

Article

mNet2FPGA: A Design Flow for Mapping a Fixed-Point CNN to Zynq SoC FPGA

Tomyslav Sledevič  and Artūras Serackis 

Department of Electronic Systems, Vilnius Gediminas Technical University, Naugarduko str. 41, LT-03227 Vilnius, Lithuania; arturas.serackis@vgtu.lt

* Correspondence: tomyslav.sledevic@vgtu.lt

Received: 24 September 2020; Accepted: 30 October 2020; Published: 2 November 2020



Abstract: The convolutional neural networks (CNNs) are a computation and memory demanding class of deep neural networks. The field-programmable gate arrays (FPGAs) are often used to accelerate the networks deployed in embedded platforms due to the high computational complexity of CNNs. In most cases, the CNNs are trained with existing deep learning frameworks and then mapped to FPGAs with specialized toolflows. In this paper, we propose a CNN core architecture called mNet2FPGA that places a trained CNN on a SoC FPGA. The processing system (PS) is responsible for convolution and fully connected core configuration according to the list of prescheduled instructions. The programmable logic holds cores of convolution and fully connected layers. The hardware architecture is based on the advanced extensible interface (AXI) stream processing with simultaneous bidirectional transfers between RAM and the CNN core. The core was tested on a cost-optimized Z-7020 FPGA with 16-bit fixed-point VGG networks. The kernel binarization and merging with the batch normalization layer were applied to reduce the number of DSPs in the multi-channel convolutional core. The convolutional core processes eight input feature maps at once and generates eight output channels of the same size and composition at 50 MHz. The core of the fully connected (FC) layer works at 100 MHz with up to 4096 neurons per layer. In a current version of the CNN core, the size of the convolutional kernel is fixed to 3×3 . The estimated average performance is 8.6 GOPS for VGG13 and near 8.4 GOPS for VGG16/19 networks.

Keywords: convolutional neural network (CNN); design flow; field-programmable gate array (FPGA); hardware-software co-design

1. Introduction

The convolutional neural networks (CNNs) have widely spread out in the last decade. They are most commonly used to solve machine vision tasks [1,2], or even for the identification of dynamic systems [3]. The training of deep neural networks is commonly performed using floating-point arithmetic operations. Most of the deep learning systems are designed to be trained and executed on systems with graphics processing units. Such devices have up to several thousand computing cores and large external memory bandwidth.

Running deep neural network-based machine vision algorithms on the edge introduces some challenges, including the requirements of energy efficiency. The design of application-specific integrated circuits introduces longer turn-around times and higher non-recurring engineering costs.

In this paper, we focus on the field-programmable gate array (FPGA)-based implementation of deep neural networks with CNNs. The widely used devices, e.g., multispectral cameras, high-end vision systems already have FPGAs on-board, which leads to a shorter time to upgrade a device with new features. In addition, many embedded RGB or thermal vision systems use image pre-processing on the edge and are implemented on an FPGA-based device. The possibility to extend the current

functionality of a vision system with a CNN yet to not change the whole system's architecture, sounds like a valuable option for developers. In comparison to application-specific integrated circuit-based upgrades, the FPGA-based solution has a shorter turn-around time, has less non-recurring engineering costs and introduces a variety of input/output ports together with different communication protocols. However, FPGAs also have some disadvantages. They run on lower frequencies and can perform fewer operations per second. Therefore, in order to implement a CNN-based machine vision algorithm, the training and classification stage should be optimized using additional techniques.

Currently, many manipulations are proposed to speed-up the training stage and classification procedure. This includes network training with normalized batches of data [4], the binarization of convolutional kernels [1,5,6], network compression using adaptive quantization schemes [7–10] and reducing a number of operations in the CNN model with neuromorphic techniques [11,12].

The known hardware cores of the CNN in most cases are designed taking into account the specifics of mid-range or ultra-scale devices with abundant computational resources [13]. The very deep CNN models are mapped on dense FPGAs to get consistent performance in aspects of the frame rate and GOPS. Table 1 summarizes the performances of VGG-16 networks implemented with various known toolflows with the same 16-bit fixed-point precision. Several studies have shown that CNN inference does not require high-precision floating-point computations and can be carried out using fixed-point arithmetic for near 1% accuracy degradation [14].

Already existing and widely used deep learning software is often employed to train CNNs for hardware accelerators. The toolflows such as fpgaConvNet [15], Caffeine [16], Angel-Eye [17], DnnWeaver [18], NEURAghe [19], MALOC [20], Nullhop [21] and Ma et al. [22] take on the input pre-trained kernel weights from the Caffe deep learning framework. The OPU [23] and FP-DNN [24] toolflows accept network architecture from the Tensorflow deep learning framework.

Table 1. Toolflows for automated mapping of CNNs to FPGAs.

Toolflow	FPGA	DSPs	Clock, MHz	GOPS
Caffeine [16]	KU060	1058	200	266
MALOC [20]	Virtex-7	2688	150	830
OPU [23]	Kintex-7	516	200	354
Nullhop [21]	Z-7100	128	60	17.2
Angel-Eye [17]	Z-7045	780	150	137
NEURAghe [19]	Z-7045	864	140	169
fpgaConvNet [15]	Z-7045	900	125	123
fpgaConvNet [13]	Z-7020	220	125	48
DnnWeaver [18]	Z-7020	140	150	31
FP-DNN [24]	Stratix-V	1036	150	164
Ma et al. [22]	Arria-10	3036	240	968

In this paper, we propose a technique to map a pre-trained CNN to an FPGA. Similarly to the Nullhop [21], Guo et al. [10], NEURAghe [19] and Angel-Eye [17] toolflows, we use hardware–software co-design for CNN implementation. In our proposed solution, the deep neural network is configured and trained in *Matlab* using functions available in Deep Learning Toolbox. Afterwards, the parameters of CNN are stored in a file and processed with a developed conversion program. That program creates all necessary sets of instructions and modified parameters of CNN, and then uploads it to the SoC FPGA board. This technique allows one to map a *Matlab*-trained CNN to a SoC FPGA (additional restrictions to layer design are discussed in a Section 3.1). We named the proposed toolflow mNet2FPGA. The current version of our CNN core is purposely designed as a single computation engine to map CNN to cost-optimized SoC FPGA devices, utilizing a single convolutional core. It utilizes only 68 DSPs, 36K LUTs and 48 RAM to store some of parameters of the CNN. There is no need to reprogram the FPGA to implement different configurations of the CNN. The same CNN core can be used to run different types of CNN if a new list of instructions and parameters is loaded to the SoC FPGA through Ethernet connection. The processing system (PS) in a hardware/software

co-design is responsible for instruction execution and advanced extensible interface (AXI) stream initialization. All the convolutional responses from intermediate layers are streamed to RAM. Therefore, no BRAM is required to store the results of the computations on the chip itself. The data transfers between hardware core and the RAM were implemented using four synchronous AXI DMA streams. Two high-performance ports are dedicated to transfer data from RAM to FPGA, and the additional two ports are used simultaneously in opposite directions. The core consists of $64 \times 3 \times 3$ size configurable kernels and forms a convolution processor with eight input and eight output channels. The core is shared through all convolution layers. In most cases, when the number of input channels is larger than eight, the parameters in a core are configured to work like a multi-channel addition core.

The main contributions of this work:

- Hardware cores (VHDL) for convolution and fully connected (FC) layers. Synthesized once and deployed in an FPGA can be re-utilized even if the structure of the CNN changes;
- A conversion program (Python) that takes a trained model of CNN, truncates and rearranges the parameters and schedules instructions for the PS that handles the hardware cores.

2. Background

CNNs are a class of deep learning networks generally used for image classification. Similarly to other networks, each CNN has a training phase and a feed-forward classification phase. In most cases, only the classification phase is placed on the edge device, and the training stage is performed separately with floating-point precision. The CNN consists of a set of repetitive layers, as shown in Figure 1. Each layer receives multiple feature maps and generates a new set of feature maps for the following layer. The multi-channel 2D feature maps are extracted with the convolution, batch normalization, activation and sub-sampling layers. The fully connected layers are used for classification.

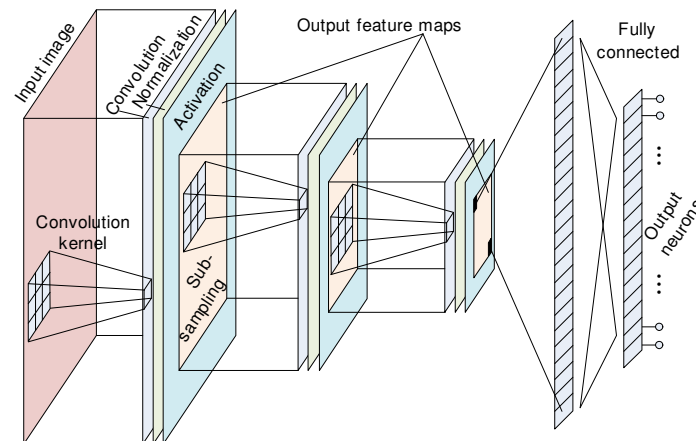


Figure 1. Typical layers in a feed-forward phase of a CNN.

The convolutional layers are the most computationally expensive. They perform filtering of a 3D input feature map and produce a 3D output map. The weights of 2D kernels are adapted during the training phase of the convolutional layer. To speed-up the training phase, the batch normalization layer is placed just after the convolution layer. Despite the batch normalization requiring the computation of mean, standard deviation, scale and offset factors for each batch of the input feature map, it acts as regularization and accelerates network training up to 14 times [4]. All the multiplication and addition operations with constant coefficients can be merged to accelerate batch normalization in a feed-forward phase, thereby reducing the FPGA resource utilization rate [14,25].

After the batch normalization comes the activation layer. In most types of CNN, it suppresses the negative values in a feature map and passes through the positive values to the sub-sampling layer. In a sub-sampling layer, the size of the feature map is downgraded four times using 2×2 size

non-overlapping average or max-pooling kernels. In the final stage of the CNN, the fully connected layers apply a linear transformation to the input feature map.

The floating-point kernels can be replaced with binary kernels to accelerate the computation of convolutional layers and reduce the requirements for arithmetical resources [5]. The process of binarization means a conversion of kernel weight to +1 if the kernel cell is positive and to −1 if the cell is negative. The binarization is especially useful for CNN implementation in hardware, where there are a lot of distributed logic resources to substitute multiplication with XNOR operation. The main disadvantage of convolution kernel binarization or replacement with XNOR gates lies in the accuracy drop [6]. Therefore, a trade-off between precision in bits and throughput is still an issue, especially in real-time edge computing applications [1].

3. The Proposed Toolflow

In this section, we describe the proposed toolflow in detail. First, the training, divided into two phases, is explained. Second, the kernel binarization and merging with the batch normalization layer are explained.

3.1. Training of the CNN

The complete configuration process is given in Figure 2. The presented diagram describes design stages featured to run custom the CNN on the SoC FPGA device. *Matlab* is used to prepare configuration files for the CNN layers and to send them to a hardware core using *Python* coded tools. *Vivado* is used to develop a CNN core and *Xilinx SDK* to create embedded glue software for PC ↔ FPGA communication.

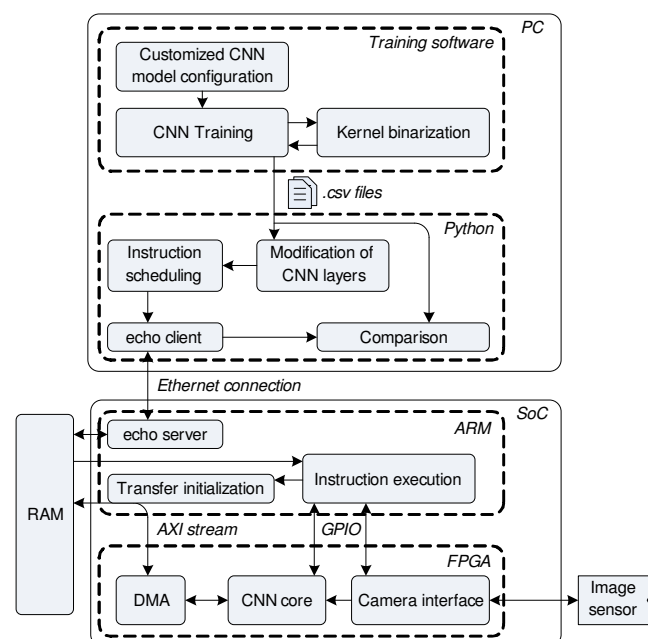


Figure 2. A schematic diagram of the complete process used to configure and place a CNN core of a custom convolutional neural network on a SoC FPGA device.

This work is based on an idea to develop a software tool that places a trained CNN into the hardware core. The Deep Neural Network Toolbox was employed to develop a custom CNN in *Matlab*. Standard functions from the toolbox should be used to add convolution, batch normalization (BN), rectified linear unit (ReLU), max-pooling (pooling) and fully connected (FC) layers. Layers such as BN, ReLU and pooling, are optional. Any of these layers can be skipped during custom design. A few architectural constraints should be considered before creating a custom CNN and placing it into the current version of the proposed CNN core:

- Size of all convolutional kernels fixed to 3×3 ;
- Stride of all convolutional kernels fixed to 1×1 ;
- Size of max-pooling kernel fixed to 2×2 ;
- One to three fully connected layers;
- Up to 4096 neurons per FC layer.

A sample of the code for a VGG network configuration is shown in Listing 1. This example shows the configuration of the few first and last layers. An image input layer inputs 224×224 RGB image. The first convolutional layer applies 64 sliding convolutional kernels. The next convolutional layer applies 128 sliding convolutional kernels to the input with the same padding so that the output has the same size as the input. Optional BN, ReLU and 2×2 stride pooling layers come next. Finally, two fully connected layers are inserted with 4096 and 1000 neurons, respectively. Any customization in default training options does not influence the structure of the created CNN.

To obtain pseudo-binary kernels with standard training functions, we applied a double training procedure to the initialized CNN. In the beginning, the training is applied to all parameters in any layer. When the first training phase using the *trainNetwork()* function is done, all the weights of convolutional kernels are changed to a pseudo-binary representation, as is shown in (2)–(5). Before the next training phase, we prohibit any updates of kernel weights in all convolutional layers by setting *WeightLearnRateFactor* and *BiasLearnRateFactor* to zero, thereby keeping pseudo-binarized weights unchanged during the final training. Only the parameters of BN layers and weights in FC layers are updated during the second run of *trainNetwork()*.

Listing 1: A sample of the code used to configure few first and last layers of VGG network.

```
layers = [
    imageInputLayer([224 224 3])
    % ...
    convolution2dLayer(3,64,'Padding','same')
    batchNormalizationLayer
    % ...
    convolution2dLayer(3,128,'Padding','same')
    batchNormalizationLayer
    reluLayer % optional
    maxPooling2dLayer(2,'stride',2) % optional
    % ...
    fullyConnectedLayer(4096)
    fullyConnectedLayer(1000)
    regressionLayer];
```

When the training is done, the parameters of every layer are stored in separate CSV files. We use *Python* functions to read those files and truncate the precision of the parameters to 16 bit. Then we apply compression to the pairs of parameters (two neighbor values as high 16 bit and low 16 bit to 32 bit) to efficiently utilize on-board RAM and throughput of an Ethernet connection. A software script, written in *Python*, handles the assignment of memory addresses in on-board RAM (Section 4.1) and prepares a list of instructions to be processed by the convolutional core (Section 4.2).

Finally, the Ethernet connection between the PC and SoC is established using the socket client and an echo server on ARM, as is shown in Figure 2. The instructions, base addresses, configurations of the convolutional core and FC layer parameters are transferred to RAM afterward. When the PC sends an image file to the SoC, the CNN starts filtering immediately and sends the response of the last FC layer back to PC for further comparison with the floating-point implementation of the same network structure. Hardware timers count a period passed to process CNN operations. The total size of CNN is limited only by on-board RAM from where the configurations are loaded to the hardware core.

In a functional application of the CNN the images can be captured from an image sensor that is connected to the camera interface (Figure 2). In that case, the input image size is cropped by a camera interface controller according to the resolution of the input layer of the CNN. The beginning of the image processing with the CNN is initialized by the camera controller, which sends a flag to the ARM when pixels of a new frame are ready to be processed. It should be noted that the CNN core receives a stream of pixels not as an AXI stream from RAM but from buffers located on the camera controller.

3.2. Kernel Binarization and Merging with BN Layer

The response y of the convolutional kernel with the number of input channels N is calculated as follows:

$$y = \sum_{c=1}^N w_c \times x_c, \quad (1)$$

where w_c is a weight array of the 2D kernel of the c^{th} input channel; x_c is a 2D input to the c^{th} kernel.

Each multiplication of the input pixel with convolutional kernel weight needs a DSP operation. Binary kernels are often used to reduce the toll of using DSP blocks on the cost of accuracy drop by 4–8% [1,5]. We introduce some modifications in the training process to generate binary kernels with the conventional *Matlab* CNN training toolbox. As mentioned before, after the first training of the CNN comes the binarization of all kernels in all convolutional layers. The 3×3 sized weights array w of a kernel is approximated by:

$$\begin{bmatrix} w^{1,1} & w^{1,2} & w^{1,3} \\ w^{2,1} & w^{2,2} & w^{2,3} \\ w^{3,1} & w^{3,2} & w^{3,3} \end{bmatrix} \approx A \begin{bmatrix} B^{1,1} & B^{1,2} & B^{1,3} \\ B^{2,1} & B^{2,2} & B^{2,3} \\ B^{3,1} & B^{3,2} & B^{3,3} \end{bmatrix}, \quad (2)$$

where $i \in [1, 2, 3]$ and $j \in [1, 2, 3]$ are vertical and horizontal indices in convolutional kernel; A is a positive scaling factor estimated as the mean of absolute weights w in a kernel:

$$A = \frac{1}{9} \|w\|. \quad (3)$$

$B^{i,j}$ is an approximated binary kernel, which takes the sign of weight $w^{i,j}$ [1]:

$$B^{i,j} = \text{sign}(w^{i,j}), \quad (4)$$

here $B^{i,j} = 1$ if $w^{i,j} \geq 0$, and $B^{i,j} = -1$ if $w^{i,j} < 0$;

According to (2)–(4) kernel weights $w^{i,j}$ are pseudo-binarized as follows:

$$\begin{bmatrix} 0 & 0.3 & -0.1 \\ -0.2 & 0.4 & 0.3 \\ 0.3 & -0.4 & 0.2 \end{bmatrix} \approx 0.24 \begin{bmatrix} 1 & 1 & -1 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \end{bmatrix}. \quad (5)$$

Before the next training, all the kernels are replaced according to (5). The approximated kernels bring in an error. Therefore, the second call of *trainNetwork()* function is required to adapt parameters in BN and FC layers and compensate for the impact of pseudo-binarization on the accuracy of CNN. We call the process pseudo-binarization because it is not a pure binary kernel that is left. However, the response from the binary kernel is extra scaled by a factor of A —a mean of absolute weights to keep the filter response amplitude on the same level as before binarization, like filtering with the primary floating-point kernel.

The BN layer implements the following expressions:

$$\hat{y} = \frac{y - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad (6)$$

$$\tilde{y} = \gamma \hat{y} + \beta, \quad (7)$$

where μ is a mean; σ^2 —variance; ϵ —numerical stability coefficient; γ —scale factor; β —offset calculated over a mini-batch; y —response from multi-channel convolution filter; \tilde{y} —batch normalized output.

Inserting (6) into (7), we get:

$$\tilde{y} = \frac{\gamma y}{\sqrt{\sigma^2 + \epsilon}} + \beta - \frac{\mu \gamma}{\sqrt{\sigma^2 + \epsilon}}, \quad (8)$$

assuming that $\frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}$ and $\beta - \frac{\mu \gamma}{\sqrt{\sigma^2 + \epsilon}}$ are unique trainable parameters for each filter in convolution layer. Then (8) gets form of linear equation, where only one multiplier and adder are required:

$$\tilde{y} = ky + b, \quad (9)$$

where $k = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}$ and $b = \beta - \frac{\mu \gamma}{\sqrt{\sigma^2 + \epsilon}}$.

Replacing (1) with a pseudo-binarized kernel, we get approximated response Y :

$$Y = \sum_{c=1}^N A_c (B_c \times x_c), \quad (10)$$

where A_c is a scaling factor; B_c —binarized kernel of c^{th} channel in a filter. Approximated and batch normalized output \tilde{Y} is gotten by the following expression, inserting (10) into (9):

$$\tilde{Y} = kY + b = k \sum_{c=1}^N A_c (B_c \times x_c) + b, \quad (11)$$

where $(B_c \times x_c) = X_c$ is a convolutional response of the binary kernel with c^{th} input channel x_c , which can be implemented as the sum of multiplexed positive/negative inputs $x_{i,j}$ depending on $B_{i,j}$'s sign. After CNN training, the parameter A_c is fixed to the input channel, while k is fixed to the kernel. Considering above modifications, the approximated convolution and batch normalization are performed as follows:

$$\tilde{Y} = \sum_{c=1}^N K_c X_c + b. \quad (12)$$

The biases are removed from all the convolutional layers. Instead, the training algorithm adapts β parameter in BN layer if the offset of kernel response is required. The multiplication operations are merged with the scaling factor A in the precedent convolutional layer and marked as a new parameter $K_c = kA_c$ in the reduced batch normalization block to save memory and reduce the number of DSPs in the BN layer (see Figure 3).

3.3. Conversion/Scheduling Program

The conversion program (Figure 4) takes a set of files with parameters of the trained CNN as the input and gives a truncated to 16 b, compressed file, with weights arranged in 512 B size blocks of data for convolutional core configuration memory and for the memory of fully connected layers. It takes care of the proper distribution of that blocks of parameters in external memory. In the beginning, the conversion program generates base addresses (Figure 5) according to the configuration of trained CNN. The conversion program generates the addresses for the DMA controller to access external memory in the desired location. It gives two read and two write addresses (because of 4 AXI streams) for every configuration of the convolutional core. It manages where the feature maps come from in an external memory and where to store the newly computed feature maps.

The program converts the trained CNN to a format supported by the proposed core. The hardware core has a limited number of input/output channels, and therefore the processing of the CNN was

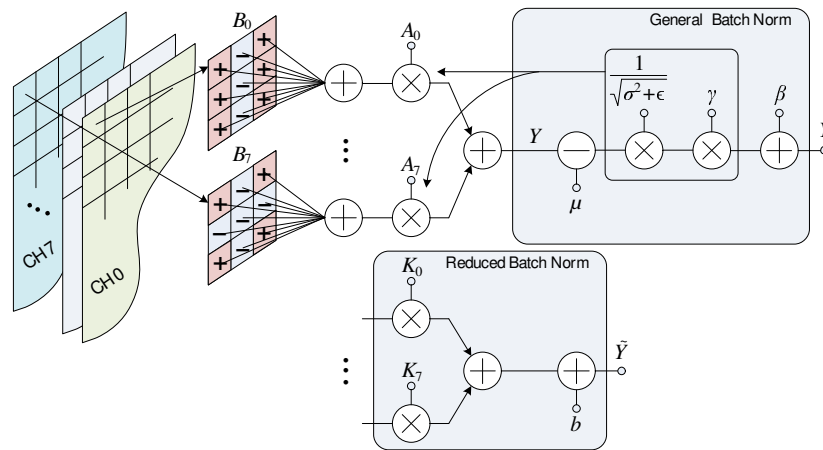


Figure 3. Signal flow graph of the binarized convolution and batch normalization.

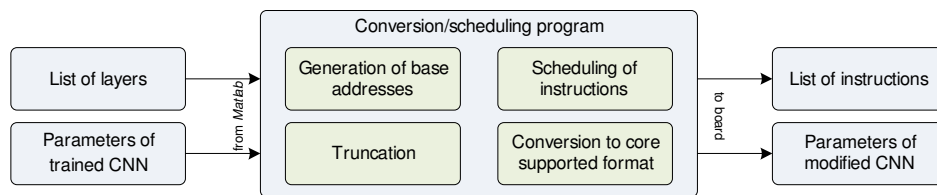


Figure 4. A schematic diagram of the program that converts a floating-point CNN to a fixed-point version for implementation on the CNN core.

sectioned into portions in an instruction scheduling stage. According to the number of input and output channels in a convolutional layer, the program computes how many times the convolutional core needs to be called to process all the multi-channel feature maps and get all the required number of channels on the output. The convolutional core has eight input and eight output channels; therefore, in all the cases when the number of input or output channels exceeds eight, the core is configured for convolution and addition alternately, as is shown in the core configuration plan.

The ARM processor is a master for the hardware CNN core. It initiates all the DMA transfers between the FPGA and external memory. Therefore, the conversion program creates a list of instructions for the ARM processor, according to the CNN's structure. The first 32 b describes a type of instruction, which switches the convolutional core to work in the desired mode. The instruction contains source and destination addresses, and the size of the data to be transferred to/from a core and external memory.

4. Design

Many aspects need to be considered before the implementation of the CNN core. The constrained computational and storage resources of the SoC FPGA force us to share arithmetical cores through all the layers and deal with limited throughput transfer controllers. Therefore, the next subsections explain the general ideas applied during the design stage of the CNN core. The CNN core was implemented with VHDL code. Then it was placed to the Vivado IP catalog and used in our further projects.

4.1. Memory Allocation

All the necessary configurations of the CNN (Listing 1) and its training were made only in *Matlab* script. Therefore, all the further functions in a toolchain interpret CNN in accordance with output files from the primary script. Depending on the image size of the CNN input, the number of layers and other settings of the created CNN, we need to dedicate memory in a flexible way. *Python* script is responsible for the efficient allocation of data in on-board RAM. It sticks together different kinds of


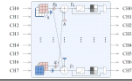

<i>IMG_BASE</i>		Image up to 8 channels
<i>NET_BASE</i>		Network description
<i>INSTR_BASE</i>		ARM instructions
<i>CORE_BASE</i>		Weights of convolution cores
<i>FC_BASE</i>		Weights of FC layers
<i>TEMP_BASE</i>		Results of intermediate layers
<i>RES_BASE</i>		Results of last FC layer

Figure 5. The arrangement of the base addresses in the on-board RAM.

data as closely as possible to reduce the memory size required to place the configurations of CNN and the results of intermediate layer computation.

Figure 5 briefly describes an arrangement of base addresses in RAM. The main seven regions can be distinguished in on-board memory, where different types of data are stored. The *IMG_BASE* pointer directs to a first pixel in the input image. Due to the architecture of the convolutional core, the channels of the image need to be stored in a specific [high 16 b, low 16 b] order: [CH1, CH0], [CH3, CH2], etc., for DMA0 AXI stream; and [CH5, CH4], [CH7, CH6], etc., for DMA1 AXI stream. *Python* script handles that procedure of channel interleaving as well before sending it to RAM. For RGB images, only the first three channels are useful (all the upper channels are filled with zeros), and only CH0 is used in the case of a grayscale image.

The *NET_BASE* points to the location where the description of the CNN is stored. It contains spatial parameters of the created network, like the resolution of the image in the input, the number of kernels in each layer, indicators of the presence of optional layers (BN, ReLU, pooling) and the number of neurons in FC layers. The processing system needs that information to properly set the batch size of AXI stream, convolutional core and FC layers.

At the *INSTR_BASE* address, a list of instructions starts that the processing system sequentially reads from RAM, and then it configures the parameters in the convolutional core and handles AXI stream transfers. Each convolutional instruction directs to a dedicated array of weights for convolutional core (that includes B_c , K_c , b) stored in a region where the *CORE_BASE* address points. To simplify the direct memory addressing, the order of weights of convolutional cores matches the order of instructions in RAM.

The *FC_BASE* address directs to a memory location where the weights of fully connected layers are stored. The *TEMP_BASE* pointer refers to temporary memory, which is required to store the responses from convolutional cores. The data in this memory region is overwritten if it is no longer necessary for further computations. At the *RES_BASE* address, the outputs of CNN are saved. When CNN processing on FPGA is done, the data from this region are streamed back to the PC to compare with the floating-point implementation of the CNN.

The configurations of convolutional core are loaded in batches from on-board RAM to FPGA BRAM-based core configuration memory, which is limited to 8 kB in the current version of the CNN hardware core (Figure 6). This memory holds 15 convolutional cores and one additional core configuration. Additionally, after every 15 convolutions PS updates the core configuration memory to load the next batch of B_c , K_c and b parameters. In a case when the summation of the channel is required, the convolutional core is configured to work as an addition. It adds input channels CH0+CH4, CH1+CH5, CH2+CH6 and CH3+CH7, and streams the CH0–CH3 outputs through DMA2 back to RAM. The parameters to the memory of addition core are loaded only once at the first configuration of BRAM and stay unique for all summation requests.

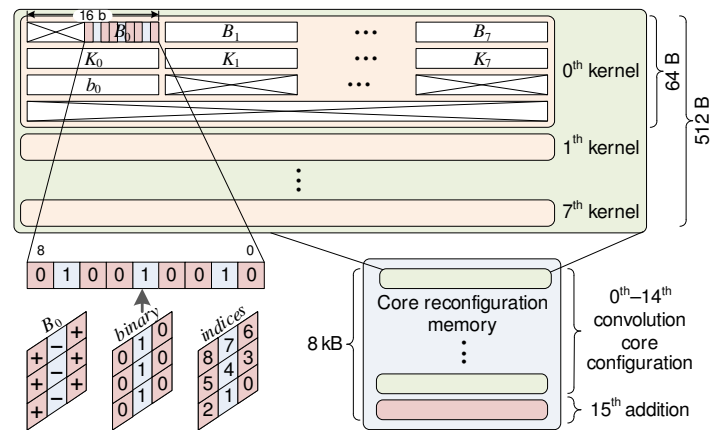


Figure 6. Parameter arrangement in the convolutional core's configuration memory.

A single configuration of the convolutional core contains parameters of eight filters and occupies 512 B of memory. One filter contains eight binary B_c kernels, eight K_c scaling factors, one b offset parameter and occupies 64 B. The unused fields of memory are crossed out in Figure 6. The single binary kernel uses 9 bits, and therefore 7 highest bits are sacrificed to simplify the addressing controller.

4.2. Instructions for the Convolutional Core

Due to the hardware limitations, a single multi-channel core is placed on FPGA. This core must be shared between convolutions in different kernels and layers. The input and output of the convolutional core have an AXI stream-based interface. All the AXI streams are initiated from PS side. To start the so called MM2S (memory map to stream) data transfer (in other words transfer from on-board RAM to FPGA), the software in PS needs to set the size of transmitted data and the source address in RAM from where the data will be pushed to FPGA. Simultaneously, to capture the responses from the convolutional core, the S2MM (stream to memory map) transfer is initiated. It sets a destination address where the data received from FPGA will be stored in RAM.

The data processing in convolutional layers on PL side is initiated by the execution of instruction on PS side. The proposed instruction format is presented in Figure 7. The *instr0* tells the convolutional core what kind of operation will be executed on the incoming stream of data. There are six distinct types of operations for the convolutional core:

- convolution + BN;
- convolution + BN + ReLU;
- convolution + BN + ReLU + pool;
- addition + BN;
- addition + BN + ReLU;
- addition + BN + ReLU + pool.

At the moment, the *instr1* is a reserved parameter. The fields *sizeTX* and *sizeRX* describe the amount of data in Bytes to be sent during MM2S and S2MM transfers respectively. The source addresses *src_0* and *src_1* in the instruction line point to memory in on-board RAM from where the data are streamed out to the core through AXI DMA 0 and DMA 1 buses. The outputs from the convolutional core are streamed back to RAM on the destination addresses *dst_0* and *dst_1* using AXI DMA 2 and DMA 3 buses.

The developed software in a *Python* recognizes one of six above-listed types of convolutional layer and creates an instruction for a core. For example, if after the convolutional layer come a ReLU layer and pooling layers, then a corresponding code is written to the *instr0*. During the CNN processing,

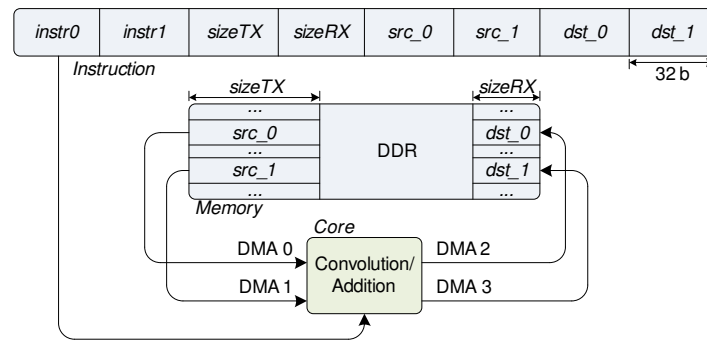


Figure 7. The format of instructions for the convolutional core.

the core receives *instr0* and enables ReLU and pooling layers to process the stream of feature maps just after convolution.

4.3. Data Exchange between RAM and FPGA

The architecture of the convolutional core and fully connected layers is based on the continuous processing of the AXI data stream. After the experimental investigation in Vivado with various settings of DMA controllers, we found one configuration suitable to deliver data from RAM to FPGA and take data back to RAM simultaneously. For that purpose, all four AXI high-performance slave interfaces are connected to the DMA controller through a 32 b width stream and memory map data buses with burst transfers of 256 data beats for each read and write channel. The DMA 0/1 work as reading channels connected to PL through HP0/1 ports, and DMA 2/3 work as write channels connected to PL through HP2/3 ports, as is shown in Figure 8. Through the read channels, data from RAM are delivered to convolutional core configuration memory, to the core or to fully connected layers. Furthermore, through the write channels, data are streamed back to RAM from convolutional core or fully connected layers. The MM2S and S2MM transfers are initialized from an application on the PS side.

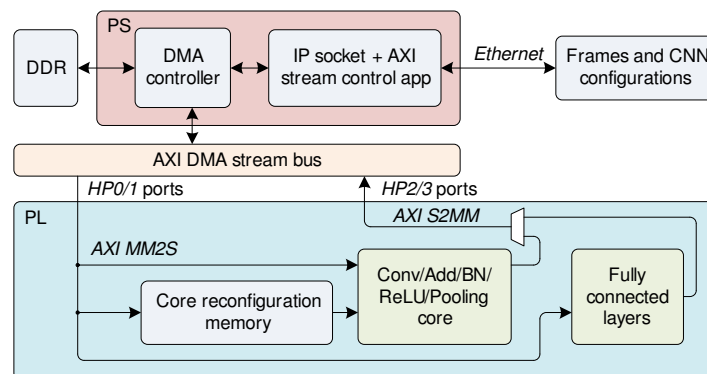


Figure 8. The diagram of data transfer between the processing system (PS) and the PL using the AXI DMA stream bus.

A 100 MHz clock is used for AXI stream bus synchronization. The tested higher frequency of the clock does not meet the timing constraints of the PS–PL interface. Since the stream data buses are 32 b width and the data precision is fixed to 16 b, two samples of data are streamed on every clock cycle through single DMA, as is shown in Figure 9. The MM2S and S2MM streams contain data from interleaved channels. The four lowest channels are transferred through DMA 0 and the four highest channels through DMA 1. The [CH1, CH0], [CH5, CH4] pairs are streamed on the even clock cycles, and the [CH3, CH2], [CH7, CH6] pairs are streamed on the odd clock cycles from the RAM to FPGA and vice versa. The demultiplexers on the input to the convolutional core separate the MM2S AXI stream into eight channels. Every second cycle of 100 MHz clock, the convolutional core receives a new

sample for each of eight channels. Therefore, the core is designed to work in a 50 MHz clock domain. The multiplexers on the output of the convolutional core combine the responses from eight channels to two S2MM AXI streams in a 100 MHz clock domain.

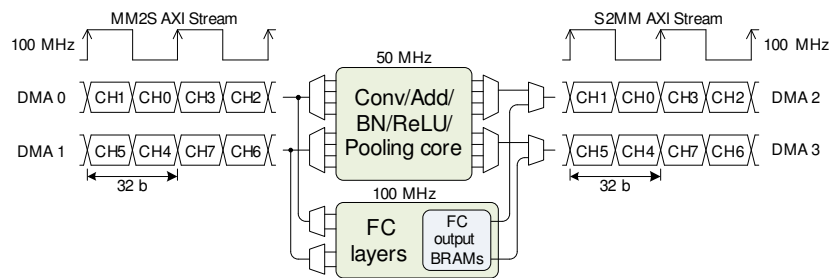


Figure 9. Four synchronous AXI streams: two to PL side; two to RAM.

The FC layers work in a 100 MHz clock domain, and on every clock cycle, they receive samples from four channels. The last FC layer returns responses of the neurons as a stream of two channels. The first half of the responses go through DMA 2, and the second half goes through DMA 3 interface.

Figure 10 illustrates how the data are transferred from RAM to multi-channel 3×3 sliding window in convolutional core. The implementation of the window is based on the 16 b depth shift registers. Every input channel has its own 3×3 window. At the beginning of the data stream, the channels of the first pixel go to the right lower corner in the window. At every cycle of the 50 MHz clock, the pixels are shifted left, according to the directions of the arrows. When pixels reach the left border of the window, they are moved to the BRAM buffers, where up to 512 pixels are reserved for a single line of the input image. The $8 \times 3 \times 3$ data from the sliding window go to the convolutional kernels when the convolution is requested by PS. In a case when the addition of two channels is required, only the central $8 \times 1 \times 1$ pixel in a sliding window is taken to the addition. The addition generates four channels on the output and uses only DMA 2 interface, in comparison with convolution where data on eight channels are generated and two output interfaces are used. While executing the addition operation, data from two parallel streams are added, as is shown by yellow squares in Figure 10. The input channels CH0 and CH4 are added and streamed back to RAM on the output CH0 and so forth on the three next output channels.

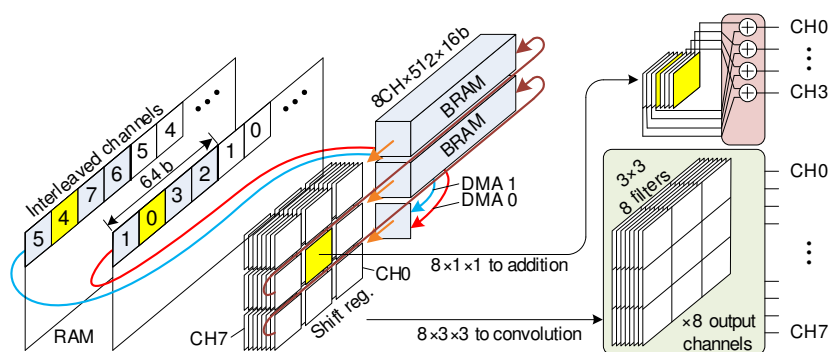


Figure 10. Data transfer to a multi-channel 3×3 sliding window and a convolution/addition core.

4.4. Multi-Channel Convolutional Core

The final architecture of the convolutional core was influenced by the throughput of AXI stream interfaces. The bottleneck lies in the bandwidth of the connection to the external DDR RAM. It is limited by the effective RAM bus width and the memory clock frequency. In the DDR configuration menu, the data width of DDR interface was set to have a maximum of 32 bits and the clock frequency had 533 MHz as a maximal value in the allowed frequency range. Therefore, the maximal bandwidth is equal to $32 \times 533 \times 10^6 \text{ b/s} = 1.98 \text{ GB/s}$. In our design, the 4 AXI interfaces work at 100 MHz and

32 b data, which demands at least $4 \times 32 \times 10^8$ b/s or 1.49 GB/s throughput. The generated stream of data meets the requirements and is smaller than the bandwidth of DDR bus. Two AXI interfaces are used in RAM to FPGA transfers preserving fixed throughput $100 \text{ MHz} \times 32 \text{ b} \times 2$ streams equivalent to 763 MB/s speed, and the next two streams work at same speed in opposite directions. The clock domain of a core was intentionally downgraded to 50 MHz on purpose to process eight input channels, instead of four channels if it worked on 100 MHz. The most important things in core design were to keep the same number of interleaved channels for the input and output, and stream back to RAM in the same order as the data were received. In order to reuse the core in the same way in the next layers, the sequence of the channels in input *MM2S* and output *S2MM* AXI streams must be kept identical.

The scaling factors A_c from the convolution layer are merged with parameters from the BN layer, creating a new factor K_c following the Equation (12). Since the 3×3 kernels are binary and the weight is reduced to one K_c , only a single DSP per filter is required. Each output channel adds responses from eight filters. Thus the proposed core utilizes 64 DSPs. The offset factors b_c are added to every output channel, as is shown in Figure 11. Depending on the requested configuration of the convolutional core, we can put on the output BN, ReLU or max pooled value. The use of optional BN and ReLU layers does not affect the total latency of the processing, since they are integrated into a core and multiplexed to the output channels if needed.

The core receives the parameters B , K and b from BRAM-based convolutional core configuration memory, presented in Figure 6. Every output \tilde{Y} of the kernel process feature maps from eight input channels, as is shown in Figure 3. The binary 3×3 kernels are implemented using 3×3 size 2 input multiplexers with selection signals controlled by the $B^{i,j}$ (4) binary values, which select the direct or negated pixels from input feature maps directly for the summation, as is shown in Figure 3. The response from binary kernel goes to the DSP for multiplication with a scaling factor K and the addition of the offset b (12). Depending on the instruction that the core receives from ARM, the output multiplexer defines the type of layer, whose results will be streamed back to RAM. It can be a batch normalized feature map or features passed through activation function or features after sub-sampling computed with the max-pooling block. The max-pooling is implemented as a 2×2 sliding window. Depending on the type of the layer requested to be computed, the valid data signals are synchronized with the outgoing stream of feature maps on the output channels. In a case in which the core is set to work in channel addition mode, the values of feature maps are passed only through the central cell of the kernel and then added with a neighbor channel on parallel DMA stream, as is shown in Figure 10.

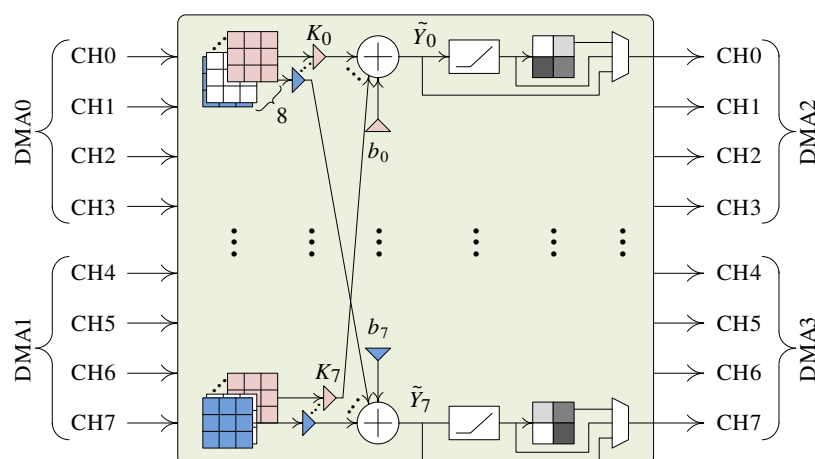


Figure 11. A core with 8 input and 8 output channels for convolution, batch normalization, ReLU and max-pooling computation.

Let us suppose that convolution is computed in a layer with eight input and eight output channels. Then only a single call of the convolutional core is needed from the PS side. What if a 4-in 4-out core were to be employed for that? Despite that, the 4-input 4-output core can be clocked at 100 MHz,

utilize 16 DSPs and maintain the same throughput on *MM2S* and *S2MM* AXI streams. However, in that case, the 4-input 4-output convolutional core needs to be called twice for every four outputs plus one call of the addition core and then repeat that procedure for the next four outputs. Each initiation of the DMA stream induces some latency to the total processing time and finally decreases the frame rate of incoming images. Therefore, we left an eight input and eight output core in a current version of CNN core. The duplication of the core makes sense when the layer outputs are not streamed to RAM to increase computational capabilities, because we cannot have twice larger throughput between the FPGA and external RAM, without affecting the synchronization between the AXI streams.

The single 8-in 8-out convolutional core must be repeatedly used to compute convolution for a larger number of input/output channels. The diagram in Figure 12 explains the process of convolution for 32-in 16-out channels. The orange blocks mark a recurrent structure that needs to be executed for every extra eight input channels. The triply repeated structure in orange blocks along with isolated start convolutional core (most left core in Figure 12) form 32 input 8 output channels. In order to get outputs for the next eight channels, the same steps need to be repeated with updated configurations of the convolutional core. The addition core needs to be called twice to form eight channels, whereas it generates only four channels for the outputs at once.

Since the convolution/addition core is based on an AXI stream, at every call of the core, the input data are received from RAM, and the responses are streamed back to the RAM (gray blocks in Figure 12). To compute convolution for 32 input and 16 output channels, the convolution and addition core must be executed 8 and 12 times, respectively. All the source and destination addresses for AXI DMA streams in conjunction with the information about streamed batch sizes are managed by *Python* developed software and coded in the instructions for convolution/addition cores.

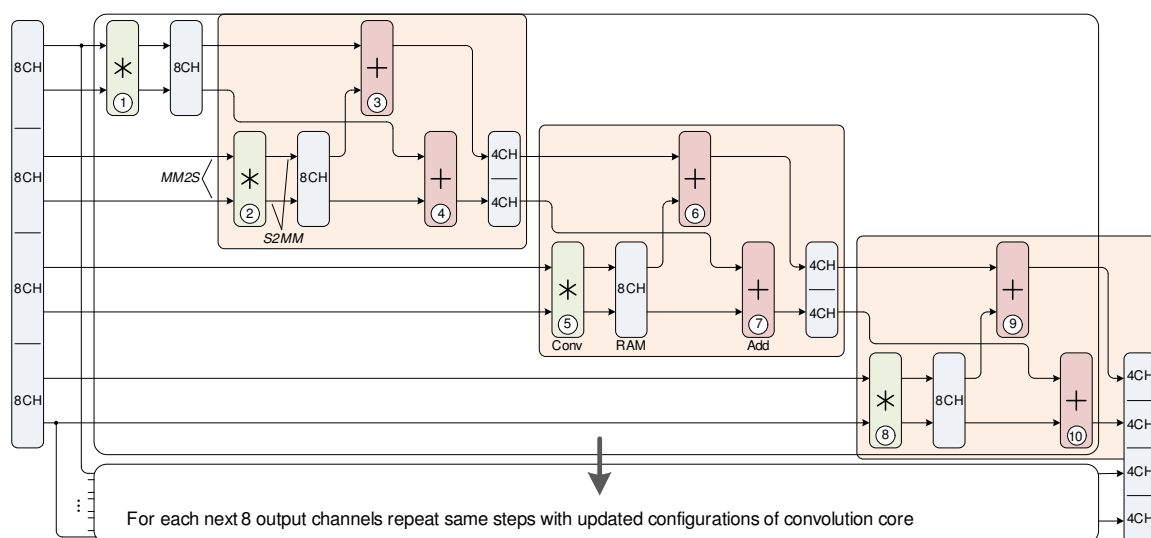


Figure 12. Core configuration plan to process 32 input and get 16 output channels. Green block—convolutional core; red block—addition core; gray block—on-board RAM; arrows—AXI DMA stream.

4.5. Implementation of Fully Connected Layers

Before running the computation of $\text{weight} \times \text{sample}$ products, the AXI DMA streams transfer bias coefficients for FC1 and FC2 layers. The 2 kB buffers are reserved for storing up to 1024 biases for each FC layer. The bias values go to the neurons through the multiplexers before the products.

It is not efficient to dedicate two separate AXI DMA streams for simultaneous transfers of samples and weights because, in that case, we need to transfer the same sample multiple times to deliver it to all the neurons in a layer. Every sample in the input in a fully connected layer is multiplied with different weights as many times as the layer has neurons. Therefore, the samples are streamed first

from RAM to FPGA just one time and then stored in BRAM. The BRAM buffers are illustrated on the most left side of the diagram in Figure 13. In a current implementation of FC layers, the maximum channel resolution on the input is limited to 64×64 pixels, and it is equivalent to $4096 \times 16 \text{ b} = 8 \text{ kB}$ of memory for each of eight reserved input channels.

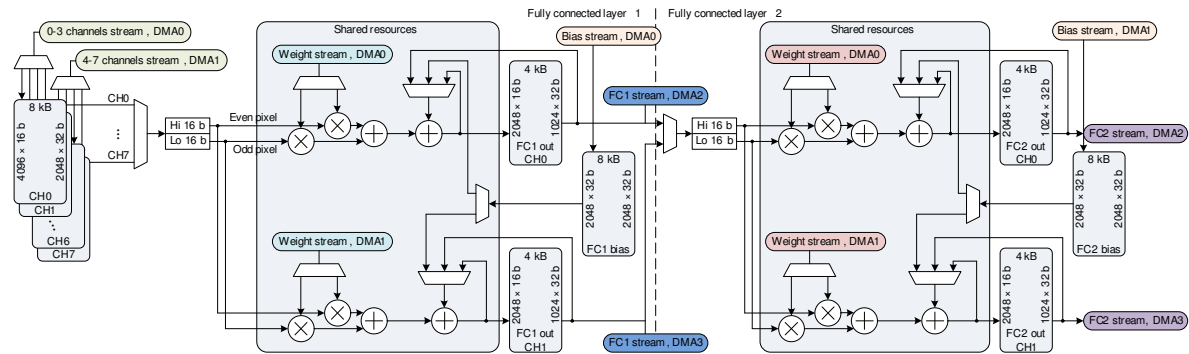


Figure 13. An example of the AXI stream-optimized architecture of two fully connected layers.

Let us say that the volume of the samples on the input to the FC1 is $32 \times 64 \times 64$, where 32 means a number of channels with 64×64 resolution. The samples from the first eight channels (CH0–CH7) are transferred to the 8 kB buffers on FPGA. Then neurons accumulate products of the AXI DMA0/1 streamed weights multiplied with samples streamed from on-chip BRAM buffers. Every AXI DMA transfers two 16 b weights per cycle at 100 MHz. Therefore, two samples (even and odd pixels) per cycle are read out from BRAM buffers to maintain the same rate. The DMA0 and DMA1 streams deliver weights for the first half of the neurons (0–511) and the second half of neurons (512–1023), respectively. A total of four DSPs and four adders are utilized to compute and accumulate four weight \times sample products simultaneously. The accumulated products of CH0–CH7 input channels are stored in two FC1 output buffers separated for DMA0 and DMA1 streams. When the computation of CH0–CH7 inputs is finished, the next eight channels (CH8–CH15) are transferred from RAM to the 8 kB buffers on FPGA. This process is repeated for all the 32 input channels in FC1. For the debugging purpose or in a case when CNN has a single FC layer, it is able to stream all the outputs of FC1 to RAM and then transfer them to the PC. The number of neurons in FC1 and FC2 is restricted to 1024 in the current implementation.

The same arithmetic resources are shared between two FC layers. To compute FC2 we do not need to stream input samples (neuron responses of FC1 layer) from RAM to FPGA, because they already are in output buffers called FC1 out CH0 and CH1 in Figure 13. The difference between FC1 and FC2 architectures lies only in a number of input channels. The FC1 has eight buffers dedicated to periodically updatable channels, and the FC2 has only two input buffers (the FC1 output buffers). When the computation of FC2 is finished, the outputs of FC2 neurons are streamed to the RAM through AXI DMA2/3.

5. Results

Different configurations of the CNN were designed, and then the parameters of the networks were modified in *Python*; afterward the kernels and FC weights were transferred to on-board RAM according to the CNN configuration process described in Figure 2. The VGG networks were selected for the implementation since they contain 3×3 size kernels suitable to run on the proposed CNN core. The VGG networks have been run on the Z-7020 Zynq SoC device on the ZedBoard. The timing results were transferred from FPGA to the PC for further comparison and performance analysis. Another reason why the VGG networks were selected for the investigation is the opportunity to compare performance and resource utilization with a few related approaches implemented on the same series of cost-optimized platforms [13,18,21].

The CNN computation engine was implemented in Vivado as an IP core and then connected with PS through AXI DMA and GPIO interfaces. Table 2 shows the summary of resource utilization after the post-implementation stage. The hardware core utilizes 67% of logic resources and 41% of flip-flops on Artix-7 FPGA. The 48 units of 4 kB block RAM were used to buffer convolution/addition core configurations, biases of FC layers and input/output data of the FC layers. The core utilizes only 31% of DSPs—64 in convolution and 4 in FC core, respectively.

Table 2. Post-implementation report of the CNN core on Z-7020 FPGA.

Resource	Available	Utilisation	Utilisation, %
LUT	53,200	35,521	67
Flip-Flops	106,400	43,405	41
4kB BRAM	140	48	34
DSP	220	68	31

It takes 19.2 μ s to transfer a single configuration batch of 15 cores from RAM to BRAM. The algorithm losses time to transfer parameters from RAM to the BRAM-based core configuration memory and next to the core. The total configuration latency increases linearly with the number of instructions in convolution layers. Table 3 provides the summarized results of the latency over all the configurations of convolution/addition cores required to run VGG networks. The first line in Table 3 takes into account only the time required to load all the convolution kernel configurations from RAM to the on-chip kernel memory based on BRAM computed as the product of number of batches times latency per batch. It does not take into consideration the PS time required to read the instruction and initialize AXI DMA transfer. It takes 0.32 us to update parameters of single convolution or addition core. Therefore last two lines in Table 3 show the total time wasted to update convolution and addition cores, respectively.

Table 3. The parameter update times (ms) of all convolution layers.

	VGG13	VGG16	VGG19
Kernel memory updates	20.91	32.70	44.51
Total convolution updates	5.23	8.17	11.12
Total addition updates	10.21	16.01	21.81

Figure 14 shows the processing time distributions of three different VGG networks. The 8-in 8-out core works in convolution mode for 0.39, 0.54 and 0.68 s in VGG13/16/19 respectively. The core is switched to work in addition mode for 0.73, 1.01 and 1.29 s, respectively. Since all tested VGG networks have the same structure of three dense layers ($4096 \times 4096 \times 1000$ neurons), we get equal 0.31 s execution delay. The major part of the latency is induced by control overhead on the PS side while reading the instruction from RAM, and then decoding the instruction and initializing the MM2S/S2MM AXI DMA streams between FPGA and DDR RAM. It takes 0.04, 0.06 and 0.08 s to update the convolutional core in VGG13/16/19 nets.

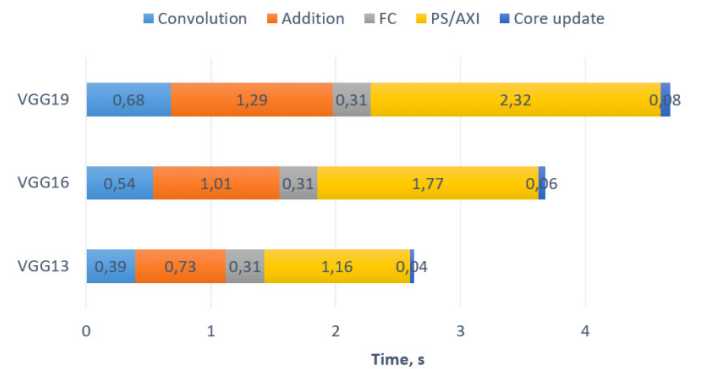


Figure 14. The distribution of processing time of VGG networks implemented with mNet2FPGA CNN core.

If we look relatively at the convolution layer execution time over all of the tested VGG13/16/19 networks, it is seen that core updating in conjunction with convolution and addition causes 44% of total latency independently from VGG net, as is shown in Figure 15. The relative time of FC layer computation decreases from 12% for VGG13 to 7% for VGG19 net, since extra convolutional layers are inserted into the nets. The execution of the PS/AXI part increases from 44% to 49% when the number of convolutional layers increases.

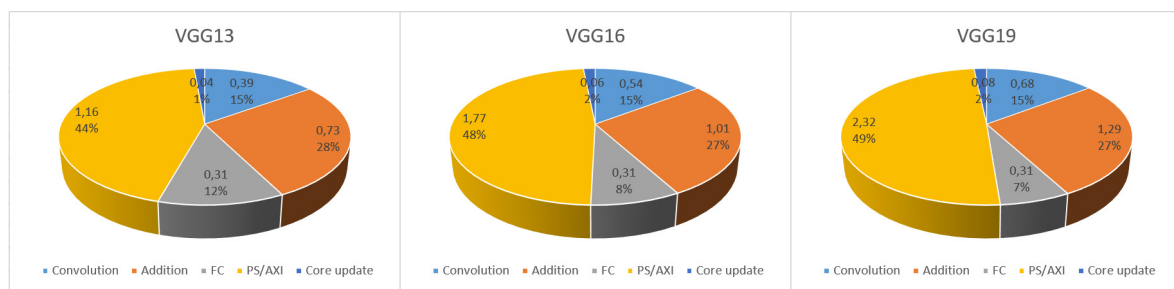


Figure 15. The comparison of relative latency required to execute VGG13/16/19 on mNet2FPGA core.

The peak throughput of the proposed convolutional core is 32.0 GOPS excluding latency of core configuration and cyclic initialization of DMA transfers from PS side. In a case when the convolutional core is configured to addition the throughput decreases to 0.2 GOPS since the data rate is tied up to AXI DMA stream and only four addition operations are executed in a cycle of 50 MHz. The peak throughput of FC core is 0.8 GOPS, whereas four multiplication and four addition operations are executed in a cycle of 100 MHz clock.

Table 4 shows the results of different VGG net implementations on Z-7020 Xilinx FPGA. VGG13/16/19 need approximately 22.62, 30.94 and 39.26 GOPS to execute a single frame. The current implementation yields 8.601, 8.408 and 8.388 GOPS respectively, averaged over all the CNN layers. Therefore, a single frame is processed in 2.63, 3.68 and 4.68 s with VGG13/16/19 nets accordingly. Due to the similar structure of FC layers in all the tested VGG, we get the same 0.31 s delay to process all dense layers. The time spent to execute convolutional layers grows linearly from 2.32 s for VGG13 to 4.37 s for VGG19. The ARM processor needs to execute more than 49k, 77k, and 105k instructions respectively with VGG13/16/19. Here 2% are the kernel memory load instructions, 33% convolution and 65% addition instructions. Up to 273 MB of free external RAM is required to store core configurations and weights of FC layers. The memory requirements are computed according to the memory arrangement diagram in Figure 5. It includes sectors beginning from *NET_BASE* and ending on *RES_BASE* address.

Table 4. Results of the CNN implementation on a Z-7020 FPGA.

	VGG13	VGG16	VGG19
Convolution instr.	16329	25545	34761
Addition instr.	31919	50031	68143
Kernel memory load instr.	1089	1703	2318
Total instructions	49337	77279	105222
Convolutional layer time, s	2.32	3.37	4.37
FC layer time, s	0.31	0.31	0.31
s/frame	2.63	3.68	4.68
Required GOP/frame	22.62	30.94	39.26
Estimated average GOPS	8.601	8.408	8.388
RAM, MB	257	265	273
Average accuracy drop, %	−3.9	−4.1	−4.4

The truncation of the parameters to 16 bit fixed-point precision decreases the accuracy of CNN's output. The quantization brings up to −4.1% accuracy loss on the images from ImageNet dataset compared to the double precision implementation of VGG16 on the PC.

Table 5 provides a summary of mNet2FPGA results in comparison with other CNN implementations on SoC FPGA for VGG16 and VGG19 networks. The compared implementations come with 8 and 16 bit fixed-point precision. All presented implementations use 7th series Xilinx programmable devices. Therefore, resources are directly comparable. The mNet2FPGA utilizes 16% of LUTs, 40% of flip-flops, 53% of DSPs and 11% of BRAMs used in NullHop. The convolution kernels are computed on 50 MHz and 60 MHz clocks on mNet2FPGA and NullHop, respectively. The dense layers in the NullHop design are computed on a PS side, while in the mNet2FPGA on the PL with the 100 MHz clock. The NullHop utilizes about two times more DSPs, six times more logic and runs on a 20% higher clock. Therefore it outperforms our implementation 4 times by a convolutional GOPS.

The summary of dynamic power shows that the PS+PL consumes 2.1 W compared to 2.3 W in NullHop. The mNet2FPGA consumes 27% less PL power than NullHop. A single frame is processed 1.82 and 1.92 times slower on mNet2FPGA with VGG16 and VGG19 networks, respectively.

Table 5. Comparison to a fpgaConvNet [13], DnnWeaver [18], Angel-Eye [17] and NullHop [21] implementations.

Implementation	mNet2FPGA	fpgaConvNet	DnnWeaver	Angel-Eye		NullHop
PL Series	Artix-7	Artix-7	Artix-7	Artix-7	Kintex-7	Kintex-7
Device	Z-7020	Z-7020	Z-7020	Z-7020	Z-7045	Z-7100
Conv. clock, MHz	50	125	150	214	150	60
FC clock, MHz	100					1 GHz (on PS)
LUTs	36K	—	35K	30K	183K	229K
Flip-Flops	43K	—	33K	35K	128K	107K
DSPs	68	220	140	190	780	128
4kB BRAMs	48	—	140	86	486	386
PS+PL Dynamic Power, W	2.129	—	—	3.5	9.63	2.339
PL Power, W	0.583	—	—	—	—	0.804
VGG16, s/frame	3.68	0.633 (CONV)	—	0.364	0.225	2.27
VGG19, s/frame	4.68	—	—	—	—	2.44
GOPS (CONV)	32	48.53	31.35	84.3	187.8	128
Precision	16 b	16 b	16 b	8 b	16 b	16 b

Due to the lack of performance results of the full VGG-16 network using the fpgaConvNet [13] and DnnWeaver [18] toolflows, we compare only the performances of convolutional layers implemented on the same Z-7020 device with 16-bit fixed-point precision. Due to the kernel binarization and normalization to the mean value, we achieved a relatively higher GOPS per DSP ratio at a lower frequency compared to fpgaConvNet and DnnWeaver implementations. Both 8 and 16 bit Angel-Eye implementations outperformed our proposed CNN design via the processing time per frame for VGG16 network due to the higher clock frequency, BRAM and DSP utilization. However, the mNet2FPGA achieved lower dynamic power.

6. Discussion

The hardware core was intentionally designed for low-density SoC FPGA and did not utilize on-board BRAM to store samples temporarily generated by the convolutional layers. All the convolutional responses are streamed back to the RAM to the address decoded from the instruction. The weak points of the proposed mNet2FPGA CNN core are the number of redundant AXI DMA transfers between FPGA and RAM in conjunction with control overhead on the PS side. Therefore, to reduce the number of transfer initializations, the scatter-gather DMA controller needs to be considered to use in the next modification of the mNet2FPGA core. In the current implementation, we have only 0.43 MB free BRAM left. The Table 6 shows minimal requirements for the size of on-chip BRAM depending on the layer index to store all the input and output samples on the chip for 16 bit VGG implementation. Due to the high resolution on the input, the most significant amount of data is generated by the first two layers. The next convolutional layers reduce memory size by two after the max-pooling. Therefore, some architectural design modifications are required to be introduced to reduce the number of DMA transfers and place some of the responses from convolutional layers in the unemployed BRAM for fast access of the samples during the execution of the next layer.

Table 6. Minimal requirements of on-board RAM for the first 6 convolutional layers' implementation of VGG-13/16/19 nets.

Layer	L1	L2	L3	L4	L5	L6
Read, MB	0.144	6.125	6.125	3.063	3.063	1.531
Write, MB	6.125	6.125	3.063	3.063	1.531	1.531
Time, ms	8.0	64.2	32.1	64.2	32.1	64.2

To reduce the number of DMA transfers, we need to add optional buffers on the outputs of the proposed CNN core. Depending on the selected SoC FPGA and the size of feature maps generated by the convolutional layers, the data on the output channels need to be directed to on-chip BRAM if free memory resources are available. Additionally, the input feature maps need to be loaded from BRAM while executing a subsequent layer. That improvement will allow speeding up the processing up to two times because the current CNN core loses time waiting for PS/AXI transfers, as shown in Figures 14 and 15.

The main reason for the relatively low 50 MHz clock frequency of the convolutional core is the synchronization of the core with the AXI stream of data. Two AXI channels give 64 bits at 100 MHz. It is four parallel streams of 16 bit interleaved sequence of samples. From those streams, the convolutional core extracts eight channel feature maps, and therefore it should run at 50 MHz ($4 \text{ CH} \times 16 \text{ b} \times 100 \text{ MHz} = 8 \text{ CH} \times 16 \text{ b} \times 50 \text{ MHz}$). If the internal buffers are used to store input/output feature maps in separate BRAMs, then clock frequency can be increased at least twice because the feature maps will not come outside the FPGA and the processing speed will be not limited by throughput between FPGA and external RAM.

7. Conclusions

In this work, we proposed an approach to map fixed-point CNNs to low-density SoC FPGA devices. To create a network, the standard layers such as convolution, batch normalization, rectified linear unit, max-pooling and fully connected from the Neural Network Toolbox may be used. The data transfers between RAM and hardware cores are based on AXI DMA streams. Moreover, the latency results show that a significant part of the time is wasted on instruction decoding and AXI stream initialization. All the responses from intermediate convolution layers are streamed back to RAM; therefore, the complexity of the CNN is restricted only by the memory size. The performance of the CNN core was tested with VGG13, VGG16 and VGG19 networks achieving 8.6, 8.4 and 8.4 GOPS, respectively. It takes 2.63, 3.68 and 4.68 s to process a single 224×224 frame with VGG13, VGG16 and VGG19 respectively. The CNN core was implemented on a cost-optimized Z-7020 FPGA with 16-bit fixed-point operations, and it utilizes 67% of logic, 41% of flip-flop, 30% of BRAM and 31% of DSP resources. The 32 GOPS performance of the single convolutional core is similar to the performance obtained with existing CNN-to-FPGA toolflows.

The future work is to make a flexible size for the convolutional kernel and append common modifications of activation layers to investigate various types of CNN. As for the introduced simplifications in RAM addressing, we have a lot of scattered unused locations in memory. We are capable of reducing the RAM utilization rate by up to two times in the current version of the CNN core after the restructuring of the order of the parameters in the kernel configuration memory. The duplication of convolutional cores in dense SoC FPGA in conjunction with on-chip BRAM utilization for the temporary storage of layer outputs and reduction of AXI streams will speed up the execution of CNNs.

Author Contributions: Conceptualization, T.S.; methodology, T.S. and A.S.; software, T.S.; validation, A.S.; investigation, T.S. and A.S.; writing—original draft preparation, T.S. and A.S.; visualization, T.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research is supported by Central Project Management Agency (Vilnius, Lithuania), project number 01.2.2-CPVA-K-703-02-0017.

Acknowledgments: The authors would like to thank editors and reviewers for many constructive suggestions and comments that helped to improve the quality of the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rastegari, M.; Ordonez, V.; Redmon, J.; Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*; Springer: New York, NY, USA, 2016; pp. 525–542.
2. Nguyen, D.T.; Nguyen, T.N.; Kim, H.; Lee, H.J. A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2019**, *27*, 1861–1873. [[CrossRef](#)]
3. Machado, J.B.; Givigi, S.N. Convolutional Neural Networks as Asymmetric Volterra Models Based on Generalized Orthonormal Basis Functions. *IEEE Trans. Neural Netw. Learn. Syst.* **2019**, *31*, 950–959. [[CrossRef](#)] [[PubMed](#)]
4. Ioffe, S.; Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv* **2015**, arXiv:1502.03167.
5. Courbariaux, M.; Hubara, I.; Soudry, D.; El-Yaniv, R.; Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv* **2016**, arXiv:1602.02830.
6. Liang, S.; Yin, S.; Liu, L.; Luk, W.; Wei, S. FP-BNN: Binarized neural network on FPGA. *Neurocomputing* **2018**, *275*, 1072–1086. [[CrossRef](#)]
7. Chen, J.; Liu, L.; Liu, Y.; Zeng, X. A Learning Framework for n-Bit Quantized Neural Networks Toward FPGAs. *IEEE Trans. Neural Networks Learn. Syst.* **2020**, *Early Access*, 1–15.

8. Wang, J.; Lin, J.; Wang, Z. Efficient hardware architectures for deep convolutional neural network. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2017**, *65*, 1941–1953. [[CrossRef](#)]
9. Lian, X.; Liu, Z.; Song, Z.; Dai, J.; Zhou, W.; Ji, X. High-performance fpga-based cnn accelerator with block-floating-point arithmetic. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2019**, *27*, 1874–1885. [[CrossRef](#)]
10. Guo, K.; Han, S.; Yao, S.; Wang, Y.; Xie, Y.; Yang, H. Software-hardware codesign for efficient neural network acceleration. *IEEE Micro* **2017**, *37*, 18–25. [[CrossRef](#)]
11. Tapiador-Morales, R.; Linares-Barranco, A.; Jimenez-Fernandez, A.; Jimenez-Moreno, G. Neuromorphic LIF row-by-row multiconvolution processor for FPGA. *IEEE Trans. Biomed. Circuits Syst.* **2018**, *13*, 159–169. [[PubMed](#)]
12. Chung, J.; Shin, T.; Yang, J.S. Simplifying deep neural networks for FPGA-like neuromorphic systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *38*, 2032–2042. [[CrossRef](#)]
13. Venieris, S.I.; Kouris, A.; Bouganis, C.S. Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 56. [[CrossRef](#)]
14. Wai, Y.J.; Bin, Z.; Irwan, S.; Kim, L. Fixed Point Implementation of Tiny-Yolo-v2 using OpenCL on FPGA. *Int. J. Adv. Comput. Sci. Appl.* **2018**, *9*, 506–512. [[CrossRef](#)]
15. Venieris, S.I.; Bouganis, C.S. Latency-driven design for FPGA-based convolutional neural networks. In Proceedings of the 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, 4–8 September 2017; pp. 1–8.
16. Zhang, C.; Sun, G.; Fang, Z.; Zhou, P.; Pan, P.; Cong, J. Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *38*, 2072–2085. [[CrossRef](#)]
17. Guo, K.; Sui, L.; Qiu, J.; Yu, J.; Wang, J.; Yao, S.; Han, S.; Wang, Y.; Yang, H. Angel-Eye: A Complete Design Flow for Mapping CNN onto Embedded FPGA. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2017**, *37*, 35–47. [[CrossRef](#)]
18. Sharma, H.; Park, J.; Mahajan, D.; Amaro, E.; Kim, J.K.; Shao, C.; Mishra, A.; Esmaeilzadeh, H. From high-level deep neural models to FPGAs. In Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 15–19 October 2016; pp. 1–12.
19. Meloni, P.; Capotondi, A.; Deriu, G.; Brian, M.; Conti, F.; Rossi, D.; Raffo, L.; Benini, L. NEURA ghe: Exploiting CPU-FPGA Synergies for Efficient and Flexible CNN Inference Acceleration on Zynq SoCs. *ACM Trans. Reconfigurable Technol. Syst. (TRETTS)* **2018**, *11*, 18.
20. Gong, L.; Wang, C.; Li, X.; Chen, H.; Zhou, X. Maloc: A fully pipelined fpga accelerator for convolutional neural networks with all layers mapped on chip. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *37*, 2601–2612. [[CrossRef](#)]
21. Aimar, A.; Mostafa, H.; Calabrese, E.; Rios-Navarro, A.; Tapiador-Morales, R.; Lungu, I.A.; Milde, M.B.; Corradi, F.; Linares-Barranco, A.; Liu, S.C.; et al. Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE Trans. Neural Netw. Learn. Syst.* **2018**, *30*, 644–656. [[CrossRef](#)] [[PubMed](#)]
22. Ma, Y.; Cao, Y.; Vruthula, S.; Seo, J.s. Automatic compilation of diverse cnns onto high-performance fpga accelerators. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *39*, 424–437. [[CrossRef](#)]
23. Yu, Y.; Wu, C.; Zhao, T.; Wang, K.; He, L. OPU: An FPGA-Based Overlay Processor for Convolutional Neural Networks. *IEEE Trans. Very Large Scale Integr. (Vlsi) Syst.* **2019**, *28*, 35–47. [[CrossRef](#)]
24. Guan, Y.; Liang, H.; Xu, N.; Wang, W.; Shi, S.; Chen, X.; Sun, G.; Zhang, W.; Cong, J. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, USA, 30 April–2 May 2017; pp. 152–159.
25. Nakahara, H.; Yonekawa, H.; Fujii, T.; Sato, S. A Lightweight YOLOv2. *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays—FPGA '18*; ACM Press: New York, NY, USA, 2018; pp. 31–40. [[CrossRef](#)]

