IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 5, 2018, accepted December 19, 2018, date of publication December 28, 2018,
date of current version January 23, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2890150

FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review

AHMAD SHAWAHNA[1], SADIQ M. SAIT[1,2], (Senior Member, IEEE), AND AIMAN EL-MALEH[1], (Member, IEEE)

[1]Department of Computer Engineering, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
[2]Center for Communications and IT Research, Research Institute, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia

Corresponding author: Sadiq M. Sait (sadiq@kfupm.edu.sa)

This work was supported by the King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia.

ABSTRACT Due to recent advances in digital technologies, and availability of credible data, an area of artificial intelligence, deep learning, has emerged and has demonstrated its ability and effectiveness in solving complex learning problems not possible before. In particular, convolutional neural networks (CNNs) have demonstrated their effectiveness in the image detection and recognition applications. However, they require intensive CPU operations and memory bandwidth that make general CPUs fail to achieve the desired performance levels. Consequently, hardware accelerators that use application-specific integrated circuits, field-programmable gate arrays (FPGAs), and graphic processing units have been employed to improve the throughput of CNNs. More precisely, FPGAs have been recently adopted for accelerating the implementation of deep learning networks due to their ability to maximize parallelism and their energy efficiency. In this paper, we review the recent existing techniques for accelerating deep learning networks on FPGAs. We highlight the key features employed by the various techniques for improving the acceleration performance. In addition, we provide recommendations for enhancing the utilization of FPGAs for CNNs acceleration. The techniques investigated in this paper represent the recent trends in the FPGA-based accelerators of deep learning networks. Thus, this paper is expected to direct the future advances on efficient hardware accelerators and to be useful for deep learning researchers.

INDEX TERMS Adaptable architectures, convolutional neural networks (CNNs), deep learning, dynamic reconfiguration, energy-efficient architecture, field programmable gate arrays (FPGAs), hardware accelerator, machine learning, neural networks, optimization, parallel computer architecture, reconfigurable computing.

I. INTRODUCTION

In recent years, due to the availability of massive amounts of credible data (Big Data: Text, Video, Audio, etc.), and tremendous advances in the area of digital electronics technologies that provide immense computing power, there has been a revival in the area of artificial intelligence (AI), particularly in the area of deep learning (DL) [1]–[3], a sub-field of machine learning (ML).

The field of DL emerged in 2006 after a long pause in the area of neural networks (NNs) research [4]. A key aspect in DL is that the networks and/or their weights are *not* designed by human beings. Instead, they are learned from data using a general purpose learning procedure [5], [6].

While ML uses algorithms to parse and learn from data, to make informed decisions, DL structures algorithms in layers to create an artificial neural network (ANN) that can learn, and similar to human intelligence, can make accurate decisions on its own [7]. Therefore, instead of designing algorithms by hand, systems can be built and trained to implement concepts in a way similar to what comes naturally to humans, and with accuracy sometimes *exceeding* human-level performance [8], [9].

In DL, each layer is designed to detect features at different levels. A layer transforms the representation at one level (starting from input data which maybe images, text, or sound) to a representation at a higher, slightly more abstract level [10]. For example, in image recognition, where input initially comes in the form of pixels, the first layer detects low level features such as edges and curves. The output of the first layer becomes input to the second layer which

produces higher level features, for example semi-circles, and squares [11]. The next layer assembles the output of the previous layer to parts of familiar objects, and a subsequent layer detects the objects. As we go through more layers, the network yields an activation map that represents more and more complex features. The deeper you go into the network, the filters begin to be more responsive to a larger region of the pixel space. Higher level layers amplify aspects of the received inputs that are important for discrimination and suppress irrelevant variations.

A. APPLICATIONS OF DEEP LEARNING NETWORKS

With the now widely used convolution neural networks (CNNs) [12], [13] and deep neural networks (DNNs) [14], [15], it is now possible to solve problems in domains where knowledge is not easily expressed explicitly and implicit information is stored in the raw data. Solutions to multifarious problems in the domain of sciences, business, etc., have been possible that were not conceivable for several years, in spite of best attempts by the AI community. This has been primarily possible due to the excellent ability of deep learning in discovering intricate structures in high-dimensional data. Examples include character recognition [16], gesture recognition [17], speech recognition (e.g., in Google Now, Siri, or click-through prediction on an advertisement) [18]–[20], document processing [21]–[23], natural language processing [24], [25], video classification [26], image classification [27]–[32], face detection and recognition [33], [34], robot navigation [35]–[37], real-time multiple object tracking [38], financial forecasting [39], and medical diagnosis systems [40]–[42], to name a few.

Other recent areas of applications include automated driving (e.g., learning to detect stop signs, traffic lights, pedestrians, etc.), aerospace and defense (e.g., identify objects from satellites and identify safe or unsafe zones), medical research (e.g., in identification of cancer cells), industrial automation (e.g., to improve worker safety by detecting when people or objects are within an unsafe distance of machines), and electronics (used in automated hearing, speech translation, etc.) [9], [43]–[46].

B. EMERGENCE OF DEEP LEARNING NETWORKS

Convolutional neural networks are considered as one of the most influential innovations in the field of computer vision [47]. The success of deep learning networks grew to prominence in 2012 when Krizhevsky *et al.* [28] utilized CNNs to win the annual olympics of computer vision, ImageNet large-scale vision recognition challenge (ILSVRC) [30]. Using AlexNet model, they achieved an astounding improvement as the image classification error dropped from 26% (in 2011) to 15%. ImageNet is a standard benchmark dataset used to evaluate the performance of object detection and image classification algorithms. It consists of millions of different images distributed over tens of thousands of object classes.

CNNs have achieved even better accuracy in classification and various computer vision tasks. The classification accuracy in ILSVRC improved to 88.8% [48], 93.3% [31], and 96.4% [49] in the 2013, 2014, and 2015 competitions, respectively. Fig. 1 shows the accuracy loss for the winners of ImageNet competitions before and after the emergence of deep learning algorithms.

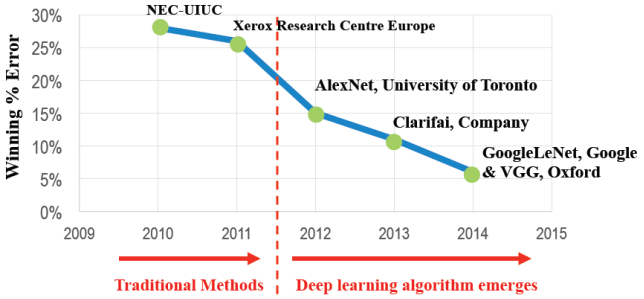

FIGURE 1. ImageNet Competition Results [50].

Thereafter, large host companies started using CNNs at the core of their services. Google, Microsoft, Facebook, Amazon, Pinterest, and Instagram are currently using neural networks for their photo search, Bing's image feeds, automatic tagging algorithms, product recommendations, home feed personalization, and for their search infrastructure, respectively [11]. However, the classic use-case of CNNs is for image and speech processing [51].

A typical CNN is a multi-layered feed-forward ANN with a pipeline-like architecture. Specifically, each layer performs a well-known computation on the outputs of the previous layer to generate the inputs for the next layer. In general, CNNs have two types of inputs; the data to be tested or classified (also named as feature maps), and the weights. Images, audio files, and recorded videos are examples of the input data to be classified using CNNs. On the other hand, the network weights are the data generated from training the CNN on a dataset containing similar inputs to the one being tested.

C. HARDWARE ACCELERATION OF DEEP LEARNING NETWORKS

To provide more accurate results as well as real-time object recognition, for example in applications such as robots and auto-piloted cars, the size of the convolution neural network needs to be increased by adding more neural network layers [28]. However, evolving more and new type of NN layers results in more complex CNN structures as well as high depth CNN models. Thus, billions of operations and millions of parameters, as well as substantial computing resources are required to train and evaluate the resultant large-scale CNN [31], [52], [53]. Such requirements represent a computational challenge for general purpose processors (GPP). Consequently, hardware accelerators such as application specific integrated circuit (ASIC), field programmable gate array (FPGA), and graphic processing unit (GPU) have been employed to improve the throughput of the CNN.

In practice, CNNs are trained off-line using the back-propagation process [54]. Then, the off-line trained CNNs are used to perform recognition tasks using the feed-forward process [55]. Therefore, the speed of feed-forward process is what matters.

GPUs are the most widely used hardware accelerators for improving both training and classification processes in CNNs [56]. This is due to their high memory bandwidth and throughput as they are highly efficient in floating-point matrix-based operations [57]–[59]. However, GPU accelerators consume a large amount of power. Therefore, their use in CNN-based applications implemented as a cloud service on large servers or in battery operated devices becomes a challenge. Furthermore, GPUs gain their performance from their ability to process a large image batch in parallel. For some applications like a video stream, input images should be processed frame by frame as the latency of the result of each frame is critical to the application's performance. For some tracking algorithms, the result of one frame affects the process of the next frame [60]. Nurvitadhi *et al.* [61] recently evaluated emerging DNN algorithms on latest generations of GPUs (i.e., NVIDIA Titan X Pascal) and FPGAs (i.e., Intel Arria 10 GX 1150 and Intel Stratix 10 2800). The experimental results show that current trends in deep neural networks favor FPGA platforms as they offer higher power efficiency (a.k.a., performance per Watt).

FPGA and ASIC hardware accelerators have relatively limited memory, I/O bandwidths, and computing resources compared with GPU-based accelerators. However, they can achieve at least moderate performance with lower power consumption [62]. The throughput of ASIC design can be improved by customizing memory hierarchy and assigning dedicated resources [63]. However, the development cycle, cost, and flexibility are not satisfactory in ASIC-based acceleration of deep learning networks [64], [65]. As an alternative, FPGA-based accelerators are currently in use to provide high throughput at a reasonable price with low power consumption and reconfigurability [66], [67]. The availability of high-level synthesis (HLS) tools, using C or C++, from FPGA vendors lowers the programming hurdle and shortens the development time of FPGA-based hardware accelerators [68]–[70].

Convolutional neural networks have a very useful property, that is, each feature map neuron shares its weights with all other neurons [71]. Hameed *et al.* [72] and Keckler *et al.* [73] proved that the highest energy expense results from accessing the off-chip DRAM memory for data movement rather than computation. In other words, the energy cost of the increased memory accesses and data movement due to the large number of CNN operations often exceeds the energy cost of computation [64], [74]. Thus, CNN accelerators need to carefully consider this to achieve efficient architecture in terms of time and power.

In this paper, we review the current status of using FPGAs as accelerators for implementing deep learning networks. We highlight the implementation challenges and design directions used to tackle those challenges. We also provide future recommendations to maximize the performance of FPGAs as accelerators for deep learning networks and simplify their use.

The remainder of the paper is organized as follows. Section II provides background information about CNNs, their key operations, and some well-known deep learning networks. In addition, it introduces the basic structure of FPGAs and highlights their features enabling them to accelerate computationally intensive applications. It also discusses the implementation challenges of deep learning networks on FPGAs and how these challenges can be overcome. Section III reviews existing CNNs compression techniques and presents the current status of accelerating deep learning networks using ASIC-based and FPGA-based accelerators. Section IV describes the use of metaheuristics in the design and optimization of CNNs implementation. Section V summarizes existing design approaches for accelerating deep learning networks and provides recommendations for future directions that will simplify the use of FPGA-based accelerators and enhance their performance. Finally, section VI concludes the paper.

II. BACKGROUND AND TERMINOLOGY

This section gives an overview of the key operations and terminology used in convolutional neural networks (CNNs) and provides examples of well-known deep learning networks. In addition, it illustrates the basic structure of field programmable gate arrays (FPGAs) and how deep learning methods can benefit from the capabilities of FPGAs. The last subsection highlights the challenges of implementing deep learning networks on FPGAs.

A. CONVOLUTIONAL NEURAL NETWORKS (CNNs)

In this subsection, we describe the key operations and terminology involved in the construction of CNNs including convolution, activation functions, normalization, pooling, and characteristics of fully connected layers.

1) CONVOLUTION (CONV)

A convolution operation can be thought of as the production of a matrix smaller in size than the original image matrix, representing pixels, by sliding a small window (called filter, feature identifier, or kernel) of size $k \times k$ over the image (called input feature map (FM)), to produce an output feature neuron value [75]. The filter is an array of numbers called weights or parameters. These weights are computed during the training phase. As the filter slides over the feature map, it multiplies the values in the filter with the original pixel values, that is, it first performs element-wise multiplication, and then sums the products, to produce a single number. The inputs and outputs of the CONV layer are a series of FM arrays.

This operation, starting from the top left corner of the FM, is repeated by moving the window S strides at a time, first in the right direction, until the end of the FM is reached, and then

proceeding downwards until the FM is completely scanned and all the elements of the FM are covered. The sliding of the filter window and performing the operation is known by the verb convolving, hence the noun *convolution* [11], [76]. Normally, the size of the kernel is very small, less than or equals to 11 × 11. Each output-input FM pair has a set of weights equal to the kernel size and each output FM is computed based on the sum of the convolution operations performed on all input FMs. Note that different CONV layers in the same CNN model vary considerably in their sizes.

In summary, the convolution operation comprises four levels of loops; the output FMs loop (*Loop-4*), the loop across the input FMs (*Loop-3*), the loop along the dimensions of a single input FM (scan operation, *Loop-2*), and the kernel window size loop (multiply-and-accumulate (MAC) operation, *Loop-1*). CONV layers are dominant in CNN algorithms since they often constitute more than 90% of the total CNN operations [28], [29], [49], [74], [77], [78]. Therefore, many attempts have been made to speedup CONV operations using loop unrolling technique [55], [79], as will be discussed later. Loop unrolling maximizes the parallelism of CONV MACs computation which requires a special consideration of processing elements (PEs) and register arrays architecture. Fig. 2 illustrates the loop unrolling of CONV loops levels.

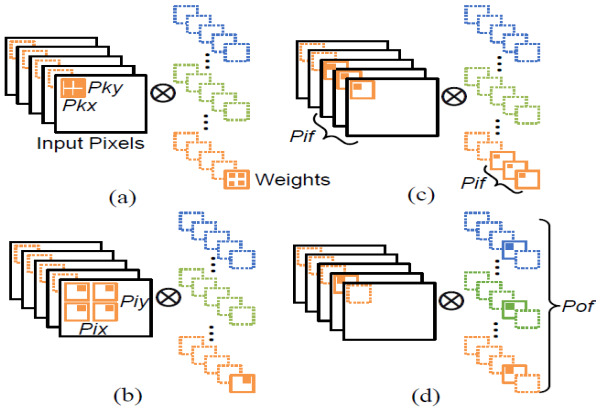

FIGURE 2. CONV Loops Unrolling [83]: (a) Unrolling *Loop-1*; (b) Unrolling *Loop-2*; (c) Unrolling *Loop-3*; (d) Unrolling *Loop-4*, where, *Pkx*, *Pky*, *Pix*, *Piy*, *Pif*, and *Pof* are loop unrolling design variables for the kernel window width, kernel window height, input FM width, input FM height, number of input FMs, and the number of output FMs, respectively.

2) ACTIVATION FUNCTIONS (AFs)

Activation function in neural networks is similar to *action potential* in animal cells such as neurons. A neuron is said to *fire* if it emits an action potential. A popularly used activation function is the *sigmoid* function which can be expressed as

$$f(x) = 1/(1 + e^{-x}) \tag{1}$$

where x represents the weighted sum of the neuron inputs and if it is a sufficiently large positive number, the *sigmoid* function approximates to unity. For sufficiently large negative values of x, the *sigmoid* function is close to 0. Another popular activation function is

$$f(x) = tanh(x) \tag{2}$$

The above standard *sigmoid* and *tanh* non-linear functions require long training time [28]. A recently proposed and commonly used AF in CNNs is rectified linear unit (ReLU) which is defined as

$$f(x) = max(x, 0) \tag{3}$$

ReLU activation function is known to converge faster in training, and has lesser computational complexity [80], [81] than standard *sigmoid* and *tanh* functions. In addition, it does not require input normalization to prevent it from saturating [28], [80], [82].

3) NORMALIZATION

In real life, a phenomenon called 'lateral inhibition' appears, which refers to the capacity of an excited neuron to subdue its neighbors, thereby creating a contrast in that area. In CNNs, to accomplish this, local response normalization (LRN) or simply *normalization* is used, particularly when dealing with ReLU neurons, because they have unbounded activation that needs normalization. It detects high frequency features with a large response. If we normalize around the local neighborhood of the excited neuron, it becomes even more sensitive as compared to its neighbors. At the same time, it will dampen the responses that are uniformly large in any given local neighborhood. If all the values are large, then normalizing those values will diminish all of them. So, basically it performs some kind of inhibition and boosts the neurons with relatively larger activations.

Normalization can be done within the same feature or across neighboring features by a factor that depends on the neighboring neurons. Expressions to compute the response normalized activity can be found in [28] and [80].

4) POOLING

Pooling, also known as *subsampling*, is employed to progressively reduce the spatial size of the representation, thereby reducing the amount of parameters and computation in the network. Pooling layers are periodically inserted in between successive convolutional layers. They operate independently on every depth slice of the input and resize it spatially using the *MAX* operation. The most common form is a pooling layer with filters of size 2 × 2 applied where the *MAX* operation would be taking a maximum over 4 samples thereby discarding 75 percent of the activations [84]. In addition to the popular *MAX* pooling, the pooling units in some CNNs are also used to perform other functions, such as *AVG* and *MIN* operations [80].

5) FULLY CONNECTED LAYER (FC)

A common form of a convolutional neural network architecture comprises stacks of a few convolutional and ReLU layers, followed by layers for pooling, and this pattern is

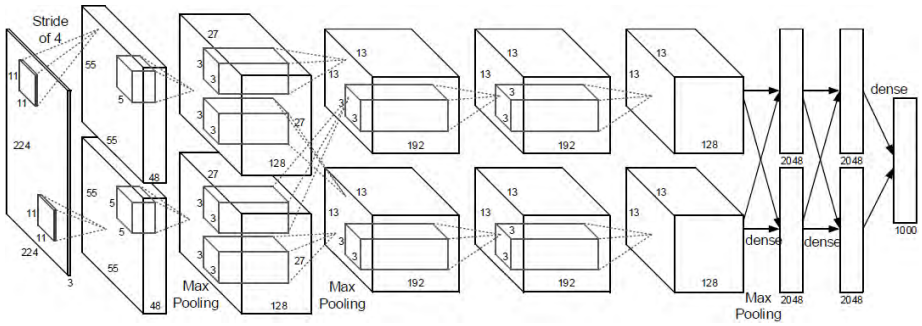

FIGURE 3. AlexNet CNN Architecture [28].

repeated until the image has merged spatially to a small size. This is followed by one or more fully connected layers, also known as inner-product layers, whose neurons have full connections to all activations in the previous layer, hence the name. The last fully connected layer is the classification layer and it holds the output such as the class scores [80].

B. EXAMPLES OF DEEP LEARNING NETWORKS

We list in this subsection some of the well-known deep learning networks.

- **AlexNet (2012)** is a convolutional neural network consisting of 5 convolutional layers, interspersed by 2 normalization layers, as well as 3 fully connected layers [28]. Each convolutional layer performs the activation function using ReLU. In addition, 3 pooling layers are employed with the first, second, and last convolutional layers. The architecture of AlexNet CNN is shown in Fig. 3. AlexNet won the 2012 ImageNet challenge by classifying 224 $\times$ 224 input color images to 1,000 different output classes.
- **VGG (2014)** is a convolutional neural network model similar to AlexNet in terms of the number of fully connected layers. However, it consists of 5 groups of convolutional layers [29], [81]. The exact number of CONV layers in each group depends on the version of the VGG, visual geometry group, model. Table 1 shows the number of CONV and FC layers for the most commonly used VGG models.
- **ResNets (2016)** are deep residual networks with extremely irregular and complex structures compared to AlexNet and VGG CNN models [49], [85], [86]. This is due to having more types of layers, where non-adjacent layers incorporate shortcuts to compute the residual functions, as well as having highly deep structures, that is, between 50 and 1000 CONV layers. Unlike AlexNet and VGG models where the layers are connected in sequence, the interconnections in ResNet layers are in the form of a directed acyclic graph (DAG). ResNet-50 and ResNet-152 are widely used, especially for image classification. ResNet-50/152 structure contains 53/155 CONV (most of them are followed by batch normalization (BatchNorm), scale, and ReLU layers), 1/1 MAX pooling, 1/1 Average pooling, 1/1 FC, and, 16/50 element-wise (Eltwise) layers, respectively.

TABLE 1. CNN layers for VGG models.

Layers	VGG-11	VGG-16	VGG-19
CONV (Group 1)	1	2	2
CONV (Group 2)	1	2	2
CONV (Group 3)	2	3	4
CONV (Group 4)	2	3	4
CONV (Group 5)	2	3	4
CONV (Total)	**8**	**13**	**16**
FC	**3**	**3**	**3**
Total	**11**	**16**	**19**

C. FIELD PROGRAMMABLE GATE ARRAYS (FPGAs)

FPGAs are off-the-shelf programmable devices that provide a flexible platform for implementing custom hardware functionality at a low development cost. They consist mainly of a set of programmable logic cells, called configurable logic blocks (CLBs), a programmable interconnection network, and a set of programmable input and output cells around the device [87]. In addition, they have a rich set of embedded components such as digital signal processing (DSP) blocks which are used to perform arithmetic-intensive operations such as multiply-and-accumulate, block RAMs (BRAMs), look-up tables (LUTs), flip-flops (FFs), clock management unit, high speed I/O links, and others. Fig. 4 shows a basic structure of an FPGA.

FPGAs are widely considered as accelerators for computationally-intensive applications as they enable models with highly flexible fine-grained parallelism and associative operations such as broadcast and collective response [88]. In [89] and [90], FPGA computing models used for applications acceleration are presented, including data streaming, associative computing, highly parallel memory access, use of standard hardware structures such as first in first out (FIFO) buffers, stacks and priority queues, and functional parallelism.

FPGAs have the advantage of maximizing performance per Watt of power consumption, reducing costs for large scale operations [91]. This makes them an excellent choice as accelerators for battery operated devices and in cloud

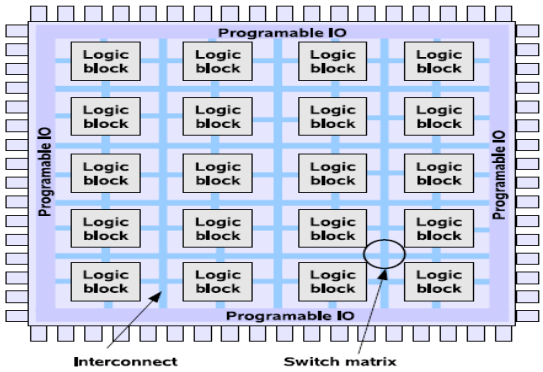

FIGURE 4. FPGA Basic Structure [87].

services on large servers. FPGAs have recently been widely used for deep learning acceleration given the flexibility in implementing architectures with large degree of parallelism resulting in high execution speeds [92].

The adoption of software-level programming models such as the open computing language (OpenCL) standard [93], [94] in FPGA tools made them more attractive to use for deep learning [95], [96]. In addition, the feed-forward nature of deep learning algorithms makes FPGAs offer a clear advantage as they can create customized hardware circuits that are deeply pipelined and inherently multithreaded [91]. FPGAs also have the capability of partial dynamic configuration, which allows part of the FPGA to be reconfigured while the rest is being used. This could be of potential benefit to deep learning methods where the next layer could be reconfigured while the current layer is being used.

D. CHALLENGES OF FPGA-BASED IMPLEMENTATION OF DEEP LEARNING NETWORKS

Implementation of deep learning networks and, in particular, CNNs on FPGAs has a number of challenges including the requirement of a significant amount of storage, external memory bandwidth, and computational resources on the order of billions of operations per second [97]. For example, AlexNet CNN has over 60 million model parameters which need 250MB of memory for storing the weights based on 32-bit floating-point representation as well as requires around 1.5 billion operations for each input image [80]. This large amount of storage required is not supported by existing commercial FPGAs and hence the weights have to be stored on external memory and transferred to the FPGA during computation. Without careful implementation of deep learning networks and maximizing resource sharing, the implementation may not fit on FPGAs due to limited logic resources.

The problem exacerbates with more complex models such as VGG CNN model which have 16 layers. For example, the VGG-16 CNN model has 138 million weights and needs over 30 GOPS [98]. Although the current trends in implementing CNNs is going toward compressing the entire CNN model with dramatically reducing data bit-width [99], it is expected that future CNN models will get more complex with larger number of layers as the amount of training data continues to grow and the problems to be solved get more complex.

In addition, different layers in CNNs have different characteristics which result in different parallelism and memory access requirements. Different layers in a CNN network exhibit vastly different amounts of intra-output and inter-output parallelism [100]. Intra-output parallelism parallelizes the computation of a single output image since it is the sum of n input-kernel convolutions. However, inter-output parallelism is based on computing multiple output FMs in parallel. Furthermore, convolutional layers are computational-centric while fully connected layers are memory centric [98]. For example, the number of operations in each group of convolutional layers in VGG-16 model are on the order of 2 to 9 GOPS while the number of weights are on the order of 0.04 to 7.08 million. However, the number of operations in fully connected layers are in the order of 0.01 to 0.21 GOPS, while the number of weights are on the order of 4.10 to 102.76 million. Thus, the developed CNN accelerator must be designed carefully to meet the varying requirements of different layers and needs to be *flexible* to maximize the performance for each CNN layer.

As technology advances, FPGAs continue to grow in size and capabilities. It is crucial to have some mechanisms for addressing the requirements for efficient implementations of deep learning networks. Addressing hardware resource limitations requires reuse of computational resources, and storing of partial results in internal memories. Data transfer and computational resource usage are significantly impacted by the ordering of operations and selection of parallelism in the implementation of CNNs on FPGAs. Careful scheduling of operations can result in significant reduction in external memory access and internal buffer sizes. External memory bandwidth requirements can be also decreased by using reduced precision for representing the weights with minimal impact on solution quality, which also results in a better energy efficiency. In addition, the number of external memory accesses can be reduced by utilizing on-chip memory and exploiting data reuse. Furthermore, the large number of weights in the fully connected layer can be reduced, based on utilizing singular value decomposition (SVD) [101] with a small impact on accuracy. In the next section, we will review various design approaches used to cope with those challenges for implementing deep learning networks.

III. ACCELERATION OF DEEP LEARNING NETWORKS: CURRENT STATUS

In this section, we will start by covering convolutional neural networks (CNNs) compression techniques as they have a significant impact on the implementation complexity of CNNs. CNNs compression techniques target the minimization of the number of operations and the memory footprint with minimal impact on accuracy. Then, we discuss hardware acceleration techniques for deep learning (DL) algorithms and CNNs based on both application specific integrated circuit (ASIC)

and field programmable gate array (FPGA) implementations. In general, hardware accelerators focus on designing specific modules and architectures that ensure data reuse, enhance data locality, and accelerate convolutional (CONV) layer operations based on performing needed operations in parallel.

A. CNNs COMPRESSION

In this subsection, we review techniques that target the compression of CNNs which results in significantly reducing their implementation complexity with minimal impact on accuracy.

Denton *et al.* [102] proposed a technique to reduce the memory footprint for the network weights in object recognition systems. They used singular value decomposition (SVD) [101] and filter clustering methods for this purpose. The results for convolutional model of 15 layers in [48] show that the proposed technique speeds up the operations in convolutional layers by a factor of 2, compared to CPU Eigen3-based library implementation [103]. In addition, it successfully achieved 13× memory footprint reduction for the fully connected layers while preserving the recognition accuracy within 99%.

In another work, Han *et al.* [104] employed network pruning techniques [105]–[107] to reduce the over-fitting and complexity of neural network models. Their results demonstrated that pruning redundant connections as well as less influential connections achieved 9× and 13× compression for AlexNet and VGG-16 models, respectively, while achieving zero accuracy loss for both.

In a subsequent work, Han *et al.* [108] proposed a deep compression technique for more reduction of the storage requirements of CNNs through the enforcement of weights sharing. Deep compression basically consists of pruning, trained weights quantization, and Huffman coding pipeline stages. The experimental results show that the proposed compression technique successfully reduced the storage requirement of AlexNet and VGG-16 CNN models by 35× and 49×, respectively, without affecting their accuracy. This also improved the power efficiency (a.k.a., performance per Watt) by 3× to 7×.

B. ASIC-BASED ACCELERATORS

In this subsection, we present some recent work in the area of hardware-based accelerators (ASICs).

An ASIC-based hardware accelerator referred to as DianNao [109] was designed for large-scale convolutional neural networks and deep neural networks. DianNao accelerates neural networks by minimizing memory transfers, which opened a new paradigm for hardware accelerators. Since the weights are repeatedly used in the computations of convolution layers, frequent memory access can significantly degrade the overall performance. Therefore, the authors exploited the locality properties of neural network layers to design custom storage structures that take advantages of these properties. In addition, they employed dedicated buffers and tiling techniques to reduce the overall external memory traffic through increasing data locality.

Chen *et al.* [109] also observed that using short fixed-point representation of feature maps (FMs) and weights can also significantly reduce computation resources and memory footprint. They found that the area and power of a 32-bit multiplier can be reduced by a factor of 0.164× and 0.136×, respectively, using 16-bit multipliers. Consequently, DianNao has been implemented using 65nm fabrication technology with 16-bit fixed-point arithmetic units, 6 bits of which are used for the integer part and the remaining 10 for the fractional part. The experimental results demonstrated that DianNao has an average performance of 452 GOPS with power consumption of 485 mW. The results depicted that using 16-bit arithmetic units instead of 32-bit ones introduced only 0.26% accuracy loss on MNIST dataset [110]. On the other hand, the scalability and efficiency of DianNao accelerator are severely limited by the bandwidth constraints of the memory system.

In a related research work, Chen *et al.* [111] and Luo *et al.* [112] proposed DaDianNao multi-chip supercomputer which offers sufficient memory capacity suitable for on-chip storage of all weights in CNNs. This system is mainly important for today's large-scale deployments of sophisticated industry and consumers services. DaDianNao uses 16-bit fixed-point numbers in the inference process like DianNao, but it is implemented using 28nm technology. The results show that DaDianNao outperforms the performance of a single GPU architecture by up to 656.63× and reduces the average energy consumption by 184.05× with only 0.01% accuracy error rate on MNIST dataset for a 64-chip system.

Another member of the DianNao family, called PuDianNao [113], has been designed using TSMC 65nm process to support multiple techniques and scenarios of machine learning (ML). PuDianNao accelerates different ML techniques through extracting their critical locality properties and computational primitives with the use of on-chip storage as well as 7 novel functional units. Experimental results show that PuDianNao is 1.20× and 128.41× faster and energy-efficient, respectively, than NVIDIA K20M GPU architecture. However, both of DaDianNao [111] and PuDianNao architectures have not been optimized to be used for embedded applications.

To improve the scalability and energy efficiency of DianNao design discussed in [109], ShiDianNao accelerator was proposed [114]. ShiDianNao is designed especially for real-time object recognition applications such as self-driving cars, smartphones, and security using 65nm CMOS technology. The proposed accelerator directly connects with a CMOS/CCD sensor in the image processing chip. In addition, all the weights of CNN layers are stored in SRAM on-chip memory, as the target here is small CNN models. ShiDianNao is embedded inside the processing chip to eliminate off-chip DRAM memory accesses and minimize data movements between the SRAM holding the CNN model and the individual processing elements from the sensor. ShiDianNao has a

power consumption of 320.10 mW with a peak performance of 194 GOPS under 1 GHz working frequency. Moreover, ShiDianNao has 1.87× speedup and is 60× more energy-efficient than DianNao [109].

However, DianNao [109], DaDianNao [111], [112], PuDianNao [113], and ShiDianNao [114] are not implemented using FPGA or any other reconfigurable hardware. Therefore, they cannot be efficiently adapted to different application demands (i.e., different neural network sizes). In addition, ASIC designs have a long development cycle and lack flexibility for handling varying DL network designs. Finally, CNN accelerators, which store all weights on-chip such as ShiDianNao [114], will not be able to support realistic large-scale CNN models.

Similar approaches to the DianNao family of techniques are presented in [115] with similar limitations. ISAAC [116] and PRIME [117] have explored in-memory processing to design an acceleration architecture for neural networks. The proposed ISAAC architecture has achieved better improvements of 14.8×, 5.5×, and 7.5× in throughput, energy, and computational density, respectively, than the state-of-the-art DaDianNao architecture.

In CNN models, fine-grained parallelism appears at feature map level, in the neuron level, and in the synapse level. Lu *et al.* [118] reviewed current accelerators that exploit the intrinsic parallelism and observed a mismatch between the parallel types supported by the computing engine and the dominant parallel types that appear in CNN workloads. They identified that most of the previous techniques proposed solutions that fall into one of the three representative architectures: (i) Systolic, (ii) 2D-mapping, and (iii) Tiling.

Due to limitations of dataflow of each of the above three architectures, most existing accelerators support only one specific parallelism. Systolic architectures exploit synapse parallelism (SP), 2D-mapping architectures exploit neuron parallelism (NP), and tiling architectures exploit feature map parallelism (FP). However, in a practical CNN, the dominant parallel type depends on the number of input FMs, the number of output FMs, the size of the output FMs, and the size of the kernel.

With three components (feature map, neuron, synapse) that can be either left serial, or parallelized, we get 2^3 possible combinations. An example of processing style could be *SFSNMS*, meaning, single feature map, single neuron, and multiple synapse.

To address the above problem, and support all possible processing styles, Lu *et al.* [118] proposed a flexible dataflow architecture, called FlexFlow, with minimal controls. FlexFlow supports all types of data paths in each type of parallelism in different layers efficiently.

As a first step, a modification to the processing element (PE) micro-architecture, and the interconnections between PEs, is proposed. PEs are arranged in rows where each row can complete one convolution and serve one output neuron. The adders in each PE row are connected to form the adder tree. Fig. 5 illustrates the proposed PE in FlexFlow and

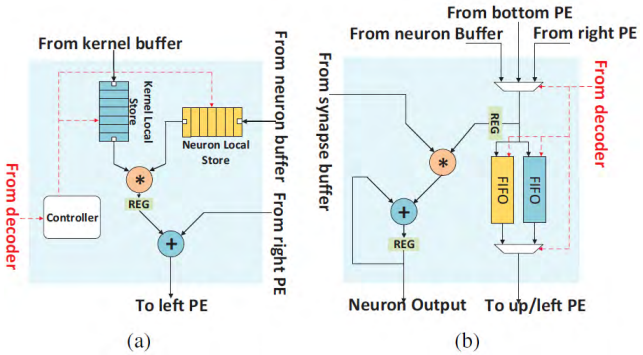

FIGURE 5. Processing Element (PE) Architecture in; (a) FlexFlow, (b) 2D-Mapping [118].

that in 2D-mapping architecture. By eliminating dependency between adjacent PEs, the proposed convolutional unit supports the comprehensive *MFMNMS* parallelisms. To cater to different types of parallelisms, they also proposed a hierarchical dataflow with high data "routability" and low control overhead. The entire dataflow can be divided into three subflows: (i) distribution to local storage in each PE, (ii) fetching of data from local storage for operators (multiplier and adder), and, (iii) dataflow from neuron and kernel buffers to the distribution layer. They also presented a method to determine parallelization type and degree (i.e., the unrolling parameters) for each CONV layer.

FlexFlow architecture was evaluated for computing resource utilization, performance, power, energy, and area. Comparison was made with three typical architectures (i.e., systolic, 2D-mapping, and tiling) using six practical workloads, including AlexNet and VGG. They also examined the scalability of FlexFlow in terms of resource utilization, power, and area with growing scales of computing engine.

From experimental results, it was found that computing resource utilization of each baseline was over 80% across all workloads in contrast to other baselines that utilized less than 60% (most of them less than 40%). In terms of performance, FlexFlow demonstrated over 420 GOPS performance with 1 GHz working frequency. It also outperformed others in terms of data reusability and power efficiency.

C. FPGA-BASED ACCELERATORS

In this subsection, we will review recent techniques employing FPGAs for the acceleration of deep learning networks. For each reviewed technique, we will highlight the key features utilized to maximize performance and throughput in the acceleration process.

FPGA implementations of CNNs appeared in the mid-1990's when Cloutier *et al.* [119] designed the virtual image processor (VIP) on Altera EPF81500 FPGA. VIP is a single-instruction stream multiple-data streams (SIMD) multiprocessor architecture with a 2D torus connection topology of processing elements (PEs). VIP improves the performance through the use of low-accuracy arithmetic to avoid

implementing full-fledged multipliers. Fortunately, recent digital signal processing (DSP)-oriented FPGAs include large numbers of multiply-and-accumulate (MAC) units which allow for extremely fast and low power CNN implementations.

Thereafter, FPGA implementations of deep learning networks have mainly focused on accelerating the computational engine through optimizing CONV layer operations. Several studies in the literature [120]–[126] have reported FPGA-based implementations of convolution operation.

Farabet *et al.* [127] presented an FPGA implementation of CNN that uses one dedicated hardware convolver and a soft-processor for data processing and controlling, respectively. The proposed implementation is referred to as convolutional network processor (CNP). CNP exploits the parallelism of CONV layers to accelerate the computational engine of CNNs while fully utilizing the large number of DSPs, the MAC hardware units on FPGA. The proposed architecture consists of Virtex4 SX35 FPGA platform and external memory. The authors designed a dedicated hardware interface with the external memory to allow 8 simultaneous read/write accesses transparently. In addition, they used first in first out (FIFO) buffers between the FPGA and the external memory chip in both directions to guarantee the steadiness of dataflow.

The vector arithmetic and logic unit in CNP implements 2D CONV, pooling, and non-linear activation function operations of convolutional networks. The implementation of 2D CONV with kernel of size 3 (i.e., $K = 3$) is shown in Fig. 6, where x is the data from input feature map (FM), y is the partial result to be combined with the current result, z is the result to the output FM, W_{ij} is the weight value in the convolution kernel, and W is the width of the input image. It can be seen that the proposed convolutional module accomplishes K^2 MAC operations simultaneously in each clock cycle. CNP represents FMs and weights using 16-bit (Q8.8) fixed-point format. The proposed accelerator has been implemented

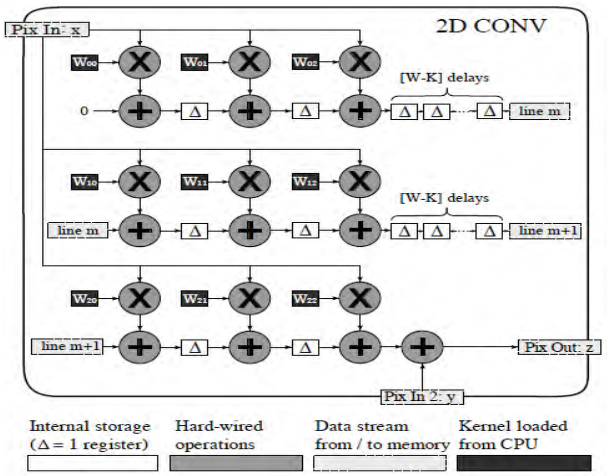

FIGURE 6. 2D Convolution Module of 3 × 3 Kernel [127].

for a face detection system with LeNet-5 architecture [128]. It utilized 90% and 28% of the general logic and multipliers, respectively. In addition, CNP consumed less than 15 Watts of power.

Sankaradas *et al.* [129] proposed a massively parallel coprocessor to accelerate CNNs using Virtex5 LX330T FPGA platform. The proposed accelerator mainly focused on optimizing computation engine by employing the parallelism within convolution kernel and FMs. The coprocessor can be considered as parallel clusters of vector processing elements (VPEs) where each cluster is designed using 2D convolvers, adders, sub-samplers, and look-up tables. Each VPE consists of multiplier-accumulator and programmable register units to hold kernel weights and FM data. To hold the massive intermediate data of CNNs, the authors employed a dedicated off-chip memory (4 DDR2 memory banks) with a large bandwidth on the coprocessor card. Moreover, the proposed accelerator uses a low precision data representation feature with memory packing to further improve the memory bandwidth as well as the throughput. 20-bit and 16-bit fixed-point representations were utilized for kernel weights and FMs, respectively.

The authors examined their architecture on CNN with 4 CONV layers and without any fully connected (FC) layer for a face recognition application. The results show that the proposed coprocessor is 6× faster than a software implementation on a 2.2 GHz AMD Opteron processor with less than 11 Watts of power dissipation. However, the proposed accelerator cannot be used to accelerate full CNNs as it uses few CONV layers without any FC layer. A full CNN model consists of both CONV layers and FC layers. Thus, an efficient CNN accelerator for real-life applications is needed to consider both. Similar approaches to the work of Sankaradas *et al.* [129] are presented in [130] and [131] to accelerate support vector machines (SVM).

MAPLE [132] is a programmable FPGA prototype system presented to accelerate both learning and classification tasks in applications with unstructured large amount of data. The authors analyzed five workload domains to help in designing MAPLE. These workloads are SVM [133], supervised semantic indexing (SSI) [134], K-means [135], generalized learning vector quantization (GLVQ) [136], and CNNs [71]. They found that their computations can be structured as parallel streams of vector or matrix operations. Thus, they architected MAPLE as a 2D grid of vector processing elements as shown in Fig. 7. To efficiently perform matrix multiplication, they allocate a private local storage to each PE which is used to store a column, or part of it, from the multiplier matrix. In this way, matrix multiplication is accomplished by streaming the multiplicand matrix rows through the PEs where each PE performs a MAC operation. The PEs are organized in clusters, where each group is served by a separate memory bank of the banked off-chip memories, which create independent streams for processor-memory computation.

Moreover, MAPLE uses on-chip smart memory blocks to process the large intermediate data on-the-fly using

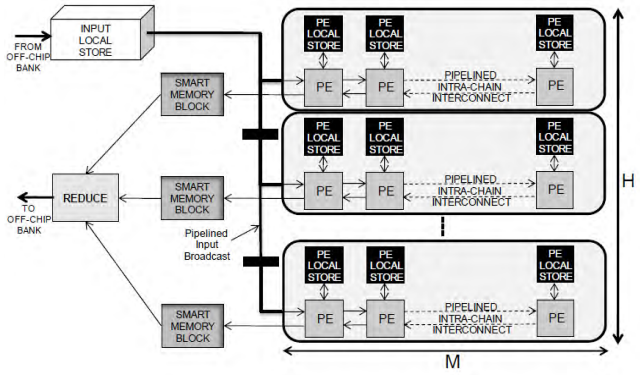

FIGURE 7. MAPLE Processing Core Architecture [132].

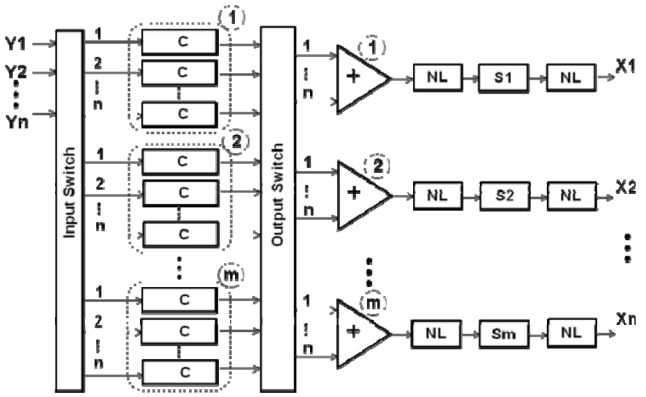

FIGURE 9. The Architecture of DC-CNN [100].

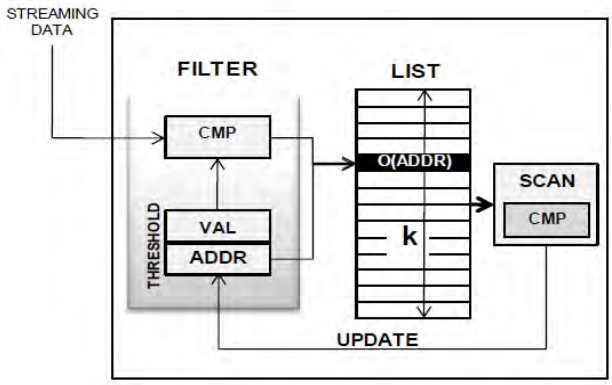

FIGURE 8. MAPLE Smart Memory Block [132].

in-memory processing. Fig. 8 shows the architecture of the smart memory block. To illustrate the idea of on-the-fly in-memory processing, lets consider finding the maximum K elements. The filter compares the input data with the threshold value (VAL). If the input value is greater than VAL, it updates the list by replacing VAL at address ADDR with the input value. Then, the scanner (SCAN) searches for the new minimum value in the list and updates the threshold VAL and ADDR accordingly. It should be mentioned here that the employment of in-memory processing reduced the off-chip memory traffic by 1.64×, 25.7×, and 76× for SSI, K-means, and CNN workloads, respectively. MAPLE prototype has been implemented on Virtex5 SX240T platform running at 125MHz. The experimental results for face and digit recognition CNNs [137]–[139] show that MAPLE is 50% faster than that for 1.3 GHz NVIDIA Tesla C870 GPU implementation.

Chakradhar *et al.* [100] proposed a dynamically configurable CNN architecture on FPGA. The proposed system consists of three main components; a coprocessor, a dynamically configurable CNN (DC-CNN) processing core, and 3-bank memory subsystem. The coprocessor is designed such that it automatically configures the software and the hardware elements to fully exploit the parallelism at the workload level. DC-CNN is responsible for executing CNN applications and its architecture is shown in Fig. 9. It consists of m computational elements (each with n 2D convolvers as well as sub-sampling (S) and non-linearity (NL) pipelined units), m adders (each with n inputs), and input/output switches. The internal structure of the switches vector encloses $m \times n$ selectors which are used to help in exploring the entire design space and to provide the configurability function across different layers of CNN model. To determine the best (m, n) feasible combination for each layer, the system analyzes the workload using integer factorization techniques because it is considered fast for small numbers [140], [141]. Dynamic programming is also used to quickly prune infeasible combinations.

The authors compared the proposed DC-CNN architecture, considering 20 2D convolvers as well as a memory subsystem of 128-bit port width, with a 1.35 GHz NVIDIA's GPU implementation. The results show that DC-CNN achieved 4.0×, 4.4×, 5.4×, 6.0×, and 6.5× speedup for face recognition [137], face detection [139], mobile robot vision [127], video surveillance [100], and automotive safety [100] workloads, respectively. It is worth mentioning that DC-CNN is the first architecture that achieves a performance suitable for real-time processing for video streaming as it processes up to 30 frames per second. In addition, DC-CNN is more energy-efficient than the GPU implementation as it consumes 14 Watts, while more than 150 Watts are consumed by the GPU. On the other hand, the authors modeled their architecture on a CNN with 3 CONV layers only without any FC layer which makes it unsuitable for today's other real-life applications.

A second-generation of CNP [127] architecture has been proposed in [142] by designing a stream processor system. The proposed design replaces the dedicated hardware convolver in CNP with multiple parallel vector processing units, named as ALUs, laid out in a 2D grid. Each ALU is composed of four local routers, one global router, and a streaming operator. The local routers are used to stream data to/from the neighbors. Streaming data to and from global data line is done through the global router. The streaming operators in the ALU are fully pipelined to produce a result per clock

cycle as described in [127] with the use of Q8.8 coding to represent FMs and weights. The proposed system also uses a multi-port direct memory access (DMA) streaming engine to allow individual streams of data to operate seamlessly within processing blocks. The results show that the proposed stream processor system can run small CNNs at up to 30 fps while consuming about 15 Watts.

An improved version of CNP architectures given in [127] and [142] was presented in [143] and referred to as neuFlow. Particularly, neuFlow has replaced the 2D grid of ALUs with a 2D grid of processing tiles (PTs). The proposed architecture contains a 2D grid of PTs, a control unit, and a smart DMA module, as shown in Fig. 10. Each PT consists of local operators and a routing multiplexer (MUX). The top three PTs have been implemented to perform MAC operation. Thus, they can be used to perform 2D convolution, simple dot-products, and spatial pooling. General-purpose operations, such as dividing and squaring, have been implemented at the middle three PTs. Therefore, the middle row of neuFlow can be used for normalization. Finally, neuFlow's bottom PTs row implements non-linear operations. Moreover, each operator employed input and output FIFOs to stall its pipeline when required. On the other hand, PT's MUX is used to connect its local operators with the neighboring PT's streaming operators and off-chip memory instead of the used local routers and global router discussed in [142].

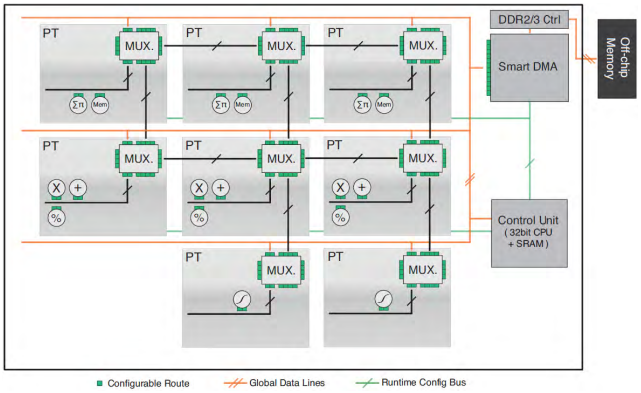

FIGURE 10. The Architecture of neuFlow [143].

NeuFlow uses a dataflow compiler, named luaFlow, to translate a high-level flow-graph representation of CNNs in Torch5 [144] into HDL scripts with different levels of parallelism. In addition, luaFlow produces a binary code configuration file and holds it in the embedded control unit. Thereafter, the control unit configures the 2D grid of PTs (connections and streaming operator) and the DMA ports through run-time configuration buses. A smart memory module has been designed to support multiple asynchronous accesses of off-chip memory through its reconfigurable ports. By targeting the larger Xilinx Virtex6 VLX240T FPGA, neuFlow achieved 147 GOPS at 10 Watts for street scene parsing CNN in [145] with the use of 16 bits to represent FMs and weights.

Peemen *et al.* [146] utilized the flexible off-chip memory hierarchy method to design a configurable memory-centric accelerator template for a variety of models of CNNs. This accelerator exploits data reuse in complex access patterns to reduce off-chip memory communication, which minimizes the bandwidth requirements. The memory-centric accelerator maximizes the efficiency of on-chip memories for better data locality using loop transformation (to optimize the tiling parameters) and block RAM (BRAM)-based multi-bank on-chip buffers [147]. At the same time, it minimizes the size of FPGA on-chip memories to optimize energy and area usage, which are key requirements for embedded platforms.

The memory-centric accelerator uses a SIMD cluster of MAC PEs with flexible reuse buffers to accelerate the CONV layer. The acceleration template has been implemented on Virtex6 FPGAs. In addition, a MicroBlaze processor has been utilized to configure and communicate with the accelerator via FIFO-based fast simplex link (FSL). The proposed accelerator has been analyzed for a CNN vision task of size 2.74 GMAC and the results show that the memory-centric accelerator is 11× faster than the standard implementation of similar FPGA resources.

Neural network next (nn-X) [148] is a real-time system-on-chip (SoC) computing system for deep learning networks on mobile devices. The architecture of nn-X consists of a host processor, a co-processor, and external memory. The co-processor accelerates the learning networks by parallelizing their operations throughout arrays of configurable processing elements referred to as *collections*. Each collection contains one convolution engine, one pooling module, and one non-linear operator. The CONV engine accelerates the CONV operation by fully pipelining the incoming data with the use of cache memories. The collections are able to communicate with one another using the collection route component to achieve cascaded pipelining, which results in reducing accesses to external memory. The data transfer between the collections and the external memory is accomplished throughout the co-processor full-duplex memory router, which provides independent data streams. The nn-X has been prototyped on Xilinx ZC706 which contains Zynq XC7Z045, two ARM Cortex-A9 processors, and 1 GB DDR3. Eight collections have been employed to achieve large parallelism. The results for face recognition model in [149] show that nn-X is 115× faster than the two embedded ARM processors.

Zhang *et al.* [55] proposed a roofline-based model to accelerate convolutional neural networks on FPGAs. The roofline model is an intuitive visual performance model used to relate the attainable performance to the peak performance that can be provided by the hardware platform and the off-chip memory traffic [150]. The focus in their work is primarily on accelerating the convolutional layers as it consumes more than 90% of the computational time during the prediction process [77]. In doing so, the authors optimized both the computation operations and the memory access operations in convolutional layers. They considered a CNN applica-

tion composed of five convolutional layers that won the ImageNet competition in 2012 [28]. The proposed accelerator uses polyhedral-based data dependence analysis [151] to fully utilize all FPGA computational resources through loop unrolling, loop pipelining, and loop tile size enumeration. Note that loop unrolling maximizes the parallel computation of CONV MAC operations. On the other hand, local memory promotion and loop transformation are used to reduce redundant communication operations and to maximize the data sharing/reuse, respectively.

Subsequently, the roofline performance model is used to identify the optimal design from all possible solutions in the design space. Specifically, the authors model all possible legal designs delivered from the polyhedral analysis in roofline to find the optimal unrolling factor $\langle T_m, T_n \rangle$ for every convolutional layer, where T_m and T_n are the tile size for the output FMs and input FMs, respectively. However, designing a CNN accelerator with different unrolling factors to each convolutional layer is challenging. Therefore, the proposed architecture enumerates all possible valid designs to find uniform cross-layer unrolling factors. Thereafter, the hardware accelerator is implemented based on the cross-layer optimal unrolling factors.

The proposed accelerator composed of a computational engine and memory sub-system is depicted in Fig. 11. The computation engine is designed as T_m duplicated tree-shaped poly structures with T_n inputs from the input FMs, T_n inputs from the weights, and one input from the bias. On the other hand, the memory sub-system is implemented as four sets of on-chip buffers; two sets to store the input FMs and weights, each with T_n buffer banks, and two buffer sets of T_m independent banks for storing the output FMs. To overlap data transfer with computation, on-chip buffers are operated in a ping-pong manner. In addition, two independent channels are implemented for load and off-load operations to increase the bandwidth utilization. Moreover, MicroBlaze processor is used to send configuration parameters and commands for the accelerator over AXI4lite bus. The CNN accelerator communicates with external data transfer engines through FIFO interfaces, where the data transfer engines are used to access DDR3 DRAM memory through AXI4 bus.

The accelerator is designed using Vivado 2013.4 high level synthesis tool and implemented on Xilinx VC707 FPGA board clocked at 100 MHz. The experimental results depict that the proposed implementation achieves a peak performance of 61.62 GFLOPS as well as a 17.42× speedup over the software implementation on Intel Xeon CPU E5-2430 at 2.20 GHz with 15 MB cache and 16 threads. In addition to this, the results show that the proposed FPGA architecture is 24.6× more energy-efficient than the software implementation as the total power consumption is only 18.6 Watts. The proposed implementation has some limitations such as designing the accelerator with new cross-layer unrolling factors for different architectures of CNNs. Furthermore, using the CNN accelerator with uniform unrolling factors might be sub-optimal for some CONV layers, which affects the overall performance.

In 2014, Microsoft research team of Catapult project integrated FPGA boards into data center applications to successfully achieve 2× speedup for Bing Ranking (the large-scale search engine) [67]. A year later, Ovtcharov *et al.* [152] at Microsoft Research utilized Catapult hardware infrastructure, a dual-socket Xeon server equipped with Stratix-V GSMD5 FPGA, to design a specialized hardware for accelerating the forward propagation of deep CNNs in a power-constrained data center.

The top-level architecture of the proposed CNN accelerator is shown in Fig. 12. Multi-banked input buffer and kernel weight buffer are used to provide an efficient buffering scheme of FMs and weights, respectively. To minimize the off-chip memory traffic, a specialized network on-chip has been designed to re-distribute the output FMs on the multi-banked input buffer instead of transferring them to the external memory. The 3D convolution operations (such as the dot-product) and other CNN operations are

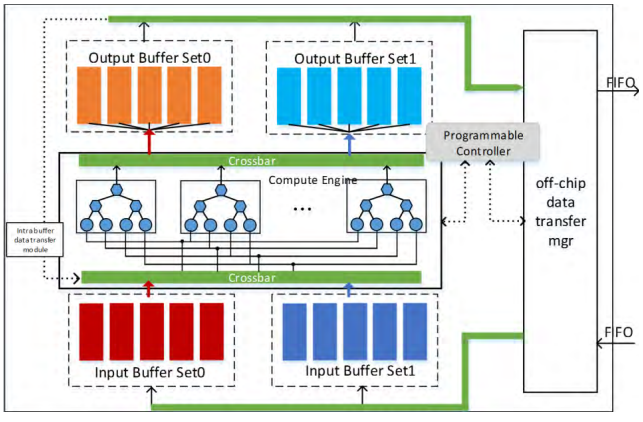

FIGURE 11. Zhang *et al.* [55] Accelerator Architecture.

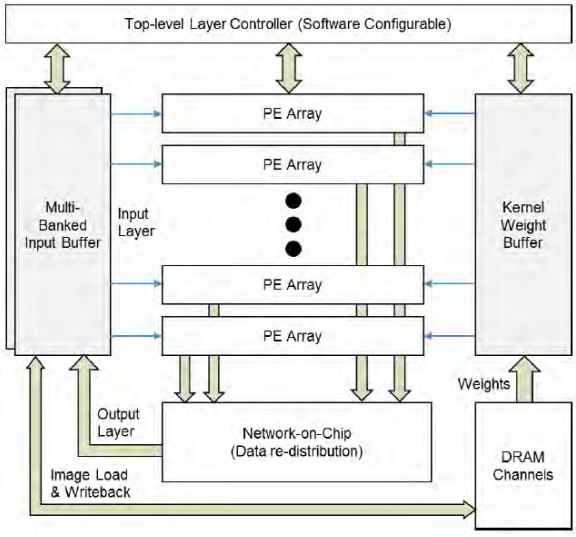

FIGURE 12. Top-Level Archeticture of Microsoft CNN Accelerator [152].

independently performed using spatially distributed scalable vectors of PEs. The controller engine is responsible for streaming and data delivery of multi-banked input buffer and kernel weight buffer data to each of the PE vectors. In addition, it supports configuring multiple CNN layers at run-time. The results show that the proposed design is able to classify 134 images/sec, while consuming about 25 Watts, for AlexNet model on ImageNet-1K dataset [28], which is 3× better than the published throughput results for the Roofline-based FPGA Accelerator [55]. The authors mentioned that using top-of-the-line FPGAs such as Arria 10 GX 1150 improves the throughput to around 233 images/sec.

Qiu *et al.* [98] proposed an FPGA design to accelerate CNNs for a large-scale image classification challenge on embedded systems. The focus was on accelerating both CONV and FC layers, since they are considered as the most computational-centric and the most memory-centric operations in CNNs, respectively. The proposed accelerator reduces the resource consumption using specific design of convolver hardware module. In addition, the authors applied singular value decomposition (SVD) to the weight matrix of FC layer in order to reduce memory footprint at this layer [101]. To further reduce memory footprint and bandwidth requirement of CNN, they proposed a dynamic-precision data quantization flow component. This component is responsible for finding the optimal fractional length for weights in each layer as well as the optimal fractional length for FMs in adjacent layers, while achieving minimal accuracy loss. Then, it converts the floating-point numbers representing weights and FMs into fixed-point numbers.

In addition, the authors proposed a data arrangement scheme that maximizes the burst length of each transaction to the external memory to accelerate CONV and FC layers, as well as to avoid unnecessary access latency. Note that maximizing the DRAM burst length raises up the effective DRAM bandwidth [55], [153].

The proposed architecture consists of a processing system (CPU) and programmable logic (FPGA). CNN computations are performed through special design of processing element modules in FPGA. The main modules in the processing element are convolver complex, max-pooling, non-linearity, data shift, bias shift, and adder tree, as shown in Fig. 13. The convolver complex is designed as a classical line buffer [154],

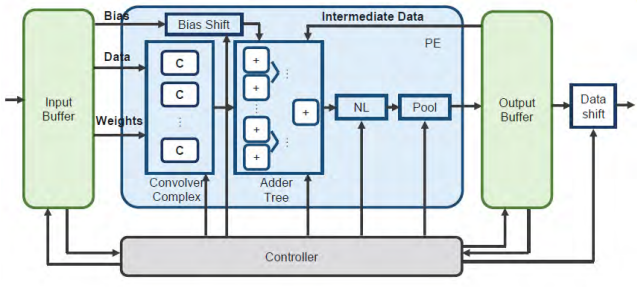

FIGURE 13. Processing Element Module of Qiu *et al.* [98] Embedded Accelerator Architecture.

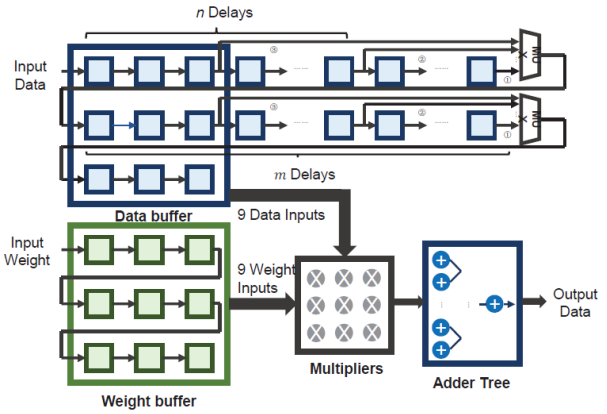

FIGURE 14. Convolver Complex Design of Qiu *et al.* [98] Embedded Accelerator Architecture.

as shown in Fig. 14, to achieve convolution operations as well as to compute FC layer multiplication of matrix-vector. The pooling layer implemented in the max-pooling module is used to output the maximum value in the input data stream with a window of size 2. The activation function of CNN is applied to the input data stream using the non-linearity module. The adder tree accumulates the partial sums generated from the convolvers. Finally, data shift and bias shift modules are responsible for accomplishing dynamic quantization.

The proposed embedded FPGA platform has been implemented using VGG-16-SVD network with 16-bit fixed-point numbers on Zynq XC7Z045 platform. The results demonstrate that applying SVD technique reduces memory footprint of FC layer by 85.8% with a compression rate of 7.04% while introducing an accuracy loss of only 0.04%. Finally, the overall performance of the proposed accelerator reported is 136.97 GOPS under 150 MHz working frequency with the top-5 accuracy of 86.66% and a total power consumption of 9.63 Watts.

DeepBurning [155] is an FPGA-based neural network (NN) design automation tool. It allows for building learning accelerators for specific NN with optimized performance and custom design parameters configuration using a pre-constructed register transfer level (RTL) module library. The RTL library holds the hardware descriptive scripts for NN reconfigurable components as well as their configuration scripts. In addition, it contains other RTL building blocks for logical and arithmetic operations such as the connection box (used to exchange data between NN layers as well as to approximate the division operation) and approximate look-up table (LUT) (used to simplify a function or operation to allow it to be mapped into hardware).

In order to design an optimized hardware, DeepBurning compresses the passed NN model to the greatest extent using temporal and spatial folding which helps also in satisfying the resource constraints and minimizing the required hardware modules. DeepBurning not only generates the hardware description for neural network scripts, but also analyzes the complex access pattern and data locality using an integrated

compiler to generate a run-time control flow which provides energy-efficient, and, better data reuse implementation. In addition, the DeepBurning compiler investigates the accelerator on-chip memory size and throughput to properly tile and partition the NN weights and feature data layouts. Moreover, DeepBurning uses the address flow component to automatically fetch and store off-chip memory and on-chip memory data. The authors compared the performance of DeepBurning with that in [55], considering AlexNet CNN model, as they both operate at 100 MHz. They considered a high budget resources constrained DeepBurning on Zynq-7045 device. The results show that DeepBurning is 1.13× slower but 1.45× more energy-efficient.

An OpenCL-based optimization framework to accelerate large-scale convolutional neural network models was proposed by Suda *et al.* [80]. They found that the number of performed CONV MAC operations in parallel (N_{CONV}), SIMD vectorization factor (S_{CONV}), normalization layer loop unrolling factor (N_{NORM}), the number of parallel pooling outputs in one cycle (N_{POOL}), and the number of parallel FC MAC operations (N_{FC}) are the key variables that determine the parallelism of the design. Subsequently, they analytically and empirically modeled the execution time for each layer as a function of the above mentioned variables. Then, genetic algorithm was used to explore the design space for finding the optimal combination of the key design variables considering the resources constraints.

The authors implemented the scalable CONV block in a similar fashion to that in [138] as a matrix multiplication by flattening and on-the-fly rearrangement of the feature data. The OpenCL software has been utilized in their work due to its parallel programming model as well as its ability to integrate the compiled RTL design with external memory interfacing IPs [156], which uses memory coalescing technique with complex load and store units. In addition, it has optimized matrix multiplication and CPU-FPGA communication libraries [157], [158].

The framework is used on both VGG-16 and AlexNet CNN models which are implemented on P395-D8 [159] and DE5-Net [160] FPGA boards with fixed-point operations according to their precision study. They compared the proposed implementation with 3.3 GHz core i5-4590 CPU implementation that uses Caffe tool [58] with ATLAS [161] optimized library for matrix/vector operations. The results show that the OpenCL optimized framework on P395-D8 achieved 5.5× (117.8 GOPS) and 9.5× (72.4 GOPS) speedups for VGG-16 and AlexNet models, respectively. On the other hand, DE5-Net FPGA achieved less throughput speedup than the P395-D8 (2.2× (47.5 GOPS) for VGG-16, and 4.2× (31.8 GOPS) for AlexNet) as it has 7.67× less DSPs than what is available on P395-D8.

Zhang *et al.* [153], [162] analyzed the transformation of CONV and FC layers to regular matrix multiplication presented in prior work [98]. For VGG-16 model, they found that such transformation necessitates up to 25× duplication of input FMs. To address this problem and improve the bandwidth utilization, they designed a uniformed matrix multiplication kernel that uses either input-major mapping (IMM) or weight-major mapping (WMM) techniques while computing FC layer. In IMM, the designed kernel batches a group of different input FMs together, and then performs the matrix multiplication. IMM technique improves the data reuse of FC weights. On the other hand, the designed kernel with WMM technique makes use of the fact that the FC layer is communication-bound in which the weight matrix is much larger than the input FM matrix. In particular, it loads input FM matrix to a weight buffer and loads weight matrix to input FM buffer. Subsequently, a regular matrix multiplication is performed on these matrices. As a result, WMM may allow for a higher data reuse than IMM, especially for input FMs that can be reused multiple times considering the limited hardware resources.

For the above, the roofline model was applied to identify the optimal mapping technique under different batch sizes and data precisions. The results demonstrate that WMM is better than IMM in term of data reuse and bandwidth utilization, especially in small batch sizes which is required for real-time inference. Hence, the same matrix multiplication kernel is utilized for the computation of both CONV and FC layers, but with the use of IMM in CONV layer and WMM in FC layer. Based on this, the authors proposed a software/hardware co-design library, which they named Caffeine, to accelerate CNNs on FPGAs.

With an easy-to-use developed tool, Caffeine aids in automatically choosing the best hardware parameters, using the model files from Caffe and FPGA device specifications obtained from the user. Caffeine FPGA engine uses a high-level synthesis (HLS)-based systolic-like architecture to implement matrix multiplication kernel. It allows changing parameters such as number of PEs, precision, and FM size. Caffeine further maximizes the FPGA computing capability by optimizing multi-level data parallelism discussed in [55] and pipeline parallelism using polyhedral-based optimization framework given in [163]. Caffeine framework also handles the weights and biases reorganization in off-chip DRAM to maximize the underlying memory bandwidth utilization. In addition, the double-buffering technique is employed to prefetch the next data tile for each PE. Caffeine has been evaluated by implementing AlexNet and VGG-16 CNNs on Ultrascale KU060 (20nm and 200 MHz) and on Virtex7 690T (28nm and 150 MHz) considering different precisions. The VGG-16 implementation with 16-bit fixed-point on Ultrascale KU060 and Virtex7 690T provided 43.5× and 65× overall throughput enhancement, respectively, compared to implementation on a two-socket server, each with a 6-core Intel CPU (E5-2609 at 1.9 GHz).

A special case of dataflow, referred to as synchronous dataflow (SDF) [164], is a paradigm of computation that allows for representing a computing system as a streaming problem. In this way, SDF model can represent the hardware implementation of CNNs using linear algebra and directed SDF graph (SDFG). Each node of SDFG represents a

hardware building block that can immediately start its computation as soon as the data are available through its input arcs. Such representation of CNN model offers a fast design space exploration. Venieris and Bouganis [165] employed SDF model to optimize the mapping of CNNs onto FPGAs based on HLS.

In particular, the proposed fpgaConvNet framework in [165] takes as input a high-level script programmed by DL expert describing the CNN model, along with specifications of the targeted FPGA platform. Thereafter, it parses the input script through a developed domain-specific language (DSL) processor to model the CNN in the form of a directed acyclic graph (DAG) where each node corresponds to a CNN layer. Then, the DAG-based CNN is transformed into an SDFG representation and modeled as a topology matrix. The topology matrix contains the number of incoming parallel streams, the width of each data stream, and the production or consumption rates at each node. In addition, the DSL processor extracts information about the platform-specific resource constraints.

Unlike other attempts, instead of exploring the design space for the optimal parameters of loop unrolling and tiling, fpgaConvNet explores the design space of the topology matrix components while considering the resource constraints. In doing so, fpgaConvNet performs graph partitioning, coarse-grained folding, and fine-grained folding. The graph partitioning splits the original SDFG into subgraphs and each subgraph is then mapped to a distinct bitstream as shown in Fig. 15. Note that the proposed multi-bitstream architecture might have multiple CONV layer processors (CLPs), as in the provided example. This away, on-chip RAM is used for intermediate results and data reuse within the subgraph, while accesss of off-chip memory is minimized and limited for input and output streams of the subgraph. However, this scheme adds reconfiguration penalty due to the need for reconfiguring the FPGA when the data flows between adjacent subgraphs. To amortize this overhead, several input data streams are processed in a pipelined manner.

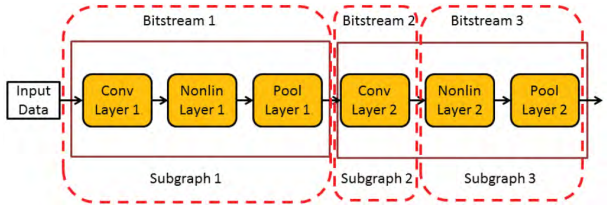

FIGURE 15. SDF Graph Partitioning [165].

Thereafter, each bitstream architecture is optimized using coarse-grained folding and fine-grained folding. In coarse-grain folding, CONV, pooling, non-linear, and other major operations of each layer are unrolled to provide the highest possible throughput by having several parallel units of each operation. The fine-grain folding controls the unrolling and pipelining of the dot-product operations inside CONV and average pooling units. Instead of fully unrolling the implementation of dot-product which produces a 1 dot-product per cycle, with the use of a high number of multipliers and adders, fpgaConvNet uses a smaller number of MAC units and schedules the execution of different operations using time-multiplexing. A trade-off between the performance and the required hardware resources can be achieved by changing the unroll factor and the degree of multiplexing. Therefore, fpgaConvNet employed simulated annealing [166] to find the optimal partitioning points and folding factors. Finally, fpgaConvNet uses optimal components to derive the configuration of PEs and buffers, and generates a synthesizable Vivado HLS hardware design.

fpgaConvNet framework has been evaluated by mapping LeNet-5 and scene labeling [167] small CNN models with Q8.8 fixed-point representation onto a Zynq-7000 XC7Z020 FPGA platform working at 100 MHz. In mapping LeNet-5, fpgaConvNet achieves up to 1.62× the performance density of CNP [127]. Compared to Tegra K1 GPU implementation of scene labeling CNN, fpgaConvNet surpasses Tegra K1's power efficiency by 1.05×.

Ma *et al.* [78] proposed a Python-based modularized RTL compiler to accelerate CNNs by employing loop unrolling optimization [55], [79] for CONV layer operations. A detailed review article of this work has been recently published and referred to as ALAMO [168]. The proposed compiler integrates both the RTL finer level optimization and the flexibility of HLS to generate efficient Verilog parameterized RTL scripts for ASIC or FPGA platform under the available number of parallel computing resources (i.e., the number of multipliers (N_m)). If N_m is greater than the number of input FMs (N_{if}), the proposed compiler fully unrolls *Loop-3* (N_{if}, refer to subsection II-A.1 for more details) while it partially unrolls *Loop-4* (N_{of}) to exploit the data reuse of shared features among N_m/N_{if} output FMs. Otherwise, it partially unrolls *Loop-3* which results in N_{if}/N_m repeated sliding of kernel window. On the other hand, *Loop-2* ($X \times Y$) is serially computed after *Loop-1* (K) to minimize the number of partial sums.

The overall modules of the proposed CNN accelerator are shown in Fig. 16. The controller is responsible for directing and ensuring in-order computation of CNN modules

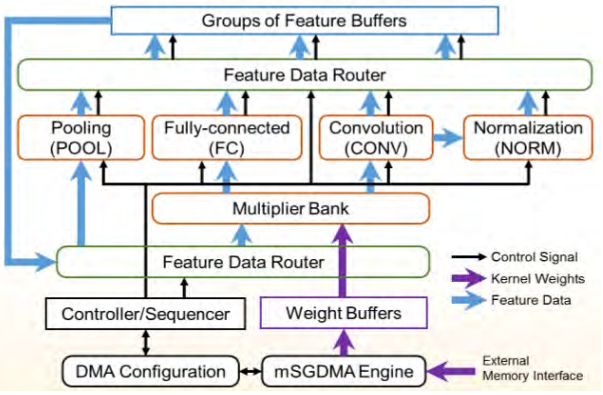

FIGURE 16. ALAMO Overall Acceleration Modules [78].

for each layer. The data routers oversee the selection of data read and data write of two adjacent modules as well as the assignment of buffer outputs to shared or pool multipliers of the multiplier bank. The feature buffers hold the FMs using on-chip RAMs. The weight buffers are used to ensure the availability of CONV and FC layers' weights before their computation as well as to overlap the transfer of FC layer weights with its computation. The CONV module consists of control logic, groups of adder trees, and ReLU components. The control logic component parameterizes the loop unrolling factors based on the configuration of each layer (N_{if}, N_{of}, X, Y, and K). The CONV module contains N_m/N_{if} adders to sum N_{if} parallel multiplier results and accumulate them. Moreover, the adder trees can be shared by layers with identical N_{if} to be as one single module. The ReLU component checks the input pixel sign bit to either output zero or the data pixel itself. The POOL module contains accumulators or comparators to perform average or maximum operation, respectively. The NORM module maintains the required components to perform the operations of local response normalization such as square, non-linear (using look-up table), and multiplication operations. Finally, the FC module shares the multiplier bank module with the CONV module to perform the matrix-vector multiplication (MVM).

ALAMO architecture permits the output pixels to be only stored in the feature buffers, which makes ALAMO suitable for CNNs with only small intermediate data volumes. The proposed RTL compiler has been tested by accelerating two CNN models; AlexNet and NiN [169]. The generated parameterized RTL scripts for AlexNet and NiN are synthesized using Altera Quartus synthesis tool and implemented on DE5-Net FPGA board. The experimental results for AlexNet model are compared with the results for OpenCL-based design [80] as both use the same FPGA board with similar hardware resources for AlexNet. ALAMO achieved 1.9× and 1.3× improvement for throughput and power consumption, respectively. Moreover, the overall throughput of NiN model is 1.03× better than that of AlexNet. This is because NiN has more CONV layers and many of them have the same N_{if}.

Liu *et al.* [170] proposed a parallel framework for FPGA-based CNN accelerators that exploits four levels of parallelism; task level, layer level, loop level, and operator level. Task-level parallelism involves executing multiple image prediction tasks simultaneously. Layer-level parallelism exploits pipelining across layers to enable parallel execution of all layers with different images. Loop-level parallelism utilizes loop unrolling in performing convolutions and this can be achieved either through intra-output or inter-output parallelism. Finally, operator-level parallelism is achieved by parallelizing the $k \times k$ MACs operations needed for convolution operation in convolutional layers or the n MACs needed for inner-product computation in fully connected layers. Fig. 17 shows the parallel framework exploiting these four levels of parallelism.

The authors have used 16-bit fixed-point format for representing pixels in input feature maps and output feature maps. However, they have used 32 bits for intermediate results which get truncated to 16 bits. In addition, they have used 8 bits for representing kernels and weights. They have presented a systematic methodology for design space exploration to find the optimal solution that maximizes the throughput of an FPGA-based accelerator under given FPGA constraints such as on-chip memory, computational resources, external memory bandwidth, and clock frequency.

The proposed technique has been evaluated by implementing three CNN accelerators on the VC709 board for LeNet, AlexNet, and VGG-S. It has achieved a throughput of 424.7 GOPS, 445.6 GOPS, and 473.4 GOPS for LeNet, AlexNet, and VGG-S accelerators, respectively. In addition, the performance has been compared with MatConvNet tool running the CNN models on Intel Core i7-4790K CPU (4.0 GHz) and NVIDIA GTX-770 GPU (1,536 CUDA cores, 2 GB GDDR5, 224.3 GB/s memory bandwidth). Compared to the CPU implementations, the accelerators for LeNet, AlexNet, and VGG-S achieved 14.84×, 6.96×, and 4.79× in performance, respectively, and 51.84×, 24.69×, and 16.46× in power efficiency, respectively. Compared to the GPU implementations, the accelerators achieved better performance in the small-scale network LeNet (3.17×), comparable

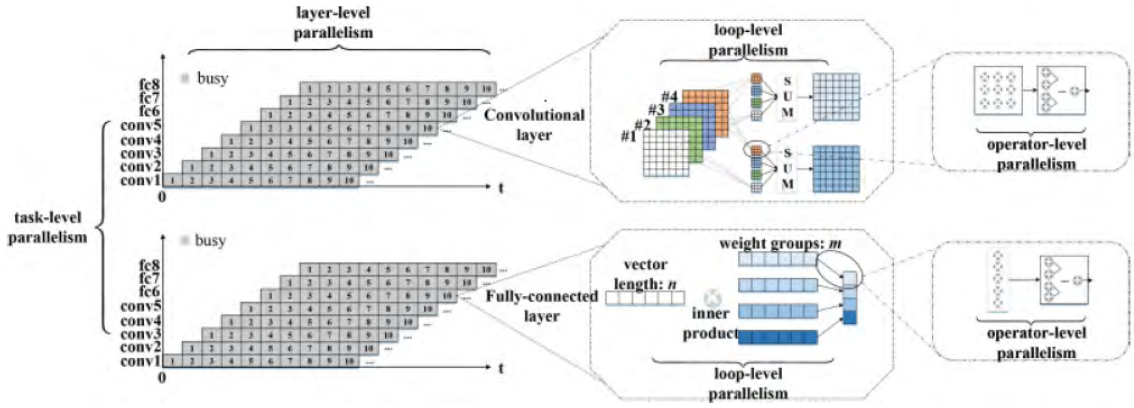

FIGURE 17. Parallel Framework Exploiting Four Levels of Parallelism [170].

performance in the medium-scale network AlexNet (0.96×), and worse performance in the large-scale network VGG-S (0.56×). However, the accelerators achieved higher power efficiency than the GPU implementations in all three networks with 28.3× for LeNet, 8.7× for AlexNet and 4.98× for VGG-S.

FP-DNN [171] is an end-to-end framework that automatically generates optimized FPGA-based implementations of deep neural networks (DNNs) using an RTL-HLS hybrid library. FP-DNN compiler, programed using C++ and OpenCL, takes TensorFlow symbolic descriptions [172] of DNNs, and then performs model inference through the use of model mapper, software generator, and hardware generator modules. The model mapper extracts the topological structure and layers configurations of DNN model from the TensorFlow descriptions and generates an execution graph for the target model. The execution graph shows layer-by-layer operations and read/write data transactions.

FP-DNN compiler allocates off-chip DRAM data buffers to store intermediate data, weights, and model parameters and configurations. The model mapper maximizes the storage resource reuse through minimizing the number of required physical buffers. Specifically, it formulates the data reuse problem as a graph coloring problem [173], and then the left-edge algorithm is applied to generate kernel configuration and kernel schedule. Subsequently, the software generator uses the kernel schedule to generate a host C++ program which initializes the model, manages the data buffers, and schedules the kernel execution. On the other hand, the hardware generator uses the kernel configuration and the execution graph to generate the FPGA hardware codes by instantiating the corresponding optimized templates from an expandable RTL-HLS hybrid library. Each template is comprised of Verilog-based computational engine and OpenCL-based control logics engine.

The architecture of the proposed FPGA-based accelerator consists of matrix multiplication and data arranger modules. Matrix multiplication module is a hand-written Verilog code that is designed and optimized based on the hardware constraints of Altera Stratix-V GSMD5 FPGA. It applies tiling and ping-pong double buffers techniques to improve the throughput. On the other hand, data arranger is an OpenCL-based module that is responsible for mapping the computational part of a layer to matrix multiplication as well as performing data communication with off-chip memory and matrix multiplication module. Mapping DNNs computational operations to matrix multiplication has been widely applied in prior studies [80], [132], [174]. FP-DNN maps FC layer to matrix multiplication by batching input vectors together. Before model deployment, FMs and weights are rearranged in DRAM using the channel-major scheme to optimize the communication between the accelerator and off-chip DRAM. On the other hand, both floating-point and fixed-point representations have been supported for implementation, and they can be adjusted by the user.

The proposed RTL-HLS hybrid framework has been evaluated by accelerating VGG-19, LSTM-LM [175], ResNet-152 DNNs on Stratix-V GSMD5 FPGA. Note that this is the first work that implements ResNet-152 on FPGA. The experimental results demonstrated that the speedup of FP-DNN for 16-bit fixed-point implementations are about 1.9× - 3.06× compared with the server that includes 2 processors each with 8-core Intel Xeon E5-2650v2 at 2.6 GHz.

In line with the current trends towards compressed neural networks, with dramatically reduced weights and activations bit-width using 1-bit or 2-bit quantization [176]–[180], Umuroglu *et al.* [181] conducted a set of experiments to estimate the trade-off between the network size and precision using the roofline model. They found that binarized neural networks (BNNs) [180] require 2 to 11 times more operations and parameters than an 8-bit fixed-point CNN to achieve a comparable accuracy on MNIST [71] dataset. However, the performance of BNN is found to be 16× faster than the fixed-point network.

Subsequently, the authors proposed a framework, referred to as FINN [181], that maps a trained BNN onto FPGA. FINN generates a synthesizable C++ network description of a flexible heterogeneous streaming architecture. The architecture consists of pipelined compute engines that communicate via on-chip data streams. Each BNN layer has been implemented using dedicated compute engines with 1-bit values for weights and FMs; $+1$ and -1 are used to represent a *set* bit and *unset* bit, respectively.

The authors have optimized accumulation, batch normalization (batchnorm), activation, and pooling operations of BNNs. In particular, the accumulation of a binary dot-product has been implemented as a counter of *set* bits (*popcount* operation). The popcount-accumulate reduces the number of required look-up tables (LUTs) and flip-flops (FFs) by a half, compared to the implementation of signed-accumulation. BNN batchnorm and activation operations have been simplified and implemented together as unsigned comparison with a threshold τ_k, $+1$ is produced when the input value is greater than or equals to τ_k, and -1 otherwise. The value of τ_k is computed during run-time. Such an implementation of batchnorm-activation operations requires much smaller number of LUTs, without the need for DSPs and FFs, compared to regular implementation of batchnorm-activation. Max-pooling, average-polling, and min-pooling have been effectively implemented with Boolean OR-operator, Boolean majority function, and Boolean AND-operator, respectively.

The accelerator architecture is composed of building blocks from the FINN hardware library. The matrix-vector-threshold unit (MVTU) is the core computational building block as matrix-vector operations followed by thresholding form the majority of BNN operations. The design of MVTU consists of an input buffer, an array of P parallel PEs each with S SIMD lanes, and an output buffer. BNN weight matrix is distributed across the PEs and stored locally in on-chip memory. Subsequently, the input images are streamed through the MVTU and multiplied with the weight matrix.

Particularly, the PE computes the dot-product between an input vector and a row of weight matrix, each of S-bits wide, using an XNOR gate, as shown in Fig. 18. Then, it compares the number of *set* bits to a threshold and produces a 1-bit output value as previously discussed.

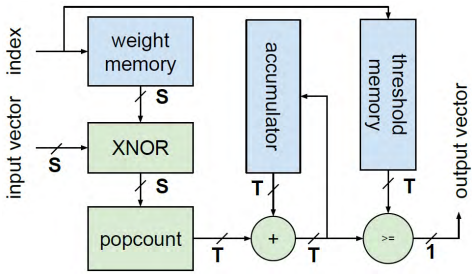

FIGURE 18. The Architecture of MVTU PE [181].

Umuroglu *et al.* [181] implemented the CONV layer using a sliding window unit (SWU) and an MVTU, where convolutional operation is transformed to matrix-multiplication of image matrix and filter matrix. SWU generates the image matrix to MVTU by moving the sliding window over the input FMs, while the filter matrix is generated by packing the weights from the convolution filters as shown in Fig. 19. In order to meet the user throughput requirement, MVTU is folded (time-multiplexed) by controlling the values of P and S. Folding of MVM decides partitioning of the matrix across PEs. Every row of matrix tile is mapped to a distinct PE and every column of PE buffer is mapped to a distinct SIMD lane. In this away, the required number of cycles to compute one MVM (total fold) is obtained as $(X \times Y)/(P \times S)$, where X and Y are the dimensions of the matrix. The folding factors of BNN layers have been determined such that every BNN layer takes nearly the same number of cycles.

To evaluate FINN, the authors implemented CNV topology on Xilinx Zynq-7000 board at 200 MHz to accelerate BNNs

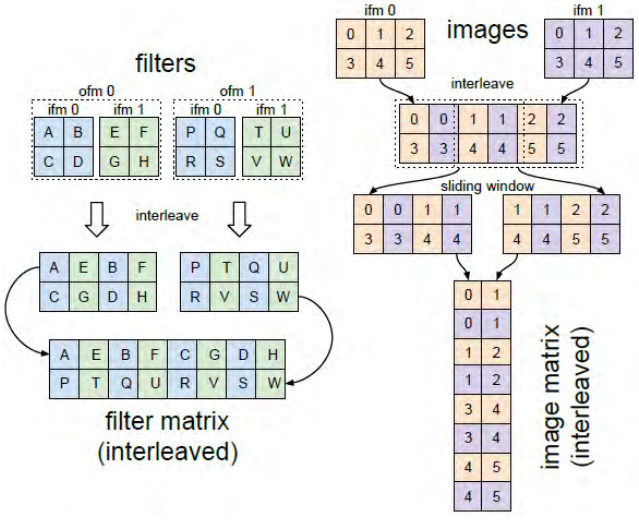

FIGURE 19. Transforming CONV to Matrix-Multiplication [181], where, ifm and ofm are the input and output feature maps, respectively.

inference on CIFAR-10 [182]. CNV contains three repetitions of two 3×3 CONVs and 2×2 max-pooling layers. Its topology is inspired by VGG-16 and BinaryNet [180]. Although CNV accepts images with 24-bits/pixel as an input and produces a 10-element vector of 16-bit values, 2-bits are used for representing intermediate results while 1-bit is used for representing CONV and FC weights. Experimental results demonstrated that the proposed design provides high performance (2.5 TOPS) while incurring low energy consumption (11.7 Watts). FINN outperforms the design by Ovtcharov *et al.* [152] by over 13.8× for throughput.

In [83], loop optimization techniques [55], [79] have been employed in FPGA to design a customized CNN accelerator through speeding up CONV layer operations. Firstly, an in-depth analysis is provided to numerically characterize loop unrolling, loop tiling, and loop interchange optimization techniques. In doing so, 8 CONV dimensions parameters (N^*), 8 loop unrolling design variables (P^*), and 8 loop tiling design variables (T^*) have been used with a constraint, as for a specific loop level, $1 \leq P^* \leq T^* \leq N^*$. Note that unrolling *Loop-1* and *Loop-3* requires $Pkx \times Pky$ and Pif multipliers, respectively, an adder tree with fan-in of $Pkx \times Pky$ and Pif, respectively, and an accumulator. On the other hand, unrolling *Loop-2* requires $Pix \times Piy$ parallel units of MAC to reuse the same weight for $Pix \times Piy$ times, while the input feature pixel can be reused by Pof times when unrolling *Loop-4* with the use of Pof parallel MAC units. Thus, $Pkx \times Pky \times Pif \times Pix \times Piy \times Pof$ multipliers are required. Please refer to Fig. 2 for more details on CONV loops levels and their parameters. In loop tile optimization, the authors have numerically set the lower bound on the required size of the input pixel buffer, the weight buffer, and output pixel buffer that ensures reading each input feature pixel and weight from the off-chip memory only once. On the other hand, loop interchange technique has a great impact on the times of memory access as well as the number of partial sums since it determines the order of computing CONV loops.

Secondly, the authors have provided a quantitative analysis of the design variables to minimize each of computing latency, partial sum storage, on-chip buffer access, and off-chip DRAM access. Subsequently, MATLAB scripts are used to randomly sample a subset of the solution space to find the optimal design configurations. This is due to the large solution space, more than 7.2×10^{13} possible configurations for loop tiling variables of width (Pox) and height (Poy) output FM alone. According to the randomly sampling results for VGG-16 CNN model on Arria 10 GX 1150 FPGA, uniform unrolling factors for CONV layers are used with $Pix = Pox = Piy = Poy = 14$ and $Pof = 16$ for *Loop-2* and *Loop-4*, respectively, to reuse input feature pixels and weights. On the other hand, *Loop-1* and *Loop-3* are serially computed to prevent the movement of the partial sums between the MAC units and consume them ASAP since both *Loop-1* and *Loop-3* need to be finished in order to obtain one final output pixel. More importantly, the order of loops computation has been found to be as follows. *Loop-1* is

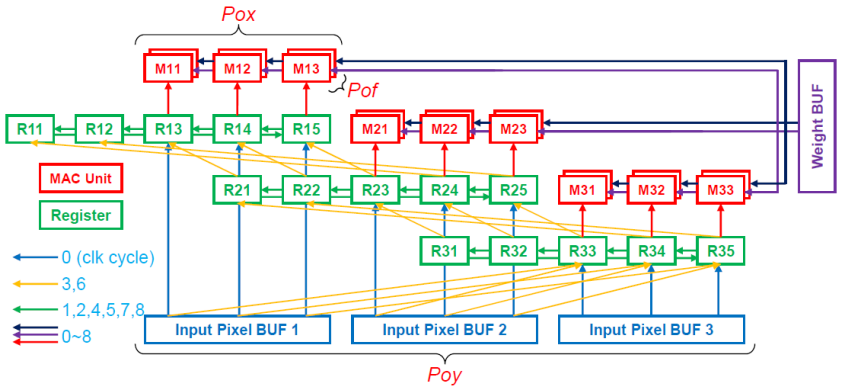

FIGURE 20. CONV Acceleration Architecture and Dataflow [83], where, $Pix = Pox = 3$, $Piy = Poy = 3$, and $Pof = 3$.

computed first, then comes *Loop-3*, and finally *Loop-2* and *Loop-4* are computed in any order.

Finally, a customized convolution accelerator module with efficient dataflow has been designed based on the previous results and used for all VGG-16 CONV layers. The CONV accelerator consists of 3,136 ($Pix \times Piy \times Pof$) independent MAC units and 14 (Pof) input pixel buffers. Fig. 20 shows an example of the designed CONV accelerator when Pix, Piy, and Pof are all equal to 3. The input pixels are shifted after fetching them out of the input pixel buffers. Subsequently, they can be reused among the input register arrays. Then, the input pixels are fed into the associated MAC units. The figure also shows that the input pixels and weights are shared by Pof and $Pix \times Piy$ MAC units, respectively.

The overall CNN acceleration system mainly consists of two SDRAM banks that hold the input feature pixels and weights, two modular Scatter-Gather DMA (mSGDMA) engines to facilitate the simultaneous read/write from/to the SDRAMs, and a controller to govern the sequential computation of layers as well as the iterations of the four CONV loops. On the other hand, dual weight buffers have been used to increase the throughput of FC layer through overlapping the inner-product computation with off-chip communication. The acceleration system has been written as parameterized Verilog scripts. The experimental results show that the proposed accelerator has a throughput of 645.25 GOPS, which is more than 3.2× enhancement compared to prior VGG-16 FPGA-based implementations [80], [98].

Venieris and Bouganis [183] further extended fpgaConvNet framework [165] to allow for optimizing either throughput or latency depending on the size of the workload. For large workloads, weights reloading transformation has been introduced to efficiently design latency-critical CNNs on FPGA. In contrast with fpgaConvNet, where a distinct architecture is designed for each subgraph, the weights reloading transformation allows for generating a single flexible architecture, named as the reference architecture and derived using pattern matching, to execute the workloads of all subgraphs by transitioning to different modes. Upon the execution of a new subgraph, the subgraph's weights are read into the on-chip memory and the multiplexers are configured to form the appropriate datapath. Fig. 21 demonstrates how weights reloading is applied. The authors have mentioned that the required time for transferring subgraph's weights is much smaller than the average time for full FPGA reconfiguration, 272.7× less when loading 4.5 MB of weights for a VGG-16 layer on Zynq XC7Z045.

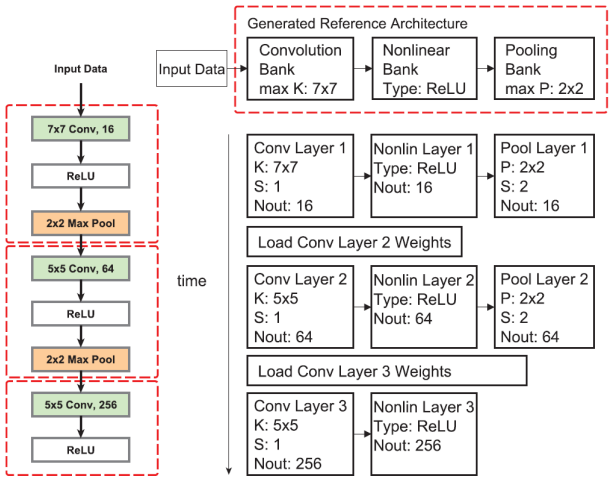

FIGURE 21. Weights Reloading [183].

In the situation discussed above, due to limited on-chip memory capacity, it might not be possible to load all weights required for a single CONV layer. To handle this, the authors introduced an input FMs folding factor (f_{in}) with each CONV layer. A CONV layer ($CONV_i$) is partitioned into f_{in_i} subgraphs in which each subgraph executes a fraction of $CONV_i$ to produce a fraction of the output FMs. The proposed latency-driven methodology has been evaluated by implementing AlexNet and VGG-16 with 16-bit fixed-point precision for both on Zynq XC7Z045 at 125 MHz. The experimental results showed 1.49× and 0.65× higher CONV throughput than DeepBurning [155] and the embedded FPGA

accelerator in [98] for AlexNet and VGG-16 implementations, respectively.

Lavin and Gray [184] demonstrated that CNN algorithms with small filters can be efficiently derived using Winograd algorithm [185] and fast Fourier transform (FFT) algorithm [186] due to their advantages in improving resource efficiency and reducing arithmetic complexity. Winograd computation involves a mix of element-wise (Eltwise) and general-purpose matrix multiplication, where some of the matrices need to be transformed. In particular, Winograd algorithm exploits the structure similarity among $n \times n$ tiled input FM pixels given a filter of size $r \times r$ to generate $m \times m$ tiled pixels of the output FM, where m represents the stride between Winograd tiles ($m = n - r + 1$), while minimizing the number of required CONV multiplications from m^2r^2 for conventional CONV algorithm to n^2. In another work, Zhang *et al.* [187] implemented FFT algorithm for CNN on FPGA platform. However, their proposed implementation shows little reduction of computation complexity with small filters such as 3×3.

Aydonat *et al.* [188] presented a deep learning architecture (DLA) based on OpenCL. Their proposed architecture reduces the external memory bandwidth requirements by an order-of-magnitude for both the convolutional and fully connected layers. This is achieved by caching all intermediate feature maps on-chip in stream buffers. For fully connected layers, image batching is used where a batch of images are processed together through the fully connected layers. The approach utilizes the Winograd transformation to reduce the multiply-accumulate operations, which could reduce the number of needed operations by about 50%. In addition, it uses half-precision (FP16) floating-point operations with shared exponents, which significantly reduces the needed computational resources.

The overall DLA architecture is shown in Fig. 22. Each PE consists of dot-product units, accumulators, and caches, for performing dot-products for convolution and fully connected layers. Caches are used for storing filter weights. To avoid idle computation cycles, double-buffering is used such that filter weights for the next convolution layer are prefetched onto the caches while filter weights are loaded from the caches for a particular convolution layer. Stream buffers store feature data and stream it to PEs. Each stream buffer is double-buffered similar to filter caches. Images are loaded from the DDR and are stored in stream buffers before the first convolution layer starts execution. During a convolution layer execution, while feature data for a convolution layer is being streamed into the PEs, the outputs of convolutions are simultaneously stored in the buffers. The StreamBuffer unit applies the Winograd transformations to features, and streams the transformed features to the first PE which are forwarded through all the PEs via the daisy-chained input connections between them. The ReLU unit receives the outputs of the PEs via daisy-chained output connections. Then, the normalization unit receives the outputs of the ReLU unit and applies the normalization formula across the feature maps. The pooling unit receives the outputs of the normalization unit and computes the maximum value in a window. The output of the pooling unit is stored back in the stream buffer for further processing, if more convolution layers are to follow. Otherwise, the outputs of the pooling unit are stored in external memory. For the fully connected layers, features data are stored on PEs caches while filter weights are stored in stream buffers. For the first fully connected layer, features data are read back from external memory and loaded onto the PE caches. The ReLU output is sent directly to DDR, without applying normalization or pooling. The sequencer generates the control signals to control the operation of the various blocks in DLA according to the topology of the executed CNN. Executing a different CNN requires just changing the sequencer configuration.

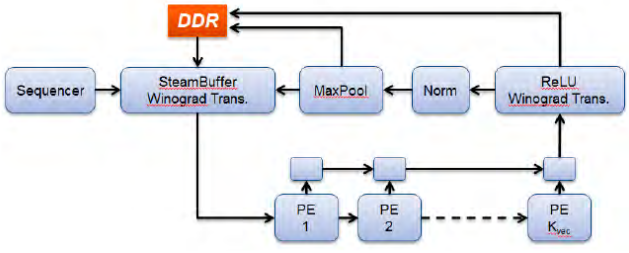

FIGURE 22. Overall DLA Architecture [188].

The DLA has been evaluated by implementing AlexNet CNN on Intel's Arria 10 dev kit which contains a A10-1150 device (20nm) using a 96 batch size for the fully connected layers. It achieved a performance of 1020 images/s. In addition, it achieved 8.4x more GFLOPS than the latest Ultrascale (KU 20nm) result reported in [162], which uses a 32 batch size for the fully connected layers, and 19× more GFLOPS than the latest Stratix V result reported in [80]. Furthermore, it has achieved energy efficiency at 23 images/s/W, which is similar to what is achieved with the best publicly known implementation of AlexNet on NVIDIA Titan X GPU.

Unlike DLA architecture [188] where a 1D Winograd algorithm was employed to reduce arithmetic complexity, Lu *et al.* [189] implemented a novel FPGA architecture with a two-dimensional Winograd algorithm [185] to accelerate convolutional computation of CNNs. The overall architecture consists of line buffer structure and Winograd PE engine, as shown in Fig. 23. Particularly, $n + m$ input lines and m output lines of on-chip buffers are used to effectively reuse FM data among different tiles. While Winograd PE engine reads the first n input lines to perform Winograd computation, the next m input lines load pixels from off-chip memory using FIFOs to overlap the data transfer and computation. Thereafter, the input lines are rotated in a circular fashion to make the next n input lines ready. On the other hand, Winograd PE engine composed of 4 pipelined stages performs transformation, element-wise matrix multiplication, additional transformation, and accumulation of output tiles, respectively.

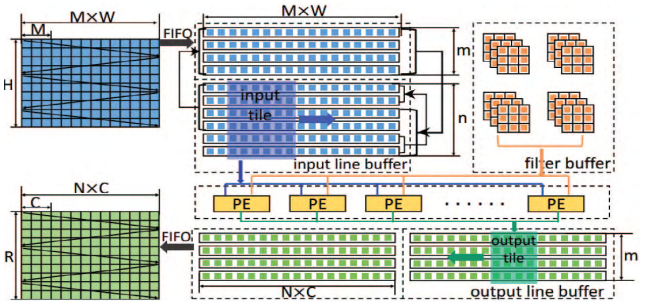

FIGURE 23. Winograd-based CNN Accelerator [189], where, *m* is the size of the input FM tile, *n* is the size of the output FM tile, *M* is the number of input channels, *N* is the number of output channels, *W* is the maximal width of all input FMs, *C* is the width of the output FMs.

A vector of PEs is employed to achieve parallelism through unrolling $Loop-4$ (Pof) and $Loop-3$ (Pif) similar to that in [55]. To implement FC layer, the proposed accelerator uses the input line buffers to hold FC weights while input neurons are stored on the filter buffers. Then, Winograd PE engine is reused to implement FC operation but with bypassing the transformation stages. Moreover, a batch (N_{batch}) of input FMs are assembled and processed together in order to improve the memory bandwidth. An analytical model has been proposed for a fast design space exploration of optimal design parameters (n, Pof, Pif, N_{batch}) constrained by FPGA configuration with a 16-bit fixed-point representation for both FM data and filter.

The proposed accelerator has been evaluated by implementing AlexNet and VGG-16 on Xilinx ZCU102 FPGA. AlexNet CONV layers have 3 different filters. Conventional CONV algorithm has been applied to the first CONV layer as it has a filter of size 11×11 while a uniform filter of size 3×3 for Winograd algorithm has been used to implement the rest of the layers. The design parameters are found to be equal to (6, 4, 8, 128) and (6, 4, 16, 128) for AlexNet and VGG-16, respectively. The experimental results demonstrated that the proposed Winograd-based CNN accelerator has an average performance of 854.6 GOPS and 2940.7 GOPS for AlexNet and VGG-16, respectively, with power consumption of 23.6 Watts for both. The proposed accelerator has also been evaluated on Xilinx ZC706 platform where the design parameters are found to be as (6, 2, 8, 32) and (7, 4, 4, 32) for AlexNet and VGG-16, respectively. The experimental results demonstrated that Winograd-based CNN accelerator has an average performance of 201.4 GOPS and 679.6 GOPS for AlexNet and VGG-16, respectively, with power consumption of 23.6 Watts for both. Compared to the implementation of VGG-16 on NVIDIA Titan X with the latest CuDNN 5.1, Titan X gives better performance than Xilinx ZC706 but the implementation on Xilinx ZC706 achieves 1.73× higher energy efficiency.

Zhang and Li [190] presented an OpenCL-based architecture for accelerating CNNs on FPGA. They also proposed an analytical performance model to identify the bottleneck in OpenCL-based acceleration of VGG-19 CCN model on modern FPGA platforms such as Altera Arria 10 GX 1150. Based on roofline mode analysis, it is shown that the bandwidth requirement of VGG-19 workload is higher than what is provided by the FPGA board. Thus, they identified on-chip memory bandwidth as the key performance bottleneck. In addition, they observed that exploited data-level parallelism in the existing Altera OpenCL library [191] leads to wasteful replication of on-chip memory (BRAM). This is due to connecting each PE with a dedicated BRAM port.

Therefore, a Verilog-based accelerator kernel has been designed and warped to an OpenCL IP in order to optimally balance on-chip memory bandwidth with workload computational throughput and off-chip memory accesses. In particular, the proposed kernel consists of a compute subsystem, a local memory subsystem, and a 2D dispatcher. The compute subsystem is organized hierarchically into compute units (CUs) and PEs. At PE level, the authors have designed a 2D multi-cast interconnection between BRAMs (32-bit data width) and PEs to improve the efficiency of on-chip BRAM usage by sharing the data of one BRAM port with several PEs as shown in Fig. 24. The CU has been designed as a 2D PE array of size 16×16 to match the computational bandwidth with the maximum streaming bandwidth (512-bit data bus) provided by off-chip memory. The 2D dispatcher divides the work items into work groups each of size (X_0, X_1) as shown in Fig. 25. Thereafter, it adaptively schedules the work items within each work group to the CUs starting with the lowest dimension to balance the memory bandwidth with capacity. The 2D dispatcher is also responsible for host/device memory data transfers. In addition, the authors have limited the maximum fan-out for registers to 100 in order to guarantee a higher frequency.

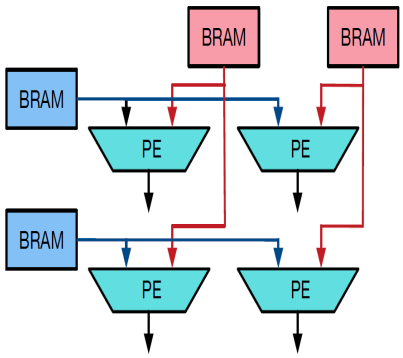

FIGURE 24. Compute Unit with a 2D BRAM-to-PE Interconnection [190].

The CONV layer has been implemented as a matrix multiplication by flattening and rearranging the data using line buffer [154], as shown in Fig. 26, in a similar fashion to that in [80]. The line buffer converts continuous address stream from external memory into a stream conducive for CONV operation to substantially reduce the bandwidth requirement of off-chip memory. To implement FC layer, the proposed accelerator uses one column of PEs in the CU. The proposed implementation has achieved 866 GOPS and 1790 GOPS

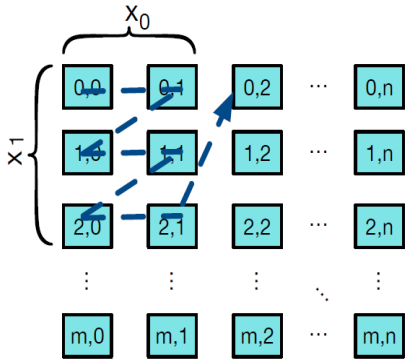

FIGURE 25. 2D Dispatcher [190], where, X_0 is the column size of kernel buffer as well as the row size of the input feature buffer, and X_1 is the row size of kernel buffer.

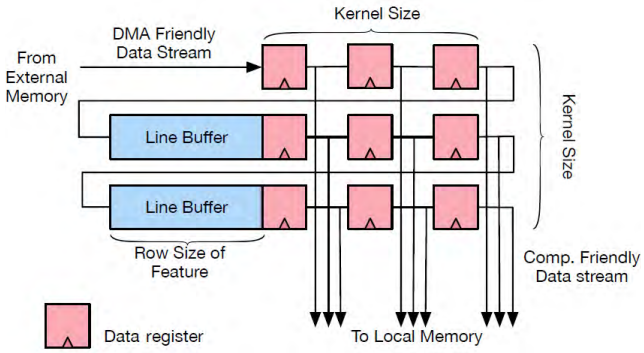

FIGURE 26. Line Buffer Design [190].

with the use of 32-bit floating-point and 16-bit fixed-point, respectively, under 370 MHz and 385 MHz working frequencies, respectively.

All previously discussed FPGA-based CNN accelerators, except the ones discussed in [165] and [170], have employed a single CLP to maximize the aggregate throughput of performed consecutive convolutional operations. However, Shen *et al.* [192] noted that using a single globally-optimized CLP design for the computation of CONV layers of radically different configurations and dimensions leads to sub-optimal performance and insufficient utilization of FPGA resources. Fig. 27a demonstrates the use of a single CLP to iteratively

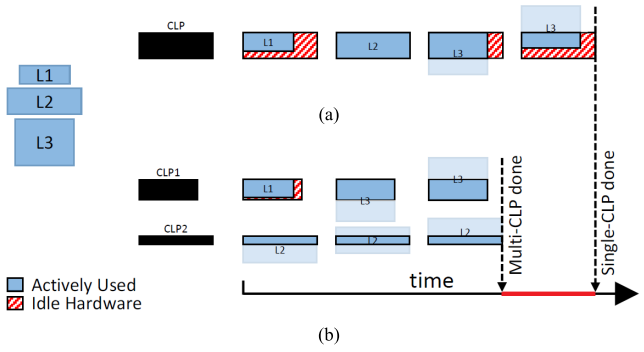

FIGURE 27. Operation of CONV Layer Processors (CLPs) on CNN with three CONV Layers [192].

process $L1$, $L2$, and $L3$ CONV layers where the dimensions of the hardware (*CLP*, *CLP*1, and *CLP*2) and the layers are represented by the size and shape of the boxes. It is clear that computing $L1$ and portions of $L3$ leaves FPGA resources unutilized as their dimensions are smaller than the dimension of the used CLP. Note that processing a CONV layer with a dimension bigger than the dimension of CLP, such as $L3$, requires the repeated use of CLP to process different portions of the layer.

The authors have also followed the methodology in [55] to derive an optimal single CLP, through finding the optimal unrolling factor $\langle T_m, T_n \rangle$, for implementing SqueezeNet [193] and AlexNet on Virtex7 690T FPGA with a single precision floating-point and 16-bit fixed-point arithmetic units, respectively. They found that one quarter of DSP slices of SqueezeNet's CLP remain unused. Even more worse utilization has been observed for AlexNet. The optimal single CLP has not utilized, on average, more than one quarter of the arithmetic unit resources.

On the other hand, they also noted that using one CLP for each stage of CONV layer in a fashion similar to that in [194] is not efficient due to three reasons. First, it reduces the on-chip BRAM buffer size of each CLP which minimizes overall data locality. Second, such one-to-one mapping of CONV layers and CLPs requires orchestrating many off-chip memory accesses which incurs latency and bandwidth overheads. Third, the overall control overhead scales with the number of CLPs which leaves insufficient resources for the computation of CNN.

To address the above inefficiencies, Shen *et al.* [192] proposed a multi-CLP accelerator system for CNNs where the available PFGA hardware resources are partitioned across multiple smaller CLPs. Each CLP is tailored with a dimension that closely matches the dimensions of a subset of CONV layers. Thereafter, these specialized CLPs are used to concurrently operate on a batch of images to achieve a higher overall throughput, as shown in Fig. 27b, where the same hardware in Fig. 27a is partitioned into two parallel CLPs; *CLP*1 and *CLP*2.

Shen *et al.* [192] developed an optimization search algorithm that uses dynamic programming to find optimal designs. For given configurations of CNN model (i.e., CONV layers descriptions) and resource constraints of the targeted FPGA platform (i.e., number of DSP slices, BRAM-18Kb units, and off-chip memory bandwidth), it derives the optimal number of CLPs (along with their $\langle T_m, T_n \rangle$ dimensions) as well as the optimal mapping between CONV layers and CLPs that maximize the performance. The assignment of CNN layers to CLPs is static, where each CNN layer is mapped and bounded to a particular CLP. Subsequently, CNN layers are pipelined to their CLP, as shown in Fig. 27b, where $L1$ and $L3$ are pipelined to *CLP*1 while $L2$ is repeatedly processed on *CLP*2 with very little idle hardware which improves the performance compared to single CLP approach. Moreover, the optimization algorithm also finds the optimal partition of on-chip BRAM resources of each CLP that minimizes

the overall off-chip memory accesses. Note that the optimal dimension of each CLP is found based on the work in [55].

Subsequently, C++ (HLS) templates are parameterized to design CLPs and to form a complete implementation of CNN. A standard AXI crossbar is used to interconnect the independent CLPs. The ping-pong double-buffering technique is also used for input FMs, output FMs, and weights to allow for transferring data while computation is in progress. The experimental results of implementing AlexNet with a single precision floating-point using multi-CLP accelerator on Virtex7 485T and 690T FPGAs at 100 MHz demonstrate 1.31× and 1.54× higher throughput than the state-of-the-art single CLP design in [55], respectively. For the more recent SqueezeNet network, the proposed multi-CLP accelerator results in speedup of 1.9× and 2.3× on Virtex7 485T and 690T FPGAs at 170 MHz with 16-bit fixed-point, respectively.

Xuechao *et al.* [195] presented a systolic architecture for automatically implementing a given CNN on FPGA based on OpenCL description, maximizing clock frequency and resource utilization. The proposed systolic architecture is shown in Fig. 28. Each PE shifts the data of the weights (W) and inputs (IN) horizontally and vertically to the neighboring PEs in each cycle. The 2D structure of PEs is designed to match the FPGA 2D layout structure to reduce routing complexity and achieve timing constraints.

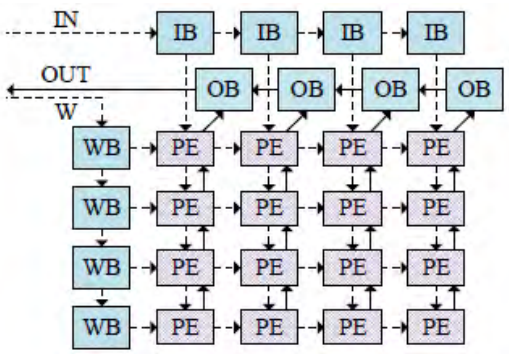

FIGURE 28. Systolic Array Architecture for CNN [195].

The technique first finds a feasible mapping for the given CNN to the systolic array to guarantee that proper data is available at specific locations in the PE array at every cycle. Then, the size of PE array (dimensions) is determined which has an impact on the required number of DSPs, the clock frequency, and the DSPs efficiency. Finally, the data reuse strategy is determined by choosing proper tiling sizes. The proposed technique has been evaluated using AlexNet and VGG16 on Intel's Arria 10 GT 1150 board. The technique has explored the use of both 32-bit floating-point and fixed-point using 8-bits for weights and 16-bits for data. Evaluation results show that, for the VGG16 CNN, the technique achieves up to 1,171 GOPS on Intel's Arria 10 device with a clock frequency of 231.85 MHZ and (8-16)-bit fixed-point representation.

In another recent research work, Ma *et al.* [196] generalized the previously proposed accelerator in [83] to efficiently accelerate ResNet-50 and ResNet-152 on Arria 10 GX 1150 FPGA. In doing so, they designed flexible and scalable CONV, ReLU, BatchNorm, scale, pooling, FC, and Eltwise primitives. In addition, local control logic and registers have been used with each primitive to control their computation order and to hold their configurations, respectively. By doing so, ResNets primitives can be efficiently reused for different parameters of each layer.

For ResNets scalable CONV primitive, there are four (kernel, stride) size configurations; (3 × 3, 1), (1 × 1, 1), (1 × 1, 2), and (7 × 7, 2). Therefore, a similar architecture and dataflow to that shown in Fig. 20 has been used for CONV but with the use of two sets of register arrays; with shifting between the registers (which is shown in Fig. 20, *Set-1*), and without shifting between the registers (*Set-2*). The CONV primitive with 3 × 3 kernel and stride of 1 uses *Set-1* register array, while *Set-2* is used with (1 × 1, 1), (1 × 1, 2), and (7 × 7, 2) configurations. In CONV primitive with *Set-2*, the input pixels are fed from the input pixel buffers into the corresponding registers without shifting, and then to MAC units. The skipped input pixels in (1 × 1, 2) configuration are not stored to the input pixel buffers. On the other hand, the (7 × 7, 2) configuration of the kernel and stride sizes is retained as the (1 × 1, 1) case while transferring repeated input pixels into the input pixel buffers and rearranging their storage patterns. The CONV primitive also takes care of zero-paddings for different (kernel, stride) size configurations.

The loop unrolling and tiling techniques in [83] have also been employed to accelerate CONV primitive with a uniform mapping of PEs to all ResNets CONV layers. However, designing of efficient CNN modules is not enough, as the memory accesses and data movements between these modules must also be minimized. Therefore, the authors have designed a layer-by-layer computation flow. The global control logic is responsible for governing the sequential operations of primitives and their dataflow through predefined and preloaded layered-based execution flowchart, as shown in Fig. 29. In addition, it has been modeled to reconfigure ResNet primitives according to the parameters of each layer during runtime. For instance, it maps a particular number of PEs to CONV layer based on loop unrolling parameters as well as it controls the selection of register array type (*Set-1* or *Set-2*) based on CONV (kernel, stride) parameters.

On the other hand, a custom DMA manager has been designed to control the operations of DMA. Note that the DMA is responsible for transferring the input FM pixels, weights, and output FM pixels between off-chip memory and on-chip buffers. Unlike ALAMO architecture [168] where the output pixels are only stored in on-chip buffers, this work as well as the work discussed in [83] store the output pixels in off-chip memory with the use of loop tiling technique in order to have a flexible architecture that can process large-scale CNNs. The dual weight buffers technique has not been used in this work due to the current trend in CNNs where either

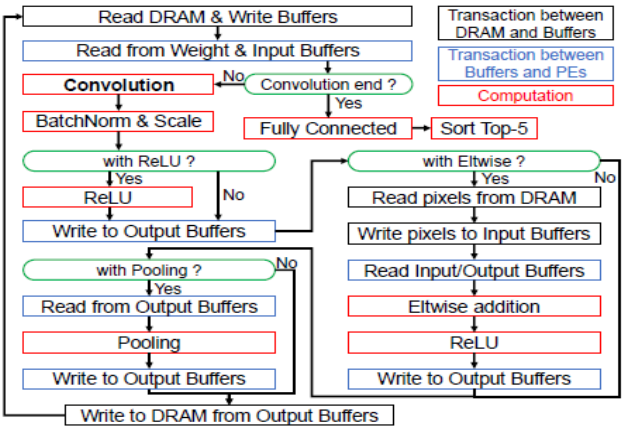

FIGURE 29. Execution Flowchart of ResNets Layers [196].

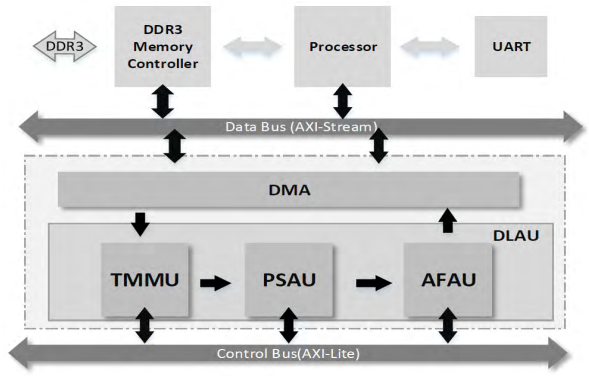

FIGURE 30. DLAU Accelerator Architecture [197].

the size of FC weights has been significantly reduced (2 M in ResNet compared with 123.6 M in VGG) or the FC layers are completely removed such as in NiN. The experimental results demonstrated that the achieved throughput for ResNet-50 and ResNet-152 are 285.1 GOPS and 315.5 GOPS, respectively. Finally, the authors mentioned that higher throughput can be achieved using batch computing [194].

Wang *et al.* [197] proposed a scalable design on FPGA for accelerating deep learning algorithms. In order to provide a scalable architecture and support various deep learning applications, the proposed architecture utilizes the tiling technique in which the large-scale input data is partitioned into small subsets. The size of the tile is configured to leverage the trade-off between the hardware cost and the speedup. Moreover, the authors explored hot spots profiling to determine the computational parts that need to be accelerated to improve the performance. The experimental results illustrated that matrix multiplication and activation functions are the key operations in deep learning algorithms as they consume about 98.6% and 1.1% of the overall execution time, respectively. Thus, the proposed accelerator is responsible for speeding up both matrix multiplication and activation function computations.

The main components of the proposed architecture are the embedded processor, the DDR3 memory controller, the DMA module, and the deep learning acceleration unit (DLAU), as shown in Fig. 30. The embedded processor utilizes the JTAG-UART to communicate with the acceleration unit [198]. The DLAU unit accesses the DDR3 memory to read the tiled input data and to write the results back through the DMA module during the execution. The DLAU utilizes three fully pipelined processing units to improve the throughput, while minimizing the memory transfer operations. These units are tiled matrix multiplication unit (TMMU), partial sum accumulation unit (PSAU), and activation function acceleration unit (AFAU). TMMU is responsible for multiplication and generating the partial sums. To optimize the performance, TMMU is structured as a pipelined binary adder tree. Moreover, it uses two sets of registers alternately to overlap the computation with the communication, one group is used for the computation, while in parallel, the other group is loaded with the next node data every clock cycle. On the other hand, PSAU is responsible for accumulating the partial sums generated from TMMU. Finally, AFAU implements the sigmoid function using piecewise linear interpolation to speed up the computation with negligible accuracy loss. Since the processing units in DLAU might have inconsistent throughput rates, each unit has input FIFO buffer and output FIFO buffer to prevent data loss.

The authors implemented the proposed architecture on Xilinx Zynq Zedboard with ARM Cortex-A9 processors clocked at 667 MHz. In addition, they used the MNIST dataset as a benchmark considering the network size as 64×64, 128×128, and 256×256. The experimental results demonstrated that the speedup of the DLAU accelerator is up to 36.1× compared with the Intel Core2 processors at 256×256 network size. In addition, the results depict that the proposed architecture is quite energy-efficient as the total power consumption was only 234 mW.

In [199], a generalized end-to-end acceleration system of the previously proposed accelerators in [78], [83], [168], and [196] has been developed to support diverse CNN models. In doing so, a user-friendly interface and an RTL-level compiler have been proposed to automatically generate customized FPGA designs. The authors have developed an expandable optimized RTL-based library containing the most commonly used CNN operations. These operations have been coded in Verilog and designed based on the quantitative analysis and optimization strategies discussed in [83]. The compiler generates a DAG-based structure for the used CNN model and then compiles it with RTL modules in the library. The proposed compiler allows the user to input the high-level information of the used CNN model (previously designed on Caffe framework [58]) as well as the design variables (i.e., loop unrolling and loop tiling variables) with the resource constrains of the targeted FPGA platform. Such utility facilitates the exploration of the best trade-off between the resource usage and the performance.

Unlike the architecture in [168] where individual CONV module is assigned to each CONV layer, the scalable RTL computing module proposed in this work is reused by

all CNN layers of the same type for different CNNs as shown in Fig. 31. Note that it is not necessary to have all these modules in the architecture. For instance, the RTL compiler will not compile or synthesize Eltwise and combined batch normalization with scale (Bnorm) modules for VGG-16 model which greatly saves the hardware resources.

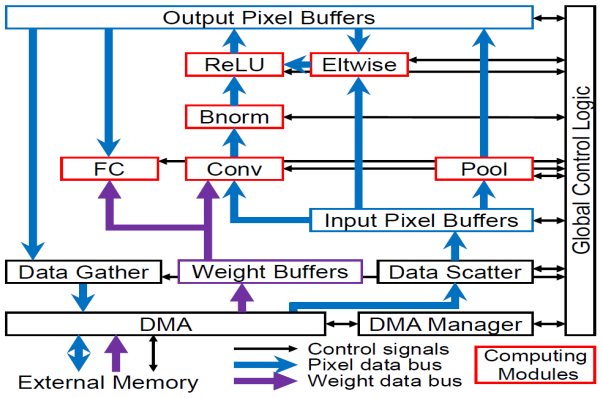

FIGURE 31. Overall Architecture and Dataflow [199].

On the other hand, the authors categorized CNN layers into key layers (i.e., CONV, pool, and FC layers) and affiliated layers (e.g., ReLU, Bnorm, Eltwise, and all other layers). They have also defined layer combos, where each combo is composed of a key layer and several affiliated layers. Layer combos are sequentially executed according to their order in the DAG. Moreover, the layer combo is also divided into several sequential tiles. The computation of each combo tile starts by reading its input pixels from off-chip DRAM and ends by writing back its output pixels to off-chip DRAM. The global control logic, inter-tile control logic, and intra-tile control logic are responsible for governing the sequential operations of layer combos and reconfiguring the modules, combo tiles, and tile layers (key and affiliated layers), respectively, through predefined flexible execution schedule similar to that in [196].

The authors have also employed special storage pattern of both input pixels and weights on off-chip memory before the acceleration process to maximize data reuse and minimize of data communication. The architecture of CONV module is designed based on the acceleration strategies in [83] and [196] but with a different organization of MAC units as shown in Fig. 32. The MAC units of CONV module have been organized into $Piy \times Pof$ independent MAC blocks, with each MAC block containing Pix MAC units to further minimize the buffer read operations and the partial sums movements. Moreover, such organization enables to handle varying (kernel, stride) sizes configurations through generating different variants of CONV register arrays during the compilation.

Experimental results demonstrated that the achieved throughput on Intel Stratix V GXA7 for NiN, VGG-16, ResNet-50, and ResNet-152 are 282.67 GOPS, 352.24 GOPS, 250.75 GOPS, and 278.67 GOPS, respectively. On the other hand, the achieved throughput on Intel Arria 10 GX 1150

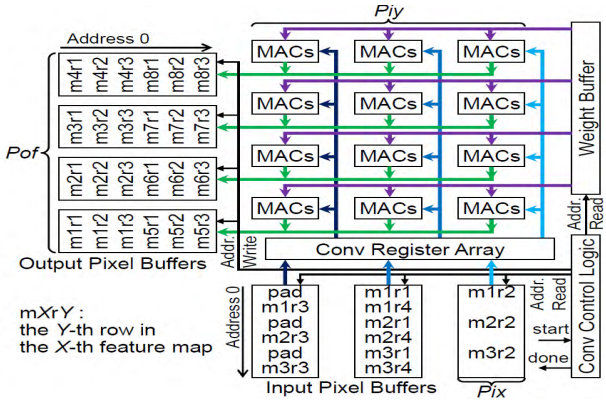

FIGURE 32. CONV Reconfigurable Computing Module [199].

was 587.63 GOPS, 720.15 GOPS, 619.13 GOPS, and 710.30 GOPS for NiN, VGG-16, ResNet-50, and ResNet-152, respectively. More than 2× throughput improvements have been achieved on Intel Arria 10 GX 1150 since it has 1.8× and 5.9× more logic elements and DSPs than the Intel Stratix V GXA7, respectively, which allows for larger loop unrolling variables.

Recently, the programmable solutions group at Intel has developed an FPGA software-programmable and run-time reconfigurable overlay for deep learning inference [200]. The developed overlay is referred to as the deep learning accelerator (DLA). For the hardware side of Intel's DLA, the team has partitioned the configurable parameters into run-time and compile-time parameters. The run-time parameters allow for easy and quick use of different neural network frameworks, while the compile-time parameters provide a tunable architecture for performance. Intel's DLA uses a lightweight very long instruction word (VLIW) network, an 8-bit unidirectional ring network, to support the control and reprogramming logic. Compared with typical overlays, Intel's DLA comes with only ∼1% overhead while other typical overlays tend to always come with larger overheads [201]. The reprogramming of Intel's DLA overlay allows for consecutive runs of multiple NNs in a single application run [202] without the need for reconfiguring and recompiling the FPGA.

Fig. 33 shows that a 1D array of PEs is used to perform convolution, multiplication, or any other matrix operations. Each PE contains a double-buffered filter cache allowing for pre-loading of next filters while computing. The stream buffer employed the double-buffering mechanism as well to store the inputs and the intermediate data on-chip. To have flexible NN architecture, Intel's DLA employs an Xbar interconnect that connects all the core functions required. Thus, deep learning functions can be easily added to the overlay through the Xbar by picking them from a suite of pre-optimized functions of the select frameworks that Intel's DLA uses. The width adaptation module has been used to control the throughput of the function. In addition, Intel's DLA supports vectorization across the input width (Q_VEC), input height (P_VEC), input depth (C_VEC), output depth (K_VEC), filter width (S_VEC), and other dimensions as

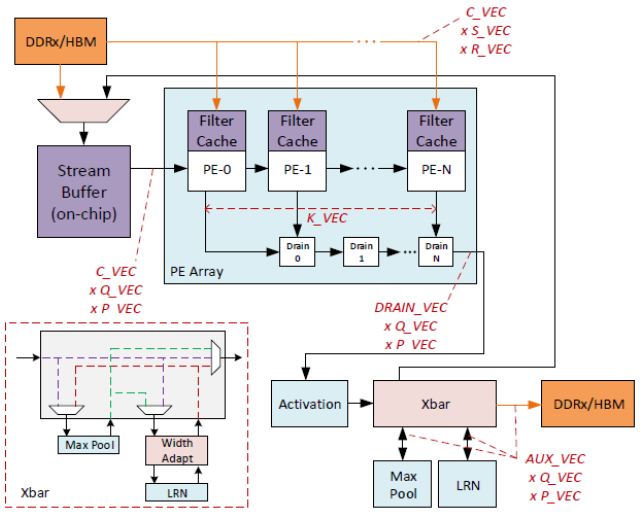

FIGURE 33. Intel's DLA: Neural Network Inference Accelerator [200].

depicted in Fig. 33. The authors mention that vectorization depends on the layers' dimensions of the considered framework. However, they did not provide a systematic way for finding the optimal balance for the number of used PEs and the size of the caches. For efficient use of resources, Intel's DLA maps AVG pooling and FC primitives to convolutions in order to avoid having under-utilized (over time) dedicated auxiliary functions.

For the software side of Intel's DLA, the proposed accelerator uses a graph compiler to map a NN architecture to the overlay for maximizing the hardware efficiency through slicing, allocation, and scheduling. In the slicing pass, the graph compiler breaks down the architecture into subgraph (a chain of functions) in such a way that they fit within the computing and storage resources of the overlay. A single CONV layer followed by a pooling layer is an example of CNN subgraph. The graph compiler optimizes the external memory spill-points by group slicing technique. The group slicing allows several sequential convolutions, for instance, of a single slice to be computed before moving onto the next slice while using the whole stream buffer. During the allocation pass, the graph compiler optimizes the use of a custom developed filter caches and stream buffer by managing the read and write from the stream buffer for each slice. Moreover, it assigns an external memory address when the stream buffer is not big enough to hold the slice data.

Finally, Intel's DLA compiler schedules the execution of subgraphs using cost-based (the ratio of the output size to the effective input size) priority queue. The authors utilized the software-programmable and run-time reconfigurable overlay to optimize the software and hardware implementation of GoogleNet [31] and ResNet [49] CNNs. The benchmark results on an Arria 10 GX 1150 FPGA demonstrated that Intel's DLA has a throughput of 900 fps on GoogLeNet. The team pointed that multi-FPGA deployment [203] might be used to further improve the throughput of Intel's DLA.

Guo *et al.* [60] proposed a complete design flow, referred to as Angel-Eye, for mapping CNNs onto FPGA. It includes a data quantization strategy, a parameterizable and run-time configurable hardware architecture to support various CNNs, FPGA platforms, and a compiler to map a given CNN onto the hardware architecture. It adopts the approach of using a flexible hardware architecture and maps different CNNs onto it by changing the software. The proposed design flow from CNN model to hardware acceleration is shown in Fig. 34.

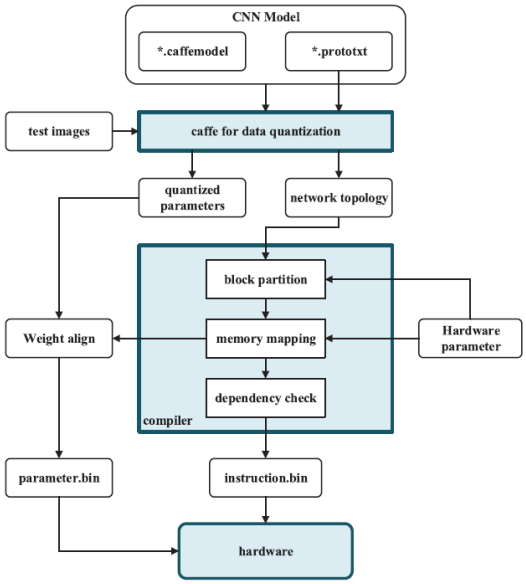

FIGURE 34. Design Flow from CNN Model to Hardware Acceleration [60].

Due to the large dynamic range of data across different layers, the best radix point is found for each layer for a given bit width. They demonstrated that their strategy can simplify state-of-the-art CNNs to 8-bit fixed-point format with negligible accuracy loss. Although 8-bits are used for representing data, 24 bits are used for representing intermediate data in layers, which is then aligned and quantized to 8 bits. Fig. 35 and Fig. 36 show the overall architecture of Angel-Eye and the structure of a single PE, respectively. The architecture is designed for supporting an instruction interface that supports three types of instructions; LOAD, SAVE, and CALC.

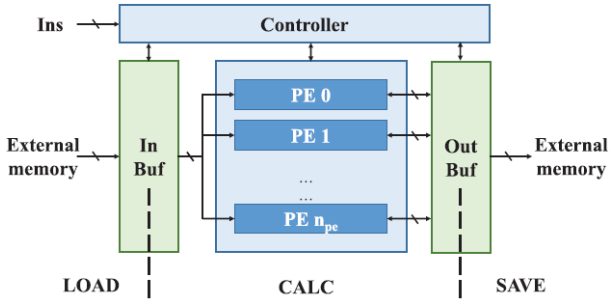

FIGURE 35. Overall Architecture of Angel-Eye [60].

The overall architecture is divided into four main components; PE array, on-chip buffer, external memory, and controller. The PE array implements the convolution operations

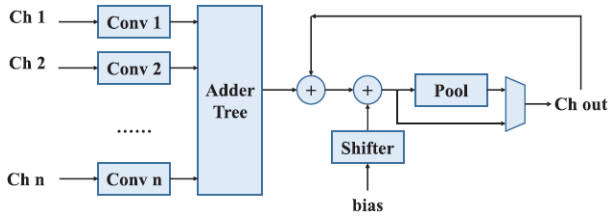

FIGURE 36. Structure of a Single PE [60].

in CNN and supports kernel level parallelism, and input and output channel parallelisms. It uses a 3×3 convolution kernel, as this size is most popular in state-of-the-art CNN models. However, larger kernel sizes are supported based on the use of multiple 3×3 kernels. The use of on-chip buffers allows data I/O and convolution calculation to be done in parallel. The controller is responsible for receiving the instructions and issuing them to the other components. The compiler maps the CNN descriptor to the set of instructions that will be executed by the hardware. It follows basic scheduling rules to fully utilize data localization in CNN and reduce data I/O.

The block partition step partitions the calculation of one layer to fit each block into the hardware. The memory mapping step allocates memory for communication between the host CPU and the CNN accelerator. Based on the block partition result, on-chip memory is allocated for the input and output feature map blocks and for the convolution kernels and bias values. The dependency check step checks data dependency among instructions and sets appropriate instruction flags to maximize parallelism between convolution calculation and data I/O. Based on experimental results, it is shown that the 8-bit implementation of Angel-Eye on XC7Z020 achieves up to $16\times$ higher energy efficiency than NVIDIA TK1 and $10\times$ higher than NVIDIA TX1. In addition, the 16-bit implementation of Angel-Eye on XC7Z045 is $6\times$ faster and $5\times$ higher in power efficiency than peer FPGA implementation on the same platform [148].

In [83] and [199], a special register array architecture has been designed to rearrange buffers data and direct them into PEs for the purpose of implementing CONV module that supports specific stride and zero-padding settings. Although the designed CONV module is not generalized for any (kernel, stride) size configurations, it is composed of complex wire routing and control logic as shown in Fig. 20. To have flexibility in directing the dataflow of CONV pixels, Ma *et al.* [204] replaced the register array architecture in [199] with a data router as shown in Fig. 37.

The data router is a scalable set of data bus from buffer to PE (BUF2PE). The BUF2PE data bus consists of simple register arrays with FIFOs in between to form a line buffer similar to that in [154]. The register array uses the FIFO to pass its input pixels to the adjacent registers. Each BUF2PE data bus has different data movements within its register arrays to implement specific stride and kernel size settings. Unlike the register array architecture in [83] where the west zero-paddings are handled by changing the storage pattern within the input pixel buffer, the BUF2PE handles such kind

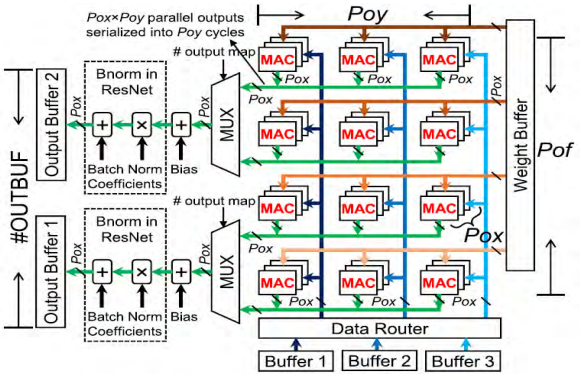

FIGURE 37. CONV Acceleration Architecture and Dataflow using Data Router [204], where, $Pix = Pox$, and $Piy = Poy$.

of paddings by shifting the connection between the register arrays and the input pixel buffers to simplify the data transferring from off-chip memory to on-chip buffers. However, there is still a need for adjusting the storage pattern within the input buffers in order to handle other zero-paddings.

The global control logic is responsible for selecting the suitable BUF2PE data bus from the data router as well as the suitable storage pattern within the input buffers based on the (kernel, stride) size configuration of CONV layer. The CONV module has also been optimized by reducing the required number of parallel adders that add the partial sums with biases as well as the number of parallel multipliers and adders needed to perform Bnorm operation by serializing the parallel outputs using multipliers. In addition, 16-bit fixed-point has been used to represent both weights and pixels, while dynamically adjusting the decimal point in different layers to fully utilize the existing data width [60]. The proposed compiler in [199] has been used to configure the parameterized Verilog scripts of the overall CNN acceleration system. Experimental results show throughput degradation on both Intel Arria 10 GX 1150 and Intel Stratix V GXA7 in comparison to the results in [199].

In Table 2 and Table 3, we summarize the reviewed FPGA-based deep learning networks acceleration techniques. For each technique, we list the year the technique was introduced, the key features employed for acceleration, the used deep learning model, the number of needed operations per image, the FPGA platform used to implement the technique, the precision used for the FMs and weights, the clock frequency used, the design entry for describing the modeled deep learning network, the type of LUT for the used platform, the number of resources available by the used platform in terms of BRAMs, LUTs, FFs, and DSPs, the percentage of each resource utilization, the performance in GOPS, the speedup in comparison to a given baseline model, and finally the power efficiency (GOPS/W).

IV. METAHEURISTICS IN THE DESIGN OF CONVOLUTIONAL NEURAL NETWORKS

Currently, convolutional neural network (CNN) structures are designed based on human expertise. For a given application,

TABLE 2. Implementation and performance summary of FPGA-based accelerators.

Technique	Year	Key Features	DL Model	Image Operations (GOP)	Platform	Precision	Frequency (MHz)	LUT Type	Design Entry	Resources				Resources Utilization				Performance (GOPS)	Speedup	Baseline	Power Efficiency (GOPS/W)
										BRAMs/ M20K	LUTs/ ALMs	FFs	DSPs	BRAMs/ M20K	LUTs/ ALMs	FFs	DSPs				
CNP [127]	2009	2D CONV modules, Memory interface with 8 simultaneous accesses	LeNet-5	0.52	Virtex4 SX35	16-bit fixed-point	200	4-input LUTs	Lush	192	30,720	30,720	192	N/A	90%	90%	28%	**5.25**	N/A		**0.35**
CONV Coprocessor Accelerator [129]	2009	Parallel clusters of 2D convolver units, data quantization, off-chip memory banks	4 CONV layers	1.06	Virtex5 LX330T	16-bit fixed-point	115	6-input LUTs	C	324	207,360	207,360	192	0.93%	17%	19.05%	55.73%	**6.74**	**6**×	2.2 GHz AMD Opteron	**0.61**
MAPLE [132]	2010	In-memory processing, banked off-chip memories, 2D array of VPEs	4 CONV layers	1.06	Virtex5 SX240T	fixed-point	125	6-input LUTs	C++	516	149,760	149,760	1,056	N/A				**7**	**0.5**×	1.3 GHz C870 GPU	N/A
DC-CNN [100]	2010	Integer factorization to determine the best config for each layer, input and output switches	3 CONV layers	0.52	Virtex5 SX240T	48-bit fixed-point	120	6-input LUTs	RTL	516	149,760	149,760	1,056	N/A				**16**	**4.0**× - **6.5**×	1.35 GHz C870 GPU	**1.14**
NeuFlow [143]	2011	Multiple full-custom processing tiles (PTs), Pipelining, Fast streaming memory interface	4 CONV layers in [145]	N/A	Virtex6 VLX240T	16-bit fixed-point	200	6-input LUTs	HDL	416	150,720	301,440	768	N/A				**147**	**133.6**×	2.66 GHz Core 2 Duo CPU	**14.7**
Memory-Centric Accelerator [146]	2013	Flexible off-chip memory hierarchy, Data reuse, Loop transformation	4 CONV layers	5.48	Virtex6 VLX240T	fixed-point	150	6-input LUTs	C	416	150,720	301,440	768	45.5%	1.1%	N/A	6%	**17**	**11**×	Standard Virtex6 Implementation	N/A
nn-X [148]	2014	Cascaded pipelining, Multiple stream processing	2 CONV layers	0.55	Zynq XC7Z045	16-bit fixed-point	142	4-input LUTs	Lua	545	218,600	437,200	900	N/A				**23.18**	**115**×	800 MHz Embedded ARM Cortex-A9 Processors	**2.9**
Roofline-based FPGA Accelerator [55]	2015	Roofline-based model, Loop transformation, Loop tiling/unrolling	AlexNet	1.33	Virtex7 VX485T	32-bit float-point	100	4-input LUTs	C	2,060	303,600	607,200	2,800	50%	61.3%	33.87%	80%	**61.62**	**17.42**×	2.2 GHz Intel Xeon	**3.31**
Microsoft Specialized CNN Accelerator [152]	2015	Multi-banked buffers, Network on-chip for re-distribution of output data, Software configurable	AlexNet	1.33	Stratix-V GSMD5	32-bit float-point	100	4-input LUTs	C	2,014	172,600	690,000	1,590	N/A				**178.63**	**3**×	Roofline-based FPGA Accelerator [55]	**7.15**
Embedded FPGA Accelerator [98]	2015	Data quantization and arrangment, SVD	VGG-16	30.76	Zynq XC7Z045	16-bit fixed-point	150	4-input LUTs	RTL	545	218,600	437,200	900	86.7%	83.5%	29.2%	89.2%	**136.97**	**1.4**×	2.9 GHz Intel Xeon	**14.22**
DeepBurning [155]	2016	NN model compression, Compiler-based library, Automatic partitioning/ tiling, Function Approx.	AlexNet	1.46	Zynq XC7Z045	16-bit fixed-point	100	4-input LUTs	RTL	545	218,600	437,200	900	N/A	17.3%	7.65%	16%	**73**	**15.88**×	2.4 GHz Intel Xeon	**6.43**
OpenCL-based FPGA Accelerator [80]	2016	Design space exploration for all CNN layers, Genetic algorithm, Altera OpenCL SDK	AlexNet[a]	1.46	Stratix-V[1] GXA7	(8-16)-bit fixed-point	120	7-input LUTs	OpenCL	2,560	234,720	938,880	256	37%	47%	N/A	100%	**31.8**[a1]	**4.2**×	3.3 GHz Intel i5-4590	**1.23**
			VGG-16[b]	30.9	Stratix-V[2] GSD8					2,567	262,400	1,050,000	1,963	N/A				**47.5**[b]	**2.2**×		N/A
														52%	46%	N/A	37%	**72.4**[b2]	**9.5**×		**3.79**
														N/A				**117.8**[b2]	**5.5**×		N/A
Caffeine [153], [162]	2016	Systolic array architecture, Loop unrolling, Double buffering, Pipelining, Roofline model	AlexNet[a]	1.46	Virtex7[1] VX690T	16-bit[a] fixed-point	150	6-input LUTs	C++	2,940	433,200	866,400	3,600	42%	81%	36%	78%	**354**[a1]	**9.7**×	Two-socket server each with a 6-core Intel CPU E5-2609 at 1.9GHz	**13.62**
			VGG-16[b]	31.1	Xilinx[2] KU060	8-bit[b] fixed-point	200			2,160	331,680	663,360	2,760	36%	60%	20%	4%	**266**[b,f]	**7.3**×		**10.64**
														36%	60%	20%	4%	**1,171.7**[b,f]	**29**×		**45.07**
														N/A				**165**[b,f]	**4.2**×		N/A
fpgaConvNet [165]	2016	Datapath optimization, Synchronous dataflow, Partitioning and folding, Design space exploration	LeNet-5	0.0038	Zynq XC7Z020	16-bit fixed-point	100	4-input LUTs	HLS	140	53,200	106,400	200	4.4%	18.2%	13%	1.2%	**0.48**	**0.09**×	CNP [127]	N/A
			Scene Labelling [167]	7.6528										6.5%	61%	36.6%	71.8%	**12.73**	**0.17**×	Tegra K1 GPU	**7.27**

TABLE 3. Implementation and performance summary of FPGA-based accelerators.

Technique	Year	Key Features	DL Model	Image Operations (GOP)	Platform	Precision	Frequency (MHz)	LUT Type	Design Entry	Resources				Resources Utilization				Performance (GOPS)	Speedup	Baseline	Power Efficiency (GOPS/W)
										BRAMs/ M20K	LUTs/ ALMs	FFs	DSPs	BRAMs/ M20K	LUTs/ ALMs	FFs	DSPs				
Throughput-Optimized FPGA Accelerator [170]	2017	Four-levels of parallelism, Memory-cache mechanism, Design space exploration	LeNet	0.04	Virtex7 VX690T	(8-16)-bit fixed-point	100	6-input LUTs	N/A	1,470	433,200	866,400	3,600	32.4%	53.8%	35.5%	80.8%	**424.7**	**14.84**×	4.0 GHz Intel Core i7-4790K CPU	**16.85**
			AlexNet	1.46										69.3%	47.7%	37.3%	79.8%	**445.6**	**6.96**×		**17.97**
			VGG-S	5.29										88.8%	51.7%	34.5%	81.9%	**473.4**	**4.79**×		**18.49**
FP-DNN [171]	2017	RTL-HLS hybrid compiler, Hand-written matrix multiply, Quantization, Tiling and double buffers	VGG-19	39.26	Stratix-V GSMD5	16-bit fixed-point	150	4-input LUTs	C++ and OpenCL	2,014	172,600	690,000	1,590	48%	27%	N/A	66%	**364.36**	**3.06**×	2 Processors each 2.6 GHz Intel Xeon 8-core E5-2650v2	**14.57**
			ResNet-152	22.62										48%	27%	N/A	66%	**226.47**	**1.9**×		**9.06**
FINN [181]	2017	Binarized CNN, Pipelining, Automatic partitioning and tiling, Approximate arithmetic	CNV	0.113	Zynq XC7Z045	(1-2)-bit precision	200	4-input LUTs	C++	545	218,600	437,200	900	34.1%	21.2%	N/A	N/A	**2,465.5**	**13.8**×	Microsoft Specialized CNN Accelerator [152]	**210.7**
Customized CONV Loops Accelerator [83]	2017	Numerical analysis of CONV loops opt., Solution space exploration, Dataflow opt.	VGG-16	30.95	Arria 10 GX 1150	(8-16)-bit float-point	150	8-input LUTs	Verilog	2,713	427,200	1,708,800	1,518	70%	38%	N/A	100%	**645.25**	**3.2**×	Energy-efficient CNN [205]	**30.44**
																			5.5×	OpenCL-based FPGA Accelerator [80][a2]	
Latency-Driven Design for FPGA-based CNNs [183]	2017	Synchronous dataflow, Weights reloading and SDF transformations, Design space exploration	AlexNet	1.3315	Zynq XC7Z045	16-bit fixed-point	125	4-input LUTs	HLS	545	218,600	437,200	900	N/A				**161.98**	**1.49**×	DeepBurning [155]	N/A
			VGG-16	30.72														**123.12**	**0.65**×	Embedded FPGA Accelerator [98]	
DLA [188]	2017	Winograd transformations, Double stream buffers, PEs double cache buffers, Daisy-chained PEs	AlexNet	1.46	Arria 10 GX 1150	Half-precision FP16 with shared exponent	303	8-input LUTs	OpenCL	2,713	427,200	1,708,800	1,518	92%	58%	40%	97%	**1,382**	**8.4**×	Caffeine [162][a2n]	**30.71**
																			19.1×	OpenCL-based FPGA Accelerator [80][a2]	
Winograd-based CNN Accelerator [189]	2017	Winograd algorithm, Loop unrolling, Double buffers, Batching, Line buffers, Design space exploration	AlexNet[a]	1.33	Zynq[1] ZCU102	16-bit fixed-point	200	6-input LUTs	C	912	274,080	548,160	2,520	N/A				**854.6**[a1]	**11.8**×	[80][a2]	**36.2**
					Xilinx[2] ZC706													**2,940.7**[f,f3]	**8.31**×	Caffeine [162][f,f3]*	**124.6**
			VGG-16[b]	30.76			167	4-input LUTs		545	218,600	437,200	900					**201.4**[f,f3]	**2.78**×	[80][f,f3]	**21.4**
																		679.6[b,f]	**0.13**×	Titan X (CuDNN 5.1)	**72.3**
OpenCL-based Architecture for Accelerating CNNs [190]	2017	2D BRAM-to-PE interconnection, 2D dispatcher, Roofline model, OpenCL	VGG-16	30.76	Arria 10 GX 1150	32-bit float-point	370	8-input LUTs	OpenCL	2,713	427,200	1,708,800	1,518	46.1%	N/A	N/A	87%	**866**	**4.41**×	Altera OpenCL [191] on Arria 10 Platform	**20.75**
						16-bit fixed-point	385			2,713	427,200	1,708,800	3,036	53.4%	N/A	N/A	90.8%	**1,790**	N/A		**47.78**
Multi-CLP Accelerator for CNNs [192]	2017	Multiple CONV processors, Pipelining, Dynamic programming, Double-buffering	AlexNet[a]	1.33	Virtex7[1] VX485T	32-bit float-point	100	4-input LUTs	C++	2,060	303,600	607,200	2,800	39.4%	58.3%	44.6%	87.3%	**85.2**[a1]	**1.31**×	Single CLP Design Based in [55]	**11.2**
			SqueezeNet[b]	0.77	Virtex7[2] VX690T	16-bit fixed-point	170	6-input LUTs		2,940	433,200	866,400	3,600	48.8%	84.7%	40.2%	88.3%	**11.02**[a2,f]	**1.54**×		**11.17**
														N/A				**708.3**[f,f]	**1.9**×		N/A
														77.7%	50.9%	18.6%	97.1%	**909.7**[f2,f]	**2.3**×		**126.3**
Automated Systolic Array Architecture for CNN [195]	2017	2D systolic array architecture, Roofline model, Automation flow Design space exploration	AlexNet[a]	1.4	Arria 10 GX 1150	32-bit[1] float-point (8-16)-bit[2,f] fixed-point	239.62[a1] 221.65[b,f] 231.85[b,f]	8-input LUTs	OpenCL	2,713	427,200	1,708,800	1,518	87%	82%	N/A	85%	**360.4**[a1]	N/A		**20.75**
			VGG-16[b]	31.1										90.3%	83%	N/A	88.3%	**460.8**[b,f]			N/A
														61.5%	73%	N/A	98.8%	**1,171.3**[b,f]			
End-to-End Scalable FPGA Accelerator [196]	2017	Flexible and scalable ResNet modules, CONV loops opt., Dataflow opt., Controlled execution flow of ResNets layers	ResNet-50	7.74	Arria 10 GX 1150	16-bit fixed-point	150	8-input LUTs	Verilog	2,713	427,200	1,708,800	1,518	80%	30%	N/A	69%	**285.07**	N/A		N/A
			ResNet-152	22.62										93%	33%	N/A	69%	**315.48**			
DLAU [197]	2017	Pipelined processing units, Tiling, FIFO buffers	DNN	N/A	Zynq XC7Z020	48-bit float-point	200	4-input LUTs	N/A	280	53,200	106,400	220	12.5%	68.4%	26.6%	75.9%	N/A	**36.1**×	2.3 GHz Intel Core2	N/A
An Automatic RTL Compiler for High-Throughput Deep CNNs [199]	2017	Library-based RTL compiler, Flexible and scalable CNN modules, Layer combo computation, CONV loops and dataflow opt., Controlled execution flow of CNN layers	NiN[a]	2.2	Stratix-V[1] GXA7	16-bit fixed-point	150	7-input LUTs	Verilog	2,560	234,720	938,880	256	59%	96%	N/A	100%	**282.67**[a1]	>**6.4**×	DeepBurning [155]	N/A
														86%	90%	N/A	100%	**352.24**[f,f]		N/A	
			VGG-16[b]	30.95										76%	74%	N/A	100%	**250.75**[f,f]		N/A	
														95%	77%	N/A	100%	**278.67**[b,f]	**1.23**×	FP-DNN [171]	
			ResNet-50[c]	7.74	Arria 10[2] GX 1150		200	8-input LUTs		2,713	427,200	1,708,800	1,518	56%	37%	N/A	100%	**587.63**[f,f2]		N/A	
														82%	50%	N/A	100%	**720.15**[f2,f]	**2.0**×	Caffeine [162][b,f,*]	
			ResNet-152[d]	22.62										71%	51%	N/A	100%	**619.13**[f2,f]		N/A	
														87%	54%	N/A	100%	**710.30**[b,f]		N/A	
ALAMO [78], [168]	2018	Modularized RTL compiler, Loop unrolling, Loop tiling	AlexNet	1.46	Stratix-V GXA7	(8-16)-bit fixed-point	100	7-input LUTs	Verilog	2,560	234,720	938,880	256	61%	52%	N/A	100%	**114.5**	**1.9**×	Roofline-based FPGA Accelerator [55]	**5.87**
			NiN	2.2										91%	48%	N/A	100%	**117.3**		N/A	**6.14**
Angel-Eye [189]	2018	Quantization, Parallel PEs, Compiler, On-chip buffers	VGG-16	30.76	Zynq XC7Z020	8-bit fixed-point	214	4-input LUTs	N/A	140	53,200	106,400	220	61%	56%	33%	86.4%	**84.3**	**3.8**×	nn-X [148]	**24.1**
					Zynq XC7Z045	16-bit fixed-point	150			545	218,600	437,200	900	89%	84%	29%	87%	**137**	6×		**14.2**
Optimizing the CONV Operation to Accelerate DNNs on FPGA [204]	2018	Scalable set of data buses (BUF2PE), Optimized CONV module for different (kernel, stride) size configurations, Flexible and scalable CNN modules, CONV loops and dataflow opt.	NiN[a]	2.2	Stratix-V[1] GXA7	16-bit fixed-point	150	7-input LUTs	Verilog	2,560	234,720	938,880	256	59%	97%	N/A	100%	**278.2**[a1]		N/A	N/A
														86%	93%	N/A	100%	**348.8**[b,f]		N/A	
			VGG-16[b]	30.95										76%	75%	N/A	100%	**243.1**[f,f]		N/A	
														95%	78%	N/A	100%	**276.6**[b,f]	**1.2**×	FP-DNN [171]	
			ResNet-50[c]	7.74	Arria 10[2] GX 1150		200	8-input LUTs		2,713	427,200	1,708,800	1,518	56%	38%	N/A	100%	**584.8**[f,f2]		N/A	
														82%	52%	N/A	100%	**715.9**[b,f]		N/A	
			ResNet-152[d]	22.62										71%	52%	N/A	100%	**611.4**[f,f]		N/A	
														87%	55%	N/A	100%	**707.2**[b,f]		N/A	

this consists of determining the number of convolution layers, number of fully connected layers, sizes of feature maps in each layer, along with other operators. Recent research has demonstrated that a large number of weights in fully connected layers could be eliminated with minimal impact on accuracy. In addition, although the suggested CNN structures

by experts perform well for various applications, the question arises whether the suggested structures could be optimized for performance with minimal impact on accuracy. Since the designed CNN has a significant impact on the complexity of its implementation, we review in this section some approaches attempting to optimize the design of CNNs using metaheuristics.

NP-hard combinatorial optimization problems [206] appear in the design of CNNs. Some examples of areas include design of CNN structures, selection of weights and bias values to improve accuracy, and determination of optimal values of variables to reduce run-time. Below, we briefly touch upon some existing literature in these areas.

A. CNN STRUCTURE OPTIMIZATION

In the design of CNNs, the number of possible network structures increases exponentially with the number of layers. Xie and Yuille [207] used genetic algorithm in learning deep network structures. The objective was to find the best CNN structure that would minimize the error rate. The cost function was the CNN accuracy. They proposed an elegant encoding of chromosome using a fixed length binary string to represent each network structure. A CNN string represents only the convolution layers.

In each generation, using standard genetic operations new individuals are generated and weak ones eliminated. The quality of an individual was assessed by its *recognition accuracy* which is obtained via the time consuming operation of training the network, and evaluating it on a validation set. Two small data sets were used (MNIST and CIFAR-10) to run the genetic implementation via which they demonstrated the discovery of new structures.

B. CNN WEIGHTS AND BIAS VALUES OPTIMIZATION

An attempt to train CNNs using metaheuristics (that is, determine weights and bias values) is presented in [208]. The objective again was to improve accuracy and minimize the estimated error. The authors experiment with three metaheuristic algorithms, namely; simulated annealing, differential evolution, and harmony search. The algorithms compute the values of weights and bias in the last layer. These values are used as the solution vector denoted by x which is to be optimized. The move comprised adding a small value of Δx to perturb the state. The cost function y is modeled as

$$y = \frac{1}{2}\left(\frac{\sum_{i=n}^{N}(o-u)^2}{N}\right)^{0.5} \tag{4}$$

where, o is the expected output, u is the real output, and N is the number of used samples. The stopping criterion is when the iteration count is reached or when the cost function goes below a pre-specified value.

C. CNN DESIGN VARIABLES OPTIMIZATION

Suda *et al.* [80] presented a systematic methodology for design space exploration with the objective of maximizing the throughput of an OpenCL-based FPGA accelerator for a given CNN model (please see subsection III-C). FPGA resource constraints such as on-chip memory, registers, computational resources and external memory bandwidth are considered. The optimization problem comprises finding the best combination of N_{CONV}, S_{CONV}, N_{NORM}, N_{POOL}, and N_{FC} variables, where

- N_{CONV} is size of the filter (or neuron or kernel);
- S_{CONV} is the factor by which computational resources are vectorized to execute in a single-instruction stream multiple-data streams (SIMD) fashion;
- N_{NORM} represents the number of normalization operations performed in a single cycle;
- N_{POOL} is the number of parallel outputs of the pooling layer in a single cycle to achieve acceleration; and,
- N_{FC} is the number of parallel multiply and accumulate (MAC) operations preformed in a single work-item within the fully connected layer.

The objective function to be minimized is the run-time (RT), and is given by

$$\sum_{i=0}^{TL} RT_i[N_{CONV}, S_{CONV}, N_{NORM}, N_{POOL}, N_{FC}] \tag{5}$$

subject to digital signal processing (DSP) slices, logic, and memory constraints, where TL represents the total number of CNN layers including the repeated layers. The convolution layer run-time (RT_{CONV}) is analytically modeled as a function of design variables as

$$RT_{CONV_i} = \frac{\#\ of\ Convolution\ Ops_i}{N_{CONV} \times S_{CONV} \times Frequency} \tag{6}$$

As for the other layers, that is, normalization, pooling, and fully connected, the following general model is proposed

$$RT_{Layer_i} = \frac{\#\ of\ Layer\ Ops_i}{Unroll\ factor \times Frequency} \tag{7}$$

The above analytical models are later validated by performing full synthesis at selective points and running them on the FPGA accelerator.

Clearly, in order to determine the best values of the discussed design variables, exhaustive search, especially if the number of variables and or FPGA resources is large, is infeasible. We have to resort to iterative non-deterministic heuristics [206] such as simulated annealing, simulated evolution, tabu search, genetic algorithm, particle swarm optimization, cuckoo search, etc., or any of the modern metaheuristics, to efficiently traverse the search space to find acceptable solutions.

The proposed methodology employing genetic algorithm was demonstrated by optimizing the implementation of two representative CNNs, AlexNet and VGG, on two Altera Stratix-V FPGA platforms, DE5-Net and P395-D8 boards, both of which have different hardware resources. Peak performance is achieved for both, for the convolution operations, and for the entire CNN network.

One major issue related to use of non-deterministic iterative heuristics in the design of neural networks and

CNNs is the large amount of memory required to store the state of solution and the amount of time taken to determine the cost of the solution, be it accuracy/error estimation, run-time, or any other objective. Reasonable estimation techniques and analytical formulations are required to efficiently traverse the design space in search of efficient solutions.

V. SUMMARY AND RECOMMENDATIONS

In this section, we highlight the key features discussed in the acceleration of convolutional neural networks (CNNs) implemented on FPGAs, and provide recommendations to enhance the effectiveness of employing FPGAs in the acceleration of CNNs.

All reviewed techniques are centered around accelerating the convolution (CONV) operation as it consumes around 90% of the computational time. This is achieved by utilizing parallel multiply-accumulate operations bounded by resource limitations. In addition, careful design of data access patterns are targeted to minimize the memory bandwidth requirements utilizing internal memory structures and maximizing data reuse. This is crucial in the acceleration process due to the large memory data that needs to be accessed including feature maps (FMs) and weights. To minimize the memory footprint and to achieve effective utilization of resources, some techniques optimize the number of bits used to represent the feature maps and weights with minimal impact on accuracy. This is combined with the optimized selection of the number of fraction bits used for each layer. Other techniques optimize the number of used weights in the fully connected (FC) layers as they are memory-intensive. Coprocessors are also employed to automatically configure both the software and the hardware elements to fully exploit parallelism [100].

To optimize parallelization of convolution operations, several approaches have been attempted. Work load analysis has been tried to determine computations that can be structured as parallel streams [132]. The roofline model based accelerator uses polyhedral-based data dependence analysis to find the optimal unrolling factor for every convolutional layer [150], and to fully utilize all FPGA computational resources through loop pipelining. To optimize performance, tiled matrix multiplication is structured as a pipelined binary adder tree for performing multiplication and generating partial sums [198]. An optimization framework has been proposed by Suda *et al.* [80] who identified the key variables of the design and optimize them to maximize parallelism.

To reduce computational complexity of CONV layers and improve resource efficiency, a number of approaches such as [184], [188], and [189] utilized Winograd transformation in performing CONV operations as this reduces the computational complexity by around 50%.

To maximize throughput, several techniques such as [165], [170], and [192] have used multiple CONV layer processors (CLPs) instead of using a single CLP that is optimized for all CONV layers. This pipelines the operation of the multiple CLPs achieving layer-level parallelism which maximizes resource utilization and enhances performance in comparison to using a single CLP.

Since the computational requirement of FC layers is significantly less than that of CONV layers, to improve performance, and maximize resource utilization, a number of techniques such as [153], [162], [188], and [189] create batches by grouping different input FMs and processing them together in FC layers.

Complex access patterns and data locality are used in DeepBurning tool [155] for better data reuse. Wang *et al.* [197] explored hot spots profiling to determine the computational parts that need to be accelerated to improve the performance. Acceleration is accomplished by reducing the memory bandwidth requirements. Techniques proposed exploit data reuse to reduce off-chip memory communications. Loop transformations have also been used by reducing tiling parameters to improve data locality, and to reduce redundant communication operations to maximize the data sharing/reuse.

Efficient buffering, where the weight buffers are used to ensure the availability of CONV and FC layers' weights before their computation, as well as to overlap the transfer of FC layer weights with its computation, helps in improving performance [78], [168]. In the Catapult project, FPGA boards were integrated into data center applications and achieved speedup. Microsoft Research's Catapult utilized multi-banked input buffer and kernel weight buffer to provide an efficient buffering scheme of feature maps and weights, respectively. To minimize the off-chip memory traffic, a specialized network on-chip was designed to re-distribute the output feature maps on the multi-banked input buffer instead of transferring them to the external memory [152].

To further reduce memory footprint and bandwidth requirement, optimal fractional length for weights and feature maps in each layer are used. Singular value decomposition (SVD) has also been applied to the weight matrix of FC layer in order to reduce memory footprint at this layer [98]. Tiling techniques have been proposed where large-scale input data is partitioned into small subsets or tiles whose size is configured to leverage the trade-off between the hardware cost and the speedup [197].

Automation tools have been developed that automatically build neural networks with optimized performance [155]. They employ pre-constructed register transfer level (RTL) module library that holds hardware (including logical and arithmetic operations) and configuration scripts. DeepBurning, for example, generates the hardware description for neural network scripts. Another modularized RTL compiler, ALAMO, integrates both the RTL finer level optimization and the flexibility of high-level synthesis (HLS) to generate efficient Verilog parameterized RTL scripts for ASIC or FPGA platform under the available number of parallel computing resources (i.e., the number of multipliers) [78], [168]. Acceleration is achieved by employing loop unrolling technique for CONV layer operations. Some of the reviewed techniques also help minimize the size

TABLE 4. Optimization mechanisms employed for FPGA-based acceleration of deep learning networks.

Technique	VIP [119]	CNP [127]	CONV Coprocessor Accelerator [129]	MAPLE [132]	DC-CNN [100]	NeuFlow [143]	Memory-Centric Accelerator [146]	nn-X [148]	Roofline-based FPGA Accelerator [55]	Embedded FPGA Accelerator [98]	DeepBurning [155]	OpenCL-based FPGA Accelerator [80]	Caffeine [153], [162]	fpgaConvNet [165]
Loop Unrolling									×			×	×	×
Loop Tiling	×					×	×		×	×	×		×	
Loop Interchange							×							
Pipelining	×	×	×	×	×	×	×	×	×			×	×	×
Batching													×	
Multi-CLPs														×
Fixed-Point Precision	×	×	×	×	×	×	×	×		×	×	×	×	×
Per-Layer Quantization										×				
Singular Value Decomposition										×				
Prefetching					×	×							×	
Rearranging Memory Data										×	×	×	×	
In-Memory Processing				×										
Line Buffer										×				
Double Buffering									×				×	
Approximating Non-Linear AF		×	×			×		×			×	×		×
Eliminating FC Layer			×		×			×	×					
Roofline Model									×				×	
Polyhedral Optimization									×				×	
Dynamic Programming					×									
Graph Partitioning														×

TABLE 5. Optimization mechanisms employed for FPGA-based acceleration of deep learning networks.

Technique	ALAMO [78], [168]	Throughput-Optimized FPGA Accelerator [170]	FP-DNN [171]	FINN [181]	Customized CONV Loop Accelerator [83]	Latency-Driven Design for FPGA-based CNNs [183]	DLA [188]	Winograd-based CNN Accelerator [189]	OpenCL-based Architecture for Accelerating CNNs [190]	Multi-CLP Accelerator for CNNs [192]	Automated Systolic Array Architecture for CNN [195]	End-to-End Scalable FPGA Accelerator [196]	DLAU [197]	An Automatic RTL Compiler for High-Throughput Deep CNNs [199]	Intel's DLA [200]	Angel-Eye [60]	Optimizing the CONV Operation to Accelerate DNNs on FPGA [204]
Loop Unrolling	×	×			×	×		×		×		×		×			×
Loop Tiling			×	×	×		×	×		×	×	×	×	×			×
Loop Interchange					×												×
Pipelining	×	×		×		×	×	×		×	×		×				
Input Batching										×							
FC Layer Batching			×				×	×									
Multi-CLPs		×								×						×	
Binarized CNN				×													
Fixed-Point Precision	×	×	×		×	×		×	×	×	×	×		×		×	×
Per-Layer Quantization																×	×
Prefetching	×						×	×							×		
Rearranging Memory Data									×			×					
Line Buffer								×	×								×
Double Buffering			×		×		×	×		×	×		×		×		
Padding Optimizations												×					×
Winograd Algorithm							×	×									
Approximating Non-Linear AF				×		×							×				
Roofline Model				×					×								
Polyhedral Optimization											×						
Dynamic Programming										×							
Graph Coloring			×														
Graph Partitioning						×									×		
Pattern Matching						×											

of FPGA on-chip memories to optimize energy and area usage [146], [147].

In Table 4 and Table 5, we list the optimization mechanisms utilized by each of the reviewed techniques to maximize performance and throughput of FPGA-based deep learning networks.

To enhance utilization of FPGAs in CNNs acceleration and to maximize their effectiveness, we recommend the

development of a framework that includes a user-friendly interface that allows the user to easily specify the CNN model to be accelerated. This includes specifying the CNN model parameters in terms of number of convolution layers and their sizes, and number of fully connected layers along with other intermediate operations. The specified CNN model weights will be read from a file. In addition, the user should have the option of specifying the FPGA platform that will be used for implementing the CNN accelerator and the maximum tolerable error, along with the selection of a library from a set of applications to be used for model optimization and evaluation. The framework then should perform optimizations to find the minimum number of bits that need to be used for representing the weights and feature maps and the number of fraction bits to be used for each layer. In addition, optimization of fully connected layers is performed to minimize the memory requirements. All such optimizations are carried out bounded by the maximum error specified by the user for the specified application library.

The framework should be designed based on the development of a scalable hardware architecture that works for any given FPGA platform and achieves higher speedup with the availability of higher resources. Based on the available resources, specified by the FPGA platform, the tool will perform optimizations to maximize parallelism and data reuse, given the resource limitations. The tool will then automatically generate the CNN model that will fit on the given FPGA platform and will allow the user to evaluate the performance based on the chosen application library. This will allow the user to evaluate the performance gains while evaluating different FPGA platforms with different resources. The tool should have the option to generate performance measures based on different performance metrics as selected by the user such as number of frames processed per second or number of operations performed per second. In addition, the tool will report other design metrics such as resource utilization, memory sizes and bandwidth, and power dissipation.

Furthermore, it is desired to have the option for the user to specify the desired performance for a given CNN model and have the tool perform necessary analysis and evaluation and recommend to the user candidate FPGA platforms for achieving the desired performance levels. This will require the development of reasonably accurate analytical models that will estimate the needed resources for achieving the desired performance. The user can then choose the recommended FPGA platform and perform complete evaluation to verify that the desired performance levels are met.

VI. CONCLUSION

In this paper, we reviewed recent developments in the area of acceleration of deep learning networks and, in particular, convolution neural networks (CNNs) on field programmable gate arrays (FPGAs). The paper begins with a brief overview of deep learning techniques highlighting their importance, key operations, and applications. Special emphasis is given on CNNs as they have wide applications in the area of image detection and recognition and require both CPU and memory intensive operations that can be effectively accelerated utilizing FPGA inherent ability to maximize parallelism of operations.

While the paper briefly touches upon the acceleration techniques for deep learning algorithms and CNNs from both software and hardware perspectives, the core of this article has been the review of recent techniques employed in the acceleration of CNNs on FPGAs. A thorough up-to-date review is provided that illustrates the employment of various possibilities and techniques such as exploitation of parallelism utilizing loop tiling and loop unrolling, effective use of internal memory to maximize data reuse, operation pipelining, and effective use of data sizes to minimize memory footprint, and, to optimize FPGA resource utilization.

The paper also presented the use of tools for generating register transfer level (RTL) scripts that not only help in automating the design process, but also help in exploring the design space and suggesting efficient hardware. The paper discusses the use of analytics such as: (i) work load analysis in determining the computations that can be parallelized, (ii) optimal loop unrolling factors, (iii) determining access patterns to improve data locality, etc. In addition, a brief review of the use of non-deterministic heuristics in solving NP-hard combinatorial optimization problems in the design and implementation of CNNs has been presented. Finally, the paper summarizes the key features employed by the various FPGA-based CNN acceleration techniques and provided recommendations for enhancing the effectiveness of utilizing FPGAs in CNNs acceleration.

ACKNOWLEDGMENT

The authors would like to thank King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, for all support. They would also like to thank Dr. Blair P. Bremberg and Ms. Sumaiya Hussain Sadiq for their help in professional English editing of this manuscript.

REFERENCES

[1] Y. Bengio, "Learning deep architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, 2009.
[2] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Netw.*, vol. 61, pp. 85–117, Jan. 2015.
[3] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning*, vol. 1. Cambridge, MA, USA: MIT Press, 2016.
[4] L. Zhang, S. Wang, and B. Liu, "Deep learning for sentiment analysis: A survey," in *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*. Hoboken, NJ, USA: Wiley, 2018, p. e1253.
[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, p. 533, 1986.
[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," in *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, pp. 696–699.
[7] M. A. Nielsen, *Neural Networks and Deep Learning*, vol. 25. Washington, DC, USA: Determination Press, 2015.
[8] T. Weyand, I. Kostrikov, and J. Philbin, "PlaNet—Photo geolocation with convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2016, pp. 37–55.
[9] MathWorks. (2018). *What Is Deep Learning?* [Online]. Available: https://www.mathworks.com/discovery/deep-learning.html/

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.
[11] A. Deshpande. (2018). *A Beginner's Guide To Understanding Convolutional Neural Networks*. https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/
[12] J. E. Dayhoff, *Neural Network Architectures: An Introduction*. New York, NY, USA: Van Nostrand Reinhold, 1990.
[13] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," in *The Handbook of Brain Theory and Neural Networks*, vol. 3361, no. 10. Cambridge, MA, USA: MIT Press, 1995.
[14] J. Hauswald *et al.*, "DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 27–40, 2015.
[15] J. Y.-H. Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici, "Beyond short snippets: Deep networks for video classification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 4694–4702.
[16] Y. LeCun *et al.*, "Handwritten digit recognition with a back-propagation network," in *Proc. Adv. Neural Inf. Process. Syst.*, 1990, pp. 396–404.
[17] P. Barros, S. Magg, C. Weber, and S. Wermter, "A multichannel convolutional neural network for hand posture recognition," in *Proc. Int. Conf. Artif. Neural Netw.* Cham, Switzerland: Springer 2014, pp. 403–410.
[18] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2013, pp. 6645–6649.
[19] P.-S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck, "Learning deep structured semantic models for web search using clickthrough data," in *Proc. 22nd ACM Int. Conf. Conf. Inf. Knowl. Manage.*, 2013, pp. 2333–2338.
[20] O. Abdel-Hamid, A.-R. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Trans. Audio, Speech Lang. Process.*, vol. 22, no. 10, pp. 1533–1545, Oct. 2015.
[21] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis," in *Proc. 7th Int. Conf. Document Anal. Recognit.*, Aug. 2003, pp. 958–963.
[22] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," in *Proc. AAAI*, vol. 333, 2015, pp. 2267–2273.
[23] Y. Kim. (2014). "Convolutional neural networks for sentence classification." [Online]. Available: https://arxiv.org/abs/1408.5882
[24] R. Collobert and J. Weston, "A unified architecture for natural language processing: deep neural networks with multitask learning," in *Proc. 25th Int. Conf. Mach. Learn.*, 2008, pp. 160–167.
[25] R. Sarikaya, G. E. Hinton, and A. Deoras, "Application of deep belief networks for natural language understanding," *IEEE/ACM Trans. Audio, Speech, Lang. Process.*, vol. 22, no. 4, pp. 778–784, Apr. 2014.
[26] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 1725–1732.
[27] J. Mutch and D. G. Lowe, "Multiclass object recognition with sparse, localized features," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 1, Jun. 2006, pp. 11–18.
[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[29] K. Simonyan and A. Zisserman. (2014). "Very deep convolutional networks for large-scale image recognition." [Online]. Available: https://arxiv.org/abs/1409.1556
[30] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.
[31] C. Szegedy *et al.* (Sep. 2015). "Going deeper with convolutions." [Online]. Available: https://arxiv.org/abs/1409.4842
[32] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 91–99.
[33] K. Korekado, T. Morie, O. Nomura, T. Nakano, M. Matsugu, and A. Iwata, "An image filtering processor for face/object recognition using merged/mixed analog-digital architecture," in *Symp. VLSI Circuits Dig. Tech. Papers*, 2005, pp. 220–223.
[34] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua, "A convolutional neural network cascade for face detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 5325–5334.
[35] U. Muller, J. Ben, E. Cosatto, B. Flepp, and Y. L. Cun, "Off-road obstacle avoidance through end-to-end learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2006, pp. 739–746.
[36] R. Hadsell *et al.*, "A multi-range vision strategy for autonomous offroad navigation," in *Proc. Robot. Appl. (RA)*, vol. 1, no. 7, 2007, pp. 457–463.
[37] P. Sermanet *et al.*, "A multirange architecture for collision-free off-road robot navigation," *J. Field Robot.*, vol. 26, no. 1, pp. 52–87, 2009.
[38] B. Blanco-Filgueira, D. García-Lesta, M. Fernández-Sanjurjo, V. M. Brea, and P. López. (2018). "Deep learning-based multiple object visual tracking on embedded system for IoT and mobile edge computing applications." [Online]. Available: https://arxiv.org/abs/1808.01356
[39] P. D. McNelis, *Neural Networks in Finance: Gaining Predictive Edge in the Market*. New York, NY, USA: Academic, 2005.
[40] P. J. G. Lisboa and E. C. Ifeachor, *Artificial Neural Networks in Biomedicine*. London, U.K.: Springer, 2000.
[41] P. W. Mirowski, Y. LeCun, D. Madhavan, and R. Kuzniecky, "Comparing SVM and convolutional networks for epileptic seizure prediction from intracranial EEG," in *Proc. IEEE Workshop Mach. Learn. Signal Process. (MLSP)*, Oct. 2008, pp. 244–249.
[42] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for LVCSR using rectified linear units and dropout," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2013, pp. 8609–8613.
[43] R. Hadsell *et al.*, "Learning long-range vision for autonomous off-road driving," *J. Field Robot.*, vol. 26, no. 2, pp. 120–144, Feb. 2009.
[44] L. Deng and D. Yu, "Deep learning: Methods and applications," *Found. Trends Signal Process.*, vol. 7, nos. 3–4, pp. 197–387, Jun. 2014.
[45] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 580–587.
[46] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer, "SqueezeDet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving," in *Proc. CVPR Workshops*, 2017, pp. 446–454.
[47] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Comput. Vis.*, Jun. 2015, pp. 1026–1034.
[48] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2014, pp. 818–833.
[49] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.
[50] Image-Net. (2018). *The ImageNet Large Scale Visual Recognition Challenge (ILSVRC)*. [Online]. Available: http://image-net.org/challenges/LSVRC/
[51] A. Mohamed, G. E. Dahl, and G. Hinton, "Acoustic modeling using deep belief networks," *IEEE Trans. Audio, Speech, Language Process.*, vol. 20, no. 1, pp. 14–22, Jan. 2012.
[52] O. Nomura and T. Morie, "Projection-field-type VLSI convolutional neural networks using merged/mixed analog-digital approach," in *Proc. Int. Conf. Neural Inf. Process.* Berlin, Germany: Springer, 2007, pp. 1081–1090.
[53] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *Proc. OSDI*, vol. 14, 2014, pp. 571–582.
[54] Y. LeCun *et al.*, "Backpropagation applied to handwritten zip code recognition," *Neural Comput.*, vol. 1, no. 4, pp. 541–551, 1989.
[55] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2015, pp. 161–170.
[56] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural acceleration for GPU throughput processors," in *Proc. 48th Int. Symp. Microarchitecture*, 2015, pp. 482–493.
[57] G. Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Nov. 2012.
[58] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.
[59] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *Proc. IEEE 28th Int. Conf. Appl.-Specific Syst., Archit. Process. (ASAP)*, Jul. 2017, pp. 19–24.

[60] K. Guo *et al.*, "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018.
[61] E. Nurvitadhi *et al.*, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 5–14.
[62] J. Misra and I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," *Neurocomputing*, vol. 74, nos. 1–3, pp. 239–255, 2010.
[63] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2012, pp. 449–460.
[64] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 243–254.
[65] L. Du *et al.*, "A reconfigurable streaming deep convolutional neural network accelerator for Internet of Things," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 1, pp. 198–208, Jan. 2018.
[66] W. Vanderbauwhede and K. Benkrid, *High-Performance Computing Using FPGAs*. New York, NY, USA: Springer, 2013.
[67] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 13–24, 2014.
[68] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level synthesis: Productivity, performance, and software constraints," *J. Elect. Comput. Eng.*, vol. 2012, p. 1, Jan. 2012.
[69] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.
[70] A. Canis *et al.*, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. 19th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2011, pp. 33–36.
[71] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[72] R. Hameed *et al.*, "Understanding sources of inefficiency in general-purpose chips," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 37–47, Jun. 2010.
[73] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep./Oct. 2011.
[74] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 367–379, 2016.
[75] T. Serre, L. Wolf, S. Bileschi, M. Riesenhuber, and T. Poggio, "Robust object recognition with cortex-like mechanisms," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 3, pp. 411–426, Mar. 2007.
[76] P. Joshi. (2018). *What is Local Response Normalization in Convolutional Neural Networks*. [Online]. Available: https://prateekvjoshi.com/2016/04/05/what-is-local-response-normalization-in-convolutional-neural-networks/
[77] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Proc. Int. Conf. Artif. Neural Netw.* Cham, Switzerland: Springer, 2014, pp. 281–290.
[78] Y. Ma, N. Suda, Y. Cao, J.-S. Seo, and S. Vrudhula, "Scalable and modularized RTL compilation of convolutional neural networks onto FPGA," in *Proc. 26th Int. Conf. Field Program. Logic Appl. (FPL)*, Aug. 2016, pp. 1–8.
[79] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, Dec. 1994.
[80] N. Suda *et al.*, "Throughput-optimized opencl-based FPGA accelerator for large-scale convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2016, pp. 16–25.
[81] M. Denil, B. Shakibi, L. Dinh, and N. De Freitas, "Predicting parameters in deep learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 2148–2156.
[82] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. 27th Int. Conf. Mach. Learn. (ICML)*, 2010, pp. 807–814.
[83] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 45–54.
[84] A. Karpathy. (2018). *Convolutional Neural Networks for Visual Recognition*. [Online]. Available: http://cs231n.github.io/convolutional-networks/
[85] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2016, pp. 630–645.
[86] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proc. AAAI*, vol. 4, 2017, p. 12.
[87] J. Villasenor and W. H. Mangione-Smith, "Configurable computing," *Sci. Amer.*, vol. 276, no. 6, pp. 66–71, 1997.
[88] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*, vol. 180. Boston, MA, USA: Springer, 2012.
[89] M. C. Herbordt, Y. Gu, T. VanCourt, J. Model, B. Sukhwani, and M. Chiu, "Computing models for FPGA-based accelerators," *Comput. Sci. Eng.*, vol. 10, no. 6, pp. 35–45, Nov. 2008.
[90] B. S. C. Varma, K. Paul, and M. Balakrishnan, *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*. Singapore: Springer, 2016.
[91] G. Lacey, G. W. Taylor, and S. Areibi. (2016). "Deep learning on FPGAs: Past, present, and future." [Online]. Available: https://arxiv.org/abs/1602.04283
[92] C. Farabet *et al.*, "Large-scale FPGA-based convolutional networks," in *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge, U.K.: Cambridge Univ. Press, 2011, pp. 399–419.
[93] A. Munshi, "The OpenCL specification," in *Proc. IEEE Hot Chips 21 Symp. (HCS)*, Aug. 2009, pp. 1–314.
[94] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66–73, 2010.
[95] A. R. Omondi and J. C. Rajapakse, *FPGA Implementations of Neural Networks*, vol. 365. Boston, MA, USA: Springer, 2006.
[96] H. M. Waidyasooriya, M. Hariyama, and K. Uchiyama, *Design of FPGA-Based Computing Systems With OpenCL*. Cham, Switzerland: Springer, 2018.
[97] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Apr./May 2017, pp. 1–8.
[98] 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.
[99] S. Han *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 75–84.
[100] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 247–257, 2010.
[101] C. F. Van Loan, *Matrix Computations* (Johns Hopkins Studies in the Mathematical Sciences). Baltimore, MD, USA: The Johns Hopkins Univ. Press, 1996.
[102] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 1269–1277.
[103] G. Guennebaud *et al.* (2015). *Eigen V3, 2010*. [Online]. Available: http://eigen.tuxfamily.org
[104] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.
[105] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Proc. Adv. Neural Inf. Process. Syst.*, 1990, pp. 598–605.
[106] S. J. Hanson and L. Y. Pratt, "Comparing biases for minimal network construction with back-propagation," in *Proc. Adv. Neural Inf. Process. Syst.*, 1989, pp. 177–185.
[107] 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.
[108] S. Han, H. Mao, and W. J. Dally. (2015). "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." [Online]. Available: https://arxiv.org/abs/1510.00149

[109] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.
[110] Y. LeCun. (1998). *The MNIST Database of Handwritten Digits*. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[111] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*. Washington, DC, USA: IEEE Computer Society, Dec. 2014, pp. 609–622.
[112] T. Luo *et al.*, "DaDianNao: A neural network supercomputer," *IEEE Trans. Comput.*, vol. 66, no. 1, pp. 73–88, Jan. 2017.
[113] D. Liu *et al.*, "Pudiannao: A polyvalent machine learning accelerator," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 369–381, Mar. 2015.
[114] Z. Du *et al.*, "Shidiannao: Shifting vision processing closer to the sensor," *Acm Sigarch Comput. Archit. News*, vol. 43, no. 3, pp. 92–104, 2015.
[115] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.
[116] A. Shafiee *et al.*, "ISAAC: A convolutional neural network accelerator with *in-situ* analog arithmetic in crossbars," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 14–26, 2016.
[117] P. Chi *et al.*, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 27–39, 2016.
[118] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 553–564.
[119] J. Cloutier, E. Cosatto, S. Pigeon, F. R. Boyer, and P. Y. Simard, "VIP: An FPGA-based processor for image processing and neural networks," in *Proc. IEEE 5th Int. Conf. Microelectron. Neural Netw.*, Feb. 1996, pp. 330–336.
[120] D. F. Wolf, R. A. Romero, and E. Marques, "Using embedded processors in hardware models of artificial neural networks," in *Proc. Simposio Brasileiro de automação Inteligente*, Brasília, Brasil, 2001.
[121] K. R. Nichols, M. A. Moussa, and S. M. Areibi, "Feasibility of floating-point arithmetic in FPGA based artificial neural networks," in *Proc. CAINE*, 2002, pp. 8–13.
[122] K. Benkrid and S. Belkacemi, "Design and implementation of a 2D convolution core for video applications on FPGAs," in *Proc. IEEE 3rd Int. Workshop Digit. Comput. Video (DCV)*, Nov. 2002, pp. 85–92.
[123] F. Cardells-Tormo, P.-L. Molinet, J. Sempere-Agullo, L. Baldez, and M. Bautista-Palacios, "Area-efficient 2D shift-variant convolvers for FPGA-based digital image processing," in *Proc. IEEE Workshop Signal Process. Syst. Design Implement.*, Aug. 2005, pp. 209–213.
[124] R. G. Gironés, R. C. Palero, J. C. Boluda, and A. S. Cortés, "FPGA implementation of a pipelined on-line backpropagation," *J. VLSI Signal Process. Syst. Signal, Image Video Technol.*, vol. 40, no. 2, pp. 189–213, 2005.
[125] H. Zhang, M. Xia, and G. Hu, "A multiwindow partial buffering scheme for FPGA-based 2-D convolvers," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 54, no. 2, pp. 200–204, Feb. 2007.
[126] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study," *IEEE Trans. Neural Netw.*, vol. 18, no. 1, pp. 240–252, Jan. 2007.
[127] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for convolutional networks," in *Proc. IEEE Int. Conf. Field Program. Logic Appl. (FPL)*, Aug./Sep. 2009, pp. 32–37.
[128] Y. LeCun *et al.* (2015). *LeNet-5, Convolutional Neural Networks*. [Online]. Available: http://yann.lecun.com/exdb/lenet
[129] M. Sankaradas *et al.*, "A massively parallel coprocessor for convolutional neural networks," in *Proc. 20th IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2009, pp. 53–60.
[130] H. P. Graf *et al.*, "A massively parallel digital learning processor," in *Proc. Adv. Neural Inf. Process. Syst.*, 2009, pp. 529–536.
[131] S. Cadambi *et al.*, "A massively parallel FPGA-based coprocessor for support vector machines," in *Proc. 17th IEEE Symp. Field Program. Custom Comput. Mach.*, Apr. 2009, pp. 115–122.
[132] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in *Proc. ACM 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 273–284.
[133] J. C. Platt, "12 fast training of support vector machines using sequential minimal optimization," *Adv. Kernel Methods*, pp. 185–208, 1999.
[134] B. Bai *et al.*, "Learning to rank with (a lot of) word features," *Inf. Retr.*, vol. 13, no. 3, pp. 291–314, 2010.
[135] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Math. Statist. Probab.*, Oakland, CA, USA, vol. 1, 1967, pp. 281–297.
[136] A. Sato and K. Yamada, "Generalized learning vector quantization," in *Proc. Adv. Neural Inf. Process. Syst.*, 1996, pp. 423–429.
[137] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *IEEE Trans. Neural Netw.*, vol. 8, no. 1, pp. 98–113, Jan. 1997.
[138] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Proc. 10th Int. Workshop Frontiers Handwriting Recognit.*, 2006.
[139] F. Nasse, C. Thurau, and G. A. Fink, "Face detection using GPU-based convolutional neural networks," in *Proc. Int. Conf. Comput. Anal. Images Patterns*. Berlin, Germany: Springer, 2009, pp. 83–90.
[140] J. D. Dixon, "Asymptotically fast factorization of integers," *Math. Comput.*, vol. 36, no. 153, pp. 255–260, 1981.
[141] P. L. Montgomery, "A survey of modern integer factorization algorithms," *CWI Quart.*, vol. 7, no. 4, pp. 337–365, 1994.
[142] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May/Jun. 2010, pp. 257–260.
[143] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jun. 2011, pp. 109–116.
[144] R. Collobert, C. Farabet, and K. Kavukcuoglu, "Torch," in *Proc. Workshop Mach. Learn. Open Source Softw. (NIPS)*, vol. 76, 2008, p. 113.
[145] D. Grangier, L. Bottou, and R. Collobert, "Deep convolutional networks for scene parsing," in *Proc. ICML Deep Learn. Workshop*, 2009, vol. 3, no. 6, p. 109.
[146] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Oct. 2013, pp. 13–19.
[147] A. Beric, J. van Meerbergen, G. de Haan, and R. Sethuraman, "Memory-centric video processing," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 18, no. 4, pp. 439–452, Apr. 2008.
[148] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 G-OPS/S mobile coprocessor for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, Jun. 2014, pp. 682–687.
[149] C. Farabet, C. Poulet, and Y. LeCun, "An FPGA-based stream processor for embedded real-time vision with convolutional networks," in *Proc. IEEE 12th Int. Conf. Comput. Vis. Workshops (ICCV Workshops)*, Sep./Oct. 2009, pp. 878–885.
[150] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.
[151] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2013, pp. 29–38.
[152] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," Microsoft Res., Washington, DC, USA, White Paper 11, 2015, vol. 2, no. 11, pp. 1–4.
[153] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, to be published, doi: 10.1109/TCAD.2017.2785257.
[154] B. Bosi, G. Bois, and Y. Savaria, "Reconfigurable pipelined 2-D convolvers for fast digital signal processing," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 7, no. 3, pp. 299–308, Sep. 1999.
[155] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family," in *Proc. 53rd Annu. Design Autom. Conf.*, 2016, p. 110.
[156] Khronos OpenCL Working Group. (2011). *The OpenCL Specification Version 1.1*. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf
[157] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL," in *Proc. Int. Workshop OpenCL*, 2014, p. 4.

[158] Altera. (2018). *OpenCL Design Examples*. [Online]. Available: https://www.altera.com/support/support-resources/designexamples/design-software/opencl.html/
[159] Nallatech. (2018). *P395-D8 OpenCL FPGA Accelerator Cards*. [Online]. Available: http://www.nallatech.com/wp-content/uploads/openclcardspb_v1_51.pdf/
[160] Altera. (2018). *DE5-Net FPGA Kit User Manual*. [Online]. Available: ftp://ftp.altera.com/up/pub/Altera_Material/Boards/DE5/DE5_User_
[161] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proc. IEEE/ACM Conf. Supercomput. (SC)*, Nov. 1998, p. 38.
[162] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2016, pp. 1–8.
[163] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2013, pp. 9–18.
[164] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.
[165] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs," in *Proc. IEEE 24th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2016, pp. 40–47.
[166] C. R. Reeves, *Modern Heuristic Techniques for Combinatorial Problems* (Advanced Topics in Computer Science). New York, NY, USA: McGraw-Hill, 1995.
[167] L. Cavigelli, M. Magno, and L. Benini, "Accelerating real-time embedded scene labeling with convolutional networks," in *Proc. 52nd Annu. Design Autom. Conf.*, 2015, p. 108.
[168] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J.-S. Seo, "ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler," *Integration*, vol. 62, pp. 14–23, Jun. 2018.
[169] M. Lin, Q. Chen, and S. Yan. (2013). "Network in network." [Online]. Available: https://arxiv.org/abs/1312.4400
[170] Z. Liu *et al.*, "Throughput-optimized FPGA accelerator for deep convolutional neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 3, 2017, Art. no. 17.
[171] 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-Program. Custom Comput. Mach. (FCCM)*, Apr./May 2017, pp. 152–159.
[172] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. OSDI*, vol. 16. 2016, pp. 265–283.
[173] M. H. Alsuwaiyel, *Algorithms: Design Techniques And Analysis*, vol. 14. Singapore: World Scientific, 2016.
[174] S. Chetlur *et al.* (2014). "cuDNN: Efficient primitives for deep learning." [Online]. Available: https://arxiv.org/abs/1410.0759
[175] W. Zaremba, I. Sutskever, and O. Vinyals. (2014). "Recurrent neural network regularization." [Online]. Available: https://arxiv.org/abs/1409.2329
[176] W. Sung, S. Shin, and K. Hwang. (2015). "Resiliency of deep neural networks under quantization." [Online]. Available: https://arxiv.org/abs/1511.06488
[177] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.*, 2016, pp. 525–542.
[178] M. Kim and P. Smaragdis. (2016). "Bitwise neural networks." [Online]. Available: https://arxiv.org/abs/1601.06071
[179] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. (2016). "DoReFa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients." [Online]. Available: https://arxiv.org/abs/1606.06160
[180] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. (2016). "Binarized neural networks: Training deep neural networks with weights and activations constrained to $+1$ or -1." [Online]. Available: https://arxiv.org/abs/1602.02830
[181] Y. Umuroglu *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 65–74.
[182] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Univ. Toronto, Toronto, ON, Canada, Tech. Rep. 4, 2009.
[183] S. I. Venieris and C.-S. Bouganis, "Latency-driven design for FPGA-based convolutional neural networks," in *Proc. 27th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2017, pp. 1–8.
[184] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 4013–4021.
[185] S. Winograd, *Arithmetic Complexity of Computations*, vol. 33. Philadelphia, PA, USA: SIAM, 1980.
[186] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, vol. 10. Philadelphia, PA, USA: SIAM, 1992.
[187] C. Zhang and V. Prasanna, "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 35–44.
[188] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL deep learning accelerator on arria 10," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 55–64.
[189] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr./May 2017, pp. 101–108.
[190] J. Zhang and J. Li, "Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 25–34.
[191] T. S. Czajkowski *et al.*, "OpenCL for FPGAs: Prototyping a compiler," in *Proc. Int. Conf. Eng. Reconfigurable Syst. Algorithms (ERSA)*, 2012, p. 1.
[192] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 535–547.
[193] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. (2016). "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size." [Online]. Available: https://arxiv.org/abs/1602.07360
[194] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *Proc. 26th Int. Conf. Field Program. Logic Appl. (FPL)*, Aug./Sep. 2016, pp. 1–9.
[195] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. Design Autom. Conf.*, 2017, pp. 1–6.
[196] Y. Ma, M. Kim, Y. Cao, S. Vrudhula, and J.-S. Seo, "End-to-end scalable FPGA accelerator for deep residual networks," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2017, pp. 1–4.
[197] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "DLAU: A scalable deep learning accelerator unit on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 3, pp. 513–517, Mar. 2017.
[198] Altera. (2018). *JTAG UART Core*. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/hb/nios2/n2cpu_nii51009.pdf
[199] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," in *Proc. 27th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2017, pp. 1–8.
[200] M. S. Abdelfattah *et al.* (2018). "DLA: Compiler and FPGA overlay for neural network inference acceleration." [Online]. Available: https://arxiv.org/abs/1807.06434
[201] A. K. Jain, S. A. Fahmy, and D. L. Maskell, "Efficient overlay architecture based on DSP blocks," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2015, pp. 25–28.
[202] W. Liu *et al.*, "SSD: Single shot MultiBox detector," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2016, pp. 21–37.
[203] E. Chung *et al.*, "Accelerating persistent neural networks at datacenter scale," in *Proc. Hot Chips*, vol. 27, 2017.
[204] 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.
[205] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient CNN implementation on a deeply pipelined FPGA cluster," in *Proc. Int. Symp. Low Power Electron. Design*, 2016, pp. 326–331.
[206] S. M. Sait and H. Youssef, *Iterative Computer Algorithms With Applications in Engineering: Solving Combinatorial Optimization Problems*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999.
[207] L. Xie and A. Yuille, "Genetic CNN," in *Proc. ICCV*, Oct. 2017, pp. 1388–1397.
[208] L. M. R. Rere, M. I. Fanany, and A. M. Arymurthy, "Metaheuristic algorithms for convolution neural network," *Comput. Intell. Neurosci.*, vol. 2016, May 2016, Art. no. 1537325.

AHMAD SHAWAHNA received the B.Sc. degree in computer engineering from An-Najah National University, Palestine, in 2012, and the M.S. degree in computer engineering from the King Fahd University of Petroleum and Minerals (KFUPM), Saudi Arabia, in 2016, where he is currently pursuing the Ph.D. degree with the Department of Computer Engineering. He is currently with the Center for Communications and IT Research, KFUPM. His research interests include hardware accelerator, deep learning, convolutional neural networks, field-programmable gate array, wireless security, network security, the Internet of Things, and cloud computing.

SADIQ M. SAIT was born in Bengaluru, India. He received the bachelor's degree in electronics engineering from Bangalore University, in 1981, and the master's and Ph.D. degrees in electrical engineering from the King Fahd University of Petroleum and Minerals (KFUPM), in 1983 and 1987, respectively. He is currently a Professor of computer engineering and the Director of the Center for Communications and IT Research, KFUPM. He has authored over 200 research papers, has contributed chapters to technical books, and has lectured in over 25 countries. He is the principle author of two books. He is a Senior Member of the IEEE. In 1981, he received the Best Electronic Engineer Award from the Indian Institute of Electrical Engineers, Bengaluru.

AIMAN EL-MALEH received the B.Sc. degree (Hons.) in computer engineering from the King Fahd University of Petroleum and Minerals (KFUPM), in 1989, the M.A.Sc. degree in electrical engineering from the University of Victoria, Canada, in 1991, and the Ph.D. degree in electrical engineering, with dean's honor list, from McGill University, Canada, in 1995. He is currently a Professor with the Computer Engineering Department, KFUPM. He was a Member of Scientific Staff with Mentor Graphics Corporation and the Leader in design automation, from 1995 to 1998. He holds five U.S. patents. His research interests include synthesis, testing, and verification of digital systems, defect and soft-error tolerance design, VLSI design, design automation, and efficient FPGA implementations of deep learning algorithms and data compression techniques. He received the Best Paper Award for the most outstanding contribution to the field of test at the 1995 European Design and Test Conference, the Excellence in Teaching Award from KFUPM, in 2001 and 2002, in 2006 and 2007, and in 2011 and 2012, the Excellence in Advising Award from KFUPM, in 2013 and 2014 and in 2017 and 2018, the Excellence in Research Award from KFUPM, in 2010 and 2011 and in 2015 and 2016, and the First Instructional Technology Award from KFUPM, in 2009 and 2010. His paper presented at the 1995 Design Automation Conference was also nominated for the Best Paper Award.