Received February 6, 2019, accepted March 12, 2019, date of publication April 2, 2019, date of current version April 19, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2907865

MulNet: A Flexible CNN Processor With Higher Resource Utilization Efficiency for Constrained Devices

MULUKEN TADESSE HAILESELLASIE AND SYED RAFAY HASAN
Department of Electrical and Computer Engineering, Tennessee Tech University, Cookeville, TN 38505-0001, USA

Corresponding author: Muluken Tadesse Hailesellasie (mthailesel42@students.tntech.edu)

ABSTRACT Leveraging deep convolutional neural networks (DCNNs) for various application areas has become a recent inclination of many machine learning practitioners due to their impressive performance. Research trends show that the state-of-the-art networks are getting deeper and deeper and such networks have shown significant performance increase. Deeper and larger neural networks imply the increase in computational intensity and memory footprint. This is particularly a problem for inference-based applications on resource constrained computing platforms. On the other hand, field-programmable gate arrays (FPGAs) are becoming a promising choice in giving hardware solutions for most deep learning implementations due to their high-performance and low-power features. With the rapid formation of various state-of-the-art CNN architectures, a flexible CNN hardware processor that can handle different CNN architectures and yet customize itself to achieve higher resource efficiency and optimum performance is critically important. In this paper, a novel and highly flexible DCNN processor, MulNet, is proposed. MulNet can be used to process most regular state-of-the-art CNN variants aiming at maximizing resource utilization of a target device. A processing core with multiplier and without multiplier is employed to achieve that. We formulated optimum fixed-point quantization format for MulNet by analyzing layer-by-layer quantization error. We also created a power-of-2 quantization for multiplier-free (MF) processing core of MulNet. Both quantizations significantly reduced the memory space needed and the logic consumption in the target device. We utilized Xilinx Zynq SoCs to leverage the one die hybrid (CPU and FPGA) architecture. We devised a scheme that utilizes Zynq processing system (PS) for memory intensive layers and the Zynq programmable logic (PL) for computationally intensive layers. We implemented modified LeNet, CIFAR-10 full, ConvNet processor (CNP), MPCNN, and AlexNet to evaluate MulNet. Our architecture with MF processing cores shows the promising result, by saving 36%–72% on-chip memory and 10%–44% DSP48 IPs, compared to the architecture with cores implemented using the multiplier. Comparison with the state of the art showed a very promising 25–40× DSP48 and 25–29× on-chip memory reduction with up to 136.9-GOP/s performance and 88.49-GOP/s/W power efficiency. Hence, our results demonstrate that the proposed architecture can be very expedient for resource constrained devices.

INDEX TERMS DCNN, MulNet, constrained devices, hybrid embedded system.

I. INTRODUCTION

Leveraging deep convolutional neural networks (DCNN) for various application areas has become a recent inclination of many machine learning practitioners due to their impressive performance [1]. These include applications ranging from (but certainly not limited to) image processing, computer vision, automotive applications to computational biology, computational finance and natural language processing. Research trends show that the state-of-the-art networks proposed in past few years are getting deeper and deeper and such networks have shown significant performance increase [2], [3]. However, as the networks get deeper it also implies more training time, increase in computational intensity and memory-footprint. This is particularly a problem for inference-based applications on resource

The associate editor coordinating the review of this manuscript and approving it for publication was Junaid Shuja.

constrained computing platforms. With the rapid formation of various state-of-the-art convolutional neural network (CNN) architectures a flexible CNN hardware processor that can handle different CNN architectures and yet customize itself to achieve higher resource efficiency and optimum performance is critically needed. In terms of hardware platform, Field Programmable Gate Arrays (FPGAs) are becoming the dominating choice for high performance and low-power deep learning processor design [4]–[7]. FPGA technology is also in a trend of rapid advancement in the past few years due to increment in its capacity and ease of usage for the designers. FPGAs are suitable for computationally intensive algorithms resulting in a faster computational speed and higher energy efficiency compared to other hardware rivals [7]–[9]. In the past three years, a number of FPGA-based CNN processor architectures have been proposed [12]–[25]. A few of the highlights of these approaches include, parameter reduction [11], [26], binary weight quantization [20], [21], [24], optimization for power [12], memory bandwidth optimization [15], [16], [23], pipelining, parallelism and batch-based processing [10], [13], [25], computation load reduction [14], and dataflow optimization [19], [22]. Among the various tools available for implementation of CNN architecture on various FPGAs, Vivado HLS and OpenCL are the most commonly used in literature for the sake of productivity at the cost of hardware efficiency and performance [5], [8], [9]. Most of the FPGA-based architectures proposed in these literatures are tailored or optimized to a particular CNN architecture. This approach, however, is not quite effective as it requires designing a tailored CNN hardware processor for every new network. This problem is more intensified by the different number of parameters each network has (different kernel sizes, input dimensions, feature dimensions, number of layers and the number of kernels in each layer), that the hardware processor has to acclimate. Hence, a highly flexible architecture that can mold itself into the given CNN and yet achieve a higher resource utilization efficiency is critically important. In this work we are proposing a highly flexible architecture, MulNet, that can process most regular CNN variants. Additionally, as stated earlier, convolution operation is the most computationally intensive task in almost every state-of-the-art CNN. The existing literature address this problem by optimizing the processing core multiplier unit for convolution. We are also exploring to relieve this problem by utilizing binary logic relation and judicious quantization to replace the actual unit entirely with inexpensive shift register units. To the best of our knowledge none of the aforementioned references have proposed a highly flexible architecture utilizing multiplier-free (MF) processing cores by formulating their own quantization format to achieve higher resource utilization on the target devices.

In this paper, a novel and highly flexible DCNN processor architecture, MulNet, is proposed. MulNet can be used to process most regular CNN variants aiming to maximize resource utilization of a target device by creating a novel architecture with configurable number of multiply and accumulate, and pooling (MAP) cores. MAP cores with multiplier and without multiplier is employed to achieve higher resource efficiency. We formulated quantization for trained weights and feature maps by analyzing the layer-by-layer quantization error between quantized network and original 32-bit float network. We also created a power-of-2 quantization for a MF processing cores of MulNet. Both quantization significantly reduced the memory space and the logic needed in the target device. We stored weights, input image and feature maps on external DDR. Weights and input features of the layer under execution are loaded on-chip for current computation with one-time transfer of each feature pixel and maximum data reuse across all weights. We utilized Xilinx Zynq SoCs to leverage the one die hybrid (CPU and FPGA) architecture. We devised a scheme that utilizes the processing system (PS) for memory-centric layers and the programmable logic (PL) for computation-centric layers. We implemented Modified LeNet, CIFAR-10 Full, ConvNet Processor (CNP), MPCNN, and AlexNet to evaluate MulNet. Our architecture with MF processing cores shows promising results, by saving 36%-72% on-chip memory and 10%-44% DSP48 IPs, compared to the architecture with cores implemented using multiplier. Comparison with the state-of-the-art showed a very promising 25-40x DSP48 and 25-29x on-chip memory reduction with up to 136.9 GOP/sec performance and 88.49 GOP/sec/watt power efficiency. Hence, our results demonstrate that the proposed architecture can be very expedient for resource constrained devices. The following are the main contributions of this work:

- A novel and highly flexible CNN processor architecture, MulNet, is proposed that can be used to process any regular CNN variant aiming to maximize resource utilization of a target device
- Implemented hardware optimized MulNet on hybrid embedded architecture by devising hardware-friendly data loading, on-chip memory addressing and task scheduling
- Formulated and evaluated different fixed point and power-of-2 quantization formats by analyzing a layer-by-layer quantization error across different bit widths
- Demonstrated the functionality of proposed MulNet on CNP, MPCNN, Modified LeNet, CIFAR-10 and AlexNet CNNs

The rest of this paper is organized as follows: Section II presents some introductory concepts on CNN and the different layers. Section III discusses the framework we developed for design and evaluation of MulNet. Section IV addresses MulNet architecture and its data flow in detail. Section V presents evaluation of MulNet and a comparison with the state-of-the-art. Section VII discusses the strategies we devised to implement AlexNet leveraging our proposed architecture, MulNet. In Section VIII we reported MulNet's accuracy performance on different CNN benchmarks and across different datawidth. Section IX concludes the paper.

II. PRIMER ON CONVOLUTIONAL NEURAL NETWORK (CNN)

History of learning systems goes back to the time when Artificial Neural Networks (ANN) were developed to emulate human brain. Our brain is composed of millions of neurons interconnected to each other allowing us to make complex decisions in a very short time. Similar to that, ANN is composed of simple connected units called perceptron, where each perceptron is activated by real-valued inputs, computing the equation below to output 1 when the linear operation of the inputs is greater than zero, and 0 otherwise [27].

$$\sigma(x1, x2, \ldots) = \begin{cases} 1, & \text{when } w_0x_0 + w_1x_1 + \ldots + w_nx_n > 0 \\ 0, & \text{otherwise} \end{cases}$$

where x_1, $x_2 \ldots x_n$ are inputs and w_1, $w_2 \ldots w_n$ are real-valued constants called weights which relate the input with the output, shown in Figure 1.

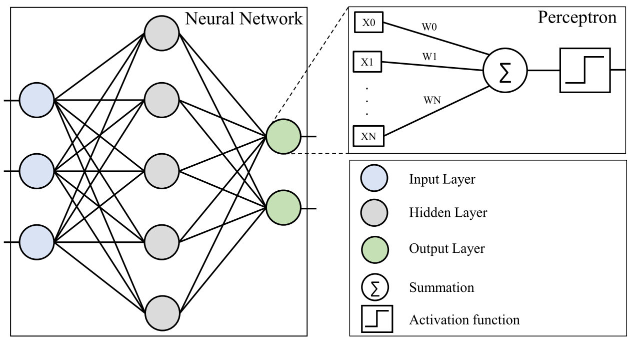

FIGURE 1. Perceptron and Artificial Neural Network Structure [27].

Basic perceptron, however, has limitation as it is not able to represent complex functions. This limitation lead to an idea of creating multi-perceptron structure called Multi-Layer Perceptron (MLP), shown in Figure 1. In MLP we have network of perceptrons divided into layers; input layer, hidden layer and output layer. Depending on the network, more than one hidden layer can be used to build the network. MLPs are able to learn very complex functions and have been used for various tasks. Training MLPs is done using back-propagation, an algorithm using gradient descent search in the network's weight space reducing the error between the network output and the expected output iteratively [27]. Researchers have shown that such small sized neural network performed well in different applications [27]. In recent years, researchers have extended this concept of conventional NN (called shallow-NN) by adding more layers and complex functions in order to learn more abstract features from the data (deep-NN or DNN). Such deep networks become popular in the past decade by out-performing the conventional shallow networks [1]. Convolutional neural network (CNN) is one of such deep neural network architectures designed to have a sequence of convolution and pooling operations (both of these operations are explained later on in this section) applied on an input data followed by fully connected layers resulting in the classification probabilities of the input data. CNN is also one of the most widely used DNN algorithm in numerous computer vision applications. Fig. 2 shows the basic convolutional neural network structure with all the above mentioned operations.

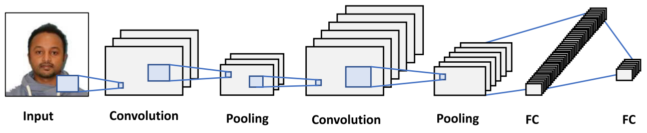

FIGURE 2. Convolutional Neural Network Structure.

A. CONVOLUTIONAL LAYER

Convolutional layer is the main layer in CNN (it is appearing twice in Figure 2), this layer performs convolution over the input image using kernels and produces an output features (distinct and useful observation grasped from the input image). Kernels also called filters or weight vectors are small sized real-valued matrix that are applied on the image to transform the information encoded in the data. Convolution operation is needed to extract the relevant information by suppressing the distracting information inside the input data. Convolutional layer is usually used in the early stages of the network. The basic operation of a convolutional layer in a feed-forward phase is shown in Figure 3. Each kernel function (weight matrix), whose dimension is represented by K, is convolved with $K \times K$ window of the input image and then added to produce one pixel of the feature map for the next layer. Each kernel function slides over the input image by stride size (the number of steps the window of the input images slides for the next convolution). This stride, dot product of $K \times K$ window of the input with $K \times K$ kernel matrix, and summation operation is repeated until the kernel function covers the whole input image. Therefore, if $N \times N$ image is convolved with $K \times K$ kernel function it results in features with dimension $(N - K + 1) \times (N - K + 1)$, for stride size of one.

B. POOLING LAYER

Convolutional layers are normally followed by pooling layers in most CNNs. Pooling layers perform down-sampling of input features. Similar to convolutional layer, the sliding window in pooling layer is also shifted by n number of rows/columns. This results in combining features by ignoring small distortions/shifts. In other words, by combining features into one, pooling layer enforces spatial invariance. Pooling layers also help in reducing the input feature dimension and resulting in less computational overhead for the following layers. The most common pooling operation is max-pooling which transforms an input feature by taking the maximum of the values in a sliding window over the input feature. In Fig. 3, a *MaxPooling* operation on a $1 \times 6 \times 6$ feature resulting in a reduced feature size $1 \times 4 \times 4$ is shown. The 3×3 green area of *MaxPooling* within the input feature in Fig. 2 is representing the sliding window.

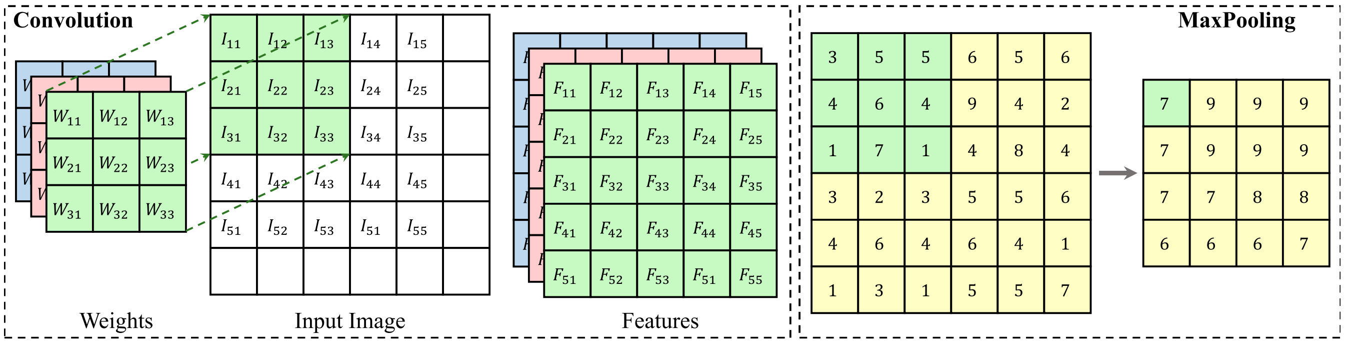

FIGURE 3. Convolutional Layer Operation for Input (7×7) with Three Kernels (3×3) Resulting in Three Features (5×5) [25] and MaxPooling Operation on a (1×6×6) Feature Resulting in a (1×4×4) Feature.

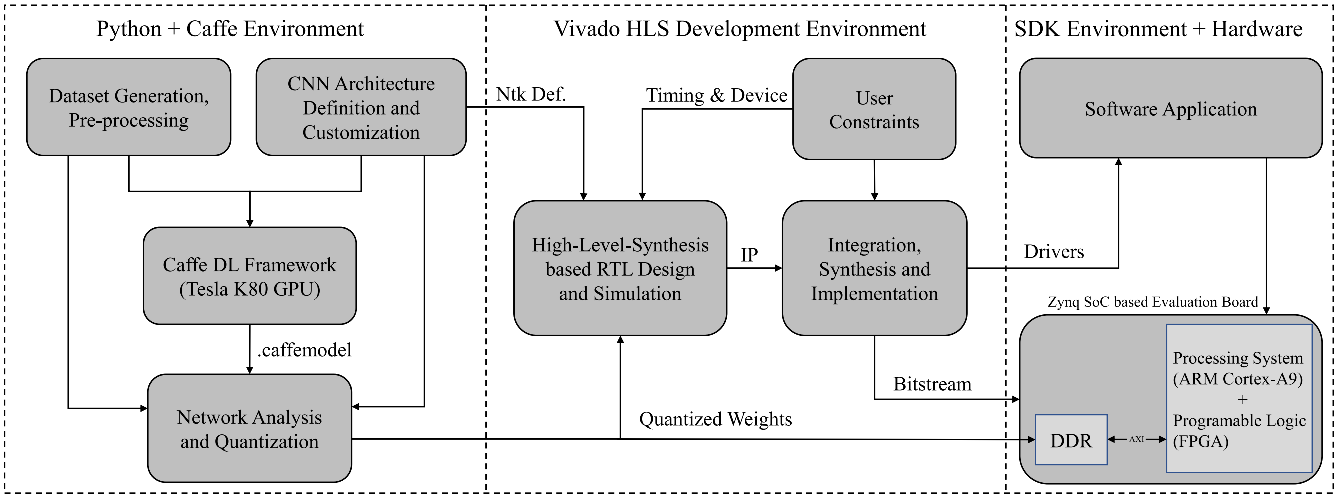
Python + Caffe Environment
Vivado HLS Development Environment
SDK Environment + Hardware
Dataset Generation, Pre-processing
CNN Architecture Definition and Customization
Caffe DL Framework (Tesla K80 GPU)
.caffemodel
Network Analysis and Quantization
Ntk Def.
Timing & Device
User Constraints
High-Level-Synthesis based RTL Design and Simulation
IP
Integration, Synthesis and Implementation
Bitstream
Quantized Weights
Drivers
Software Application
Zynq SoC based Evaluation Board
Processing System (ARM Cortex-A9) + Programable Logic (FPGA)
DDR
AXI

FIGURE 4. Our Methodology for Design and Evaluation.

C. FULLY CONNECTED(FC) LAYER

Fully connected or inner product layers are used in the last stage of CNN for classification operations. Fully connected layer performs linear transformation over the input features by applying matrix multiplication with the weight vectors.

III. FRAMEWORK OF MULNET

This work proposes a highly flexible CNN processor architecture with higher resource utilization efficiency. The methodology we developed for both design and evaluation of our proposed CNN processor is shown in Figure 4. This step includes creating a new network or utilizing existing CNN networks. Creating or collecting a dataset is for training the network is the next process. This involves collecting a large data, cleaning the data and generating the right training and validation data format. For training the network we used Caffe from Berkeley Vision and Learning Center [28] which we built on Tesla K80 GPU. Caffe is one of the early deep learning frameworks used for training deep learning networks. Since it uses a simple text-based file format to define the network and the training optimization hyper-parameters it eases the training complexity. After training Caffe returns the trained model in a file called *caffemodel*. This file contains the individual weight values that the network learned during training which are represented in 32-bit float. The last process in this stage is Network Analysis and Quantization. We developed two quantization formats. For the first one, we converted the 32-bit weight vectors into fixed point Q(I.F) format for arbitrary I and F size where I is the integer part bit width and F is the fractional part bit width. This conversion however is made based on the network analysis which performs the layer-by-layer quantization error. For the second one, we quantized the 32-bit weight vectors into power-of-2 exponent format for MF MAC operations. Both techniques are discussed in detail in Section V. All the aforementioned tasks are purely software based accomplished using Python and Caffe IDE.

The next stage of our framework is designing the hardware processor for the CNN network defined in the previous stage. The first task in this stage is register transfer level (RTL) design and simulation using Vivado HLS development environment. Vivado HLS is a tool from Xilinx that transforms a C specification in C, C++, SystemC into an RTL implementation that can synthesize into a Xilinx field programmable gate array (FPGA). Using this tool, we developed MulNet, a highly flexible CNN processor architecture both in terms of mapping various regular CNN architectures into one hardware and its capability to compute on various weight and feature quantization. A more detailed discussion on MulNet is presented in Section IV. The main challenge in creating MulNet is to make it generic enough and yet be able to tailor it to a specific network with optimum hardware consumption and faster computation time. This is particularly a challenge as different networks have different kernel sizes, input dimensions, feature dimensions, number of layers and the number of kernels in each layer. We created customizable functions that are common and can be used in most CNN variants, for instance data loading functions from DDR to on-chip memory, on-chip memory addressing functions, a MAC computing functions, data offloading function from on-chip to DDR and interfacing functions. We utilize these functions and develop a new scheduling for that particular network depending on its work load and network structure. Regarding the actual implementation, this process requires the network definition, the target hardware and a timing constraint as an input, as shown in Figure 4. The network definition, as stated above, is used to create task scheduling of a CNN network in the hardware. The target device and the timing constraint are used by Vivado HLS Synthesizer to estimate if the hardware created fits in that target device and if the hardware created meets the clock frequency specified in the timing constraint file, respectively. This is followed by verifying the functionality of the designed hardware through simulation and Co-Simulation. Vivado perform C/RTL Co-Simulation by generating RTL testbench from the C testbench for simulating the RTL and verifying if the C code produce the same result as the synthesized RTL. A successful run through these tasks lead to generating an IP core with the right interfaces for integration. The FPGAs we targeted are Zynq SoC based FPGAs. This FPGA family is based on the Xilinx All Programmable system-on-chip (AP SoC) architecture. This hybrid architecture integrates a dual-core ARM Cortex-A9 MPCore-based processing system (PS) and Xilinx programmable logic (PL) in a single device. ARM Cortex-A9 MPCore CPUs are the central hub of the PS which includes on-chip memory, external memory interfaces, and a set of I/O peripherals [35]. We leveraged this architecture by using the PS as a central coordinator for the whole computation, and memory centric tasks while using the PL for computationally intensive tasks. The Vivado HLS generated IP is then required to be integrated with the CPU cores through the processing system wrapper in Vivado Design Suite, DDR external memory and the system reset unit. This integration is done in Vivado IP Integrator. After the integration one top-level HDL wrapper is generated which needs to be synthesized and implemented using the timing and target device as constraints. This generates a bitstream that is used to configure the Zynq device.

The last stage of our methodology is application development to coordinate the whole computation using Software Development Kit (SDK) on the actual hardware, shown in Figure 4. In general, SDK provides a variety of Xilinx software packages, including drivers, libraries, BSP (board support packages), and complete operating systems for developing a software platform. When the SDK is launched, after the bitstream is generated, it starts with an auto generated BSP files for a target device, and functions to access peripherals. These functions expose the IP level function, port level interfaces and interrupt functions. In our framework we utilize these functions as follows: all the interface ports defined in our CNN processor IP are exposed using their corresponding get/set functions to write and read data from the IP in that port. Using these functions that run on the CPU cores we then write a C/C++ based application to coordinate the CNN computation which includes functions to reading and loading weight vectors and input data to DDR, to initialize the IP, to start the IP and to read the computed output from the IP.

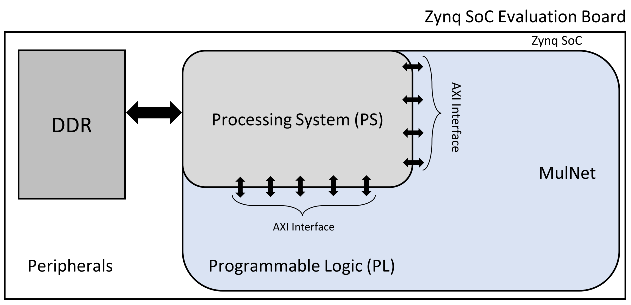

FIGURE 5. Block diagram showing PL, PS, and DDR inside Zynq SoC System. MulNet is fully implemented in PL interfaced via AXI with the other subsystems.

IV. MULNET

The framework and the MulNet architecture we presented in this work target a Zynq SoC based platform. This is depicted in Figure 5. MulNet is fully implemented using Zynq programmable logic (PL). It is interfaced with PS and external DDR through AXI4. The PS runs a C++ application that controls the start and end of MulNet computation. MulNet is designed to process CNN networks by computing the computation centric part of the network inside PL and memory centric part using PS. This is mainly because convolution operations take more than 90% of the total computation, and weight and feature maps in convolutional layers are reusable unlike fully connected layers where all weights are unique for every single multiplication. Moreover, in Zynq SoC platforms the external memory, DDR, is physically connected to PS hence memory intensive tasks could take benefit of this design by reducing memory traffic. Therefore, all convolutional, pooling and activation layers are computed in PL and all fully connected layers are computed in PS. The high-level MulNet architecture is shown in Figure 6. The core units in MulNet architecture are on-chip memory, processing cores and interfacing modules. The on-chip memory is clustered into Input Block RAM (IBRAMs), Weight Block RAM (WBRAM), Output or Feature Block RAM (OBRAM) and Bias Block RAM (BBRAM)). The processing core we called

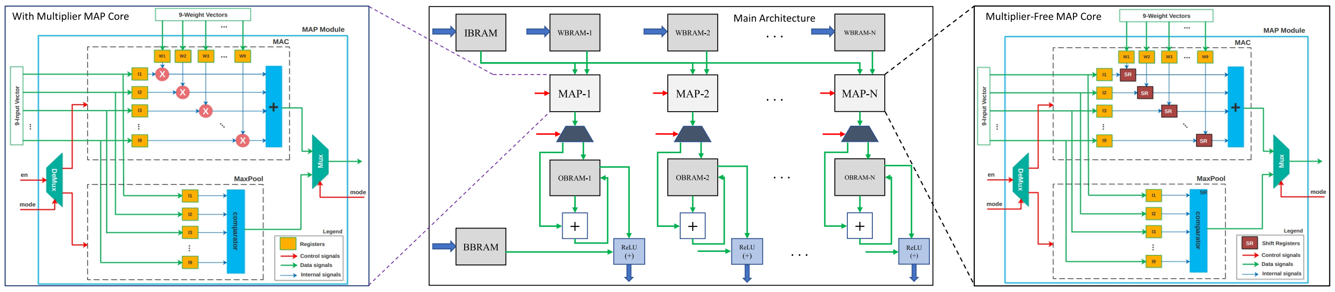

FIGURE 6. **MulNet RTL and Conceptual Architecture.**

them MAP Cores (Multiply and Accumulate, and Pooling) which compute MAC operations or max pooling operations based on the control signal asserted. Our architecture also utilizes external memory, DDR, for its computation by initially storing the quantized weight vectors and input image. As the computation continues, each layer generates intermediate feature maps, which we stored these values in DDR as well.

A. ON-CHIP MEMORY UNITS

The IBRAM is an on-chip block ram that is used to store the input image or input features. MulNet's IBRAM consumption vary in number depending on the size of the input features that needs to be loaded on-chip. A single Xilinx's block RAM consists of a 36Kb storage area two independent access ports [34]. The total number of on-chip memory required for IBRAM, hence can be estimated as follows:

$$\#IBRAM = I.Quant.bitwidth * Words/36K \qquad (1)$$

$$Words = MAX(\forall_{layers}(feature\_size)) \qquad (2)$$

where *Words* imply the maximum layer feature size in the network and *I.Quant.bitwidth* is feature map bitwidth. Hence, we allocated the on-chip memory for the size of the largest layer in-terms of feature size where by all other layer's features can be accommodated.

The WBRAM is a cluster of block RAMs that is used to store quantized weights of the network. Similarly, number of WBRAM consumption depends on the number of weight vectors a layer has. For MulNet most of the available block rams on the target device are consumed for weight storing. This is partly because our scheme of storing all weights of a layer on-chip and the memory partitioning applied for concurrent reading of weights for parallel convolution operation. The total number of on-chip memory required for WBRAM, hence can be estimated as follows:

$$WBRAM = W.Quant.bitwidth * Words/36K \qquad (3)$$

$$Words = MAX(\forall_{layers}(weight\_size)) \qquad (4)$$

where *Words* imply the maximum number of weights of a single layer in a network and *W.Quant.bitwidth* is feature map bitwidth. Hence, we allocated the weight on-chip memory to the size of the layer with maximum number of weight where by all other layer's weights can be accommodated.

Similarly, OBRAM and BBRAM are on-chip block RAMs used in our architecture to store output feature maps and bias values of a particular layer, respectively.

B. PROCESSING CORES - MAP CORES

A single MAP unit, shown in Figure 6, functions as a MAC (Multiply and Accumulate) operator or as a Pooling operator depending on the mode bit asserted. It is worth mentioning here that Figure 6 shows two types of implementation of MAP unit, cores implemented using multiplier and MF implementation. Each MAP unit can be implemented with anyone of these two ways. As a MAC module it computes the basic multiply and accumulate operation of a $K \times K$ convolution and as a pooling operator it computes a max pool operation of a $K \times K$ receptive area of the input features. A single MAP unit, as a MAC operator, can take up to K parallel inputs and K weight vectors to produce a single output using the dot product. For a MAP cores implemented with multipliers, the dot product is computed by performing a multiplication operation between each i^{th} weight from $K \times K$ weight arrays and each i^{th} pixel from the $K \times K$ feature map arrays. However, with MF MAP cores, we are shifting each i^{th} pixel by the magnitude of the power-of-2 exponent fed to the cores as weight vectors. The sign of the weight is identified based on the sign of the power-of-2 exponent which are appended artificially. The later approach results in a higher resource utilization efficiency, discussed in Section VI, and brings substantial memory space savings. The dimensions of the input vectors for the MAC can be adjusted to any $K \times K$ convolution. The networks we chose to demonstrate the functionality of our architecture on has a 7×7, 6×6 and 5×5 kernel dimensions. For the sake of simplicity, Figure 6 illustrates a 3×3 convolving MAP cores only. On the other hand, as a Pool operator MAP cores can compute MaxPool by comparing and selecting the maximum value from up to K parallel input vectors, given the mode bit is set to Pool. As shown in Figure 6, the number of MAP cores, N, is configurable. This helps to increase the number of MAP cores computing in parallel which is only limited by the resources of the target device. This is also one of the features that makes MulNet highly flexible to fit in a small or large target device.

C. INTERFACING WITH SUBSYSTEMS

Our processor is interfaced with subsystems inside the Zynq system based on efficient industry standard Advanced eXtensible Interface (AXI) connections. This has been made handy in Vivado HLS as the arguments of the top-level function are synthesized into the IP's external interfaces. Among the three-port level AXI4 interfaces supported by Vivado HLS (AXI4-Stream (axis), AXI4-Lite (s_axilite), and AXI4 master (m_axi) interfaces) we utilized AXI master as a main data transfer interface between external DDR and on-chip memory, and AXI4-Lite is used for transferring configuration data from the PS to MulNet IP.

D. SCHEDULING TECHNIQUE

Before MulNet starts computing, quantized and pre-arranged weight vectors and input images are stored in DDR. The initial memory size in bytes can be obtained if the input image dimension, N, is known using Equation 5. Similarly, total weight vector bytes can be calculated using Equation 6 by multiplying the total number of individual weights, #weight, with the bitwidth.

$$Input\_memory\_req : N^2 * \frac{bitwidth}{8} \tag{5}$$

$$Weight\_mem\_req : \#weights * \frac{bitwidth}{8} \tag{6}$$

This calculation is used for offset calculation for the right indexing of data from DDR. On the other hand, binary format of each weight vector and input image are generated and re-arranged in channel-major fashion before being loaded to DDR. Channel-major arranges weights or feature maps in all channels first and all rows second manner, this provides an opportunity to compute multiple convolutions at a time and/or without pulling new weight vectors from external memory. All these data loading functions to DDR are executed from the PS.

Figure 7 shows the task scheduling of MulNet once it gets started from the processing system. MulNet has two interfaces, AXI4 interface for data loading from DDR to on-chip and vice versa, and AXI4-Lite interface for configuring MulNet from PS. As shown in Figure 7, the first task is configuring MulNet modules which includes Weight Module, Feature Module, Input Module and MAP Module represented as W, F, I and M, respectively. The configuration information contains all the layer specific data which includes input dimension, number of input channels, number of output channels, kernel dimension, stride size, number of padding, layer type and memory addressing offsets. All these data are packed using DATA_PACK directive feature of Vivado HLS. It packs all the configuration elements into one wide vector and allows simultaneous read and write on all elements. For instance configuration data for the first convolutional layer of CIFAR10 network looks like (32, 1, 20, 5, 1, 0, 0, 520, 15680, 1024, 0, 2264), which corresponds to input dimension, number of input channels, number of output channels, kernel dimension, stride size, number of padding, layer type, number of weights, weight memory offset address, feature memory offset address, and output memory offset address, respectively.

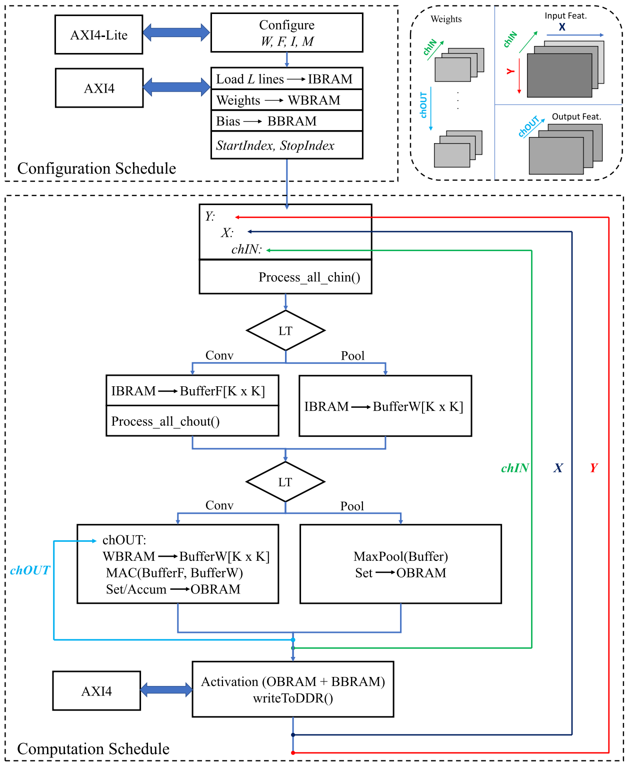

FIGURE 7. MulNet Task Schedule for Convolution and Pooling Layers. W, F, I, M, and LT represents Weight Module, Feature Module, Input Module and MAP Module and Layer Type, Respectively.

Once the modules are configured to the currently executing layer parameters, the next step is loading the input data and weights from DDR to on-chip memory via the AXI4 interface of the MulNet. The scheme we created transfers all input feature maps and weight vectors of a specific layer to on-chip memory before starting computation of that layer. This approach works for small to medium sized networks but when the network gets larger and the number of weights and input features of a layer cannot fully fit over on-chip memory. For that we devised another scheme where we load L lines (rows + channels) of the input features at a time. The number of lines to be loaded at a time, L, is configurable per layer basis. Following a layer-specific start and stop index is calculated as shown in Algorithm 1. These indexes are memory addresses in IBRAM corresponding to the first and the last $K \times K$ convolution features in that layer. Algorithm 1 shows how both start and stop indexes are calculated.

An alternative way of reading the current $K \times K$ feature map is by calculating (x * y * ci) to read pixel value at (x, y, ci) and iterating through K^2 pixels by calculating (x+i * y+j * ci+p) where i and j are between [0, $K - 1$], and p ranging from the first to last channel of the feature map.

Algorithm 1 IBRAM Indexing for 5 × 5 Convolution

```
kernel size = kernelDim
feature size = inputDim = N
number of input channels = chIn
chInTimesN = chIn * N
if noPadding then
    startIndex = ceil(kernelDim/2) - 1
    stopIndex = (inputDim - 1) - startIndex
end
else if withPadding then
    startIndex = ceil(kernelDim/2) - pad - 1
    stopIndex = (inputDim - 1) - startIndex
end
// In Figure 8: (left) (X, Y) = (2, 2) and right (1, 1)
centerPixelAdd = X*chInTimesN + (chIn*Y) + chIn
for 0:2 do
    $x' = X + i - 1$;
    for 0:2 do
        $y' = Y + j - 1$;
        //Calculated using adders and shifters only
        addIBRAM=($x'$*chInTimesN) + (chIn*$y'$) + chIn
        if noPadding then
            pixelValue = IBRAM(addIBRAM)
        end
        else if withPadding then
            if $x' < 0 | x' > N | y' < 0 | y' > N$ then
                pixelValue = 0
            end
            else
                pixelValue = IBRAM(addIBRAM)
            end
        end
    end
end
```

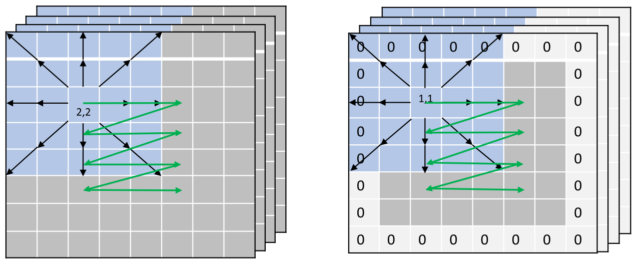

FIGURE 8. Input Feature Indexing with and without Padding from IBRAM for Less Hardware Consumption.

However, executing this calculation for every pixel in parallel consumes a lot of multiplier and is not hardware friendly. For instance, for a 5 × 5 kernel, 2*25 multipliers are required to read all 25 pixel values. Hence, we created the scheme shown in Figure 8. In our scheme we only calculate the central pixel (for instance in a 5 × 5 convolution, (2,2) is the central pixel) once, which is derived from the start and stop index, and use adders and shifters to calculate all the neighboring pixels as shown in Figure 8 by black arrows. The pseduo code for this address calculation is shown in Algorithm 1. To complete the whole convolution, this indexing iterates over the whole feature map as shown in the green line in Figure 8. The indexing variation when the layer's input features is zero padded is also shown both in Algorithm 1 and Figure 8.

The next step in the task scheduler of Figure 7 is to compute convolution or pooling based on the layer type. For convolutional layers, a $K \times K$ feature is pulled from IBRAM and get convolved with *chOUT* number of $K \times K$ weight vectors repeated across each input channel. The main advantage of this approach is it maximizes computational efficiency, as all the *chOUT* convolutions can be computed concurrently with no data dependency between each other. Moreover, each $K \times K$ feature is accessed just once for all convolution operations involving it. This saves a number of clock cycles both from the memory access and convolution computation concurrency. The MAC result for each output channel then is stored in OBRAM. This process continues across each input channel where each *chOUT* sum is accumulated on the previous one and written back to OBRAM after the summation. The RTL for this computation is shown in Figure 6, Main Architecture. Depending on the current computing layer architecture, pixel wise activation is applied. MulNet has the two most common activation functions, Sigmoid and ReLU, implemented. Sigmoid is implemented using piecewise linear approximation with the whole function taking a few hundred LUTs and flops based on the technique presented in [29]. ReLU is implemented using simple comparators. Once all the convolution for each *chOUT* number weight vectors across each input channel, *chIN*, is computed, the data in BRAM is offloaded back to DDR to the right offset address through AXI4 interface of MulNet. This data flow continues until all the convolution operations of the layer are completed, i.e. across *chIN*, *X*, *Y* as shown in Figure 7 where *X*, *Y* are the dimensions of each feature map and *chIN* is the number of individual feature maps. In our proposed technique the next layer starts execution taking the output of the previous feature map as an input. Pooling is computed in a similar fashion as shown in Figure 7.

V. QUANTIZATION

In this work, we explored two quantization formats. The first one is Q(I.F) fixed-point format where I is the integer part bit width and F is the fractional part bit width, and the second one is power-of-2 quantization format. For the former one, as depicted in Figure 4, we quantized the trained weights into 8, 16 and 32 bits as follows:

$$decimal\_value = float\_value * 2^{-F} \quad (7)$$

$$rounded\_dec = round(decimal\_value) \quad (8)$$

$$binary\_value = dec2bin(rounded\_dec) \quad (9)$$

where F is the number of fractional digits in QI.F fixed-point format, *float_value* is the actual weight value, *dec2bin* is a decimal to binary converter function and *binary_value* is the final quantized weight value that we stored in memory for computation.

For power-of-2 quantization we converted the trained weight values into the nearest power-of-2 values, and then calculated the power-of-2 exponent that we stored in memory. This exponent is later used in our MAP cores to know how many times a feature value needs to be shifted. For all the cases a signed 8-bit is enough to store the exponents. This quantization allows such convolution computation using MF processing cores in hardware. The resource utilization efficiency of the two quantization techniques is compared in Figure 9. We calculated Power-of-2 quantization error based on 32-bit weight distribution and the quantized weight distribution percentage difference for each layer as follows:

$$w\_perc\_diff = \frac{\sum \frac{(\forall(origW - quantizedW))}{\forall origW}}{num.of.weights} \tag{10}$$

$$std\_dev = \sqrt{\frac{\sum(quantizedW - w\_perc\_diff)^2}{num.of.weights}} \tag{11}$$

where *origW* is originally trained 32-bit float weight values, *quantizedW* is the corresponding quantized weight values and *num.of.weights* is number of weights in that layer.

We plotted our quantization error and averaged out the percentage value. Our technique provides an average of 3.5% error with a 3 σ of 11. The reason we are only using a power-of-2 bit width (8, 16 and 32) is that AXI4 only supports 32, 64, 128, 256, 512, or 1024 data bits and AXI4-Lite only supports 32 bits and 64 bits. Hence the bit width should always be a power of two with a support of data width conversion that doesn't match the internal crossbar between our AXI master MulNet IP to any memory-mapped slave.

TABLE 1. Workload of CNP, modified LeNet, MPCNN, and CIFAR10 convolutional neural networks.

Workload	Layers	Dimension	Kernel	Num. of MAC	Num. Weights	Model Size (KB)
Modified LeNet [26]	Conv1	1×28×28	5×5	288000	520	102.28
	Pool1	20×24×24	2×2	-	-	
	Conv2	20×12×12	5×5	1600000	25050	
	Pool2	50×8×8	2×2	-	-	
CIFAR-10 [31]	Conv1	1×32×32	5×5	81920	2432	317.056
	Pool1	32×32×32	3×3	-	-	
	Conv2	32×16×16	5×5	6553600	25568	
	Pool2	32×16×16	3×3	-	-	
	Conv3	32×8×8	5×5	3276800	51264	
	Pool3	64×8×8	3×3	-	-	
CNP [30]	Conv1	1×42×42	7×7	381024	300	16.24
	Pool1	6×36×36	2×2	-	-	
	Conv2	6×18×18	7×7	677376	800	
	Pool2	16×12×12	2×2	-	-	
	Conv3	16×6×6	6×6	46080	2960	
MPCNN [32]	Conv1	1×42×42	7×7	392000	520	4.96
	Pool1	6×36×36	2×2	-	-	
	Conv2	6×18×18	7×7	1000000	520	
	Pool2	16×12×12	2×2	-	-	
	Conv3	16×6×6	6×6	3276800	200	

VI. EVALUATION AND STATE-OF-THE-ART COMPARISON

In order to evaluate our proposed architecture we implemented five well known CNNs, ConvNet processor (CNP) [30], Modified LeNet [26], CIFAR-10 [31] and MPCNN [32]. Table 1 contains these workloads with their layers, feature dimension, kernel size, number of MAP operations, number of individual weights and total model size of the layers. All of them are implemented using both techniques, MAP cores implemented using multipliers and MF MAP cores, of our proposed architecture, MulNet, targeting Zynq XC7Z020 device running at 100MHz. The Zynq XC7Z020 devices has 106400 Flops, 53200 LUTs, 140 BRAMs and 220 DSP slices. The CNP is a network that is designed for face detection [30]. It has three convolutional layers and two pooling layers. Modified LeNet is another network created for hand written digit recognition, built from two convolutional layers and two pooling layers. CIFAR-10 is a 6-layer CNN that can classify 32 × 32 images into 10 different classes. MPCNN is a CNN developed for visual-based hand gesture recognition on a 32 × 32 gray scale images. The following sections discuss the MulNet's resource utilization efficiency, on-chip power estimate and state-of-the-art comparison.

A. RESOURCE UTILIZATION

The resource utilization for the four benchmark CNN architectures is shown in Figure 9. Our architecture with MF processing cores show a promising result by saving 36%-72% on-chip memory and 10%-44% DSP48 IPs compared to the architecture with cores implemented using multiplier. We also observed that for lower bit precisions the difference between the two versions of our architecture is very small while as the computation bit width of the architecture increases, we see significant resource difference between with multiplier and MF architecture. It can also be seen from our result that, for power-of-2 quantization the on-chip memory requirement stays constant across bit width variation, the reason for this is that we only stored the power-of-2 exponent, not the actual value for which an 8-bit is always enough. This brings considerable on-chip memory saving and certainly comes in very handy for resource constrained devices. From all the four benchmarks, the maximum on-chip BRAM saving is 72% (comparing with multiplier and MF architectures) for CIFAR-10, the maximum DSP48 saving is 44% for Modified LeNet and maximum saving for FF and LUT is 39% and 27%, respectively, on CIFAR-10 architecture. Based on our experiment, it can be concluded that the technique presented here can be more beneficial for deeper networks (more number of layers) with a higher feature precisions (i.e. our MF technique is better for 32-bit feature compared to 8-bit). Although the BRAM and DSP48 consumption always have an improved trend for MF architecture, but in few cases FF and LUT requirements remained in similar range for both the techniques.

Comparison with the state-of-the-art is made with [36] and [37]. Both reported 16-bit fixed point quantized computation on the same target device as our work. Table 2 shows the improvement MulNet obtained over [36] and [37]. MulNet achieves 4.8-7.8x better on-chip memory utilization, 25-29x less DSP48 and significant Flops and LUTs saving in both cores compared to [36] on CNP workload. For LeNet, MulNet shows a 1.7-2.7x and 12.7-20x better on-chip memory utilization compared to [36] and [37], respectively.

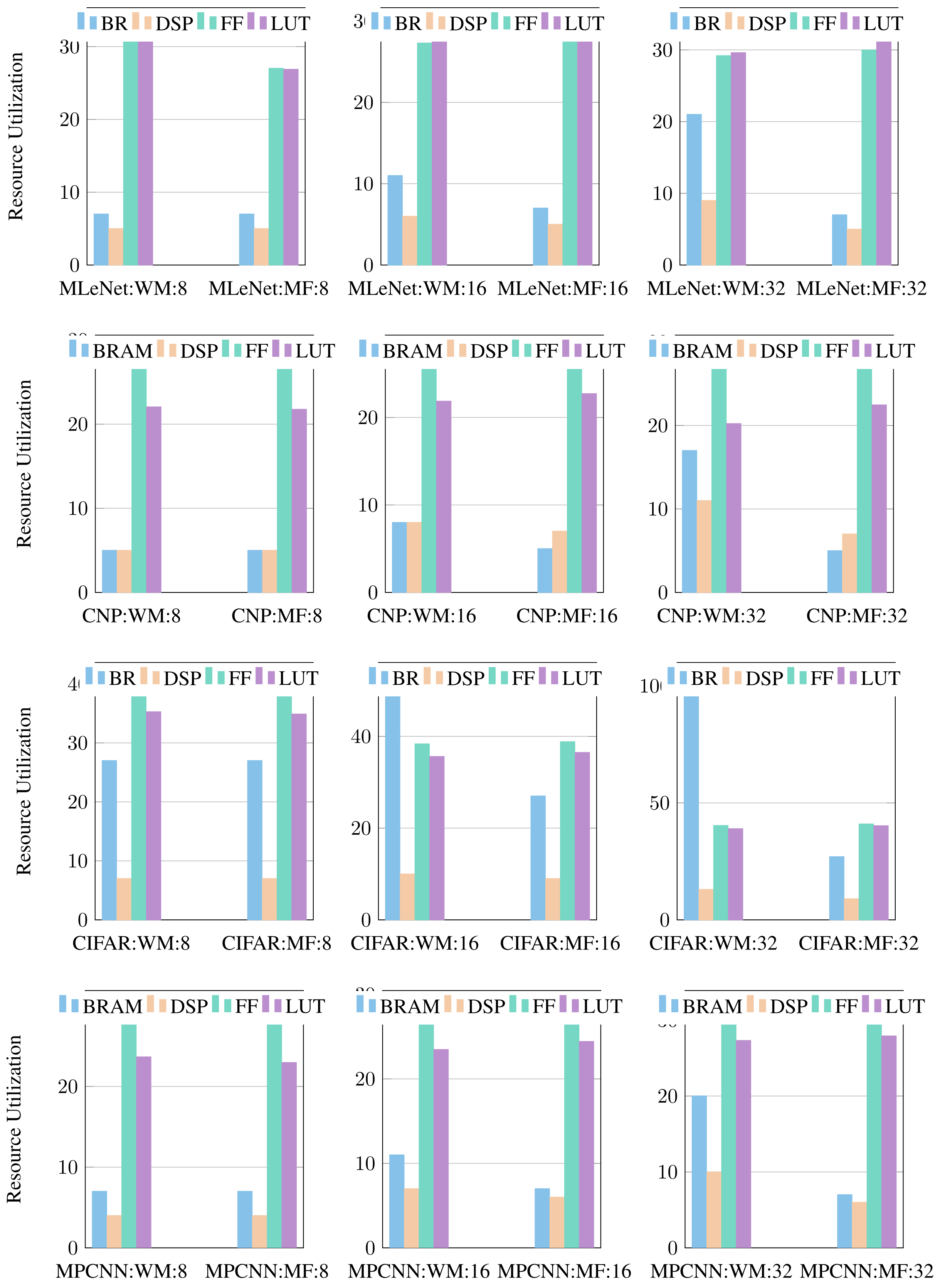

FIGURE 9. Resource Utilization for Modified LeNet (MLeNet), CNP, CIFAR-10, and MPCNN for 8-bit, 16-bit, and 32-bit Computation with Multiplier (WM) and Multiplier-Free (MF) MAP Cores: FFs and LUTs are in x100 magnitude.

TABLE 2. Resource Utilization Comparison with the State-of-the-Art for 16-bit computation.

	[36]			[37]		**This Work**							
Workload	MLeNet	CNP	MPCNN	MLeNet	CIFAR-10	MLeNet		CNP		CIFAR-10		MPCNN	
Core	-	-	-	-	-	WM	MF	WM	MF	WM	MF	WM	MF
BRAM	19	39	24	140	138	11	7	8	5	51	27	11	7
DSP	8	202	12	136	136	6	5	8	7	10	9	7	6
FF	29K	108K	80.6K	23K	31K	2.7K	2.7K	2.6K	2.6K	3.8K	3.8K	2.7K	2.7K
LUT	41K	81K	51.36K	25K	32K	2.8K	2.8K	2.2K	2.2K	3.5K	3.6K	2.3K	2.4K

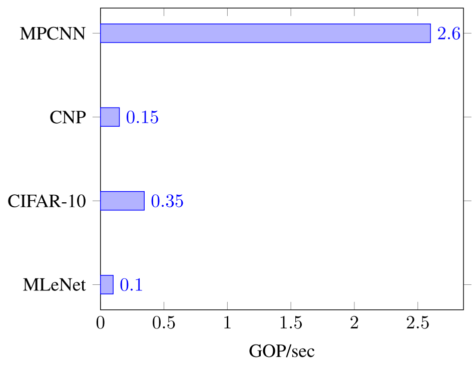

FIGURE 10. Performance in GOP/sec of MulNet for MPCNN, CNP, CIFAR-10, and MLeNet.

Likewise, 1.3-1.6x and 22.6-27.2x less DSP48 compared to [36] and [37], respectively. For CIFAR-10, MulNet gains an improvement of about 2.7-5.1x on-chip memory and 13.6-15x DSP48 resource saving compared to [37]. MulNet obtains an improvement of about 2.2-3.4x on-chip memory, 1.7-2x DSP48, 29.8x Flops and 21.3x LUT resources reduction compared to [32] on MPCNN workload. These results evidently show a significant improvement both in on-chip memory and PL resources.

B. PERFORMANCE

In order to measure the performance, we calculated GOP/sec of MulNet for all the four benchmarks presented in this work. Figure 10 shows the results we obtained. In [36] Venieris *et al.* reported 0.48 GOP/sec for LeNet, 0.74 GOP/sec for MPCNN and 3.53 GOP/sec for CNP. Comparison of MulNet with [36] shows a 3.5x better performance on MPCNN while Venieris *et al*'s work shows a better performance on CNP. However, if the reduction in resource utilization is considered for CNP, in parallel with the performance trade off, we find out that the performance-resource joint metric (a variant of Area-Delay Product) is still favorable to our technique. Just to give a numerical perspective, least amount of DSP reduction, which is 25 times reduction for CNP, is better than 23.5 times (3.53/0.15) increment in the latency of the same architecture. Moreover, in Table 2 it can be seen that MulNet offers even further reduction (about 40 times) in logic elements (LUT and Flops). Hence, we claim that our overall performance-resource metric (i.e. Area-Delay Product) is better than the architecture in [36].

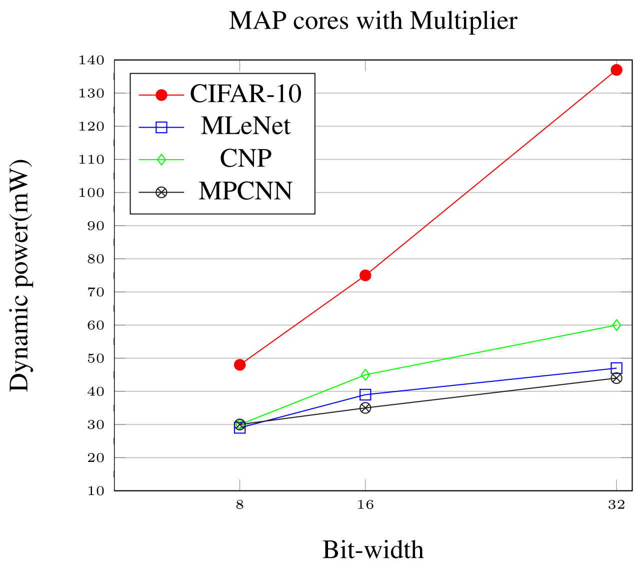

FIGURE 11. On-chip Power Estimate of MulNet on CIFAR-10, MLeNet, CNP, and MPCNN across different bit-widths.

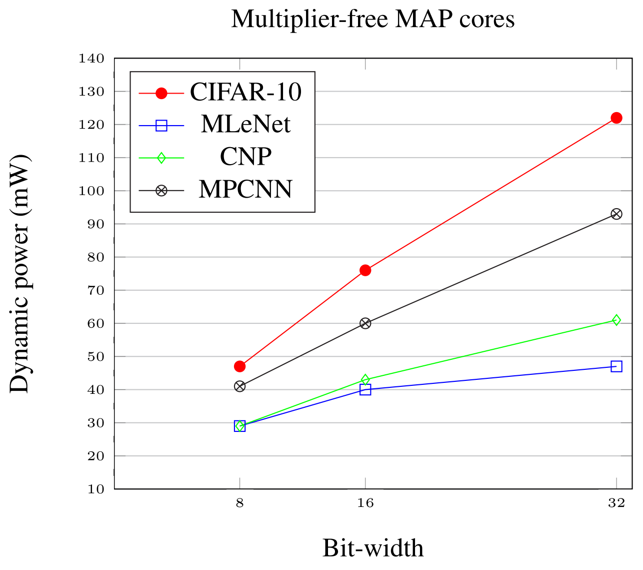

FIGURE 12. On-chip Power Estimate of MulNet on CIFAR-10, MLeNet, CNP, and MPCNN across different bit-widths.

C. ON-CHIP POWER ESTIMATE

This section presents the On-Chip dynamic power of the target devices estimated using Vivado power analyzer. The results we obtained are plotted in Figure 12. The power estimate is generated by extracting the power consumption of MulNet IP only, from the whole Zynq SoC subsystem. We observed the power consumption increases as the number of computation bits increase. The power estimation is mainly contributed by clock, signal, logic, BRAM and DSP of the target device. To further investigate, we analyzed the estimated power from the power analyzer for each component separately. We observed a distinct trend in power consumption due to signal transitions, which is denoted as signals' power in the log file. As an example, we found out that signal power for CNP network doubled as we increased number of bits from 8 to 16. While at the same time the signal power for MLeNet increased only 1.5 times. Although the power consumption for rest of the components, e.g. number of DSPs, and BRAMs, are within the similar range, but the higher increment in signal transition power resulted in a trend showing higher power consumption for wider bit widths for CNP network compared to MLenet. We attribute this fact due to higher dimension of weight matrix size in CNP network, which consequently require more frequent memory access, hence more power consumption. Our results also show that MulNet with MF processing cores achieved 2-10% dynamic power reduction for CIFAR-10 and 3.3-4.4% for CNP compared to MulNet with cores implemented using multipliers.

TABLE 3. MulNet Power Efficiency (GOP/sec per watt) for MLeNet, CIFAR-10, CNP, and MPCNN across 8, 16, and 32 bitwidth for Cores implemented using multiplier and MF Cores .

Benchmark	Power Efficiency (GOP/sec/watt)					
	Core with Multiplier			Multiplier-free Core		
	8-bit	16-bit	32-bit	8-bit	16-bit	32-bit
MLeNet	3.45	2.56	2.13	3.45	2.5	2.12
CIFAR-10	7.29	4.67	2.55	7.45	4.6	2.8
CNP	5.0	3.33	2.5	5.17	3.48	2.45
MPCNN	86.6	74.28	59.09	63.4	43.3	27.9

By leveraging the performance results discussed in Section VI-B and the power estimate metric presented in this section, we calculated the power efficiency for our proposed MulNet. The power efficiency figures are reported in Table 3 in GOP/sec per watt. This metric signifies the amount of energy MulNet dissipates to perform each of the four benchmarks across different bitwidth. We observed that the power efficiency of MulNet increases with increase in computation bitwidth for all the four benchmarks. On the other hand, MulNet with MF core tends to show slightly better power efficiency compared to cores implemented using multipliers.

VII. ALEXNET ON MULNET

All workloads discussed above and implemented to evaluate MulNet are relatively smaller networks compared to AlexNet. This section describes our strategy of utilizing MulNet to compute AlexNet on XC7Z045 Zynq FPGA, and any deeper networks in general. Since the main challenge of deploying deeper networks in target FPGAs is the model size, therefore we investigated the number of parameters for each layer in AlexNet architecture to devise an efficient deployment. In MulNet scheduling, as discussed in Figure 7, computation of each layer is completed before moving to the next layer. Hence, the requirement in fitting a trained model is to fit the layer with the higher number of weights in the network. To accomplish that, we calculated the number of parameters in each layer of AlexNet to see if the layer with the maximum number of weights can fit inside the on-chip memory available in our targeted device. In AlexNet architecture there are 34848, 614400, 884736, 1327104 and 884736 number of weights in *Conv1* to *Conv5* layers, respectively. The total available on-chip memory in our targeted FPGA device is 545 block RAMs with 36K bits per block. Hence, from our analysis we found out that the number of parameters in layer *Conv4* can't fit in the 545 blocks of on-chip memory. As a result, we devised a technique of splitting this convolutional layer (*Conv4*) into two (*Conv4_1* and *Conv4_2*) separate tasks, so we can fit each layer's (task's) weights on-chip. Table 4 shows this task division. It also shows each layer's input feature dimension, kernel size, number of MAC operations, number of weights, and total model size of the architecture with 32-bit representation. After splitting *Conv4* layer, the maximum number of weights in a single layer is 884736 (in *Conv3* and in *Conv5*) which we stored in 512 block RAMs with 16-bit quantization. The lowest bit quantization for AlexNet is chosen as 16 bits. This is because of a significant drop in accuracy for 8-bit fixed point quantization (38.6%) and below. For the sake of optimized hardware computation, we split the weight tensor in the 4^{th} dimension i.e. from K × N × H × W into two K/2 × N × H × W dimension tensors, where K is the number of filters with H × W spatial dimension (height and width) and N is the number of inputs. The splitting in the K^{th} dimension is selected in order to avoid extra accumulation (addition) operations in the 3D convolution that occurs if other dimensions are chosen. The K^{th} dimension splitting of weight tensor results in two divided output features (1 × N/2 × H × W) in the N^{th} dimension. Due to implementation details, such approach is also efficient in memory reading of the parameters and memory writing of the feature maps.

TABLE 4. Layer Tasks for AlexNet Convolutional Neural Networks on MulNet.

Workload	Layers	Dimension	Kernel	Num. of MAC	Num. Weights	Model Size (MB)
AlexNet [33]	Conv1	3×227×227	11× 11	105415200	34848	119.866
	Pool1	96×55×55	3×3	-	-	
	Conv2	96×27×27	5×5	447897600	614400	
	Pool2	256×27×27	3×3	-	-	
	Conv3	256×13×13	3×3	149520384	884736	
	Conv4_1	384×13×13	3×3	112140288	663552	
	Conv4_2	384×13×13	3×3	112140288	663552	
	Conv5	384×13×13	3×3	149520384	884736	
	Pool3	256×13×13	3×3	-	-	

As discussed in section IV-D, when full parameters and features of the currently executing layer are larger in size compared to the available on-chip memory we utilize a scheme where we load *L* lines (rows + channels) of the input features at a time. We followed the same approach for implementing AlexNet using our proposed architecture,

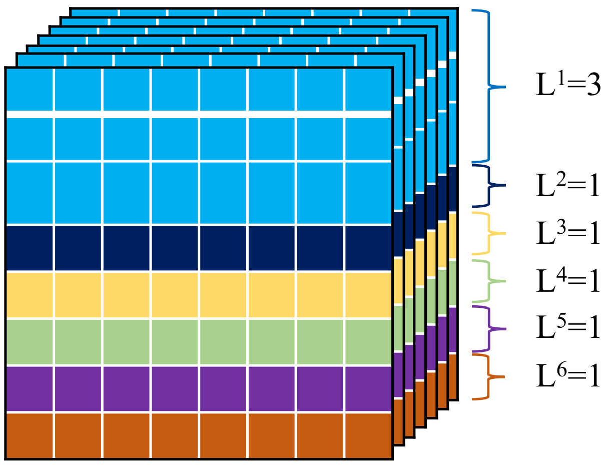

FIGURE 13. Input buffer reading sequence example with channel major fashion (shown in different colors). A line represents one full row with all its channels. In the first iteration (L^1) three lines ($3 \times 8 \times 7 = 168$ pixels) and in each of the following data reading iterations ($L^2 - L^6$) one line ($8 \times 7 = 56$ pixels) is loaded to On-chip memory.

Algorithm 2 Efficient Input Buffer Reading Algorithm with ONE Memory Read Traffic for Each Pixel in Feature Maps. Example Feature Map $7 \times 8 \times 8$ which is shown in Figure 13 is used.

```
1  feature size = inputDim = N
2  Line = L = 3
3  linePerIter = 1
4  loadLinesToIBRAM(DRAM, L);
   //Other configuration
5  if $X < N$ then
6      if $totalReadLines < N$ then
7          totalReadLines+ =linePerIter;
8          loadLinesToIBRAM(DRAM, linePerIter);
       end
9      if $Y < N$ then
           //Other configuration
10         if $ch < chIN$ then
11             processFullChannel(X, Y, ch);
           end
       end
   end
```

MulNet. Figure 13 and Algorithm 2 illustrates this approach of reading a certain portion of the input features from DDR to on-chip memory. The main advantage of this input buffering technique is that a pixel in a feature map (stored in DDR) is read only once (and stored on-chip) for its computation in the PL which significantly reduces the memory traffic between the off-chip and on-chip memory. Figure 13 shows a demonstrative example of our algorithm on $7 \times 8 \times 8$ features. As shown in Figure 13, in the first iteration of input buffering three lines of pixels are read, and in all the subsequent iterations require to read a single line of pixels. The input buffering iterations stops when the last line of pixel is read. The number of lines read at a time are configurable for each layer with the help of Algorithm 2 (line number 2 to 3 of Algorithm 2). The other constraints for selecting the number of lines are as follows: 1) how many more block RAMs are available after storing the layers weights; 2) the kernel size of the layer and 3) the stride size. In the first iteration at least a line greater than the kernel size has to be read in order to compute the convolution over that filter without reading features more than once. Similarly, the number of lines per iteration (the *linePerIter* in line number 3 of Algorithm 2) should be greater or equal to the stride size of the currently executing layer for the same reason. Lastly if in the current buffering iteration, the available input block RAMs (IBRAMs) are all occupied by the previously read pixels, memory space of pixels whose computation has completed is overwritten by the currently read pixels. This iteration continues until all pixels of the input features map are read. Any other line variation is possible if it meets these constraints. In our AlexNet implementation, we set $L^1 = 16$ and *linePerIter* of 4 for *Conv1*, $L^1 = 4$ and *linePerIter* of 2 for *Pool1*, $L^1 = 8$ and *linePerIter* of 4 for *Conv2*, $L^1 = 8$ and *linePerIter* of 2 for *Pool2* and $L^1 = 13$ and *linePerIter* of 0 for *Conv3* to *Pool3* layers. Because of implementation details (i.e. efficient in hardware) we select power of two numbers except in the case of loading full feature maps on-chip (for instance, 13 in our design).

The resource utilization of our AlexNet architecture is shown in Figure 14. We acquire 545 BRAMs and 40 DSP48 blocks usage in 16-bit fixed point quantized AlexNet implementation. Our MF implementation, as shown in the last column of Table 5, uses 8-bit power-of-2 quantization. This implementation consumes 274 BRAMs and 20 DSP48 blocks. The 8-bit quantized AlexNet is implemented only using power-of-2 quantization as the accuracy drop in that case is very minimal i.e. 1.16% compared to the original trained 32-bit model. The performance of 16-bit fixed point quantized and 8-bit power-of-2 quantized AlexNet using MulNet architecture in GOP/sec and in GOP/sec/watt are comparable to the state-of-the-art methods shown in Table 5. As shown in Table 5, our implementation obtains 2.8x lesser on-chip memory requirement compared to [18]. In our MF implementation (with power-of-2 quantization) our architecture consumes 2x lesser on-chip memory in comparison to any best-case method available in the literature, to the best of our knowledge. Comparison based on DSP blocks shows that our architecture consumes 11.8x lesser blocks compared to both [10] and [17], and 6.7x lesser blocks compared to [18]. The power efficiency of our implementation is calculated to be 86.15 GOP/sec/watt and 88.49 GOP/sec/watt for WM and MF mode of MulNet architecture, respectively. This power performance estimation is calculated with the total power consumption of PL and PS together. Moreover, MulNet achieves comparable performance to other state-of-the-art implementation. Our results show 1.26x, 1.02x and 0.844x better performance (GOP/sec) compared to [17], [18] and [10], respectively. Therefore, our

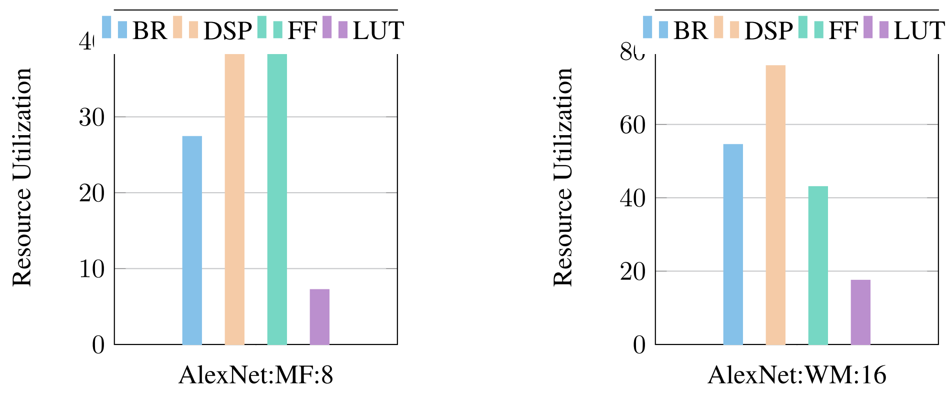

FIGURE 14. Resource Utilization of MulNet on AlexNet for 8-bit (power-of-2 quantization) and 16-bit (Fixed point quantization) Computation with Multiplier-Free (MF) MAP Cores: FFs and LUTs are in x100 magnitude and BRAMs are in x10 magnitude.

TABLE 5. Resource Utilization, Performance, and Power Efficiency Comparison of MulNet on AlexNet with Previous Works.

	[17]	[18]	[10]	**This Work**	
Core Mode	-	-	-	WM	MF
Target Device	Zynq XC7Z045	Stratix-V GXA7	Zynq XC7Z045	Zynq XC7Z045	Zynq XC7Z045
Clock	100MHz	100MHz	125MHz	100MHz	100MHz
Bitwidth(Quantization)	16 (Fixed Point)	16 (Fixed Point)	16 (Fixed Point)	16 (Fixed Point)	8 (Power-of-2)
BRAM	545	1552[1]	545	545	274
DSP	900	512	900	76	20
Performance (GOP/sec)	108.25	134.10	161.98	136.9	136.9
Power Efficiency (GOP/sec/watt)	-	-	-	86.15	88.49
Performance Density (GOP/sec/DSP)	0.12	0.26	0.18	1.8	3.4

TABLE 6. Accuracy versus Bitwidth for MLeNet, CIFAR-10, and AlexNet for Originally Trained and Quantized Models with Fixed Point and Power-of-2 Quantization Mode.

Benchmark	Data and Weight Bitwidth	Quantization Mode	Quantized Model Accuracy(%)	Original Model Accuracy(%)	Accuracy Drop
MLeNet [26]	8-bit (4.4)	Fixed Point (I.F)	69.66	98.76	29.1
	16-bit (8.8)		98.68		0.08
	32-bit (16.16)		98.76		0
	8-bit	Power-of-2	98.5		0.26
CIAFR-10 [31]	8-bit (4.4)	Fixed Point (I.F)	10.17	78.98	68.81
	16-bit (8.8)		78.28		0.7
	32-bit (16.16)		78.98		0
	8-bit	Power-of-2	72.72		6.26
AlexNet [33]	8-bit (4.4)	Fixed Point (I.F)	51.56	90.16	38.6
	16-bit (8.8)		89.32		0.84
	32-bit (16.16)		90.16		0
	8-bit	Power-of-2	89.0		1.16

results objectively testify our claim of significant reduction in utilizing resources, while maintaining at par performance level.

VIII. ACCURACY EVALUATION

The accuracy of the proposed processor for different bitwidth and quantization formats on MLeNet, CIFAR-10 and AlexNet workloads are shown in Table 6. The MLeNet CNN is trained on MNIST dataset which is a dataset of handwritten digits composed of 60,000 training 28×28 gray scale images and 10,000 similar test images. As shown in Table 6, the accuracy of the 16-bit and 32-bit fixed point quantized model has very negligible drop of 0.08%. On the other hand, the power-of-2 quantization results in 0.26% accuracy drop. CIFAR-10 is trained on tiny images of CIFAR-10 dataset which consists of 60,000 images of size 32×32 in 10 classes, with 6,000 images per class. The dataset is divided into 50,000 training images and 10,000 test images. Similarly, we trained CIFAR-10 network and computed 8-bit, 16-bit and 32-bit fixed point, and 8-bit power-of-2 quantization. The change in accuracy with respect to bitwidth changes are also summarized in Table 6. As shown in the table, for 16-bit and 32-bit fixed point quantization, the accuracy drop for the quantized model is very minimal with 0.7% and no drop, respectively. In the same way, we trained AlexNet on Cats and Dogs dataset from Kaggle [38]. The training dataset contains 25,000 images of dogs and cats with image size of 227 × 227. The total computation workload and the number of weight parameters for AlexNet trained on this dataset is analyzed in Table 4. We quantized the trained model using 8-bit, 16-bit and 32-bit fixed point, and 8-bit power-of-2. As shown in Table 6, for 8-bit fixed point quantization the AlexNet network does not perform well, with accuracy drop of 38.6% compared to the 32-bit Caffe trained model. The original Caffe trained model has an accuracy of 90.16%. Our quantization has resulted in a negligible drop in accuracy for 32-bit fixed point quantization, 16-bit fixed point quantization and power-of-2 quantization. Their respective accuracy drops are 0.84%, 0.0% and 1.16%. Comparatively, MLeNet is more resilient for lower bit quantization in both fixed point and power-of-2 quantization modes compared to AlexNet and CIFAR-10. On the other hand, AlexNet has smaller accuracy drop compared to CIFAR-10 for power-of-2 quantization. In summary, the accuracy drops from our proposed processor, MulNet, is comparable or even better than other state-of-the-art designs. At the same time MulNet gives the higher resource utilization efficiency and offers flexibility across different CNN architectures.

IX. CONCLUSION

A novel and highly flexible CNN processor architecture that can compute most regular CNN variants with MF processing cores aiming for attaining higher resource utilization efficiency is presented. We formulated fixed-point and power-of-2 quantization techniques to achieve a MF operation. We also devised a scheme that utilizes the processing system (PS) for memory-centric layers and the programmable logic (PL) for computation-centric layers. We implemented Modified LeNet, CIFAR-10 Full, ConvNet Processor (CNP), MPCNN, and AlexNet to evaluate our architecture. Our results show promising performance of our proposed, MulNet, with MF processing cores. It saves 36%-72% on-chip memory and 10%-44% DSP48 IPs compared to MulNet with cores implemented using multiplier. Comparison with the state-of-the-art showed a very promising 25-40x DSP48 and 25-29x on-chip memory reduction with up to 136.9 GOP/sec performance and 88.49 GOP/sec/watt power efficiency. Hence, our results demonstrate that the proposed architecture can be very expedient for resource constrained devices. One limitation of our architecture is that for each CNN model it requires customization of the design parameters to fit the characteristics of MulNet. We plan to alleviate this limitation by developing an automated framework (utilizing MulNet as a template), that can generate a synthesized hardware IP using an optimization algorithm over the network architecture and the target FPGA without the need for designer customization, and with an aim of achieving higher resource utilization efficiency.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015.

[2] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 1–9.

[3] 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.

[4] A. Ling and J. Anderson, "The role of FPGAs in deep learning," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, p. 3.

[5] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions," *ACM Comput. Surv.*, vol. 51, no. 3, 2018, Art. no. 56.

[6] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 2, pp. 326–342, Feb. 2019.

[7] E. Wang *et al.* (2019). "Deep neural network approximation for custom hardware: Where we've been, where we're going." [Online]. Available: https://arxiv.org/abs/1901.06955

[8] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang. (2017). "A survey of FPGA-based neural network accelerator." [Online]. Available: https://arxiv.org/abs/1712.08934

[9] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry. (2018). "Accelerating CNN inference on FPGAs: A survey." [Online]. Available: https://arxiv.org/abs/1806.01683

[10] S. I. Venieris and C.-S. Bouganis, "Latency-driven design for FPGA-based convolutional neural networks," in *Proc. IEEE 27th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2017, pp. 1–8.

[11] 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

[12] C. Shea, A. Page, and T. Mohsenin, "SCALENet: A scalable low power accelerator for real-time embedded deep neural networks," in *Proc. ACM Great Lakes Symp. VLSI.* , 2018, pp. 129–134.

[13] 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. EPFL Field Program. Logic Appl. (FPL)*, 2016, pp. 1–9.

[14] 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.

[15] 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.

[16] 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.

[17] 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. ACM 53rd Annu. Design Automat. Conf.*, 2016, p. 110.

[18] 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. IEEE 26th Int. Conf. Field Program. Logic Appl. (FPL)*, Aug./Sep. 2016, pp. 1–8.

[19] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu. (2017). "An OpenCL(TM) deep learning accelerator on Arria 10." [Online]. Available: https://arxiv.org/abs/1701.03534

[20] 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.

[21] H. Yonekawa and H. Nakahara, "On-chip memory based binarized convolutional deep neural network applying batch normalization free technique on an FPGA," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May/Jun. 2017, pp. 98–105.

[22] 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.

[23] J. Zhang and J. Li, "Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network," in *Proc. FPGA*, 2017, pp. 25–34.

[24] R. Zhao *et al.*, "Accelerating binarized convolutional neural networks with software-programmable FPGAs," in *Proc. FPGA*, 2017, pp. 15–24.

[25] M. Hailesellasie and S. R. Hasan, "A fast FPGA-based deep convolutional neural network using pseudo parallel memories," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2017, pp. 1–4.
[26] M. Hailesellasie, S. R. Hasan, F. Khalid, M. Shafique, and F. A. Wad, "FPGA-based convolutional neural network architecture with reduced parameter requirements," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.
[27] T. Mitchell, *Machine Learning*. New York, NY, USA: McGraw-Hill, 1997, p. 997.
[28] *Berkeley Vision and Learning Center*. Accessed: Mar. 17, 2017. [Online]. Available: http://caffe.berkeleyvision.org/
[29] H. Amin, K. M. Curtis, and B. R. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEE Proc.-Circuits, Devices Syst.*, vol. 144, no. 6, pp. 313–317, 1997.
[30] 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.
[31] *CIFAR-10*. [Online]. Available: https://github.com/BVLC/caffe/tree/master/examples/cifar10
[32] J. Nagi *et al.*, "Max-pooling convolutional neural networks for vision-based hand gesture recognition," in *Proc. IEEE Int. Conf. Signal Image Process. Appl. (ICSIPA)*, Nov. 2011, pp. 342–347.
[33] 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.
[34] *Xilinx 7 Series FPGAs Block RAM, UG473 7Series Memory Resources*. Accessed: Oct. 2018. [Online]. Available: https://www.xilinx.com/support/documentation/
[35] *Zynq-7000 SoC Product*. Accessed: Oct. 2018. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html
[36] 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.
[37] H. Sharma *et al.*, "From high-level deep neural models to FPGAs," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2016, Art. no. 17.
[38] *Dogs vs. Cats Dataset*. Accessed: Jan. 2018. [Online]. Available: https://www.kaggle.com/c/dogs-vs-cats/data

MULUKEN TADESSE HAILESELLASIE received the B.S. degree in electrical and computer engineering from Addis Ababa University, Addis Ababa, Ethiopia, in 2009, and the M.S. degree in communications engineering from Ulm University, Ulm, Germany, in 2015. He is currently pursuing the Ph.D. degree in electrical engineering with Tennessee Tech University, TN, USA. From 2012 to 2015, he was a Research Assistant with the Institute of Electron Devices and Circuits, Ulm University. He joined Intel as an SoC Design Engineer, in 2018, as an Electrical Validation Engineer, in 2017, and as a Product Development Engineer, in 2016. Since 2015, he has been a Research Assistant with the Electrical and Computer Engineering Department, Tennessee Tech. He has reviewed peer-reviewed conference papers and journals for DAC, the IEEE ACCESS, and the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS. His research interests include applied AI, deep learning, architecture design for computationally intensive workloads, low-power hardware design, field programmable gate array (FPGA) design, and digital signal processing.

SYED RAFAY HASAN received the B.Eng. degree in electrical engineering from the NED University of Engineering and Technology, Pakistan, and the M.Eng. and Ph.D. degrees in electrical engineering from Concordia University, Montreal, QC, Canada. From 2006 to 2009, he was an Adjunct Faculty Member with Concordia University. From 2009 to 2011, he was a Research Associate with the Ecole Polytechnique de Montreal. Since 2011, he has been with the Electrical and Computer Engineering Department, Tennessee Tech University, Cookeville, TN, USA, where he is currently an Associate Professor. He has published more than 67 peer-reviewed journal and conference papers. His current research interests include hardware design security in the Internet of Things (IoT), hardware implementation of deep learning, deployment of convolution neural networks in the IoT edge devices, and hardware security issues due to adversarial learning. He is a Full Member of Sigma Xi and a Life Member of the Pakistan Engineering Council. He received the Postdoctoral Fellowship Award from the Scholarship Regroupment Stratgique en Microsystmes du Quebec, in 2011, the Faculty Research Award from Tennessee Tech University, from 2013 to 2014 and from 2015 to 2016, the Kinslow Outstanding Research Paper Award from the College of Engineering, Tennessee Tech University, in 2015, and the Summer Faculty Fellowship Award from the Air force Research Lab (AFRL). He was a recipient of the Sigma Xi Outstanding Research Award, in 2012. He has received research and teaching funding from NSF, ICT-funds UAE, AFRL, and Intel Inc. He has been part of the funded research projects, as a PI or a Co-PI, that worth more than $1.1 million. He is the Session Chair and a Technical Program Committee Member of several IEEE conferences including ISCAS, ICCD, MWSCAS, and NEWCAS, and a Regular Reviewer for several IEEE TRANSACTIONS and other journals including TCAS-II, IEEE ACCESS, *Integration*, the VLSI Journal, *IET Circuit Devices and Systems*, and IEEE EMBEDDED SYSTEMS LETTERS.