# High Performance CNN Accelerators Based on Hardware and Algorithm Co-Optimization

Tian Yuan, Weiqiang Liu, *Senior Member, IEEE*, Jie Han, *Senior Member, IEEE*, and Fabrizio Lombardi, *Fellow, IEEE*

***Abstract*—Convolutional neural networks (CNNs) have been widely used in image classification and recognition due to their effectiveness; however, CNNs use a large volume of weight data that is difficult to store in on-chip memory of embedded designs. Pruning can compress the CNN model at a small accuracy loss; however, a pruned CNN model operates slower when implemented on a parallel architecture. In this paper, a hardware-oriented CNN compression strategy is proposed; a deep neural network (DNN) model is divided into "no-pruning layers ($NP$-layers)" and "pruning layers ($P$-layers)". A $NP$-layer has a regular weights distribution for parallel computing and high performance. A $P$-layer is irregular due to pruning, but it generates a high compression ratio. Uniform and incremental quantization schemes are used to achieve a tradeoff between compression ratio and processing efficiency at a small loss in accuracy. A distributed convolutional architecture with several parallel finite impulse response (FIR) filters is further proposed for the regular model in the $NP$-layers. A shift-accumulator based processing element with an activation-driven data flow (ADF) is proposed for the irregular sparse model in the $P$-layers. Based on the proposed compression strategy and hardware architecture, a hardware/algorithm co-optimization (HACO) approach is proposed for implementing a $NP-P$ hybrid compressed CNN model on FPGAs. For a hardware accelerator on a single FPGA chip without the use of off-chip memory, a 27.5× compression ratio is achieved with 0.44% top-5 accuracy loss for VGG-16. The implementation of the compressed VGG-16 model on a Xilinx VCU118 evaluation board processes 83.0 frames per second (FPS) for image applications, this is 1.8× superior than the state-of-the-art design found in the technical literature.**

***Index Terms*—Convolutional neural network (CNN), field programmable gate array (FPGA), network compression, hardware acceleration.**

## I. Introduction

CONVOLUTIONAL neural networks (CNNs) have been extensively used for image classification [1], [2] and recognition [3]. For better accuracy, CNNs require intensive and extensive computation. For real-time processing, CNNs are usually accelerated by parallel processors such as graphic processing units (GPUs) [4]. Although GPUs accelerate computation, a substantial increase of power limits its application to embedded systems. For low power and high performance digital systems, field programmable gate arrays (FPGA) [5]–[7] and application specific integrated circuit (ASIC) [8]–[10] have been used for CNN accelerators in recent years.

However, the on-chip memory resources in current FPGAs are not sufficient to completely store a large-scale CNN model. Therefore, off-chip memory is generally used in an FPGA implementation of CNNs; this causes a limitation in terms of bandwidth and speed. Therefore, model compression methods have been studied quite extensively. Among them, network pruning [11] is one of the most widely applied compression methods [12]–[14]. As a cost of improving compression ratio, the irregularity caused by pruning affects the performance of parallel computing. A compressed sparse model not only requires decoding but it also causes an imbalanced weights load and a difficulty in activations reading. In [15], it has been shown that processing a sparse layer takes dozens of milliseconds and requires a large memory utilization.

To achieve a better tradeoff between model size and performance of large CNNs, hardware-oriented compression and hybrid quantization strategies are proposed in this paper by requiring a smaller memory. By considering the processing feature of a CNN, the size of feature maps is reduced, but the model size expands as the layers deepen. The reduced feature maps require less computation, and the expanded models require a larger memory. As per the above characteristic, all layers are divided into two categories: "no-pruning layers ($NP$-layers)" and "pruning layers ($P$-layers)". With a regular weights distribution, a $NP$-layer utilizes parallel computing for high performance. A $P$-layer is irregular due to the pruning but it has a high compression ratio.

Leveraging the proposed compression strategy, the VGG-16 [1], one of the most useful CNN models for image classification, is implemented on a Xilinx VCU118 evaluation board without off-chip memory such as DRAM to store weight data. The proposed CNN accelerators achieve high performance because only on-chip memory in an FPGA is used. Based on the hardware-oriented compression-based architecture, a hardware/algorithm co-optimization scheme (HACO) is proposed for implementation of the CNNs. To the best of the author's knowledge, this is the first work that implements VGG-16 on

Manuscript received April 9, 2020; revised September 7, 2020; accepted October 7, 2020. Date of publication October 21, 2020; date of current version December 21, 2020. This work was supported in part by the National Natural Science Foundation of China under Grant 62022041 and Grant 61871216, and in part by the Six Talent Peaks Project in Jiangsu Province under Grant 2018XYDXX-009. This article was recommended by Associate Editor P. Li. *(Corresponding author: Weiqiang Liu.)*

Tian Yuan and Weiqiang Liu are with the College of Electronics and Information Engineering, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China (e-mail: liuweiqiang@nuaa.edu.cn).

Jie Han is with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB T6G 1H9, Canada.

Fabrizio Lombardi is with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115 USA.

Color versions of one or more of the figures in this article are available online at https://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2020.3030663

a single FPGA chip without the use of off-chip memory. The main contributions of this work are summarized as follows:

- A hardware-oriented compression method and a uniformly-incremental hybrid quantization strategy are proposed.
- The proposed compression strategy has been applied in the VGG-16 model and achieves a 27.5× compression ratio with a 2.04% top-1 accuracy loss and a 0.44% top-5 accuracy loss compared to the single-precision floating-point VGG-16 model using the ISVRC2012 test data set.
- A distributed convolutional architecture is proposed for FPGAs with a fast pipeline data path of CNNs.
- A shift-accumulator based processing element and an efficient activation-driven data flow are proposed for a sparse model.
- A hardware/algorithm co-optimization approach is proposed for high performance implementations on FPGAs.
- The proposed VGG-16 network is implemented on the Xilinx VCU118 evaluation platform achieving 30.3∼83.0 frames per second (FPS) with a compression ratio of 34.5∼27.5×, so attaining the highest performance compared with the state-of-the-art designs.

This paper is organized as follows. Section II provides the background of CNNs. The proposed compression strategy is presented in Section III. Section IV proposes a distributed FIR based hardware architecture for the $NP$-layers and a shift-accumulator based processing element for the $P$-layers. Section V evaluates the computation time and the hardware requirement. Section VI proposes a hardware/algorithm co-optimization method. Section VII provides the experimental results and analysis. Comparison with the state-of-the-art designs is also provided in this section. Section VIII concludes this paper.

## II. Background

### A. CNN Basics

CNNs extract the features of images and process these feature maps to classify images by finding and using the weights in each layer. In a deep learning algorithm, the CNN weights are found through training. A typical CNN has three types of layers: convolutional (Conv) layers, pooling layers and fully connected (FC) layers.

Convolutional layers are used to extract image features. A convolutional layer receives a feature map $X_{W,H,M}$ and generates a feature map of $Y_{W,H,N}$ by the filter $H_{K,K,M,N}$, which is calculated by:

$$Y(i,j,n)=\sum_{m=1}^{M}\sum_{q=1}^{K}\sum_{p=1}^{K}H(p,q,m,n)X(i\times s+p,\ j\times s+q,m) \tag{1}$$

where $W$ and $H$ indicate the width and the height of the feature map; $K$ indicates the size of convolution kernel; $M$ and $N$ indicate the input channels and output channels; $s$ is the stride of filter.

Pooling layers compute a local field of feature map to output a pixel, so reducing the size of feature maps. Average and max pooling are the two typical pooling operations that are commonly used in CNNs. Average pooling computes the average value of the local field while max pooling selects the largest value of the local field as the output. Max pooling is used in this work due to its high efficiency.

FC layers are the last few layers in a CNN. All input neurons are fully connected to every neuron in the next layer through weights. Therefore, FC layers have many weights to be stored.

$O$ denotes the number of multiply-accumulate (MAC) operations required in each layer (including both $O_{FC}$ and $O_{Conv}$):

$$\begin{aligned} O_{FC} &= U_{in} \times U_{out} \\ O_{Conv} &= (W \times H \times N) \times (K \times K \times M) \end{aligned} \tag{2}$$

where $U_{in}$ and $U_{out}$ denote the number of input and output neurons, respectively.

### B. Related Works

A tiling technique [5], [16] has widely been used to address insufficient memory in embedded architectures. For image compression, the adaptive joint photographic experts group (JPEG) method [17] has been proposed to dynamically adjust the compression ratio for the desired quality. For speeding up convolution, a fast finite impulse response (FIR) algorithm (FFA) [6] has been proposed. The Winograd algorithm has been studied for sparse networks [18]. To achieve a high throughput, [7] has presented a design method to fully exploit the limited resources in FPGAs. Approximate computing has widely been studied in recent years; its objectives are to achieve low energy and high performance at an acceptable accuracy loss [19]. Neural networks require a significant large-scale computation and have high error resilience, so suitable for approximate computing. For example, approximate multipliers [20] can be used in CNNs with a very small loss of accuracy [21]; however, this method has only been applied to small-scale neural networks.

Deep compression has been proposed in [12] to reduce the model size of CNNs. Using network pruning [11], weight quantization and Huffman coding, a high compression ratio has been achieved. A so-called dynamic network surgery has been used to accelerate training and avoid unacceptable pruning [13]. Binary neural networks (BNNs) have been proposed to train deep neural networks (DNNs) with weights and activations constrained to $+1$ or $-1$ [22]. At a reduced complexity and a small number of weights, BNNs achieve a high performance for embedded systems [23], [24]. Activations for very low bit-width has also been proposed, thus saving memory and accelerating training.

Incremental Network Quantization (INQ) [25] has been proposed for efficient CNN models with low precision weights. INQ divides weights into several groups, and incrementally quantifies each set of data. At each time of quantization process, a network is retrained and weights that have not been quantified are updated to compensate for the accuracy loss. Using the INQ algorithm, weights in a network can be quantified as $\pm 2^n$ with only a small accuracy loss, where $n$ is an integer. The MAC is the main arithmetic unit in CNNs. With the quantized data form, multiplication can be replaced

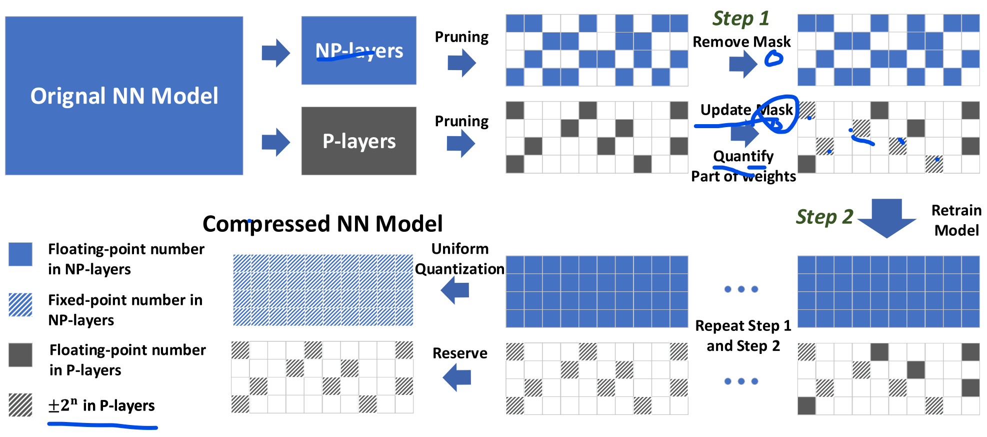

Fig. 1. Overview of the hardware-oriented compression strategy.

by shift operations in a MAC; therefore, INQ has a high speed performance on hardware.

The significant difference between INQ and the proposed compression strategy is that the proposed strategy targets sparse neural network models, whereas the original INQ causes an unacceptable accuracy loss for sparse models. Due to the use of non-pruning layers (NP-layers) defined in this work, the proposed method compensates the accuracy loss from the pruning-layers (P-layers) in sparse networks. Furthermore, the implementation of NP-layers can increase parallel processing performance due to its regular structure.

## III. The Proposed Compression Strategy

We propose a hardware-oriented compression strategy for large-scale CNNs to store weights on a chip. The front layers incur a significant level of computation but with a smaller model size. Also, the front layers are less error-resilient: [12] has applied pruning (albeit at a smaller rate) in the front layers of VGG and AlexNet. [26] shows that by pruning the filters in the front layers, the accuracy significantly decreases. In addition, an irregular sparse model affects the performance of parallel computation. Hence, pruning the front layers is not efficient. The principle of the proposed strategy is to apply different compression strategies to different layers, so making the network to be front-regular but back-irregular.

All layers are divided into two types: $NP$-layers and $P$-layers. In the beginning, all layers are pruned. Then INQ is used for quantifying the $P$-layers. During the incremental quantization, the error introduced by pruning and quantization in the $P$-layers is compensated by the weight update of $NP$-layers, in which the $NP$-layers are no longer sparse. Therefore, the proposed compressed model has more error resilience, hence making the model easier to converge. When the quantization in the $P$-layers is completed, the $NP$-layers are quantified as fixed-point numbers so ready for computation.

In general, the $NP$-layers have a regular structure for higher parallel performance, while the $P$-layers are highly compressed for higher compression ratio. The $NP$-layers are generally Conv layers, but the $P$-layers can include Conv layers and FC layers. The overview of the proposed strategy is shown in Fig. 1.

### A. Hybrid Quantization Strategy

For $P$-layers, INQ is applied to achieve a higher compression ratio. For a layer $l$, the weights are stored in an array $W_l$. The network sparsity is determined by $T_l$ as a binary array with the same size of $W_l$. $T_l$ is calculated by the following equation:

$$T_l(i) = \begin{cases} 0 & W_l(i) = 0 \\ 1 & W_l(i) \neq 0 \end{cases} \quad (3)$$

A 0 and 1 in $T_l$ indicates that the corresponding weight is zero or non-zero, respectively. At each time of quantization, 1s in the array $T_l$ are randomly divided into two groups: $A_1$ and $A_2$. The data in $A_2$ will be modified to zero which indicates that the corresponding weight will be quantified. The weight quantization in the $P$-layers is represented as:

$$W_l(i) = \begin{cases} +2^{\lfloor \log_2 |W_l(i)| \rfloor} & W_l(i) > 0 \\ -2^{\lfloor \log_2 |W_l(i)| \rfloor} & W_l(i) \leq 0 \end{cases} \quad (4)$$

By using quantization, the original multiplication in a MAC is replaced by a shift operation. Therefore, only the sign bit and exponent need to be stored for the direction and the number of bits to shift, respectively. After quantization, the model is retrained to compensate for the error. Weights are updated as per the following equation:

$$W_l(i) \leftarrow W_l(i) - \eta \frac{\partial E}{\partial W_l(i)} T_l(i) \quad (5)$$

where $\eta$ is the learning rate that depends on the learning policy; $E$ is the objective function. $T_l(i)$ denotes the mask of weights, which determines whether the weight must be updated. For zeros entries in $T_l$, the corresponding weight will not be updated because it is already zero or has been quantified. For the $NP$-layer $l$, the weights are quantified after the quantization of the $P$-layers. The following uniform quantization method is applied:

$$W_l(i) = round(\frac{W_l(i)}{Q})Q \quad (Q = 2^{-n}) \quad (6)$$

where $Q$ indicates the quantization factor and $n$ is the fraction bit. For the case that most of weights are less than 1, the following quantization method is used:

$$W_l(i) = \begin{cases} 1 - Q & W_l(i) \geq 1 \\ -1 & W_l(i) < -1 \\ round(\frac{W_l(i)}{Q})Q & W_l(i) \in [-1, 1) \end{cases} \tag{7}$$

**Algorithm 1** Hybrid Sparse Network Quantization Strategy

```
Input: $P$-layers and $NP$-layers
Output: Quantized network model $QM$
 1: for all $layers \in P$ do
 2:   Compute $T_l$ through Eq. (3)
 3: end for
 4: Set grouping ratio $r$
 5: for all $layers \in P$ do
 6:   Randomly divide index of 1 from $T_l$ into two parts: $A_1$, $A_2$ by grouping ratio $r$
 7:   Set $T_l(i)$ $(i \in A_2)$ to 0
 8:   Quantify $W_l(i)$ $(i \in A_2)$ through Eq. (4)
 9: end for
10: Retrain and update weights by Eq. (5)
11: Goto 4 until all weights in $P$-layers have been quantified
12: for all $layers \in NP$ do
13:   Quantify $W_l$ through Eq. (6) or Eq. (7)
14: end for
15: return $QM = P \cup NP$
```

In the proposed compression strategy, weights in the $NP$-layers are given by fixed-point numbers, so processed without decoding and faster compared to the compressed weights in the $P$-layers. The weights in the $P$-layers are discrete in $\pm 2^n$, so requiring less memory due to the lower bit width and smaller quantity. Moreover, the multiplication is replaced by shift operations in the $P$-layers, requiring less resources. Overall, the compressed CNN models have a high performance on hardware implementation. The hybrid quantization strategy is detailed in Algorithm 1.

### B. Compression Experiments and Results

We implemented the proposed compression strategy on the Caffe [27] platform to compress the VGG-16 [1] model; this has been trained and tested with ILSVRC2012 data set [28]. By using the proposed strategy, 96%, 96% and 77% of the weights are pruned in three FC layers. For quantization, weights bit-widths are set to 8-bit and 4-bit for $NP$-layers and $P$-layers. Moreover, 5 bits are set to store the index for a weight in the pruned layers, which indicates the number of zeros between two adjacent no-zero weights. Most of the weights in the Conv layer are less than 1; therefore, we employ Eq. (7) for the $NP$-layers. Overall, weights are incrementally quantified from 50%, to 75%, to 87.25%, and to 100%. In addition, all activations bit-widths are set to 16-bits. In these experiments, a 20.1MB compressed model is utilized with a 33.94% top-1 error and a 12.44% top-5 error, as shown in Table I. Compared to the single-precision floating-point VGG-16 model, we achieve a 27.5× compression ratio with 2.04% top-1 accuracy loss and 0.44% top-5 accuracy loss.

TABLE I
VGG-16 COMPRESSION RESULT

| Data Type | Top-1 error | Top-5 error | Size (MB) |
|---|---|---|---|
| Baseline | 31.90% | 12.00% | 553MB |
| Ours | 33.94% | 12.44% | 20.1MB |

The standard VGG-16 model used in this work is downloaded from Caffe Model Zoo[1] (as widely used). We tested this model on the ImageNet dataset 2012 through the framework Caffe; and obtained the same accuracy as in [29].

## IV. THE PROPOSED HARDWARE ARCHITECTURE AND IMPROVED DATA FLOW

Most state-of-the-art CNN designs on FPGAs use off-chip memory to store weights and feature maps. A key advantage of the proposed compression strategy is to implement VGG-16 on the latest Xilinx UltraScale FPGA using on-chip memory only.

Based on the proposed compression strategy, a new hardware architecture for FPGA implementation is developed with a fast pipeline dataflow of the CNNs without off-chip memory. As mentioned in Section III, the $NP$-layers are in front of the $P$-layers. Therefore, the compressed model can be processed in a pipelined manner.

Two hardware architectures are proposed to process these two types of layers. Due to the regular convolution model in $NP$-layers, an FIR based convolution processing element and an improved data flow are proposed for the $NP$-layers. For the irregular sparse model in $P$-layers, a parallel shift-accumulator based processing element is proposed to reduce the redundant computation in the parallel processing of the sparse models.

### A. Conv Processing Element

A 1-D convolution is processed by a FIR filter, and multiple parallel FIR filters (the number of FIR filters is denoted as $F$) compute the 2-D convolution. The FIR filter is efficient in convolution processing due to its stringent requirement on bandwidth. Considering a $3 \times 3$ convolution, three inputs are required for an output in the FIR filters, while nine inputs are required in traditional methods.

Reference [6] has proposed a parallel fast FIR algorithm (FFA) for CNNs, which can save multiplications in convolution. Compared with [6], the proposed FIR filter design uses cut-set retiming for the convolution kernel; this approach requires less resources and can be used to process larger convolution kernels very efficiently. So, it is possible to highly parallelize processing in higher levels with less resources. As shown in Fig. 2, in our design, a 3-tap FIR filter is retimed to improve the frequency, and 3 parallel retimed FIR filters constitute a $3 \times 3$ convolution processing element (Conv-PE) to compute the 2-D convolution.

[1]https://github.com/BVLC/caffe/wiki/Model-Zoo

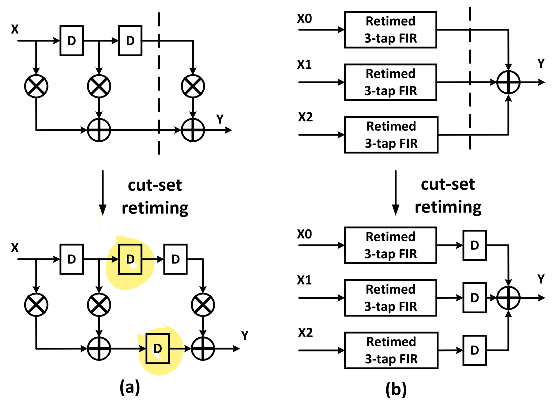

Fig. 2. Convolution processing element (Conv-PE): (a) a cut-set retimed 3-tap FIR; (b) a cut-set retimed 3 × 3 convolution processing element.

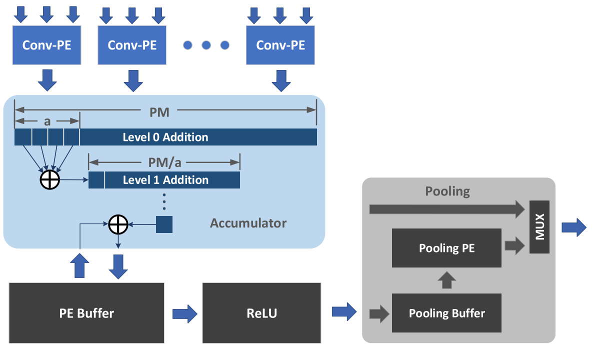

Fig. 3. Complex convolution processing unit (CCPU).

A complex convolution processing unit (CCPU) consists of several Conv-PEs; the number of Conv-PEs in a CCPU is denoted by $M_{np}$. As Fig. 3 shows, a CCPU computes $M_{np}$ channels of a feature map in parallel. The $M_{np}$ data output from the Conv-PEs is also accumulated. To reduce the delay, the $M_{np}$ data are divided into $M_{np}/a$ groups, where $a$ denotes the number of data that is added in a group. Therefore, the entire addition is divided into $\lceil \log_a(M_{np}) \rceil$ levels. Additionally, when $M_{np}$ is less than the input channel $M$, the accumulated data will be temporarily stored in the CCPU buffer; it will be sent to the accumulator in the next iterations. After $M/M_{np}$ iterations, the CCPU outputs a channel of the feature map.

Pooling layers may occasionally follow the Conv layers. When pooling is required, the data output from ReLU is stored in a pooling buffer. Several rows of data need to be stored prior to computing.

## B. Improved Data Flow in Conv Layers

For a better performance of 2-D convolution by FIR filters, an improved data flow in Conv layers is proposed; it integrates a 2-D feature map matrix into an array. To integrate each row of the feature map, zeros are filled between two neighbor rows, which are of no use after convolution. These values are reset to zero in the ReLU module.

A Conv-PE receives three parallel arrays and output one array. To acquire an array with a length of $W$, the process

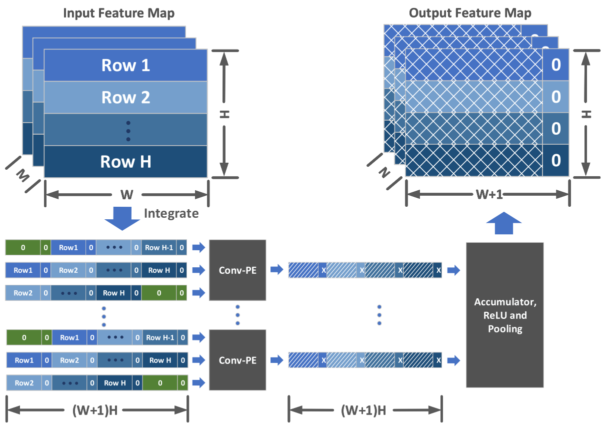

Fig. 4. Feature map integration for 3 × 3 kernel and the data flow in Conv layers.

takes $W + C_{fir}$ cycles, where $C_{fir}$ denotes the latency of a FIR. To acquire a $W \times H$ matrix, the process costs $(W + C_{fir}) \times H$ cycles. To eliminate the redundancy caused by the FIR latency $C_{fir}$, neighbor rows of the feature map are connected by filling $K$ zeros. By applying the so-called integrated rows of the feature map, the $W \times H$ matrix becomes three $(W + K - 2) \times H$ arrays that requires $(W + K - 2) \times H + C_{fir}$ cycles, where $K$ is the width of the Conv kernel. As shown in Fig. 4, Row 1 to Row $H$ are integrated as the second input array of the Conv-PE. In particular, the first and last arrays must be padded with zeros before processing. When $W$ is small and $C_{fir}$ is large, such as in the last several layers of VGG-16, performance can be significantly improved.

It is worth mentioning that for convolution kernel of different sizes, zero padding is also different. The processing speed has been increased with the improved data flow by slightly requiring more memory.

## C. F×F Ping-Pong Buffer for Conv-PEs

$F$ parallel FIR filters require an $F$ times sampling rate of the non-parallel version. As the on-chip storage is limited, an $F \times F$ ping-pong buffer (FPPB) is proposed.

A $F$ parallel FIR requires $F$ lines of input data in the proposed hardware architecture. The data in the feature map is stored in rows. Then the $F$ data in different addresses must be read to match the processing speed for parallel performance. As shown in Figs. 5 and 6, we first integrate the $F$ adjacent data in the feature map. Then the integrated data (A, B, C…) are sequentially sent to the FPPB, which consists of two $F \times F$ blocks, namely, BLOCK0 and BLOCK1. These two blocks are controlled to alternatively receive and output data. Once a block is full, it outputs data in columns while the other block receives the data.

This scheme can transform spatial relations of data in a feature map to match the parallel FIR processing, while requiring less memory compared with the conventional caching method [6]. Through the FPPB, the bandwidth requirement is also relaxed with a small hardware utilization.

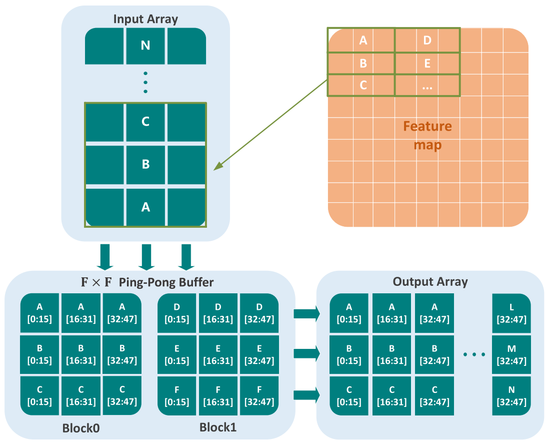

Fig. 5. F×F ping-pong buffer for parallel FIRs.

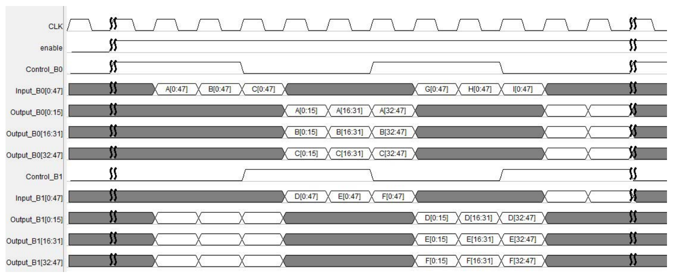

Fig. 6. The timing diagram of the proposed FPPB.

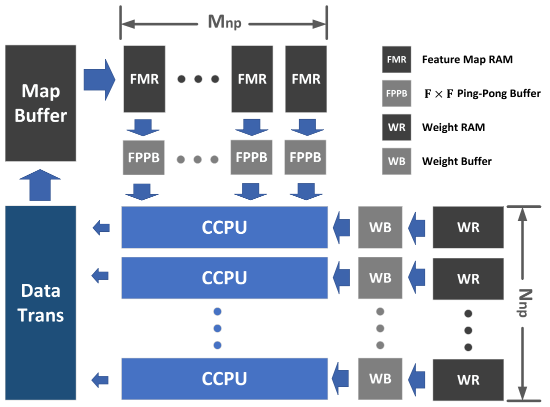

Fig. 7. The distributed convolution architecture.

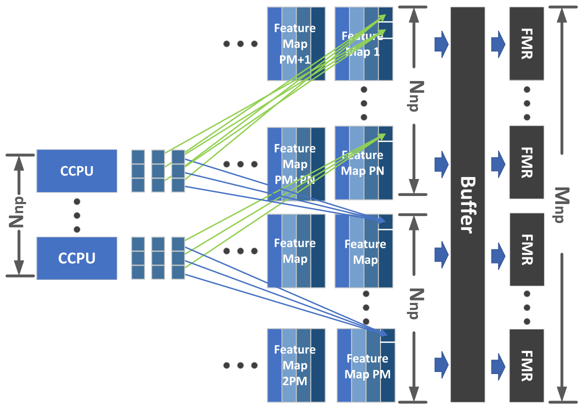

Fig. 8. Feature map storage format in FMRs.

## D. Distributed Conv Architecture

With no off-chip memory, a distributed Conv architecture is proposed for high speed processing. Each Conv-PE in the same CCPU has a RAM to transmit the feature map. In addition, different CCPUs receive weights from different RAMs; so, the speed of data transmission matches the processing speed, hence relieving the restriction of bandwidth.

As shown in Fig. 7, there are $M_{np}$ feature map RAMs (FMRs) to store the feature map, and each one corresponds to a FPPB. The $M_{np}$ data in the feature map is sent to each Conv-PE after the conversion of FPPB, so the $M_{np}$ input channels are processed at the same time.

In the Conv architecture, each CCPU processes a 3-D feature map to output a 2-D feature map, as briefly discussed in Section IV-A. $N_{np}$ denotes the number of CCPUs that process the convolution in parallel. Similarly, $N_{np}$ weight RAMs (WRs) store the weights in a distributed fashion, such that each WR corresponds to a CCPU. All CCPUs receive the same data of the input feature map. Due to the different convolution kernels, this implies that $N_{np}$ channels are output at the same time.

The entire convolution architecture processes $M_{np}$ channels of input feature map in parallel, and outputs $N_{np}$ channels of the output feature map.

$N_{np}$ is generally smaller than the number of output channels $N$; therefore, a buffer is required for temporarily storing an intermediate feature map. Using the data integration method of Section IV-C, the serial data that is output by a CCPU, is converted into $F$ parallel streams and stored in the map buffer. The map buffer receives $N_{np}$ channels of data and output $M_{np}$ channels of data. If the input feature map data of the current layer is of no use, then they are replaced by the data from the map buffer for the next layer computation.

In the proposed distributed architecture, the feature map storage format is determined by $M_{np}$. As shown in Fig. 8, the first $M_{np}$ feature maps are set as a group, stored in the bottom of FMRs. The next groups are subsequently stored in a stack fashion. Due to the different size of feature maps between layers, the indexes of the next groups are hard to be determined. To ensure the architecture work correctly, $M_{np}$ should be a multiple of $N_{np}$. Dual port RAMs are used in this implementation and there is no read and write conflict.

## E. Shift-Accumulator Based Processing Architecture for P-Layers

Based on the proposed compression strategy, the $P$-layers make a highly compressed sparse model with $\pm 2^n$ weight types. Multiplications are replaced by shift operations to reduce resources. To accelerate the processing of the $P$-layers, there are two levels in this parallel computing scheme:

- Unrolling the multiple shift-accumulations.
- Unrolling the different weight kernels (Conv kernel in Conv layers or all weights connected to a output neuron in FC layers). Due to the irregular sparse model, computations of each weight kernels are different, so decreasing the utilization of the hardware units.

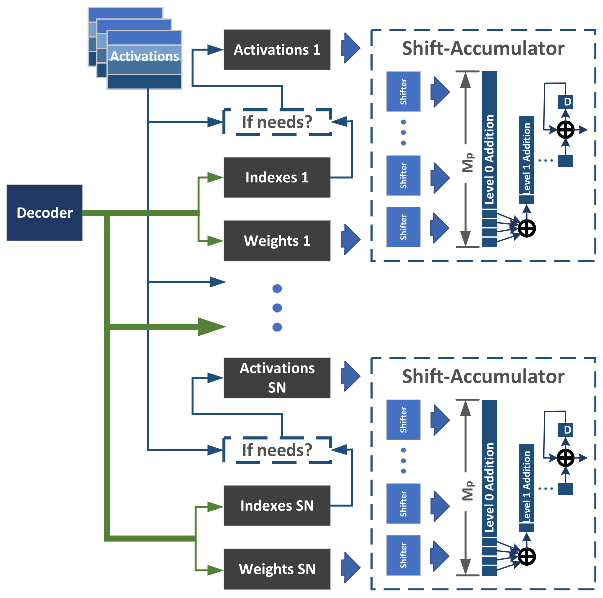

Fig. 9. The architecture of SAC-PU and activations-driven data flow (ADF).

Fig. 9 shows the $M_p \times N_p$ shift-accumulator based processing unit (SAC-PU). A shift-accumulator consists of $M_p$ shifters while a SAC-PU consists of $N_p$ shift-accumulators. $M_p$ shift-accumulations are executed in a shift-accumulator; the $N_p$ shift-accumulators process different sparse weight kernels.

To fully exploit the sparsity of the model, computations in the shift-accumulator are selected by the non-zero weights. Thus, activations must be read, and they require specific indexes that depend on weights. As the sparse weight kernels must be considered together, different indexes cause issues when reading activations. So, an activations-driven data flow (ADF) is proposed for the $P$-layers to reduce redundant computations and improve the read speed of activations. When an activation is not needed in a sparse weight kernel, it may still be needed in other weight kernels; therefore, an activation is processed at every shift-accumulator to assess whether needed. Activations are processed in an active mode, so this is referred to as ADF.

The efficiency of ADF is determined by $N_p$ and the sparsity of $P$-layers. Eq. (8) is used to evaluate the efficiency, where $R_{kp}$ denotes the pruning rate of each weight kernel. When $E_{adf}$ is larger than 1, ADF is faster than traditional methods. A large $N_p$ value implies that the activation is likely to be accepted by other weight kernels, so increasing the efficiency of ADF. However, $N_p$ cannot take a very large value because it would require a substantial increase hardware; therefore, $M_p$ can be reduced for a larger $N_p$.

$$E_{adf} = \frac{Activations \times \sum^{N_p}(1 - R_{kp})}{Activations} \tag{8}$$

The pruning rate of each weight kernel's is difficult to find analytically; therefore, the average of the pruning rate in a layer can be used to evaluate the efficiency, that can be used to determine $N_p$.

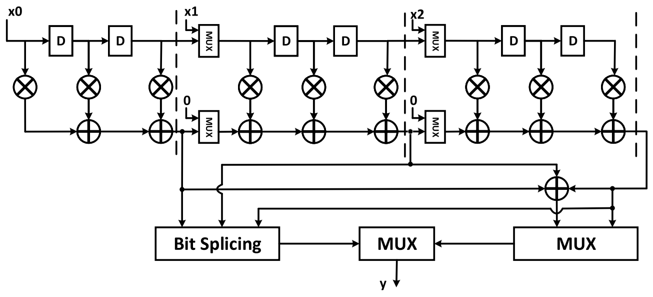

Fig. 10. The transformed Conv-PE for general CNN computation.

The architecture of SAC-PU has high fanouts when $N_p$ is large. To reduce the influence of a large net delay, the output registers of the activations and decoder can be copied for sharing the loads.

## F. *Generalization of the Proposed Architecture*

The proposed $F \times F$ Conv-PE also computes smaller Conv kernels, as determined by the input weights. For example, 4 input non-zero weights and 5 input zeros correspond to a $2 \times 2$ Conv kernel. The $3 \times 3$ Conv kernel is widely used in CNNs, so with good efficiency when the size of the Conv-PE is $3 \times 3$.

Some CNNs use larger Conv kernels in some layers, such as $7 \times 7$ in the first layer of ResNet [2], but this is well beyond the capability of a Conv-PE; however, it is not always effective to increase the size of Conv-PE due to rare use of large Conv kernels; therefore, a general Conv-PE is proposed for CNNs at a high efficiency.

Consider a $3 \times 3$ Conv-PE and three 3-tap FIR filters connected in series, as shown in Fig. 10, MUXs are inserted in each connection to select the functions of the Conv-PE.

- **Parallel Mode:** When Conv kernels are smaller than or equal to $3 \times 3$, the FIR filters receive multiple data from the feature map to compute at the same time multiple rows. A Conv-PE computes 2-D convolution. However, when the size of Conv kernel is $1 \times 1$, this Conv-PE is only 11.1% as powerful as the $3 \times 3$ scheme; therefore, the output bit-width must be expanded 3 times for 3 output data when Conv kernel is $1 \times 1$.
- **Serial Mode:** When the Conv kernels are larger than $3 \times 3$, each FIR filter receives the data from the previous filter to configure as a larger FIR filter. Each Conv-PE loads one row of weights for 1-D convolution. If the kernel size is smaller than $9 \times 9$, zeros will be filled for the FIR filter. Thus, there may be some inefficiency when the Conv-PE operates in serial mode; however as large Conv kernels are a small part in most networks, the performance will not be significantly affected. Multiple executions of the 1-D convolution are required for a 2-D convolution. For example, if the size of the kernel is $7 \times 7$, the feature maps should be input 7 times for a 2-D convolution. However, in some cases, the multiple 1-D convolutions can be unrolled; for example, if $M_{np} = 32$ and $M = 3$, each seven Conv-PEs can load 49 weights for the $7 \times 7$ convolution. The 7 outputs from Conv-PEs are added in the accumulator. These seven Conv-PEs

compute a 2-D convolution; also, the 3 input channels should be expanded to 21 for parallel computing.

The proposed general Conv-PE can switch modes between 1-D and 2-D convolution with a processing capability for a Conv kernel smaller than $9 \times 9$. The serial FIR filters cause a significant delay, that must be retimed for high performance.

## V. Evaluation

Under the $NP$-$P$ hybrid model, the performance of each module is analyzed. In this section, computation time and hardware utilization are quantitatively evaluated for the proposed optimization algorithm of Section VI.

### A. Time Analysis of $NP$-Layer

For processing the Conv layers using the proposed CCPUs, a 2-D convolution takes $(W+1) \times H + C_{fir}$ clock cycles, so all data of the feature map is read once. Consider the input and output channels, then the clock cycles of all $NP$-layers can be computed as follows:

$$C_{np} = \sum^{NP} \left[(W+1) \times H + C_{fir}\right] \times \left\lceil \frac{M}{M_{np}} \right\rceil \times \left\lceil \frac{N}{N_{np}} \right\rceil \quad (9)$$

If Conv-PEs work in a serial mode, it will take an additional time given by a number of clock cycles with a factor of $KH$ compared to the parallel mode. Consider the unrolling of convolution, then Eq. (9) is computed as follows:

$$C_{np} = \sum^{NP} \left[(W+1) \times H + C_{fir}\right] \times \left\lceil \frac{M \times K}{M_{np}} \right\rceil \times \left\lceil \frac{N}{N_{np}} \right\rceil \quad (10)$$

As weights are loaded in the buffer prior to the next convolution, no additional time overhead is encountered. Due to the map buffer, the feature map can be loaded in FMRs when the map data in the last iteration is of no use. Therefore, the total time is given by:

$$T_{np} = C_{np}(M_{np}, N_{np}) \times t_{np} \quad (11)$$

For storage, the largest feature map and the number of weights determine the size of the used memory. As per reuse, $N_{np}$ affects the size of the map buffer. Additionally, each CCPU requires a buffer to store at least a feature map for accumulation. Therefore, the entire memory is given by:

$$MS_{np} = \frac{2N - N_{np}}{N} \times \max_{NP}(W \times H \times M) \times b_a + N_{np} \max_{NP}(W \times H) \times b_a + \sum^{NP} Q_l \times b_w \quad (12)$$

where $b_a$ and $b_w$ denote the activation bit-width and weight bit-width; $Q_l$ denotes the weight number of a layer. In general, the largest feature map can be found in the first several layers; thus the layer dividing method does not affect the size of the memory in the $NP$-layers.

Computation hardware is determined by $M_{np}$ and $N_{np}$, which mostly originate from the Conv-PEs and the accumulator. The number of multipliers and adders are given as follows:

$$Mul_{np} = 9M_{np} \times N_{np}$$
$$Add_{np} = \left(8M_{np} + \sum_{i=1}^{\lceil \log_2(M_{np}) \rceil} \left\lceil M_{np}/2^i \right\rceil\right) \times N_{np} \quad (13)$$

In Eq. (13), all adders are converted into a two-input form for evaluation.

### B. $P$-Layers Computation Time Evaluation

The computation time in the $P$-layers can be divided into three parts: weights decoding, activation reading and SAC computing. The compressed weights are decoded into weights and indexes, stored in caches. Then the activations are read by ADF. After all data is cached, the SAC-PU processes such data. Each part operates based on the previous parts. Therefore, the time consumption in each part must be accumulated.

For weights decoding, the required number of clock cycles depends on the number of weights after pruning:

$$C_w = \sum^{P} Q_l \times (1 - R_{lp}) \quad (14)$$

where $R_{lp}$ denotes the pruning rate of a layer.

The $P$-layers may include Conv layers and FC layers, so two cases of clock cycle utilization must be differentiated:

$$C_a = \sum^{P_{Conv}} \max(L) \times W \times H \times \left\lceil \frac{N}{N_p} \right\rceil + \sum^{P_{FC}} \max(L) \times \left\lceil \frac{U_{out}}{N_p} \right\rceil \quad (15)$$

where $L$ is the length of the activations (equal to the last index value of the weight kernels).

Similarly, the number of clock cycles required for SAC computing is given by:

$$C_s = \sum^{P_{Conv}} \left\lceil \frac{\max(Q_k(1 - R_{kp}))}{M_p} \right\rceil \times W \times H \times \left\lceil \frac{N}{N_p} \right\rceil + \sum^{P_{FC}} \left\lceil \frac{\max(Q_k(1 - R_{kp}))}{M_p} \right\rceil \times \left\lceil \frac{U_{out}}{N_p} \right\rceil \quad (16)$$

where $Q_k$ is the weight number and $R_{kp}$ is the pruning rate of a weight kernel, different from Eq. (14).

As per the above equations, the total time execution of the $P$-layers is given as follows:

$$T_p = C_w(R_{lp})t_w + C_a(L, N_p)t_a + C_s(R_{kp}, M_p, N_p)t_s \quad (17)$$

A pipelined design can be achieved between reading activations and the SAC computation for the processing time to be overlapped. Therefore, the time execution is given by:

$$T_p = \begin{cases} C_w(R_{lp})t_w + C_a(L, N_p)t_a & C_a t_a \geq C_s t_s \\ C_w(R_{lp})t_w + C_s(R_{kp}, M_p, N_p)t_s & C_a t_a < C_s t_s \end{cases} \quad (18)$$

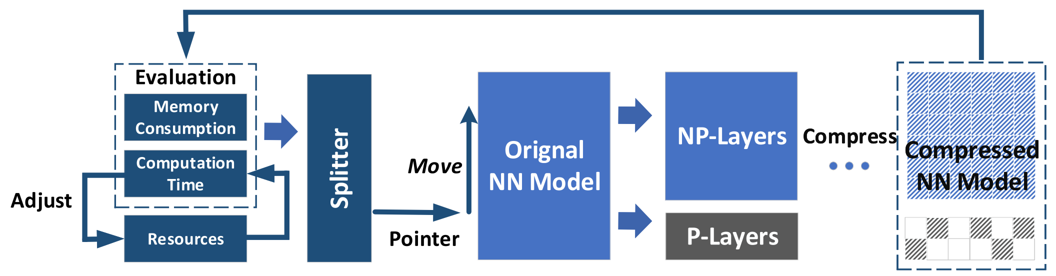

Fig. 11. Overview of hardware/algorithm co-optimization (HACO).

Storage in the $P$-layers is function of the largest feature map, the number of weights and the size of the weight cache. The memory size is given by:

$$MS_p = 2\max_{P}(W \times H \times M) \times b_a + \sum^{P} Q_l \times b_w + \sum^{N_p} N_p \times \max(Q_k) \times (b_w + b_i) \quad (19)$$

where $b_i$ denotes the index bit-width.

Computational resources are determined by the number of shifters and accumulators, which is the same as the $NP$-layers:

$$Sht_p = M_p \times N_p$$
$$Add_p = \sum_{i=1}^{\lceil \log_2(M_p) \rceil} \left\lceil M_p/2^i \right\rceil \times N_p \quad (20)$$

In Eq. (20), all adders are also converted into a two-input form for evaluation.

## VI. Hardware/Algorithm Co-Optimization

In this section, HACO is proposed to find the best design by dividing the layers into $NP$-layers and $P$-layers and by proper sizing the CCPUs and SAC-PU, hence reducing the computation time while retaining a high efficiency. The overview of HACO is shown in Fig. 11.

### A. Optimization Objective

For pipeline processing, $NP$-layers and $P$-layers are processed by CCPUs and SAC-PU, respectively. As Fig. 12 shows, the $NP$-layers are processed by the CCPUs first. After the computation of the $NP$-layers has been completed, the feature maps as output of the CCPUs are sent to the SAC-PU. Meanwhile, the CCPUs execute the next computation. Thus, the entire computation time is determined by the slower step in the entire process.

Consider the limited resources on FPGA, then this problem can be formulated as:

$$\begin{aligned} &min\ T_{np}(M_{np}, N_{np}) \\ &s.t.\ T_{np}(M_{np}, N_{np}) \geq T_P(L, R_{lp}, R_{kp}, M_p, N_p) \\ &\qquad Resources\ Utilization(M_{np}, N_{np}, M_p, N_p) \leq V_r \\ &\qquad Accuracy(L, R_{lp}, R_{kp}) \geq A_r \end{aligned} \quad (21)$$

where $A_r$ and $V_r$ are constant, that are determined by user requirements.

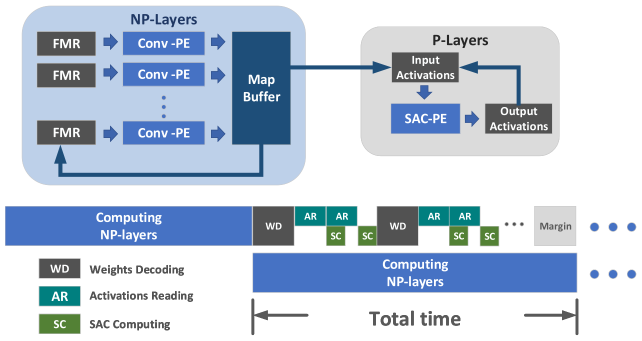

Fig. 12. Time consumption of the pipelined hybrid system.

$T_{np}$ and $T_p$ are affected by multiple factors; among them, the frequency is the most difficult to be evaluated quantitatively. However, it can be assessed after synthesis. By utilizing more hardware, frequency could decrease, so a loss in performance, which is often unavoidable. Therefore, only parameters such as $M_{np}$, $N_{np}$, $L$, $R_{lp}$, $R_{kp}$, $M_p$, $N_p$ and layer divisions need to be considered.

### B. Goal Simplification

For efficiency of the hardware, the considered parameters are analyzed under a few sampling assumptions. As mentioned in Section IV-D, $M_{np}$ is a multiple of $N_{np}$. Therefore, $T_{np}$ is denoted as $T_{np}(k, N_{np})$, where $k$ is a positive integer. However, to fully reuse the memory, $N_{np}$ must be increased as much as possible; due to hardware limitation, it can be achieved when $k$ is equal to 1.

$L$ and $R_{kp}$ are related to each weight kernel (that must be considered when implementing pruning). To ensure correctness, $L$ is given by the largest amount such that all activations are read once per shift-accumulator. Therefore, there is a difference in values between the actual $T_{np}$ and $T_p$.

$R_{kp}$ is difficult to determined prior to training due to the irregular pruning. Provided that SAC computing is faster than the process of activation reading, the proposed architecture operates correctly. In the case that the same frequencies are used for them, $M_p$ should be less than or equal to the number of activations read out in a clock cycle. Due to the above simplification, the process is now given by:

$$\begin{aligned} &min\ T_{np}(N_{np}) \\ &s.t.\ T_{np}(N_{np}) - T_p(R_{lp}, M_p, N_p) \geq Margin \\ &\qquad Resources\ Utilization(k, N_{np}, M_p, N_p) \leq V_r \\ &\qquad Accuracy(R_{lp}) \geq A_r \end{aligned} \quad (22)$$

The actual accuracy is hard to determine unless training and an implementation are pursued. However, the least value of the accuracy can be acquired before retraining. For example, the requirement of accuracy is given by $A_r$. A network (all layers are sparse), whose accuracy is $A$ and $A \geq A_r$, can be selected as a baseline sparse network. Assume the actual accuracy is $A^*$, and $A^*$ is always larger than $A$ because some layers are $NP$-layers. Therefore, it can be ensured that $A^*$ is larger than $A_r$ ($A^* \geq A \geq A_r$).

## C. Hardware/Algorithm Co-Optimization

The utilization of CCPUs and the number of $NP$-layers are two important parameters that determine the performance, but it is difficult to consider them together. Therefore, the proposed approach considers them separately.

A spliter pointer is used to determine whether the layers belong to either the $NP$-layers or the $P$-layers. At the beginning, the pointer must be initialized. As the Conv layers incur in significantly more computation than the FC layers, and the number of weights is large in the FC layers, the spliter initially indicates that $NP$-layers are Conv layers and FC layers are $P$-layers.

Then available hardware is allocated for CCPUs by a $RA$. $RA$ is defined as follows:

$$RA = \frac{Resources_{np}}{Resources_p} \tag{23}$$

In FPGA, the $Resources$ can be evaluated by the number of DSPs and LUTs, and the number of DSPs should be converted into the equivalent number of LUTs.

A higher $RA$ means that more $NP$-layers can be processed with the restrictions of $T_{np} > T_p$, which is faster due to higher efficiency in $NP$-layers. A smaller $RA$ means more $P$-layers, which have a smaller model size.

After initialization, the spliter pointer moves to the front layers with a fixed $RA$, so reducing $T_{np}$. Moreover, $T_p$ is calculated by using the prior value of $R_{lp}$ which is evaluated by the pruning rate of a full sparse model. When the pointer is fixed, the model must be retrained. Due to the $NP$-layers, the final $R_{lp}$ is larger than the previous value. Then, $RA$ is finely tuned for the final result.

During the pointer adjustment process, the activation memory increases, but the weight memory decreases. If the total memory increases, the pointer returns to the back layers, finding the least size for the memory. Therefore, the farthest distance the spliter pointer can be adjusted, is rather limited, so $RA$ cannot be too small. Similarly, if $T_{np}-T_p < Margin$ at the beginning, the pointer is not adjusted but more hardware is allocated to SAC-PE for satisfying this condition. This process is described in Algorithm 2.

A Larger $N_{np}$ may improve performance. However, the efficiency may decline due to the ceils in Eq. (9), which means a waste in computation. The best scenario is that there are no remainders of $\frac{M}{N_{np}}$ and $\frac{N}{N_{np}}$ in every $NP$-layers. Computation is different between layers due to the different values for $W$ and $H$. If the efficiency needs to be taken into account to determine $N_{np}$, it can be evaluated by:

$$E_{Conv} = \frac{\sum^{NP} [(W+1) \times H + C] \times \frac{M}{N_{np}} \times \frac{N}{N_{np}}}{\sum^{NP} [(W+1) \times H + C] \times \left\lceil \frac{M}{N_{np}} \right\rceil \times \left\lceil \frac{N}{N_{np}} \right\rceil} \tag{24}$$

**Algorithm 2** Hardware/Algorithm Co-Optimization Algorithm

```
Input: spliter pointer, $RA$, prior $R_{lp}$
Output: $N_{np}, M_p, N_p$ and new $R_{lp}$
1: Allocate available hardware to CCPUs and SAC-PU by $RA$.
2: for all $M_p, N_p$ and $N_{np}$ do
3:   Find the largest value under the restriction of each available hardware.
4: end for
5: for all layers do
6:   Compute the required memory and find the minimal value (the furthest spliter pointer can move).
7: end for
8: Compute $T_{NP}, T_P$.
9: if $T_{NP} - T_P \geq Margin$ then
10:   while $T_{NP} - T_P \geq Margin$ and the pointer is under the valid range do
11:     Move the spliter pointer to the front layer.
12:     Compute $T_{NP}, T_P$.
13:   end while
14:   Retrain the model through the spliter, acquiring new $R_{lp}$.
15:   Compute $T_{NP}, T_P$.
16:   if $T_{NP} - T_P < Margin$ then
17:     repeat
18:       Reduce $N_{np}$ and enlarge $M_p, N_p$.
19:     until Minimal $T_{NP} - T_P$
20:   else
21:     repeat
22:       Enlarge $N_{np}$ and reduce $M_p, N_p$.
23:     until Minimal $T_{NP} - T_P$
24:   end if
25: else
26:   while $T_{NP} - T_P < Margin$ do
27:     Reduce $N_{np}$ and enlarge $M_p, N_p$.
28:     Compute $T_{NP}, T_P$.
29:   end while
30: end if
31: return $N_{np}, M_p, N_p$ and new $R_{lp}$
```

## VII. Experiment and Results

### A. Experiment Set

The proposed processing architecture is scalable with different numbers of $N_{np}$ and $M_{np}$, so affecting performance and the utilization of the DSP units. Memory depends on the size of the network model and the input image; so, the number of on-chip memory resources of an FPGA determines whether the network can be implemented on the FPGA without off-chip DRAM.

HACO is implemented on VGG-16 with the Xilinx VCU118 platform. Performance varies as function of $RA$.The largest

TABLE II
COMPARISON WITH OTHER VGG-16 DESIGNS WITH THE SAME LEVEL ACCURACY (66.06% TOP-1)

| **Approaches** | **FPGA'16 [29]** | **TCAS-I'17 [6]** | **TCAD'18 [7]** | **Max $RA$** | **Min $RA$** |
|---|---|---|---|---|---|
| **Platform** | Zynq ZC706 | Virtex VC707 | Virtex VC709 | Virtex VCU118 | |
| **LUTs** | 182616 | 215556 | 337152 | 695320 | 249432 |
| **FFs** | 127653 | 66792 | 606307 | 243802 | 127105 |
| **DSPs** | 780 | 2296 | 2877 | 4096 | 1024 |
| **BRAMs** | 486 | - | 882.5 | 1779 | 2045 |
| **URAMs** | - | - | - | 779 | 661 |
| **DRAM used** | Yes | Yes | Yes | **No** | |
| **FPS** | 8.9 | 33.8 | 45.5 | **83.0** | **30.3** |

value of $RA$ has the best performance and the largest model size (i.e., 21.1MB as mentioned in Section III-B). The least value of $RA$ has a slower speed but also the smallest model size. Initially, the $NP$-layers include Conv layers only, and $P$-layers are FC layers. With the decrease of $RA$, the spliter pointer moves to the front layers. When the pointer moves to layer Conv 3-3, the needed memory increases due to the large feature map. Therefore, the layer Conv 3-3 is the farthest distance that the pointer can move in VGG-16.

For the largest model, VCU118 requires significant on-chip memory. URAMs and BRAMs are utilized for VCU118. Due to the large model size, the memory for the weights is given by URAMs. To fully use the RAMs resources, 8 9-bit weights are utilized. The remaining memories are allocated for activations. 3 parallel activations are utilized for a bit-width of 48-bit. Therefore, every 3 FMRs with a bit-width of 144-bit are synthesized as two groups of URAMs or BRAMs. This allocation is adjusted by balancing URAMs and BRAMs. Computation in VCU118 is performed by mostly DSPs and LUTs. For high performance in multiplication, DSPs are allocated for $NP$-layers, and LUTs are allocated for $P$-layers for shift operation. The available hardware are given by 60% of the VCU118. As DSPs cannot be used in $P$-layers, a decrease of $RA$ is possible. For the highest efficiency in convolution, $N_{np}$, $M_p$ and $N_p$ should be given by $2^n$, where $n$ is a positive integer. $N_p$ is limited to a range of 32∼512 due to the ADF efficiency and the largest number of output channels. In the $P$-layers, activations and weights are required to be cached before computing but accounting for large memory. When the available BRAMs are insufficient, distributed RAMs can be used as alternatives.

To improve the speed of weight decoding and activation reading, data must be integrated in RAMs for parallel computing, as the improvement in frequency is not effective. If activations are needed to be integrated, weights should be decoded in the same form. However, the weights in a sparse model are not serial; this causes the number of useful weights to be difficult to calculate. Therefore, parallel decoding cannot be achieved when multiple activations are integrated. Since the feature map shrinks during pooling, the integration of activations results in less efficiency in the $P$-layers. Therefore, only the parallel weight decoding is used in our experiment.

## B. Performance Analysis and Comparison

In HACO, the largest and least values of $RA$ are utilized as shown in Table IV. For the largest value of $RA$, $T_{np}$ is larger than $T_p$ initially. Therefore, the spliter pointer does not move, so staying at the end of the Conv layers. In this case, $N_{np} = 33$ and $N_p = 51$ are obtained by HACO. For the least value of $RA$, $N_p = 256$ and $N_{np} = 16$ are obtained, and finally the pointer moves to the layer Conv 4-3. To achieve the highest efficiency, $N_{np}$ and $N_p$ are mapped to $2^n$ in our experiments. For the $P$-layers, the caches are synthesized as BRAMs and distributed RAMs. The results are shown in Table IV.

The FMRs and CCPUs use the same frequency (200 MHz for $N_{np} = 16$; 150 MHz for $N_{np} = 32$). Weights are loaded in the buffer during convolution. As weight loading is faster than convolution computation and requires a large area and longer delay in the design, the weight RAM uses a slower frequency (100 MHz in this work). All frequencies are shown in Table V.

Table II shows the hardware resources and performance of previous VGG-16 FPGA designs and the proposed design with the largest and least values of $RA$. The proposed design achieves the highest FPS of 83.0 at 150MHz; this is 1.8× faster than [7]. 4,096 DSPs are used due to a large number of CCPUs. All other three state-of-the-art designs use off-chip DDR3 DRAM so limiting the processing speed due to bandwidth; the proposed design uses only on-chip BRAMs and URAMs for storing the feature map and weights.

$RA$ is reduced at a smaller model size. As shown is Table II, the design with the least $RA$ has a smaller memory (URAM is 8 times larger than a BRAM, so the least $RA$ saves 728 BRAMs). Using HACO, redundancy in $NP$-layers is removed to attain a higher efficiency; therefore, the number of DSPs is reduced to 1,024. Compared with [6], the least $RA$ achieves 90% performance with only a 46% usage of DSPs. Compared with [7], the least $RA$ achieves 67% performance with a 36% usage of DSPs and, a 20% usage of FFs.

TABLE III
COMPARISON BETWEEN PROPOSED GENERAL CCPUS AND OTHER DESIGNS

| **Approaches** | [30] | [31] | [31] | [31] | Ours | | |
|---|---|---|---|---|---|---|---|
| **Platform** | Stratix V GSMD5 | Arria 10 GX 1150 | Arria 10 GX 1150 | Arria 10 GX 1150 | Virtex VCU118 | | |
| **Network (# of Operations (GOP))** | ResNet-152 (22.62) | VGG-16 (30.70) | ResNet-50 (7.74) | ResNet-152 (22.62) | VGG-16 (30.70) | ResNet-50 (7.74) | ResNet-152 (22.62) |
| **Frequency (MHz)** | 150 | 200 | 200 | 200 | 150 | | |
| **Logic Elements**[a] | 45.7K | 138K | 221K | 235K | 781K | | |
| **DSPs**[b] **(# of MAC Units)** | 1044 (2088) | 1518 (3036) | 1518 (3036) | 1518 (3036) | 4096 (4096) | | |
| **Latency (ms)** | - | 29.8 | 11.5 | 29.7 | **12.0** | **9.1** | **20.3** |
| **Throughput (GOPS)**[c] | 226.5 | 1030.2 | 672.0 | 761.6 | **2558.3** | **850.5** | **1114.3** |

[a]Xilinx FPGA in LUTs and Intel FPGA in ALMs.
[b]One DSP block in Intel FPGA can be configured as two independent 18-bit×19-bit multipliers.
[c]Throughput(GOPS) = Operations(GOP) / Latency(s)

TABLE IV
RESULTS ON VGG-16 BY HACO WITH LARGEST AND LEAST $RA$

| | Max $RA$ | | Min $RA$ | |
|---|---|---|---|---|
| | $NP$-layer | $P$-layer | $NP$-layer | $P$-layer |
| **LUTs** | 668079 | 27241 (86978) | 144032 | 105400 (342286) |
| **FFs** | 225570 | 18232 (68176) | 55637 | 71468 (271143) |
| **DSPs** | 4096 | - | 1024 | - |
| **BRAMs** | 1587 | 192 (37) | 1280 | 765 (125) |
| **URAMs** | 604 | 175 | 414 | 247 |
| **Model Size** | 20.1MB/ 27.5× | | 16.0MB/ 37.5× | |
| **FPS** | 83.0 | | 30.3 | |

The results, of which caches are synthesized as distributed RAM, are shown in parentheses.

TABLE V
FREQUENCIES OF $NP$-LAYERS AND $P$-LAYERS

| | $NP$ Frequency (MHz) | | $P$ Frequency (MHz) | | |
|---|---|---|---|---|---|
| | $f_W$ | $f_{Conv}$ | $f_W$ | $f_A$ | $f_{SAC}$ |
| **Max** $RA$ | 100 | 150 | 170 | 300 | 300 |
| **Min** $RA$ | 100 | 200 | 150 | 250 | 250 |

Although there are some designs of embedded BNNs which could achieve even higher FPS, the accuracy loss is higher than the designs in Table II. For example, [24] can only achieve 55.8% top-1 accuracy for VGG-16. Therefore, these designs are not considered; only the designs with the same level of accuracy are compared in Table II.

CCPUs are also applied to ResNet. In the first layer of ResNet, the size of the Conv kernels is 7 × 7, therefore, the CCPUs will operate in a serial mode. Seven copies of the input image are loaded into the FMRs, and each image is processed by 1-D convolution. Then, the output channels is added to obtain an output feature map. For ResNet, an extra RAM is utilized to store the feature map of the shortcut connection. The FMRs must store up to 224 × 224 × 32 data for input feature maps; map buffer and shortcut buffer must store up to 112 × 112 × 64 data, separately. The required total memory is 67% of that used by VGG-16. As per the time evaluation mentioned in Section V, computation in all the Conv layers only requires 4.0 ms for ResNet-34, 9.1 ms for ResNet-50 and 20.3ms for ResNet-152 when $N_{np} = 32$ (largest $RA$). The proposed general CCPUs are compared to other state-of-the-art FPGA designs in Table III.

## VIII. CONCLUSION

A hardware-oriented compression strategy has been initially proposed in this paper. This strategy achieves high performance and 27.5× compression ratio for VGG-16. It has been shown that the proposed strategy incurs in a very small accuracy loss compared to the single-precision floating-point implementation as tested on the ILSVRC2012 data set through the Caffe framework.

As a case study, the architecture of the proposed compressed VGG-16 has been designed with no off-chip memory on the Xilinx FPGA VCU118 platform. It has been shown that the design achieved using the proposed tool HACO achieves the highest performance of 83.0 FPS for the same level of accuracy in image processing. The proposed general Conv-PE structure has a high efficiency in processing large Conv kernels; therefore, the entire architecture can process a wide range of CNN models with small additional hardware. The proposed hardware design can be applied to several real-time resource constrained image processing applications.

The proposed compression method is applicable to other CNN models; new quantization and pruning methods can be also used in the proposed HACO framework to obtain designs with even higher performance levels.

## References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*. [Online]. Available: http://arxiv.org/abs/1409.1556

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[3] R. Girshick, "Fast R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1440–1448.

[4] S. Chetlur *et al.*, "CuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*. [Online]. Available: http://arxiv.org/abs/1410.0759

[5] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[6] J. Wang, J. Lin, and Z. Wang, "Efficient hardware architectures for deep convolutional neural network," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 6, pp. 1941–1953, Jun. 2018.

[7] S. Yin *et al.*, "A high throughput acceleration for hybrid neural networks with efficient resource management on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 4, pp. 678–691, Apr. 2019.

[8] D. Han, J. Lee, J. Lee, and H.-J. Yoo, "A low-power deep neural network online learning processor for real-time object tracking application," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 5, pp. 1794–1804, May 2019.

[9] Y. Wang, J. Lin, and Z. Wang, "FPAP: A folded architecture for energy-quality scalable convolutional neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 1, pp. 288–301, Jan. 2019.

[10] Y.-J. Lin and T. S. Chang, "Data and hardware efficient design for convolutional neural network," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 5, pp. 1642–1651, May 2018.

[11] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Proc. Adv. Neural Inf. Process. Syst.*, 1993, pp. 164–171.

[12] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*. [Online]. Available: http://arxiv.org/abs/1510.00149

[13] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient DNNs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 1379–1387.

[14] S. Ye *et al.*, "Progressive DNN compression: A key to achieve ultra-high weight pruning and quantization rates using ADMM," 2019, *arXiv:1903.09769*. [Online]. Available: http://arxiv.org/abs/1903.09769

[15] L. Lu and Y. Liang, "SpWA: An efficient sparse winograd convolutional neural networks accelerator on FPGAs," in *Proc. 55th ACM/ESDA/IEEE Des. Autom. Conf. (DAC)*, Jun. 2018, pp. 1–6.

[16] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 269–284, Feb. 2014, doi: 10.1145/2654822.2541967.

[17] J. H. Ko, D. Kim, T. Na, and S. Mukhopadhyay, "Design and analysis of a neural network inference engine based on adaptive weight compression," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 1, pp. 109–121, Jan. 2019.

[18] S. Li, J. Park, and P. Tak Peter Tang, "Enabling sparse winograd convolution by native pruning," 2017, *arXiv:1702.08597*. [Online]. Available: http://arxiv.org/abs/1702.08597

[19] W. Liu, F. Lombardi, and M. Shulte, "A retrospective and prospective view of approximate computing [Point of View]," *Proc. IEEE*, vol. 108, no. 3, pp. 394–399, Mar. 2020.

[20] W. Liu, L. Qian, C. Wang, H. Jiang, J. Han, and F. Lombardi, "Design of approximate Radix-4 booth multipliers for error-tolerant computing," *IEEE Trans. Comput.*, vol. 66, no. 8, pp. 1435–1441, Aug. 2017.

[21] Z. Liu, K. Jia, W. Liu, Q. Wei, F. Qiao, and H. Yang, "INA: Incremental network approximation algorithm for limited precision deep neural networks," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, Nov. 2019, pp. 1–7.

[22] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or –1," 2016, *arXiv:1602.02830*. [Online]. Available: http://arxiv.org/abs/1602.02830

[23] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: Imagenet classification using binary convolutional neural networks," in *Eur. Conf. Comput. Vis. (ECCV)*, 2016, pp. 525–542.

[24] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, 2018, pp. 163–1636.

[25] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless CNNs with low-precision weights," 2017, *arXiv:1702.03044*. [Online]. Available: http://arxiv.org/abs/1702.03044

[26] M. A. Hanif, R. Hafiz, and M. Shafique, "Error resilience analysis for systematically employing approximate computing in convolutional neural networks," in *Proc. Des., Autom. Test Eur. Conf. Exhib. (DATE)*, 2018, pp. 913–916.

[27] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.

[28] J. Deng, W. Dong, R. Socher, L. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.

[29] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2016, pp. 26–35.

[30] Y. Guan *et al.*, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Proc. IEEE 25th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 152–159.

[31] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 7, pp. 1354–1367, Jul. 2018.

**Tian Yuan** received the B.S. degree in information engineering from the Nanjing University of Aeronautics and Astronautics (NUAA), Nanjing, China, in 2018, where he is currently pursuing the M.S. degree in circuits and systems. His research interests include hardware architectures design for deep learning, neural network compression and quantization, and approximate computing.

**Weiqiang Liu** (Senior Member, IEEE) received the B.S. degree in information engineering from the Nanjing University of Aeronautics and Astronautics (NUAA), Nanjing, China, and the Ph.D. degree in electronic engineering from Queen's University Belfast (QUB), Belfast, U.K., in 2006 and 2012, respectively. In December 2013, he joined the College of Electronics and Information Engineering, NUAA, where he is currently a Professor and the Vice Dean of the College of Electronics and Information Engineering. His research interests include emerging technologies in computing systems, computer arithmetic, hardware security, and VLSI design for digital signal processing and cryptography. He has published one research book by Artech House and over 100 leading journal and conference papers. He is a Senior Member of the Chinese Institute of Electronics. One of his papers was selected as the Feature Paper of IEEE TC in the 2017 December issue. He received the prestigious Outstanding Young Scholar Award by the National Natural Science Foundation China (NSFC) in 2020. He is the Program Co-Chair of the IEEE Symposium on Computer Arithmetic (ARITH), and a program member for a number of international conferences. He is a member of both the Circuits & Systems for Communications (CASCOM) Technical Committee and the VLSI Systems and Applications (VSA) Technical Committee, IEEE Circuits and Systems Society. He has served as a Guest Editor for the Proceedings of the IEEE and an Associate Editor for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS I: REGULAR PAPERS, the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON EMERGING TOPIC IN COMPUTING AND COMPUTERS, and the IEEE OPEN JOURNAL OF COMPUTER SOCIETY, and a Steering Committee Member of the IEEE TRANSACTIONS ON MULTI-SCALE COMPUTING SYSTEMS.

**Jie Han** (Senior Member, IEEE) received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 1999, and the Ph.D. degree from the Delft University of Technology, The Netherlands, in 2004. He is currently a Professor with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada. His research interests include approximate computing, stochastic computing, reliability and fault tolerance, nanoelectronic circuits and systems, as well as novel computational models for nanoscale and biological applications. He was a recipient of the Best Paper Award at the International Symposium on Nanoscale Architectures (NanoArch) 2015 and Best Paper Nominations at the 25th Great Lakes Symposium on VLSI (GLSVLSI) 2015, NanoArch 2016, and the 19th International Symposium on Quality Electronic Design (ISQED) 2018. He was nominated for the 2006 Christiaan Huygens Prize of Science by the Royal Dutch Academy of Science. His work was recognized by Science, for developing a theory of fault-tolerant nanocircuits (2005). He has served as the General Chair for GLSVLSI 2017 and the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT) 2013, and a Technical Program Committee Chair for GLSVLSI 2016, DFT 2012, and the Symposium on Stochastic & Approximate Computing for Signal Processing and Machine Learning, 2017. He is currently an Associate Editor of the IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING (TETC), the IEEE TRANSACTIONS ON NANOTECHNOLOGY, the *IEEE Circuits and Systems Magazine*, and the IEEE OPEN JOURNAL OF THE COMPUTER SOCIETY and Microelectronics Reliability (Elsevier journal).

**Fabrizio Lombardi** (Fellow, IEEE) received the B.S. degree (Hons.) in electronic engineering from the University of Essex, U.K., in 1977, the master's degree in microwaves and modern optics, the Diploma degree in microwave engineering from the Microwave Research Unit, University College London, in 1978, and the Ph.D. degree from the University of London in 1982. He is currently the International Test Conference (ITC) Endowed Chair Professorship with Northeastern University, Boston, MA, USA. His research interests are bio-inspired and nano manufacturing/computing, VLSI design, testing, and fault/defect tolerance of digital systems. He has extensively published in these areas and coauthored/edited seven books. He is currently the Vice President for Publications of both the IEEE Computer Society and a member of the executive committee of the IEEE Nanotechnology Council. He was the Editor-in-Chief of the IEEE TRANSACTION ON COMPUTERS from 2007 to 2010, the IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING from 2013 to 2017, and the IEEE TRANSACTIONS ON NANOTECHNOLOGY from 2014 to 2019.