$$y_{ij}^{(m)} = \sum_{c} y_{ij}^{(m,c)} \quad = \sum_{c} \sum_{s=0}^{k-1} \sum_{t=0}^{k-1} w_{s\,t}^{(m,c)}\, x_{(i+s)(j+t)}^{(c)} + b^{(m)}.$$

B. Maximum Pooling

The forward 2D maximum pooling layer is a form of non-linear down-sampling of an input feature map. 2D max pooling partitions the input feature map into 2D sub-feature maps along the dimensions k^2, selects an element with the maximum value in each sub-feature map, and transforms the input value to the output feature map y_{ij} by replacing each sub-feature map with its maximum element.

C. Global Average Pooling (GAP)

GoogLeNet [39] uses a GAP layer [25]. A conventional CNN has this type of structure to obtain an output of 1000 class classifications by stacking multiple FC layers after the convolutional layers. However, these FC layers require many parameters, and this causes an over-fitting problem [47]. The GAP layer performs an average pooling with a size similar to that of the input feature map (the output size equals $1 \times 1 \times m$). It constitutes best practice for modern CNNs for classifications.

D. Batch Normalization

There is an impact from the difference in the distribution of data on each batch (internal covariate shift); thus, the convergence of the training tends to be slow, and the trainer must carefully determine the initial value of the parameters. These problems are solved by *batch normalization* [20], which corrects the difference in the distribution by a shift and a scaling operation. Furthermore, it is used for a low-bit precision CNN to retain the hidden weight convergence.

E. Separable Convolution

Conventional convolution operation performs in both spatial and channel directions of the input feature map at a time, while separable convolution performs convolution independently in spatial and channel directions. It is based on the hypothesis that convolutions can be separated in these directions. Fig. 7 shows a separable convolution operation. Spatial convolution

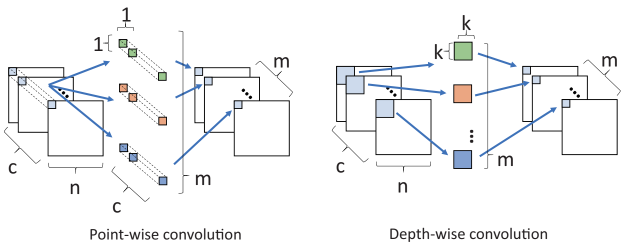


Fig. 7. Separable convolution.

and channel direction convolution are also referred to as *depth-wise convolution* and *point-wise convolution*, respectively. Depth-wise convolution is a process of performing convolution in the spatial direction independently for each channel of the feature map, while point-wise convolution is applied to 1×1 convolution. Consider $c \times n^2$ input feature map size and m output feature maps. For $k \times k$ convolution, its computation order becomes $O(cn^2k^2m)$. Conversely, the computation order for a depth-wise and a point-wise convolution are $O(n^2ck^2)$ and $O(n^2cm)$, respectively. By converting the conventional $k \times k$ convolution to a separable convolution (a pair of depth-wise and point-wise convolutions), its computation order can be relaxed to $O(n^2ck^2 + n^2cm)$. Typically, $m \gg k^2$ (e.g., $k = 3$ and $m = 64$); thus, the separable convolution can be reduced by $\frac{1}{k^2}$

F. Channel Shift Operation

Unlike the convolutional operation, it only moves the value of the neuron stored in the memory, so it requires no multiplications. Wu et. al proposed a ShiftNet [42], which replaces a depth-wise convolution into a uniform shift operation which requires no parameters. It is suitable for custom hardware implementation.

G. Channel Split and Shuffle Operation

We use a channel split and shuffle operation [48] to improve accuracy for the proposed CNN. First, separate in-coming channels into two. Then, one of them is bypassed to the channel concat layer, while another one is handled by conventional convolutional operations. It can be applied to the spatial down-sampling layer as a pooling operation.

IV. Our CNN Model Towards On-chip Memory Realization

Since we are implementing a fully pipelined circuit, it is necessary to store all the parameters into on-chip memory. We apply the following two optimization techniques to the ShuffleNetV2 [48] based CNN.

A. Approximation by Channel Uniform Shift and Point-wise Convolution

We replace all existing $k \times k$ convolutional operations with shift and point-wise convolutional operations to reduce computational complexity and parameter size. As shown in Fig. 8, when we shift for all directions with k^2 feature maps,

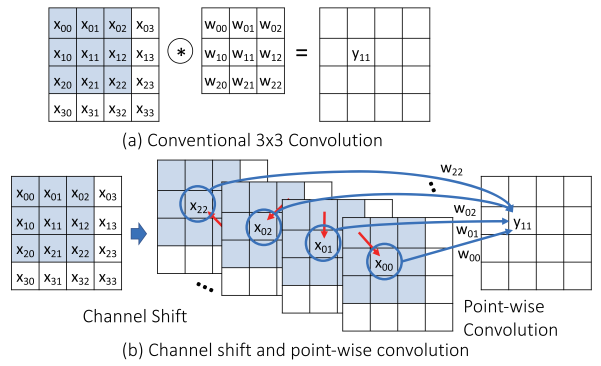

(a) Conventional 3x3 Convolution
Channel Shift
Point-wise Convolution
(b) Channel shift and point-wise convolution

Fig. 8. Channel shift and point-wise operations equivalent to the conventional $k \times k$ convolution.

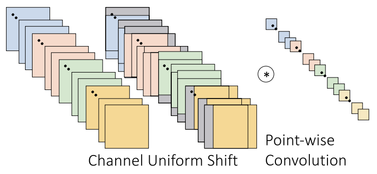


Fig. 9. Channel uniform shift and point-wise operations.

this replacement is equivalent to the original k^2 convolution. However, shift operations increase the number of channels by k^2. We introduce pruning to suppress the increase in the number of channels. It approximates weight values close to zero and skips the convolution. However, typical pruning does not consider the location of zero-weights, and the corresponding addresses of the remaining weights must be held in additional memory and this causes memory access overhead. In this paper, we predetermine the shift direction before training (see *uniform shift operation* as shown in Fig. 9).

B. Quantization Strategy

Both a trained weight and an activation value should ideally be represented by a low-precision bit. However, error-rates usually significantly increase especially for a single bit (binary). We train our low-precision model from scratch by following several rules as follows: We apply scaling factors for each layer with constant unlearned values equal to the layer-specific standard deviations used for initialization [30]; we use an 8-bit precision layer for the first and the last layers to keep information loss to a minimum [30]; we follow the guidelines for low-precision network training [5]; we choose a plain block (used in the original *ResNet* [17], not a bottleneck one, and we appropriately increase the network width (the number of feature maps).

C. Structure of Our CNN Model

Based on the ShuffleNetV2 [48], we apply proposed hardware-friendly optimizations. Fig. 10 (a) shows a plain block, while the bottom side shows a spatial down-sampling block. Note that, *PWConv* denotes a point-wise convolution

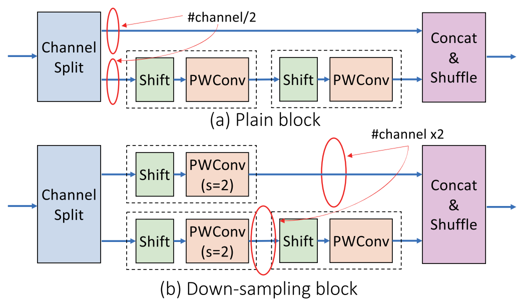


Fig. 10. Building blocks used in our CNN model

TABLE I
STRUCTURE OF OUR CNN. IT CONSISTS OF 2.54 M PARAMETERS AND 0.616 GMACS. NOTE THAT, WE APPLY A TRAINING-AWARE QUANTIZATION INCLUDING A BINARY WEIGHT AND 8-BIT ACTIVATION VALUE EXCEPT FOR THE FIRST STAGE AND THE LAST STAGE (8-BIT FOR BOTH A WEIGHT AND AN ACTIVATION).

Layer	Output size	Kernel size	Stride	#Output Channel
Image	224			3
PWConv	224	1	2	24
Norm	224	1	1	24
ReLU	224	1	1	24
Shift	224	3	1	24
Maxpool	112	2	2	24
PWConv	112	1	1	24
Norm	112	1	1	24
ReLU	112	1	1	24
Shift	112	3	1	24
Maxpool	56	2	2	24
Stage 2 (4 repeats)	28			116
Stage 3 (8 repeats)	14			232
Stage 4 (16 repeats)	7			464
Ave. Global Pool	1	7	1	464
PWConv	1	1	1	1000

layer. We apply batch normalization (*Norm*) and a *ReLU* activation function [31] for the output of a point-wise convolution layer. In the down-sampling layer, we use a point-wise convolution with stride two and double the number of channels.

We design a deeper and low-bit precision CNN because a design space exploration has been reported that a lower-bit precision deep network (ResNet-50 with 2-bit weights, 8-bit activations) outperforms a higher-bit precision shallow network (ResNet-18 with 8-bit weights, 8-bit activations), both in terms of lower compute cost and lower error rate [45]. Table I shows the overall structure of our CNN. At each stage, we repeat the process with plain blocks, i.e., it consists of one down-sampling block and many plain blocks. Our ShuffleNetV2-based CNN consists of only point-wise convolution layers. Thus, the number of parameters and computational complexities are less than those of the conventional CNN, which has a conventional $k \times k$ convolution operation. It consists of 2.54 M parameters and 0.616 GMACs. Note that,

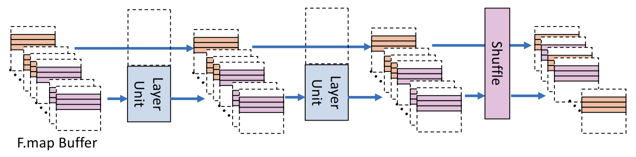


Fig. 11. Dataflow for a residual stage of a plain block.

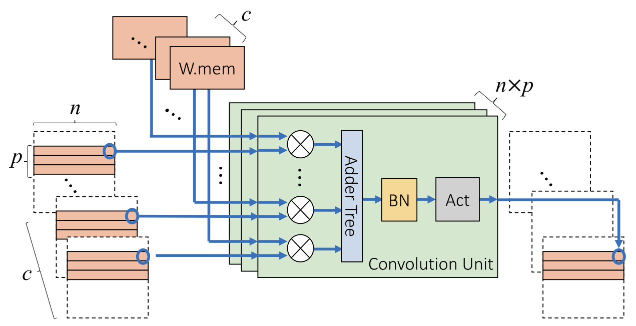


Fig. 12. Layer unit for a 2D convolutional layer.

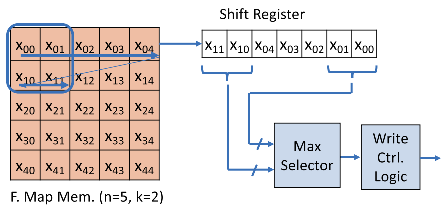


Fig. 13. Layer unit for a maximum pooling layer.

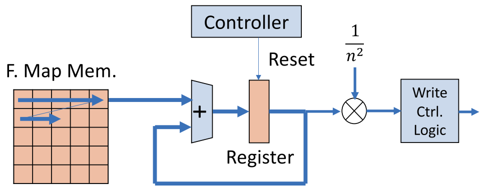


Fig. 14. Layer unit for a global average pooling layer.

we apply a training-aware quantization including a binary weight and 8-bit activation value except for the first stage and the last stage (8-bit for both a weight and an activation).

V. Fully Pipelined CNN Architecture

A. *Fully Pipelining for a Residual Model*

Fig. 11 shows a fully pipelined circuit for a residual structure. A typical Residual structure has a branching dataflow. No operations are performed in one of them, and convolutional operations are performed in another flow. Then, the flows merge through a channel shuffling unit. To realize a fully pipelined architecture consisting of a single branch, we insert a wide pipeline buffer in which no operation is performed in half of them. Although the area overhead is required to realize the residual structure, our pipelining circuit has only a few lines that corresponding to the pipeline processing. The number of additional storage elements can be suppressed.

B. *Architectures for Layer Units*

As for the 2D convolutional layer, although we used a binarized weight and multi-bit precision activation MAC operation instead of a floating-point one, much hardware is consumed in realizing the fully parallel MAC operation. Since the typical CNN has a different number of feature maps in a layer, a heterogeneous streaming architecture requires many LUTs for large-size operations.

To realize high-performance with less hardware, we design a parallel circuit supporting a streaming operation for each layer as shown in Fig. 12. It consists of a weight memory (*W.mem*), a line buffer for a feature map, a MAC operator, an adder tree, a normalization unit (*BN*), and an activation unit. Note that, our target is an inference operation, and we realize a BN unit by using a multiplier and an adder with trained parameters. Since the used activation function is *ReLU*, we implement a selector in the activation unit.

To further increase performance, we propose shared streaming. We adjust parallel parameters c and p efficiently use the available hardware resources and satisfy the performance requirements. To have flexible access to all feature maps and the weights, multiple on-chip BRAMs are used to realize multi-port memories with high bandwidth memory access. Since we use a binarized weight CNN, the memory size is drastically reduced as compared with a non-binarized one.

Fig. 13 shows a maximum pooling unit. Since a binarized maximum pooling operation is realized by a shift register and a maximum value selector.

Fig. 14 shows an average pooling unit. We implement a sequential manner to sum of all feature map values with a register. Then, we multiplied by $\frac{1}{n^2}$ to obtain the average value of one feature map.

The uniform shift operation can be realized by an additional addressing to the next layer. In other words, it can be realized by adding or subtracting each of the four directions in the address space. When shifting in any direction is available, it requires additional memory that can hold the shift direction and its access cost. In the paper, we use a uniform shift which does not cause such overhead because the shift direction is determined in advance.

Similar to the shift operation, the shuffle layer can be realized by additional addressing. We replace the set of channels divided into the first half of channels and the second half ones into even index channels and odd index channels. That is, it can be realized only by replacing the most significant bit (MSB) with the least significant bit (LSB) at the time of channel access from the next layer.

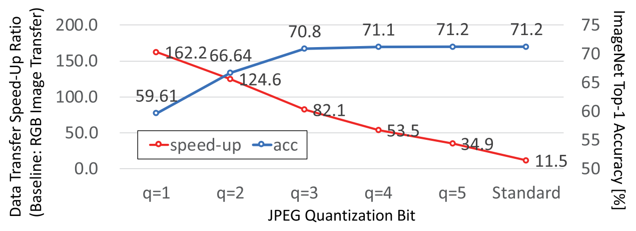


Fig. 15. Trade-off between accuracy and data transfer speed-up ratio for different JPEG quantizations.

TABLE II
RESOURCE CONSUMPTION (XILINX VIRTEX ULTRASCALE+ XCVU9P).

Module	# LUTs	#FFs	#DSPs	#18Kb BRAMs	#URAMs
JPEG Decoder					
Huffman Decoder	6,794	2,378	0	0	0
2D-IDCT	4,881	4,278	34	2	0
Pipelined-CNN	263,120	266,784	2,336	2,744	0
Total†	274,795	273,440	2,370	2,746	16
(Utlized Ratio)	(23.2%)	(11.5%)	(34.6%)	(63.5%)	(1.6%)

†(Including Buffer RAMs)

VI. EXPERIMENTAL RESULTS

A. *Compression Ratio versus Model Parameters*

We train our CNN using the ImageNet 2012 classification dataset. Hyper-parameters including optimizer are the same as *ResNet* implementation [19]. In the experiments, we used the Intel Corei7 CPU, NVIDIA TITAN RTX (24 GB memory) GPU, 64 GB main memory, and Ubuntu 18.04 LTS OS. We used the PyTorch version 1.4.0 deep learning framework to develop our CNN.

We assume that 224×224 pixel images, and Fig. 15 shows the trade-off between accuracy and data transfer speed-up ratio for different JPEG quantizations. In Fig. 15, *speed-up* denotes data transfer acceleration ratio compared with the straightforward RGB image transfer considering our target FPGA board, and *Standard* denotes a standard quantization used in *libjpeg* library. From this experiment, we set a constant quantization as $q = 3$ which only decreases 0.3 point of accuracy and achieves 82.1 times speed-up of data transfer. As shown in Fig. 15, recognition accuracy of our system for $q = 3$ achieved 70.8% of top-1 accuracy.

B. *FPGA Implementation Results*

We implemented the proposed CNN inference accelerator on the Xilinx Inc. Virtex UltraScale+ FPGA VCU1525 acceleration development kit, which has the Xilinx Virtex UltraScale+ FPGA (XCVU9P, 1,182,240 LUTs, 2,364,480 FFs, 4,320 18Kb BRAMs, 960 UltraRAMs, 6,840 DSP48Es). We used the Xilinx Inc. SDAccel 2018.2. The host PC consists of the Intel Xeon CPU E5-2690 v4 running at 2.60 GHz, whose DDR4 memory size is 32 GB. The operating system is Ubuntu 18.04 LTS 64 bit version. Our implementation used 274,795 LUTs, 273,440 FFs, 2,746 18Kb BRAMs, 16 UltraRAMs and 2,370 DSP48Es, and it operates at 300 MHz. Table II

TABLE III
COMPARISON WITH OTHER FPGA IMPLEMENTATIONS

Method	AlexNet [24]	FINN-R [6]	Synetgy [46]	MobNet V2 [43]	CloudDNN [11]	Ours
FPGA	StratixV	Zynq ZU3EG	Zynq ZU3EG	Zynq ZU9EG	VirtexUS+ XCVU9P	VirtexUS+ XCVU9P
Throughput (FPS)	864.7	200.0	96.5	809.8	123.1	**3321.2**
Top-1 Acc.	42.90%	50.30%	68.30%	68.1	—	**70.8%**
Top-5 Acc.	66.80%	—	88.12%	—	—	**90.1%**
Precision (W/Act)	16/16	1/2	4/4	8/8	16/16	1/8
Performance (GOPs)	1963.96	400	418	—	1828.61	**2419.2**
Freq. (MHz)	150	220	250	333	214	300
Power (W)	26.2	10.2	5.5	—	49.25	75.0

shows the resource consumption. As shown in Table II, the JPEG decoder part of the LUT was only 4.2% of total system resource. Thus, our proposed decoder is not a bottleneck of hardware consumption.

We measured total system power consumption which was 75 Watt including the FPGA board and the host workstation. To implement the fully pipelined CNN, we used *#pragma HLS dataflow* pragma and its interval time for the next image was 301,091 (ns). The system throughput was 3321.25 frames per second (FPS) and its bottleneck was the first convolutional layer (301,091 (ns)) and not the JPEG decoder part. Also, the system performance was 2419.2 (GOPS) for a line parallel convolution with 300 MHz operation clocks but it not included the shuffle and shift, and JPEG coding operations. Our compressed JPEG image transfer achieved $81,120$ FPS, while the straight forward RGB data transfer was 1242.8 FPS. Thus, our customized JPEG compression architecture was 65.27 times faster and the proposed decoder architecture improved serious problems concerning data transfer bandwidth.

C. *Comparison with Conventional FPGA Implementations*

Table III compares our accelerator with conventional FPGA implementation for ImageNet classification. CNNs for ImageNet classification are usually orders of magnitude more complex than CIFAR10 classification. Therefore, we can only compare accelerators targeting CNNs for ImageNet classification with reasonable accuracy. Our work focuses on achieving competitive accuracy while improving inference speed in terms of frames per second (FPS). Our training scheme reduces accuracy degradation compared with other neural network models and it is the only design achieving more than 70% of top-1 accuracy and 90% of top-5 one. As for system throughput, our system 3.84-34.41 times higher than existing FPGA implementations.

D. *Comparison with Other Platforms*

We compare our FPGA-based accelerator with other platforms. We use the NVIDIA Tesla V100 desktop GPU and Intel Xeon CPU E5-2690 v4 running at 2.60 GHz, whose DDR4 memory size is 32 GB. The Operating System used in the experiment is Ubuntu 18.04 LTS with PyTorch version 1.4.0

TABLE IV
COMPARISON WITH PLATFORMS FOR OUR CNN FORWARDING (BATCH SIZE IS 128 AND INT8 PRECISION FOR THE CPU AND THE GPU).

Platform	CPU	GPU	FPGA
Device	Xeon E5-2690	Tesla V100	Virtex US+ XCVU9P
Clock Freq.	2.6 GHz	1.53 GHz	0.3 GHz
Memory	32GB DDR4	16GB HBM2	9.49 MB BRAM
Throughput [FPS]	24.0	350.0	**3321.25**
Power [W]	95	295	75
Efficiency [FPS/W]	0.25	1.18	**44.28**

with an INT8 quantization. We measure the system throughput and total power consumption for both platforms. To make a fair comparison, we used the same workstation for every platforms. Note that, for both the CPU and the GPU, we did not use our JPEG compression scheme because such transfer did not increase the system throughput.

Table IV compares our FPGA implementation with other platforms. To ensure the comparison is fair, we use the same CNN on all platforms. Compared with the Xeon CPU, it is 138.38 times higher throughput, and it dissipates 1.2 times lower power, so its efficiency was 177.12 times better. As for the Tesla V100 GPU, it is 9.48 times higher throughput, dissipates 3.9 times lower power, and its efficiency is 37.52 times better. Thus, we show that our pipelined architecture with a custom transfer data compression machine on the FPGA has superior throughput for the acceleration of a CNN.

VII. RELATED WORK

Many FPGA-based CNN inference accelerators have been studied, and these are covered in the surveys [15], [1]. Accelerators with low-bit precision were reported as a binary CNN accelerator [40] and ternary CNN fully-pipelined implementations [8], [7]. A multi-bit fully pipelined design was proposed [16]. The clock frequency is limited by the routing between the on-chip SRAM and DSP units, and many designs remain at 200-300 MHz. Xilinx researchers have used different operating frequencies for the DSP and its routing-channel. The DSP has been able to operate at peak operating frequencies (Xilinx UltraScale (741 MHz), UltraScale+ (891 MHz)) on different speed-graded FPGA chips [44]. There have been many reports of the optimization of CNN for an FPGA implementation. For example, DiracDeltaNet [46] combining shift operation, shuffle operation and point-wise convolution, DoReFaNet [21] optimized for low-bit parameters, MobileNetV2 combining separable convolution and residual structure [43], a binary precision VGG [6] and a binary ResNet [14] have been reported.

As for FPGAs on a cloud system, there are design frameworks for cloud FPGAs [11] and the implementation of a scalable FPGA inference system by Microsoft [13]. Datamining is one of the promising applications. For example, there is a report [26] that the accuracy of traffic volume prediction is improved by image super-resolution processing by a CNN. Features from images and texts are extracted by a CNN in order to build a predictive model based on these features [33].

As for the bandwidth problem, data compressors for algorithms and FPGA implementations have been reported. Specially designed compression algorithms for floating-point (FP) data achieved better performance than general-purpose ones such as GZIP and BZIP [27]. In addition, prediction-based compression algorithms have also been proposed, which predict the next input based on the previous input [18]. An FPGA-based lossless compressor has been developed which directly compressed floating-point data streams to enhance the actual memory bandwidth of the lattice Boltzmann method accelerator [22]. A design method of a parameterizable high-performance decoder with variable-length FPGA packets has been reported [37].

VIII. CONCLUSION

We compressed a transfer image introducing a constant quantization value of JPEG coding protocol and designed a customized image decoder architecture. We analyzed the trade-off between data transfer speed-up and recognition accuracy drop as for compression ratio. We implemented a fully pipelined architecture, which maps maps the computation from different layers to their own hardware unit with independent optimization. We trained a customized CNN including a channel shift and point-wise approximation. We also applied a binary weight training-aware quantization to reduce on-chip memory size. We implemented the proposed CNN inference accelerator on the Xilinx Inc. Virtex UltraScale+ FPGA VCU1525 acceleration development kit, which has the Xilinx Virtex UltraScale+ XCVU9P FPGA. Our compressed JPEG image transfer only consumed 4% of the system resource, dropped 0.3 point of accuracy and achieved 81,120 FPS which is 65.27 times faster than the conventional straightforward RGB data transfer. Thus, our proposed decoder architecture is sufficient to increase the system performance. As for the system throughput, our system 3.84-34.41 times higher than existing FPGA implementations. Compared with the Xeon CPU, it achieved 138.38 times higher throughput, and it dissipated 1.2 times lower power, so its efficiency was 177.12 times better. Compared with the Tesla V100 GPU, it achieved 9.48 times higher throughput, dissipated 3.9 times lower power, and its efficiency was 37.52 times better. Thus, we showed that our parallel architecture with transfer data compression on an FPGA has superior throughput for the acceleration of a CNN.

ACKNOWLEDGMENT

This research is supported in part by the Grants in Aid for Scientific Research of JSPS, Industry-academia collaborative R&D programs centre of innovation (COI) program, and Core Research for Evolutional Science and Technology (CREST). Thanks to NVIDIA support and Xilinx University Program (XUP).

REFERENCES

[1] K. Abdelouahab, M. Pelcat, J. Serot and F. Berry, "Accelerating CNN inference on FPGAs: A Survey," *arXiv:1806.01683*, 2018.

[2] Application Note 922, "A Fast Precise Implementation of 8x8 Discrete Cosine Transform Using the Streaming SIMD Extensions and MMX Instructions," https://www.cs.cmu.edu/ barbic/cs-740/ap922.pdf

[3] M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," *ICML*, 2017.

[4] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *TPAMI*, Vol. 32, No. 12, 2017.

[5] J. Bethge, H. Yang, M. Bornstein, and C. Meinel, "Back to Simplicity: How to Train Accurate BNNs from Scratch?," *arXiv: arXiv:1906.08637*, 2019.

[6] M. Blott, T. Preusser, N. Fraser, G. Gambardella, K. O'Brien, and Y. Umuroglu, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," 2018.

[7] A. P-. Boucle, A. Bourge and F. Ptrot, "High-Efficiency Convolutional Ternary Neural Networks with Custom Adder Trees and Weight Compression," *TRETS*, Vol. 11, No. 3, 2018, pp.1–24.

[8] A. P-. Boucle, A. Bourge, F. Ptrot, H. Alemdar, N. Caldwell, and V. Leroy, "Scalable high-performance architecture for convolutional ternary neural networks on FPGA," *FPL*, 2017, pp.1–7.

[9] Z. Cao, T. Simon, S.-E. Wei, and Y. Sheikh, "Realtime multiperson 2d pose estimation using part affinity fields," *CVPR*, 2017.

[10] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Semantic image segmentation with deep convolutional nets and fully connected crfs," *ICLR*, 2015.

[11] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An Open Framework for Mapping DNN Models to Cloud FPGAs," *FPGA*, 2019, pp.73–82.

[12] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations," *NIPS*, 2015, pp.3105–3113.

[13] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung and D. Burger, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," *ISCA*, 2018.

[14] M. Ghasemzadeh, M. Samragh and F. Koushanfar, "ReBNet: Residual Binarized Neural Network," *FCCM*, 2018, pp. 57–64.

[15] K. Guo, S. Zeng, J. Yu, Y. Wang and H. Yang, "A Survey of FPGA-based Neural Network Inference Accelerators," *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, Vol. 12, No. 1, 2018.

[16] L. Gong, C. Wang, X. Li, H. Chen and X. Zhou, "MALOC: A Fully Pipelined FPGA Accelerator for Convolutional Neural Networks With All Layers Mapped on Chip," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 37, No. 11, 2018, pp.2601–2612.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *CVPR*, 2016, pp. 770–778.

[18] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak, "Out-of-core compression and decompression of large ndimensional scalar fields," *Proc. of Eurographics*, Vol. 11, No. 2, pp.343–348, 2003.

[19] ImageNet training in PyTorch:
https://github.com/pytorch/examples/tree/master/imagenet

[20] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *ICML*, 2015.

[21] L. Jiao, C. Luo, W. Cao, X. Zhou, and L. Wang, "Accelerating low bit-width convolutional neural networks with embedded FPGA," *In Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, 2017, pp. 1-4.

[22] K. Katahira, K. Sano, and S. Yamamoto, "FPGA-based Lossless Compressors of Floating-Point Data Streams to Enhance Memory Bandwidth," *ASAP*, 2010, pp. 246–253.

[23] Y. Kim, "Convolutional neural networks for sentence classification," *EMNLP*, 2014.

[24] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "Fp-bnn: Binarized neural network on FPGA," *Neurocomputing*, 275:10721086, 2018.

[25] M. Lin, Q. Chen, and S. Yan, "Network in network," *ICLR*, 2014.

[26] Y. Liang, K. Ouyang, L. Jing, S. Ruan, Y. Liu, J. Zhang, D. S. Rosenblum and Y. Zheng, "UrbanFM: Inferring Fine-Grained Urban Flows," *ACM SIGKDD Conf. on knowledge discoverty and data mining (KDD)*, 2019, pp.3132–3142.

[27] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Trans. on Visual and Computer Graphics*, Vol. 12, No. 5, pp.1245–1250, 2006.

[28] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," *ECCV*, 2016.

[29] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation,'" *CVPR*, 2015.

[30] M. D. McDonnell, "Training wide residual networks for deployment using a single bit for each weight," *ICLR*, 2018.

[31] V. Nair and G. Hinton, "Rectified linear units improve restricted boltzmann machines," *ICML*, 2010.

[32] A. Newell, K. Yang, and J. Deng, "Stacked hourglass networks for human pose estimation," *ECCV*, 2016.

[33] M. Okawa, T. Iwata, T. Kurashima, Y. Tanaka, H. Toda and N. Ueda, "Deep Mixture Point Processes: Spatio-temporal Event Prediction with Rich Contextual Information," *KDD*, 2019, pp. 373–383.

[34] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv:1609.03499*, 2016.

[35] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," *arXiv preprint arXiv:1612.08242*, 2016.

[36] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *NIPS*, 2015.

[37] R. Sierra, F. Mangani, C. Carreras and G. Caffarena, "High-Performance Decoding of Variable-Length Memory Data Packets for FPGA Stream Processing," *FPL*, pp. 307–313, 2019.

[38] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, No. 550, 2017, pp.354–359.

[39] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CVPR*, 2015.

[40] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," *FPGA*, 2017.

[41] S.-E. Wei, V. Ramakrishna, T. Kanade, and Y. Sheikh, "Convolutional pose machines," *CVPR*, 2016.

[42] B. Wu, A. Wan, X. Yue, P. H. Jin, S. Zhao, N. Golmant, A. Gholaminejad, J. Gonzalez, and K. Keutzer, "Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions," *CVPR*, 2018, pp. 9127-9135.

[43] D. Wu, Y. Zhang, X. Jia, L. Tian, T. L, L. Sui, D. Xie, and Y. Shan, "A High-performance CNN Processor Based on FPGA for MobileNets," *29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp.136-143.

[44] E. Wu, X. Zhang, D. Berman and I. Cho, "A High-Throughput Reconfigurable Processing Array for Neural Networks," *FPL*, 2017, pp. 1–4.

[45] Xilinx Inc., "FPGAs in the Emerging DNN Inference Landscape," *WP514 (v1.0)*, 2019.

[46] Y. Yang, Q. Huang, B. Wu, T. Zhang, L. Ma, G. Gambardella, M. Blott, L. Lavagno, K. A. Vissers, J. Wawrzynek and K. Keutzer, "Synetgy: Algorithm-hardware Co-design for ConvNet Accelerators on Embedded FPGAs," *FPGA*, pp. 23-32, 2019.

[47] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," *FPGA*, 2015, pp. 161–170.

[48] X. Zhang, X. Zhou, M. Lin and J. Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," *CVPR*, 2018.

[49] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," *NIPS*, 2015.