Energy-Efficient, High-Performance, Highly-Compressed Deep Neural Network Design using Block-Circulant Matrices

Siyu Liao*[1], Zhe Li*[2], Xue Lin[3], Qinru Qiu[2], Yanzhi Wang[2], Bo Yuan[1]
[1]City University of New York, New York, NY, USA {sliao2@gradcenter, byuan@ccny}.cuny.edu
[2]Syracuse University, Syracuse, NY, USA {zli89, qiqiu, ywang393}@syr.edu
[3]Northeastern University, Boston, MA, USA xue.lin@northeastern.edu

***Abstract*—Deep neural networks (DNNs) have emerged as the most powerful machine learning technique in numerous artificial intelligent applications. However, the large sizes of DNNs make themselves both computation and memory intensive, thereby limiting the hardware performance of dedicated DNN accelerators. In this paper, we propose a holistic framework for energy-efficient high-performance highly-compressed DNN hardware design. First, we propose block-circulant matrix-based DNN training and inference schemes, which theoretically guarantee Big-O complexity reduction in both computational cost (from $O(n^2)$ to $O(n \log n)$) and storage requirement (from $O(n^2)$ to $O(n)$) of DNNs. Second, we dedicatedly optimize the hardware architecture, especially on the key fast Fourier transform (FFT) module, to improve the overall performance in terms of energy efficiency, computation performance and resource cost. Third, we propose a design flow to perform hardware-software co-optimization with the purpose of achieving good balance between test accuracy and hardware performance of DNNs. Based on the proposed design flow, two block-circulant matrix-based DNNs on two different datasets are implemented and evaluated on FPGA. The fixed-point quantization and the proposed block-circulant matrix-based inference scheme enables the network to achieve as high as 3.5 TOPS computation performance and 3.69 TOPS/W energy efficiency while the memory is saved by $108\mathbf{X} \sim 116\mathbf{X}$ with negligible accuracy degradation.**

I. Introduction

Proposed in 1940's, neural networks (NNs) [1] are the most representative connectionism model in the artificial intelligence field. With their inherent parallel processing architectures and provable capability of well-approximation to arbitrary functions [2], the multi-layer NNs had gained tremendous attractions in 1980's. However, due to their limited progress on performance (especially accuracy and speed) and the competition from other simple but mathematically solid machine learning techniques (e.g. support vector machine and random decision forest), the research on NNs gradually diminished in 1990's and had lost its original prosperity for more than one decade.

Driven by the unprecedented advance in semiconductor technology and the explosive development of available data, NNs are now experiencing their exciting resurgence in the big data era: The superior processing power of modern computers and the massive amount of data jointly enable the *deep neural networks (DNNs)*, as the large-scale NNs, to achieve much stronger learning capability than the early-year small-size NNs within the affordable training time. As a result, to date the DNNs have produced several state-of-the-art accuracy results on various tasks [3], [4], thereby making them become the most popular and powerful machine learning technique in numerous artificial intelligence applications.

As mentioned before, one of the key enablers for the success of DNNs is their large-scale structures. Typically, a DNN consists of at least several cascade-connected layers and each layer contains hundreds or thousands of neurons, and hence resulting in very large number of parameters (or so-called weights) for the entire DNN. Although such large model sizes enable strong representing and learning capability, they also make DNNs both computation and memory intensive. As a result, it is very challenging to design DNN hardware with high energy efficiency, high computation performance and low memory footprint. Such emerging challenge, if not being properly addressed, would greatly impede the widespread deployment of DNNs in many resource-constrained applications, such as embedded or mobile systems.

Targeting at this problem, several work have been proposed to improve the hardware performance of DNNs via model compression. In [5], [6], low-precision weight-based DNN designs were proposed to reduce the weight memory size and critical path delay. In [7], [8], the strategies of pruning DNNs were used to develop the corresponding hardware since a large amount of energy and area can be saved after removing unnecessary weights or neurons. In [9], a reduced-memory DNN design was proposed by performing low-rank approximation for the weight matrices. However, aforementioned methods need additional processing which requires additional training effort. Under certain situations, these methods may lead to the irregularity of the model.

In this paper, we propose a holistic framework for energy-efficient, high-performance, memory-compressed DNN hardware design. Different from prior compressed DNN hardware that were based on the unstructured weight matrices, the model reduction in this paper results from the use of *structured* weight matrices: In the underlying network structure, the weight matrices, which were original unstructured, are now constructed in the *block-circulant* format, thereby leading to order-of-magnitude reduction in computational cost and storage requirement, which further translates to very significant improvement in energy efficiency, computation performance and memory compression ratio for the entire DNN accelerators. In short, the contributions of this paper are summarized as below:

1) We propose block-circulant matrix-based DNN training and inference schemes, which theoretically guarantee Big-O complexity reduction in both computational cost (from $O(n^2)$ to $O(n \log n)$) and storage requirement (from $O(n^2)$ to $O(n)$) of DNNs. In addition, the proposed schemes has great flexibility on the size of weight matrices and the controllable balance between test accuracy and compression ratio.

2) We dedicatedly optimize the hardware architecture, especially on the key fast Fourier transform (FFT) module, to improve the overall performance in terms of energy efficiency, computation performance and resource cost. The quantization

*Siyu Liao and Zhe Li contributed equally to this work

978-1-5386-3093-8/17/$31.00 ©2017 IEEE

scheme is also carefully investigated to achieve good balance between precision loss and hardware performance.

3) We propose a design flow to perform hardware-software co-optimization with the purpose of achieving good balance between test accuracy and hardware performance of DNNs. The proposed design flow can always ensure the finding of the maximally compressed DNN model that satisfies the predefined accuracy requirement, thereby achieving the optimal hardware performance (in terms of energy efficiency, memory size etc.) under the given constraint on accuracy.

4) We demonstrate the computation capability of the proposed block-circulant matrix-based DNN by implementing two design examples on two different datasets on FPGA. Compared with peer work, our FPGA results show the proposed block-circulant matrix-based DNN hardware can achieve very high energy efficiency and computation performance.

The rest of this paper is organized as below. Section II gives a brief review of the prior work on model reduction and efficient DNN hardware designs. The preliminary and background information on DNN training and inference is introduced in Section III. Section IV presents the proposed training and inference schemes for block-circulant matrix-based DNNs. In Section V, the hardware architecture of the DNNs, including the key FFT module and quantization scheme, is discussed and optimized. Section VI presents the design flow for software/hardware co-optimization. Experimental results are presented and discussed in Section VII. Finally, Section VIII draws the conclusions.

II. Related Work

The efforts of compressing neural networks can be traced back to 1980's. In [10], the "Optimal Brain Damage" was proposed to use second derivative information to delete part of weights in the training phase. Consider a large portion of weights in the trained model are close to zero, network pruning is an efficient approach to reduce the memory size of DNN hardware [7], [8]. However, because of the inherently irregular pruning pattern, the additional indexing operation results in extra memory cost that offsets the saving from pruning. In addition, the required re-training step after pruning also increases the complexity of the overall training phase.

Besides pruning, lowering the representation precision of weights is also very efficient for reducing both the model size and computational cost of DNNs. In [5], fixed-point implementations were developed to reduce the memory cost of DNN hardware. Further, in [11], the hardware architecture of binary neural networks (BNNs), which only uses binary (-1/+1) weight representation, were proposed to achieve extremely low area and energy cost. However, the BNN usually suffers from severe accuracy loss because of the ultra-low representation precision scheme.

Low rank approximation (LRA) is another useful DNN compression technique. By approximately representing the weight matrix as the product of two small-size matrices, the overall memory footprint and processing latency of DNN hardware may be reduced if the two component matrices could be properly found [9]. However, in order to control the inherent approximation error incurred by the LRA, the improvement on the hardware performance of the LRA-based DNNs is usually not significant.

III. Preliminaries

A. DNN Inference

In general, the DNN inference is performed in the forward propagating format: The test data is input to the first layer of DNN and the final classification results are output from the last layer. During the entire forward propagation, the key computation is the matrix-vector multiplication. For instance, the computation in the fully-connected (FC) layers in the inference phase is described as below:

$$\mathbf{y} = f(\mathbf{a}) = f(\mathbf{W}\mathbf{x}), \tag{1}$$

where $f(\cdot)$ is the activation function, $\mathbf{W} \in \mathbb{R}^{m \times n}$ is the weight matrix, $\mathbf{x} \in \mathbb{R}^n$ is the input vector, and $\mathbf{a}$ is the product of $\mathbf{W}$ and $\mathbf{x}$, respectively.

B. DNN Training

Different from inference, the DNN training is performed by using the backward propagation: The gradient descents of the weights to be updated are calculated from the last layer to the first layer. More specifically, the update of the weight matrix $\mathbf{W}$ can be performed as below:

$$\mathbf{W} = \mathbf{W} - \epsilon \frac{\partial L}{\partial \mathbf{W}}, \tag{2}$$

where L is the loss function measuring how close the prediction is compared to the ground-truth, and ϵ is the learning rate. Notice that due to the different choices of $f(\cdot)$, the explicit mathematic format of $\frac{\partial L}{\partial \mathbf{W}}$ will be different; but the key computation in equation (2) is still the matrix-vector multiplication [12], [13].

C. Complexity of DNN Inference/Training

Equation (1) and (2) describe the general computation procedure for DNN inference and training, respectively. Since typically only the weight matrix $\mathbf{W}$ is stored in the feedforward or backward propagation, the space complexity of DNN inference and training is $O(n^2)$ (m and n typically have the same order of magnitude) for the FC layer. Consider that the values of n (and m) are typically hundreds to thousands; the number of parameters in the weight matrix $\mathbf{W}$ is typically very large, thereby resulting in huge memory consumption in DNN hardware. In addition, from the perspective of computation, DNNs are also very computationally intensive. Consider that matrix-vector multiplication is the key computation during the forward and backward propagation processes, the computational complexity of DNN training and inference is $O(n^2)$ for the layer. Therefore, the large value of n incurred by the large model size of DNN also results in very high demands on computation.

IV. Reduced-Complexity DNN Inference and Training using Block-Circulant Matrices

As analyzed in Section III-C, the theoretical computational complexity and space complexity for DNN inference and training are $O(n^2)$. Although the existing techniques, such as network pruning, lowering representation precision, etc., can help to improve the hardware performance of DNN accelerators, those approaches do not fundamentally lower the theoretical bound of space or computational complexity, thereby limiting the potential in hardware performance improvement.

Different from prior efforts, in this section we propose to leverage the unique mathematic property of *block-circulant*

matrices [14] to reduce the theoretical computational and space complexities of DNN training and inference. The key idea is to impose the block-circulant structure to DNNs: The weight matrix $\mathbf{W}$ is always in the format of block-circulant matrices. In this scenario, because i) block-circulant matrices only require $O(n)$ parameters, and ii) ultra-fast matrix-vector multiplication algorithm with $O(n \log n)$ computational complexity exists for this family of special matrices, such imposing of the block-circulant structure immediately enables order-of-magnitude reduction in both storage requirement and computational cost.

Consider the inference and training schemes described in equation (1)(2) are for the general unstructured weight matrices, in this section we present low-space-cost, ultra-fast, block-circulant matrix-based DNNs-oriented inference and training schemes. First, the forward and backward propagation processes that target to square circulant weight matrices [15] are introduced. Then, the general block-circulant matrix-based inference and training schemes for DNNs that contain arbitrary-size weight matrices are developed.

A. Circulant Matrix-based DNN Inference and Training

In general, a circulant matrix $\mathbf{W} \in \mathbb{R}^{n \times n}$ [14] is defined by a vector $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ as the following:

$$\mathbf{W} = \begin{bmatrix} w_1 & w_n & \ldots & w_3 & w_2 \\ w_2 & w_1 & w_n & & w_3 \\ \vdots & w_2 & w_1 & \ddots & \vdots \\ w_{n-1} & & \ddots & \ddots & w_n \\ w_n & w_{n-1} & \ldots & w_2 & w_1 \end{bmatrix}. \tag{3}$$

From equation (3) it is seen that an n-by-n circulant matrix only has n parameters because of its strong structure. Clearly, when such structure is imposed to the weight matrices of DNNs, the required space cost for storing the weights is immediately reduced from $O(n^2)$ to $O(n)$.

Besides the advantage on low space cost, the use of circulant matrices as weight matrices can also lead to low computational complexity for both inference and training, which are described as below:

Inference: As indicated in equation (1), the dominating computation during the forward propagation in the inference is the matrix-vector multiplication ($\mathbf{Wx}$). According to [14], when $\mathbf{W}$ is a circulant matrix, $\mathbf{Wx}$ can be performed as below:

$$\mathbf{a} = \mathbf{Wx} = \text{ifft}(\text{fft}(\mathbf{w}) \circ \text{fft}(\mathbf{x})), \tag{4}$$

where $\circ$ denotes the element-wise multiplication; fft$(\cdot)$ denotes the fast Fourier transform; and ifft$(\cdot)$ denotes the inverse fast Fourier transform. Notice that since the computational complexity of n-point FFT/IFFT is only $O(n \log n)$, the computational complexity of DNN inference can achieve order-of-magnitude reduction (from $O(n^2)$ to $O(n \log n)$).

Training: For backward propagation in the training, recall that its key procedure is to perform the chain rule-based calculation for the gradient of loss function L with respect to the weight vector $\mathbf{w}$ as below:

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{w}}, \tag{5}$$

where $\frac{\partial L}{\partial \mathbf{a}}$ is the gradient back-propagated from the subsequent layer. Notice that in the scenario that $\mathbf{W}$ is a square circulant matrix, as indicated in [15], $\frac{\partial \mathbf{a}}{\partial \mathbf{w}}$ is a circulant matrix defined by the vector $\mathbf{x}' = (x_1, x_n, x_{n-1}, \ldots, x_2)$. Therefore, according to [14], equation (5) can be simplified as below:

$$\frac{\partial L}{\partial \mathbf{w}} = \text{ifft}(\text{fft}(\frac{\partial L}{\partial \mathbf{a}}) \circ \text{fft}(\mathbf{x}')), \tag{6}$$

where $\mathbf{1}$ is a column vector full of ones. In addition, the gradient of input $\mathbf{x}$ which is back-propagated to the previous layer, should be calculated as:

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{x}}. \tag{7}$$

Notice that here $\frac{\partial \mathbf{a}}{\partial \mathbf{x}}$ is also a circulant matrix that is defined as $\mathbf{w}' = (w_1, w_n, w_{n-1}, \ldots, w_2)$. Hence equation (7) can also be simplified as below:

$$\frac{\partial L}{\partial \mathbf{x}} = \text{ifft}(\text{fft}(\frac{\partial L}{\partial \mathbf{a}}) \circ \text{fft}(\mathbf{w}')). \tag{8}$$

From equation (6) and (8) it is seen that, when $\mathbf{W}$ is a circulant matrix, the updating scheme for the gradients of $\mathbf{w}$ and $\mathbf{x}$, as the key part of DNN training, can also be calculated using FFT/IFFT, thereby rendering order-of-magnitude reduction in computational cost for training (from $O(n^2)$ to $O(n \log n)$).

B. Block-Circulant Matrix-based DNN Inference and Training

Section IV-A presents the forward and backward propagation for circulant matrix-based DNNs. However, in many practical applications such schemes cannot be directly used because: 1) It is very common that the weight matrices of DNNs are non-square matrices due to the need of specific applications; and 2) Even if the weight matrices are square, as indicated in Section VI, in many cases the compression effect led by the approach in Section IV-A is too aggressive and hence causes non-negligible accuracy loss.

To address the above challenges, in this subsection, we present the block-circulant matrix-based forward and backward propagation for inference and training. The key idea here is to partition the original arbitrary-size unstructured weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ into 2D blocks of square sub-matrices and then replace those sub-matrices with different circulant matrices. Such partition strategy has the following two advantages: 1) It is suitable for arbitrary-size weight matrices without any requirement on the shape of $\mathbf{W}$; and 2) It is an adjustable approach that can conveniently control the compression ratio and potential accuracy loss by only changing the partition size of sub-matrices. Such flexibility is very useful for designing different types of compressed DNNs with different resource budgets and/or target accuracy.

Next we present the details of the proposed block-circulant matrix-based inference and training schemes. Let k be the *partition size* and there are $p \times q$ blocks after partitioning $\mathbf{W}$, where $p = m \div k$ and $q = n \div k$. Then $\mathbf{W} = [\mathbf{C}_{ij}]$, $i \in \{1 \ldots p\}$, $j \in \{1 \ldots q\}$. Assume each circulant matrix $\mathbf{C}_{ij}$ is defined by a vector $\mathbf{w}_{ij}$. Correspondingly, the input $\mathbf{x}$ is also partitioned so $\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_q]^T$. As a result, the forward propagation process in the inference phase can be performed as the following:

$$\mathbf{a} = \mathbf{Wx} = \begin{bmatrix} \sum_{j=1}^{q} \mathbf{C}_{1j}\mathbf{x}_j \\ \sum_{j=1}^{q} \mathbf{C}_{2j}\mathbf{x}_j \\ \ldots \\ \sum_{j=1}^{q} \mathbf{C}_{pj}\mathbf{x}_j \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \ldots \\ \mathbf{a}_p \end{bmatrix}, \tag{9}$$

where $\mathbf{a}_i \in \mathbb{R}^k$ is a column vector.

Algorithm 1: Block-circulant matrix-based forward propagation process

Input: $\mathbf{w}$, $\mathbf{x}$, p, q, k
Output: $\mathbf{a}$
Initialize $\mathbf{a}$ with zeros.
for $i \leftarrow 1$ *until* p **do**
  for $j \leftarrow 1$ *until* q **do**
    $\mathbf{a}_i \leftarrow \mathbf{a}_i + \text{ifft}(\text{fft}(\mathbf{w}_{ij}) \circ \text{fft}(\mathbf{x}_j))$
  end
end
return $\mathbf{a}$

Algorithm 2: Block-circulant matrix-based backward propagation process

Input: $\frac{\partial L}{\partial \mathbf{a}_i}$, $\mathbf{w}$, $\mathbf{x}$, p, q, k
Output: $\frac{\partial L}{\partial \mathbf{w}}$, $\frac{\partial L}{\partial \mathbf{x}}$
Initialize $\frac{\partial L}{\partial \mathbf{w}}$ and $\frac{\partial L}{\partial \mathbf{x}}$ with zeros.
for $i \leftarrow 1$ *until* p **do**
  for $j \leftarrow 1$ *until* q **do**
    $\frac{\partial L}{\partial \mathbf{w}_{ij}} \leftarrow \text{ifft}(\text{fft}(\frac{\partial L}{\partial \mathbf{a}_i}) \circ \text{fft}(\mathbf{x'}_j))$
    $\frac{\partial L}{\partial \mathbf{x}_j} \leftarrow \frac{\partial L}{\partial \mathbf{x}_j} + \text{ifft}(\text{fft}(\frac{\partial L}{\partial \mathbf{a}_i}) \circ \text{fft}(\mathbf{w'}_{ij}))$
  end
end
return $\frac{\partial L}{\partial \mathbf{w}}$, $\frac{\partial L}{\partial \mathbf{x}}$

Now consider the backward propagation process in the training phase. Let a_{il} be the l-th output element in $\mathbf{a}_i$. Then by using the chain rule we can derive the backward propagation process as follows:

$$\frac{\partial L}{\partial \mathbf{w}_{ij}} = \sum_{l=1}^{k} \frac{\partial L}{\partial a_{il}} \frac{\partial a_{il}}{\partial \mathbf{w}_{ij}} = \frac{\partial L}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_{ij}}, \tag{10}$$

$$\frac{\partial L}{\partial \mathbf{x}_j} = \sum_{i=1}^{p} \sum_{l=1}^{k} \frac{\partial L}{\partial a_{il}} \frac{\partial a_{il}}{\partial \mathbf{x}_j} = \sum_{i=1}^{p} \frac{\partial L}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{x}_j}. \tag{11}$$

Recall that $\frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_{ij}}$ and $\frac{\partial \mathbf{a}_i}{\partial \mathbf{x}_j}$ are circulant matrices as pointed out before. Therefore, $\frac{\partial L}{\partial \mathbf{w}_i}$ and $\frac{\partial L}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{x}_j}$ can be calculated using the same way as described in (6) and (8).

It should be noted that sometimes k may not divide m or n. A general solution to this case is to pad zeros along the dimension that k doesn't divide. In this paper, $\mathbf{W}$ is padded with zeros to the size such that $p = \lceil \frac{m}{k} \rceil$ and $q = \lceil \frac{n}{k} \rceil$.

In summary, the above described forward and backward propagation processes for block-circulant matrix-based DNN models are shown in Algorithm 1 and Algorithm 2, respectively. Note that by applying these partition-based processes, the space complexity and computational complexity are reduced to $O(pqk)$ and $O(pqk \log k)$.

V. Hardware Architecture Design

In this section, the hardware architecture of the block-circulant matrix-based DNNs is presented. Because in practical applications the dedicated DNN accelerators are usually in charge of on-line inference tasks to achieve low-latency response and high throughput, while the CPU/GPU clusters are in charge of off-line training tasks for flexible deployment [16], in this section we focus on the hardware design for the inference process.

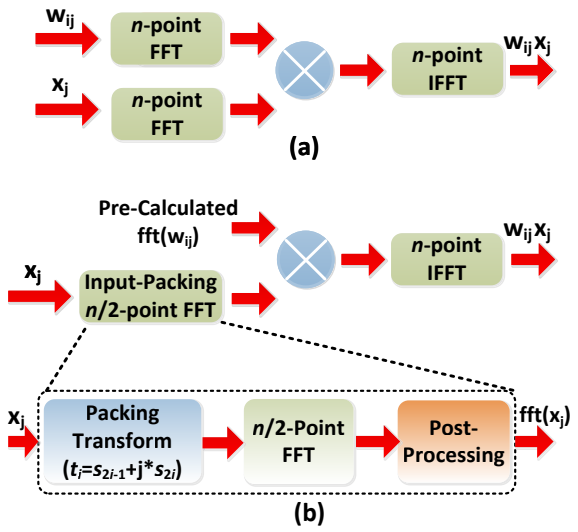

Fig. 1. (a) Straightforward design of 3-step FFT-involved operation. (b) Optimized design using pre-calculation and input-packing strategy.

A. Optimization on FFT/IFFT Module

As indicated in Algorithm 1, the key computation of the forward propagation process in the proposed block-circulant matrix-based inference scheme is the FFT/IFFT calculation. Consider that all the other operations in the inference, including element-wise multiplication and addition and ReLU, have a computational complexity of $O(n)$, the optimization on the FFT module is the key to improve the hardware performance of the entire DNN accelerators.

Fig. 1 (a) shows a straightforward design for the 3-step FFT-involved operation: (FFT -¿ Element-wise Multiplication -¿ IFFT). It can be seen that, in order to perform the inference for a size-n input vector $\mathbf{x}_j$ on one layer, totally 3 copies of FFT or IFFT are needed (since IFFT can be easily performed on the FFT module via very slight modification [17]). Although the hardware performance of this design is already very promising because of its $O(n \log n)$ computational complexity, we present the following optimization techniques that can further improve the hardware performance.

Pre-Calculation of fft($\mathbf{w}_{ij}$): Recall that in the inference phase the weights of DNNs have already been determined from the training phase. This phenomenon implies that fft($\mathbf{w}_{ij}$) can be actually off-line pre-calculated. By using this property, we only need to implement one FFT and one IFFT in the hardware (as shown in Fig. 1 (b)), thereby resulting in 33% reduction in the hardware resource of the data-path.

Packing the Real-valued Inputs: The standard FFT/ IFFT hardware is typically used for the complex input signals that widely exist in communication or signal processing systems. Different from those two types of applications, most artificial intelligence applications, such as computer vision, natural language processing etc., only deal with real-valued data. Based on this observation, the hardware architecture of FFT module can be further optimized by using the *input-packing* strategy.

More specifically, the packing strategy utilizes the property that a size-n real-input FFT (rFFT) can be computed by using a size-n/2 standard (complex-input) FFT [18]. In general, in order to calculate the n-point FFT of a real-valued size-n vector $\mathbf{s}$, we can first generate a complex-valued size-n/2

TABLE I
HARDWARE PERFORMANCE AND PRECISION LOSS OF 128-POINT FFT USING DIFFERENT QUANTIZATION SCHEMES

Bits	Scheme (sign,int.,frac.)	Precision loss (Avg. L^2 distance)	performance (equivalent GOPS)	energy efficiency (equivalent GOPS/W)
8	(1,3,4)	0.405	3,506.18	5,156.14
12	(1,5,6)	0.157	3,047.42	4,481.50
16	(1,7,8)	0.113	2,686.97	3,951.43

vector $\mathbf{t}$, where $t_i = s_{2i-1} + j \cdot s_{2i}$ for $i = 1, 2, \ldots, n/2$. And then perform n/2-point FFT for $\mathbf{t}$. After simple post-processing with $4n-1$ additions and $4n-4$ multiplications, the n-point fft($\mathbf{s}$) is calculated. Fig. 1 (b) shows the block diagram of the packing-based FFT design. Obviously, such packing strategy leads to additional nearly 25% reduction in the required computing resource.

B. Quantization Scheme

The choice of quantization scheme is critical for the fixed-point hardware design since it greatly affects the hardware performance and the potential accuracy loss as compared to the floating-point software implementation. Therefore, in this paper we explore different quantization schemes to achieve the maximum hardware performance within the affordable precision loss. Consider FFT is the key computation block in the proposed design, we investigate the performance of fixed-point FFT in terms of computation performance, energy-efficiency, and precision loss when using different quantization schemes. As shown in Table I, we explore three different quantization schemes to represent numbers in the computation of an example 128-point FFT. Here (1,i,f) means a (1+i+f)-bit fixed-point binary numbers where 1 bit is used as sign bit, i bits are used for integral parts, and f bits are used for fractional parts. To measure the performance, giga operations per second (*GOPS*), giga operations per Watt (*GOPS/W*) and the average L^2 distance among all the outputs, are used as the metrics for computation performance, energy efficiency and precision loss, respectively. Notice that here 10000 sets of random inputs within the range [-1, 1] are used to calculate the precision loss incurred by the quantization scheme. Also note that, as other related works like [19] did, we calculate the number of operations using the *equivalent operations* which means we count operations in the original matrix multiplications instead of in our proposed circulant matrix based FFT/IFFTs.

From Table I it is seen that, 8-bit quantization scheme can lead to the best computation performance and energy efficiency, but it meanwhile causes the most significant precision loss. Also, the computation performance and energy efficiency of 16-bit quantized design are only half of those of 8-bit design though it has the smallest precision loss. Therefore, in order to achieve good balance between the hardware performance and precision loss, 12-bit (1,5,6) scheme is adopted as the underlying quantization scheme in the proposed DNN hardware design.

C. Overall Hardware Architecture

Fig. 2 shows the overall hardware architecture of the proposed block-circulant matrix-based DNN accelerator. Here the FFT/IFFT module is responsible for performing FFT/IFFT operations. All the other operations in the inference, such as element-wise multiplication, activation function etc., are performed by peripheral computing module. The memory module consists of ROM that stores the constant coefficients

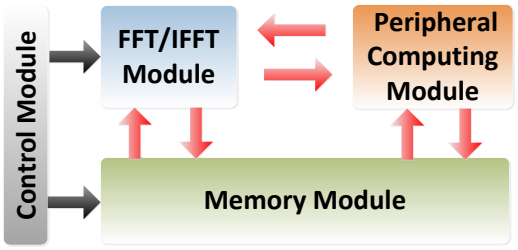

Fig. 2. Proposed overall hardware architecture.

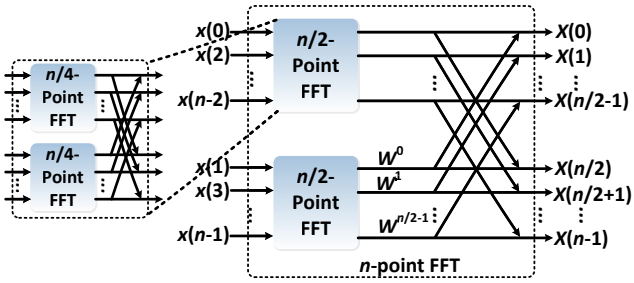

Fig. 3. Recursive property of FFT.

of FFT and RAM that stores the weights in the frequency domain (fft($\mathbf{w}_{ij}$)).

Notice that the requirement on the reconfigurability has become very important for the modern DNN accelerator design because 1) The size of the deployed network may be changed due to the change of target applications; and 2) the area constraint may require to map the function of two or several layers on the same hardware. Fortunately, the proposed FFT-based DNN hardware is well-suitable for achieving the reconfigurability because of the inherent recursive property of FFT: As indicated in Fig. 3, the computation of n-point FFT can be viewed as the parallel processing of two n/2-point FFT plus a butterfly-style post-processing, and each n/2-point FFT can be further decomposed to the smaller size FFT using the similar way. Such strong recursive property offers great benefits to improve the reconfigurability of the proposed DNN accelerator. This is because for any given-size FFT computation that is needed in the inference process, even if it is not specifically implemented on the FFT hardware module, it is still easy to calculate it by performing recursive computation (for large-size FFT) or directly using partial hardware resource of the existing FFT module (for small-size FFT). Also, the design of control signal and memory storage organization also become quite convenient because of the ultra-regular structure of FFT.

VI. SOFTWARE-HARDWARE DESIGN OPTIMIZATIONS

As indicated in Section IV, the compression ratio of the block-circulant matrix-based DNN can be adjusted by changing the partition sizes (k) of the weight matrices. In general, increasing k can lead to a higher compression ratio, which further translates to smaller memory requirement, better energy efficiency and higher computation performance. However, though maximizing k (=$min(m,n)$) for each layer can always achieve the best hardware performance, in many cases such aggressive compression needs to be relaxed in order to avoid the potentially non-negligible test accuracy loss. Therefore, it is very necessary to perform software-hardware design co-

optimization to achieve the best balance between the compression ratio and test accuracy.

Motivated by this demand, in this section we propose a design flow to explore the the impact of partition size k on the software-level test accuracy and the hardware performance of the network. Fig. 4 shows our proposed design flow, which mainly includes two phases, *greedy partitioning* (blue arrow) and *back-trace adjustment* (yellow arrow).

Next, we describe this design flow in details. Initially, the baseline network accuracy (without any compression) is precomputed. Then we sort weight matrices by size. If we index d weight matrices of a DNN from input end to output end, namely $1, 2, ..., d$; then after sorting, ranked by the size, the sequence of the matrices' indices are $i_1, i_2, ..., i_d$, where $i_j \in \{1, 2, ..., d\}$, and $size_{i_j} \geq size_{i_{j+1}}$.

Greedy Partitioning: Starting from matrix i_1, we apply the block-circulant matrix with the largest possible partition size (=$min(m, n)$), denoted as p_{i_1}, and then train the network. When we are setting up the partition size of layer i_j ($i_j \in \{1, 2, ..., d\}$), if the modified model can achieve a negligible degradation (less than a pre-set $t\%$) in test accuracy, we move to the next matrix i_{j+1}. Notice that here we intentionally start this greedy partitioning from the larger-size matrices first, and then gradually move to the smaller-size matrices. Such arrangement can guarantee the overall compression ratio always remains at a high level since the compression on the large-size matrices tends to be more significant.

Back-trace Adjustment: Sometimes the current selected partition size is too aggressive and hence causes non-negligible accuracy loss. In this scenario, we reduce the partition size p_{i_j} of layer i_j by a factor of 2. Such reduction in p_{i_j} may need to be performed using several rounds until the entire network can achieve the degradation within $t\%$. Notice that if the partition size of matrix i_j is reduced to 1, the procedure goes back to matrix i_{j-1} and $p_{i_{j-1}}$ is reduced by a factor of 2. After that, we move to matrix i_j again with resetting the partition size of matrix i_j to the the largest possible value.

Next we use an example seven-layer DNN on MNIST dataset to illustrate this proposed software/hardware co-design flow. The layer sizes of the example DNN are set as 784-2048-2048-1024-1024-512-10, which are close to the configuration in [20]. Here the shapes of the weight matrices between adjacent layers are labeled in Table II. The input data for the model is normalized by scaling the pixels of each data case down to the range [0,1]. Both the original and compressed models are trained with the learning rate set as 0.01. All experiments are averaged over 5 times of training 100 epochs from scratch.

Without loss of generality, we set the goal of this example design flow as finding a compressed model that achieves the maximum compression ratio (and hence the maximum energy efficiency) with accuracy degradation less than 1% ($t = 1$). Table III illustrates some compressed models and the corresponding test accuracies when following the design flow in Fig. 4. Here each model is denoted by the combination of the partition size of each weight matrix. For example, for model 1, which is actually the original uncompressed model, is denoted as 1-1-1-1-1 since it can be viewed as the compressed model with setting partition size as 1 for all weight matrices. Consistent with the protocol of the design flow, the matrix-wise compression procedure follows the ranking of weight matrix size: In this example we try to compress each individual weight matrix in the order of matrices $2, 3, 1, 4, 5$. When we

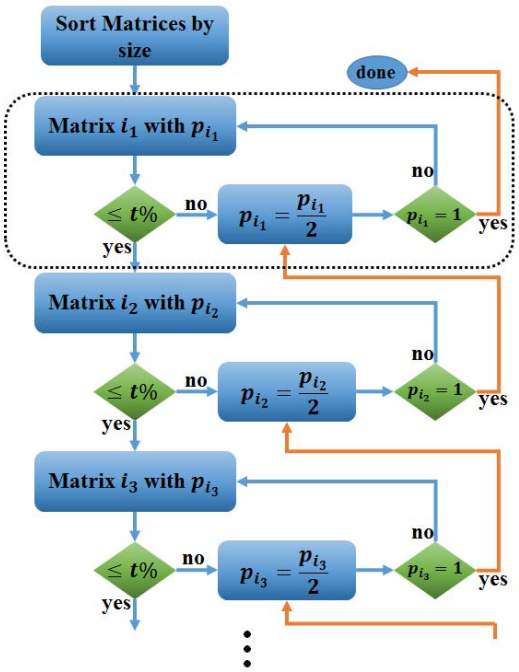

Fig. 4. Design flow to set up partition size given a network.

TABLE II
SHAPES OF MATRICES IN THE EXAMPLE DNN MODEL

Matrix ID	Layer	Matrix shape	Type
1	1-2	784×2048	FC
2	2-3	2048×2048	FC
3	3-4	2048×1024	FC
4	4-5	1024×1024	FC
5	5-6	1024×512	FC
6	6-7	512×10	Softmax

begin to compress the current weight matrix, we always start with the largest partition size (greedy partitioning) to get the highest compression ratio, and may reduce its partition size or even change the partition size of previous compressed weight matrices to achieve the balance between compression ratio and accuracy (back-trace adjustment). As shown in Table III, by following the proposed design flow, finally we get a model with 108.29 compression ratio and 97.67% accuracy. In general, the design flow in Fig. 4 can always ensure the finding of the maximally compressed DNN model that satisfies the pre-defined accuracy requirement, thereby achieving the optimal hardware performance (in terms of energy efficiency, memory size etc.) under the given constraint on accuracy.

VII. EXPERIMENTAL RESULTS

With the optimal configurations of the partition size determined by the aforementioned design flow, in this section we show the software and hardware performance of DNNs on commonly used dataset. Software accuracy and hardware performance (computation performance in term of $GOPS$ and energy efficiency in term of $GOPS/W$) are measured for each network. The hardware results are based on the mapping results of a low-power and low-cost Intel® (Altera) Cyclone V FPGA, which can accommodate the weight storage

The last softmax layer in this example is not considered for compression since the size of softmax layer is already very small.

TABLE III
SAMPLES OF CONFIGURATIONS OF THE PARTITION SIZE IN THE DESIGN FLOW

ID	Model Partition size for matrix $1 \sim 5$	Accuracy (%)	Compression Ratio
1	1-1-1-1-1	98.34	1.00
2	1-2048-1-1-1	97.93	1.79
3	1-2048-1024-1-1	97.81	2.97
4	784-2048-1024-1-1	96.25	5.96
5	512-2048-1024-1-1	96.41	5.95
	...	...	...
	128-128-128-128-128	97.67	108.29

TABLE IV
PERFORMANCE ON MNIST DATASET W.R.T. DIFFERENT QUANTIZATION SCHEMES

Scheme	Accuracy (%)	Performance (equivalent GOPS)	Energy Efficiency (equivalent GOPS/W)
(1,3,4)	96.33	3,506.18	3,690.71
(1,5,6)	97.19	3,047.42	3,017.25
(1,7,8)	97.58	1,343.49	2,442.71

Total Equivalent Operations: 18,939,904
Parameter Compression Ratio: 108.29

TABLE V
PERFORMANCE ON SVHN DATASET W.R.T. DIFFERENT QUANTIZATION SCHEMES

Scheme	Accuracy (%)	Performance (equivalent GOPS)	Energy Efficiency (equivalent GOPS/W)
(1,3,4)	93.61	3,506.18	3,690.71
(1,5,6)	94.19	3,047.42	3,017.25
(1,7,8)	94.85	1,343.49	2,442.71

Total Equivalent Operations: 56,623,104
Parameter Compression Ratio: 118.48

requirement of representative deep learning applications after performing the proposed block-circulant matrix-based compression. Please note that the Cyclone V FPGA exhibits a low static power consumption less than $0.15W$ and typical operating frequency between 227MHz and 250MHz, making it a good choice for energy efficiency optimization of FPGA-based deep learning implementations.

A. MNIST

The MNIST [21] is a handwritten digit dataset (10 classes) that consists of 28×28 grey-scale images with 60,000 images for training and 10,000 images for testing. The baseline model here we use is the example model (784-2048-2048-1024-1024-512-10) discussed in Section VI. Clearly, as indicated in Table II, the optimal partitioning scheme for this example model is 128-128-128-128-128 (for weight matrix $1 \sim 5$), which means the required FFTs/IFFTs for this configuration are 128-point FFT/IFFT.

Compression Ratio & **Accuracy:** With the use of this partitioning scheme, the number of parameters of DNN can be compressed by $108X$, which leads to $108X$ saving in memory size. Such very high compression in model size, however, does not cause severe accuracy loss. As shown in Table IV, compared with the baseline uncompressed model with accuracy of 98.34% (see Table III), the FPGA-based DNN inference hardware can achieve test accuracy as high as 97.58% with the quantization scheme of (1,7,8) (see Table IV). That means the proposed DNN hardware can achieve $100+$X compression in memory size with less than 1% accuracy loss.

Computation Performance & **Energy Efficiency:** Table IV also shows that the proposed block-circulant matrix-based DNN hardware can achieve as high as $3,690$ (equivalent) GOPS/W energy efficiency and $3,506$ (equivalent) GOPS with 8-bit quantization scheme. Besides, the 12-bit quantized design can give a better trade-off among accuracy, the computation performance, and the energy efficiency. It should be noted that depending on the choice of specific quantization scheme, one 128-point FFT/IFFT module in the Cyclone V FPGA utilizes $44\% \sim 62\%$ resources. Thus only one of FFT/IFFT with quantization scheme (1,7,8) can reside in the FPGA; while for other schemes two of FFT/IFFT can be mapped to the FPGA. That explains why the computation performance with quantization scheme (1,7,8) drops significantly as compared to the ones using other schemes.

B. SVHN

The SVHN [22] is a color image dataset that are collected from house numbers in Google Stree View Images. The data set, which has 10 classes, contains $73,257$ training images and $531,131$ additional training images plus $26,032$ testing images. In this experiment, we follow the same augmentation method as adopted in [23]: padding 4 pixels on each dimension and randomly sampling 32×32 crops. The baseline model we use for this dataset is a nine-layer DNN with the size of 3072-2048-2048-2048-2048-2048-2048-512-10.

Compression Ratio & **Accuracy:** The uncompressed baseline software model gives 96.17% classification accuracy. Again, we perform the design flow described in Section VI to explore the optimal partition scheme. Notice that here considering the complexity of multi-channel inputs and enormous number of parameters, we set $t = 2$. Our experiments show that, by following the design flow in Section VI, 128-128-128-128-128-128-128 (for weight matrix $1 \sim 7$) is identified for the optimal partition scheme. As a result, the overall DNN model is compressed by $118.48X$. As shown in Table V, such very high compression still incur negligible performance loss even after using quantization scheme.

Computation Performance & **Energy Efficiency:** Based on the aforementioned partition scheme, the FFTs/ IFFTs needed for this experiment is also 128-point. Since 1) we use same the FPGA platform to build the hardware for this SVHN dataset and MNIST dataset and 2) the required type of FFT/IFFTs are identical (128-point), it is not surprising that the computation performance and energy efficiency for the DNN hardware target to these two datasets are the same (see Table IV and Table V), even though their total equivalent operations are different. Also, Table V shows that 12-bit quantization scheme gives the best trade-off among accuracy, the computation performance, and the energy efficiency, which is consistent with the phenomenon we observe from Table IV.

C. Comparison with state-of-the-art works

The reference FPGA-based implementations are state-of-the-arts represented by [FPGA16] [24], [ICCAD16] [25], [FPGA17, Han] [19], and [FPGA17, Zhao] [26]. In those work, large-scale AlexNet, VGG-16, or a custom-designed recurrent neural network [19] were implemented. For fair comparison, we use equivalent GOPS and GOPS/W for all model compression methods, including ours. Although those references may focus on different deep learning models and

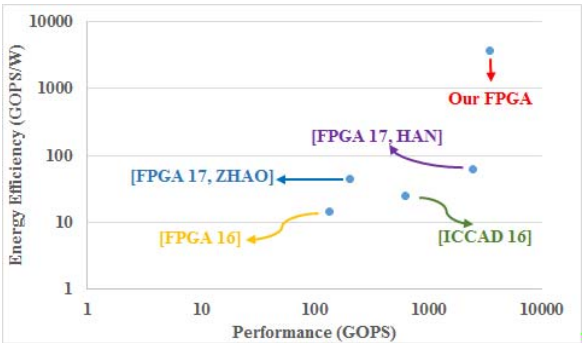

Fig. 5. Comparison on performance and energy efficiency with state-of-the art FPGA results.

structures, both GOPS and GOPS/W are general metrics that are (in general) independent of the differences of models.

In Fig. 5, we can observe significant improvement achieved by the proposed FPGA-based implementations as compared with prior arts in terms of energy efficiency. Compared with prior work with heuristic model size reduction techniques [19], [26] (reference [19] uses the heuristic weight pruning method, and [26] uses a binary-weighted neural network XORNet.), our approach achieves 60X $\sim$ 83X improvement in energy efficiency. When comparing with prior arts with uncompressed (or partially compressed) deep learning system [24], [25], the energy efficiency improvement of our approach can reach 150X $\sim$ 260X. These results demonstrate a clear advantage of the proposed framework using block-circulant matrices on improving computation performance and energy efficiency.

VIII. Conclusion

In this paper, we presented a holistic framework for energy-efficient high-performance highly-compressed DNN hardware design. First, we propose block-circulant matrix-based DNN training and inference schemes, which theoretically guarantee Big-O complexity reduction in both computational cost and storage requirement of DNNs. Second, we dedicatedly optimize the hardware architecture, especially on the key fast Fourier transform (FFT) module, to improve the overall performance in terms of energy efficiency, computation performance and resource cost. Third, we propose a design flow to perform hardware-software co-optimization with the purpose of achieving good balance between test accuracy and hardware performance of DNNs. Based on the proposed design flow, two block-circulant matrix based DNNs on two different datasets are implemented and evaluated on Intel Cyclone V FPGA. The fixed-point quantization and block-circulant matrix inference scheme enables the network to achieve as high as 3.5 TOPS (equivalent) computation performance and 3.69 TOPS/W energy efficiency while the memory is saved by 108X and 116X, respectively. Accuracies in both networks are negligibly degraded.

Acknowledgement

This work is supported by the NSF Algorithm-in-the-Field (AitF) program, the DARPA SAGA program, CUNY Research Foundation, and Syracuse University.

References

[1] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[2] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, p. 62, 2016.

[3] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of Machine Learning Research*, vol. 12, no. Aug, pp. 2493–2537, 2011.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[5] K. Hwang and W. Sung, "Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1," in *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*. IEEE, 2014, pp. 1–6.

[6] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An ultra-low power convolutional neural network accelerator based on binary weights," in *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*. IEEE, 2016, pp. 236–241.

[7] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[8] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, pp. 262–263.

[9] J. Chung and T. Shin, "Simplifying deep neural networks for neuromorphic architectures," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.

[10] Y. LeCun, J. S. Denker, S. A. Solla, R. E. Howard, and L. D. Jackel, "Optimal brain damage." in *NIPs*, vol. 2, 1989, pp. 598–605.

[11] 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," *arXiv preprint arXiv:1602.02830*, 2016.

[12] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[13] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[14] V. Pan, *Structured matrices and polynomials: unified superfast algorithms*. Springer Science & Business Media, 2012.

[15] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S.-F. Chang, "An exploration of parameter redundancy in deep networks with circulant projections," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2857–2865.

[16] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

[17] A. V. Oppenheim, *Discrete-time signal processing*. Pearson Education India, 1999.

[18] E. O. Brigham, "The {F} ast {F} ourier {T} ransform and its applications," 1988.

[19] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 75–84.

[20] M. W. Gardner and S. Dorling, "Artificial neural networks (the multilayer perceptron)a review of applications in the atmospheric sciences," *Atmospheric environment*, vol. 32, no. 14, pp. 2627–2636, 1998.

[21] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," *URL http://yann. lecun. com/exdb/mnist*, 1998.

[22] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS workshop on deep learning and unsupervised feature learning*, vol. 2011, no. 2, 2011, p. 5.

[23] C.-Y. Lee, S. Xie, P. W. Gallagher, Z. Zhang, and Z. Tu, "Deeply-supervised nets." in *AISTATS*, vol. 2, no. 3, 2015, p. 5.

[24] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26–35.

[25] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks," in *Proceedings of the 35th International Conference on Computer-Aided Design*. ACM, 2016, p. 12.

[26] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable fpgas," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 15–24.