MDPI

Article

A Heterogeneous Hardware Accelerator for Image Classification in Embedded Systems

Ignacio Pérez and Miguel Figueroa *

Department of Electrical Engineering, Universidad de Concepción, Concepción 4070386, Chile; ignperez@udec.cl
* Correspondence: miguel.figueroa@udec.cl

Abstract: Convolutional neural networks (CNN) have been extensively employed for image classification due to their high accuracy. However, inference is a computationally-intensive process that often requires hardware acceleration to operate in real time. For mobile devices, the power consumption of graphics processors (GPUs) is frequently prohibitive, and field-programmable gate arrays (FPGA) become a solution to perform inference at high speed. Although previous works have implemented CNN inference on FPGAs, their high utilization of on-chip memory and arithmetic resources complicate their application on resource-constrained edge devices. In this paper, we present a scalable, low power, low resource-utilization accelerator architecture for inference on the MobileNet V2 CNN. The architecture uses a heterogeneous system with an embedded processor as the main controller, external memory to store network data, and dedicated hardware implemented on reconfigurable logic with a scalable number of processing elements (PE). Implemented on a XCZU7EV FPGA running at 200 MHz and using four PEs, the accelerator infers with 87% top-5 accuracy and processes an image of 224 × 224 pixels in 220 ms. It consumes 7.35 W of power and uses less than 30% of the logic and arithmetic resources used by other MobileNet FPGA accelerators.

Keywords: convolutional neural network; MobileNet V2; field-programmable gate array; power consumption

check for updates

Citation: Pérez, I.; Figueroa, M. A Heterogeneous Hardware Accelerator for Image Classification in Embedded Systems. *Sensors* **2021**, *21*, 2637. https://doi.org/10.3390/s21082637

Academic Editor: Stefania Perri

Received: 29 January 2021
Accepted: 5 April 2021
Published: 9 April 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Convolutional neural networks (CNNs) [1] have been widely used in object detection, image classification, and semantic segmentation because of their high accuracy. Compared to other image-classification techniques such as Local Binary Patterns (LBP), Scale Invariant Feature Transform (SIFT), K-Nearest Neighbor (KNN) or Support Vector Machines (SVM), CNNs show better robustness when performing classification with large databases, achieving better accuracy in their results [2–4]. Therefore, CNNs have occupied a fundamental role in the development of different applications, such as video surveillance [5], autonomous and assisted driving [6], assistance navigation for blind and visually impaired people [7], detection of defects in structures [8], and clinical assistance [9,10]. CNNs perform image classification during inference, a process that depends on the architecture of the network and yields different results depending on the CNN type. In particular, MobileNet [11,12] is a CNN architecture that features similar accuracy to VGG, GoogLeNet, and ResNet [13–15], and better accuracy compared to AlexNet and SquezeNet [16,17], while using fewer parameters than these other networks.

Even with the reduced number of parameters of MobileNet, inference is a computationally expensive process because it performs a large number of mathematical operations on the input and intermediate data, and requires fast access to a large number of parameters. In many cases, this inference is performed online on the input data. A notable case is embedded systems, which frequently operate on images or video acquired from their environment, and uses inferences to make decisions in real time [18]. Because embedded processing hardware faces severe restrictions in computational resources and power consumption, performing CNN inference in real time is a challenging task.

Graphics processing units (GPUs) are an attractive platform to implement CNN inference with high performance because they can exploit the large data parallelism available in these algorithms to perform more than one order of magnitude faster than traditional processors [19,20]. However, they reach power consumptions of up to 200 W [21], making it difficult to use them in portable and mobile devices [22]. Embedded GPUs use custom acceleration, such as the Nvidia TensorRT environment in the Jetson family, to reduce power consumption compared to traditional GPUs [23], but their power consumption is still high compared to dedicated hardware solutions on reconfigurable hardware platforms such as field-programmable gate arrays (FPGAs) [24]. Dedicated hardware accelerators for neural networks, such as Google Coral [25], offer very good performance and power efficiency, but their architecture limits their programmability and the ability to dynamically retarget the hardware for other tasks in a video-processing pipeline.

FPGAs are hardware platforms that can implement custom architectures for a wide variety of algorithms with a high level of fine-grained data parallelism. Moreover, because an FPGA can be dynamically reconfigured, its hardware can be shared between different tasks in an application using time multiplexing. Indeed, recent published work has shown implementations of MobileNet and other CNN inference algorithms on FPGA [26,27]. However, these implementations use large devices with relatively high power consumption that makes it difficult to use them in edge devices for applications that need to achieve a balance between energy efficiency and inference speed [28]. One such application is object or face recognition in mobile devices, where a trade-off between video frame rate and power consumption can be achieved, and often an inference speed of one frame per second (fps) is sufficient to classify all faces of interest in the input images [29]. Another example is the semantic segmentation of images in drones and nanosatellites, which favor architectures with high accuracy, a small number of parameters, and compact implementation on low-power edge devices [30].

In this paper, we describe the architecture of a heterogeneous hardware accelerator for CNN inference using the MobileNet V2 network. The architecture combines an embedded processor and reconfigurable hardware to achieve low power and resource utilization, with an inference speed suitable for most embedded video applications. Our design uses loop tiling to reuse data, pruning to eliminate parameters in CNN, and quantization to compress the parameters of the network. These techniques allow us to reduce the effective size of the CNN and efficiently implement it on an FPGA. As a result, our accelerator uses fewer on-chip memory and logic/arithmetic resources other MobileNet implementations in the literature and has lower power consumption compared to other embedded devices. We designed and implemented the accelerator using high-level synthesis (HLS) and describe the design space exploration to maximize inference performance. The main contributions of our work are:

- We designed a heterogeneous architecture on programmable hardware to accelerate MobileNet V2 inference with lower resource utilization and power consumption than other published work. This allows our design to be synthesized on edge devices for applications that favor low resource usage over high inference speed.
- We use loop tiling, loop unrolling, pruning, and quantization techniques to maximize the inference performance of the MobileNet V2 network and, at the same time, maintain low power consumption and resource usage on our accelerator.
- Our implementation on a Xilinx XCZU7EV FPGA running at 200 MHz consumes 29 times less power than a desktop processor, and 5 times less than an embedded GPU. It also uses 30% of the on-chip memory resources and 25% of the arithmetic resources of other published MobileNet FPGA accelerators.

The rest of this paper is organized as follows: Section 2 shows related work, detailing design techniques and CNN FPGA accelerators. Section 3 details the MobileNet V2 model, shows the base data for training and inferring, and explains the techniques used to reduce, accelerate, and implement the CNN in the architecture. Section 4 describes the architecture and the design space exploration of our custom CNN accelerator.

Section 5 shows experimental results and compares them with other published works. Finally, Section 6 presents the conclusions and future work.

2. Related Work

2.1. CNN Inference on FPGAs

Inference process in CNNs is done in two stages: convolutional layers, which are used to extract patterns or features maps of the images, and classification layers, used to classify the features. In the convolutional stage, each layer applies N convolutions on the input map, where N is the number of channels or depth of the output map. Then, an activation function removes the pixels of the output map that are not relevant, and a reduction function reduces the size of the activation map. The CNN repeats this stage according to the model it uses. In the classification stage, the CNN linearly associates the feature maps to obtain C output data elements, where C is the number of categories that the CNN can recognize. The outputs of the classification layer represent the probability of classification of the input image in each category.

Performing CNN inference on an FPGA is a challenging problem, due to the limited logic, arithmetic, and memory resources available on the device, and the performance limitations imposed by the reconfigurable hardware [31]. CNN inference requires performing millions or billions of arithmetic operations in each layer [32]; moreover, a CNN typically uses millions of parameters making it impossible to store all weights and activations on the on-chip memory available on most FPGAs [33,34]. Therefore, most accelerators store data off chip, which increases inference time due to the limited memory bandwidth available on the device [35].

Recently, research has used different methods to solve these problems. Published work [32,36–38] proposes using loop tiling and loop unrolling to reuse weights and activations, reducing the size of data in on-chip memories, removing bubbles in the pipeline and parallelizing operations. Many approaches [32,39–43] use quantization strategies, which reduce the number of bits used to represent weights and activations. Other optimizations [44–48] apply pruning and fine-tuning techniques to reduce the number of parameters of the network. Because quantization and pruning reduce the size of the CNN, these techniques both speed up computation and increase fraction of the parameters that can be stored in on-chip memory.

CNN accelerators described in the literature have used the techniques listed above for CNN inference in FPGAs. For example, the architecture described in [32] implements VGG16 [13] using singular value decomposition (SVD), loop tiling, and loop unrolling, achieving inference at 4.45 fps. The authors in [39] use a design flow with quantization and pruning techniques to implement different CNNs on dedicated hardware and reach 2.75 fps in VGG16. The architecture described in [26] uses quantization, pruning, and loop tiling to store all feature maps and weights in on-chip memories, processing 32 channels in parallel on each processing element (PE) to speed up MobileNet [11] inference, and achieve 127 fps. The hardware accelerator described in [27] uses MobileNet V2 [12] with 16-bit quantization. This design stores all the feature maps in on-chip memories using a large FPGA. The architecture stores the weights in off-chip memories and transfers them to a buffer in the FPGA using direct memory access (DMA) in Scatter-Gather (SG) mode. Moreover, the accelerator uses four PEs, where each can process 32 channels in parallel, and achieves 266 fps. DiracDeltaNet [40] is based on ShuffleNet [49], but replaces convolutions with shift operations and uses PACT quantization to classify at 58.7 fps on an FPGA. The architecture described in [50] uses reverse-pruning and peak-pruning strategies to improve the compression factor in AlexNet [16] without sacrificing accuracy. The authors of [51] create a design flow to implement CNN inference in FPGA-based SoCs using high-level synthesis (HLS). The design flow uses Matlab to quantize and binarize the parameters and the algorithm of CNN. The workflow transforms the algorithm into an HLS C/C++ implementation to implement it on an FPGA. The authors use the design flow in SqueezeNet [17] and achieve a throughput of 14.2 fps.

Table 1 summarizes the results of the FPGA-based CNN implementations described above using the standard ImageNet dataset [52]. It includes the FPGA platform used, the CNN architecture and techniques used in the designs, the resource utilization, and the frame rate supported reported by the accelerator. CNN implementations with more parameters, such as AlexNet and VGG, infer each image in more time than smaller CNNs such as MobileNet V1 and V2. This is because the AlexNet and VGG implementations need to store weights and activations in off-chip memory, adding data-access latency. On the other hand, the reported MobileNet implementations operate at over 100 fps because they store all the weights and activations in on-chip memory using large FPGAs, which reduces latency and increases parallelism. However, these implementations are resource-intensive, particularly using a large number of on-chip memory blocks (BRAMs) and multiplier/adders (DSP slices), which makes it difficult to implement them on lower-end devices with limited resources. General-logic (LUT) resource usage is high on both types of CNN architectures.

Table 1. FPGA-based CNN inference described in the literature (N/A: Not available).

	Qiu et al. [32]	**Guo et al. [39]**	**Su et al. [26]**	**Bai et al. [27]**	**Yang et al. [40]**	**Zhang et al. [50]**	**Panagiotis et al. [51]**
Year	2016	2018	2018	2018	2019	2019	2020
FPGA	XC7Z045	XC7Z045	XCZU9EG	Arria 10 SX	XCZU3EG	XCZU7EV	XC7Z020
Freq. (MHZ)	150	214	150	133	250	300	100
CNN	VGG-16	VGG-16	MobileNet	MobileNet V2	DiracDeltaNet	AlexNet	SqueezeNet
Reduction	SVD	N/A	Pruning	N/A	N/A	Pruning	N/A
Quantization	16 bits	8 bits	8–4 bits	16 bits	1–4 bits	8 bits	8 bits
LUT	182,616	29,867	139,000	163,506	24,130	101,953	34,489
BRAM	486	85.5	1729	1844	170	198.5	97.5
FF	127,653	35,489	55,000	N/A	29,867	127,577	25,036
DSP	780	190	1452	1278	37	696	172
Power (W)	9.63	3.5	N/A	N/A	5.5	17.67	N/A
Perf. (GOPS)	136.97	84.3	91.2	170.6	47.09	14.11	N/A
fps	4.45	2.75	127	266	58.7	9.73	14.2
Top-1	N/A	67.72%	64.6%	71.8%	70.1%	55.99%	56.94%
Top-5	86.66%	88.06%	84.5%	91.0%	88.2%	N/A	79.94%

2.2. Other Image Classification Algorithms on FPGAs

As mentioned in Section 1, algorithms like SIFT or SVM can also be used to classify images. Like CNNs, these algorithms have been implemented on FPGAs to speed up the classification. The architecture described in [53] combines SIFT to extract image features and SVM for classification. The authors implement the architecture on a Virtex 5 FPGA, using 38,000 LUTs, 9000 registers, and 52 DSP blocks. The hardware implementation can infer an image of the Caltech-256 database in 0.25 ms, 5.72 times faster than an equivalent software implementation. The hardware accelerator described in [54] uses SVM to detect melanoma on an FPGA. The authors use HLS to design an SVM classifier and implement it on an FPGA as a heterogeneous system using a Zynq-7 ZC702 evaluation board. The architecture uses 17,500 LUTs, 48 BRAMs, and 5 DSPs to classify an image in 11.46 µs, consuming 2 W of power. Although these implementations use fewer resources and are faster than CNNs, they achieve lower classification accuracy on large databases, thus limiting their applicability [2,4].

3. Methods

This section shows the MobileNet V2 CNN architecture, the dataset used in our evaluation, the modifications that we made to the network to implement it on dedicated hardware, and the architecture of the accelerator.

3.1. MobileNet V2

MobileNet V2 has two types of layers: (1) convolutional layers, grouped into bottleneck blocks that combine standard, depthwise, and expansion/projection convolutions with batch normalization and activation functions, and (2) classification layers, which use pooling and fully connected layers.

3.1.1. Convolutional Layers

Figure 1 shows the standard, depthwise, and expansion/projection convolutions of MobileNet V2. N_i and N_o are the number of input and output channels, respectively, M_i and M_o are the size of the input and output feature maps, and K is the size of convolutional masks ($K = 3$ in MobileNet V2). MobileNet V2 uses standard convolutions only in the first layer to combine the RGB channels of the input images. To calculate each output channel, the CNN performs the sum of the N_i convolutions between each input channel and the corresponding convolutional mask. Depthwise convolutions use $K \times K = 3 \times 3$ masks and a single convolution to extract features on each output channel. Expansion/projection convolutions calculate each output channel by adding the N_i convolutions of each input channel, but replacing the 3×3 mask with 1×1 weights, transforming convolutions into multiplications. Expansion convolutions increase the number of channels, while projection convolutions reduce it.

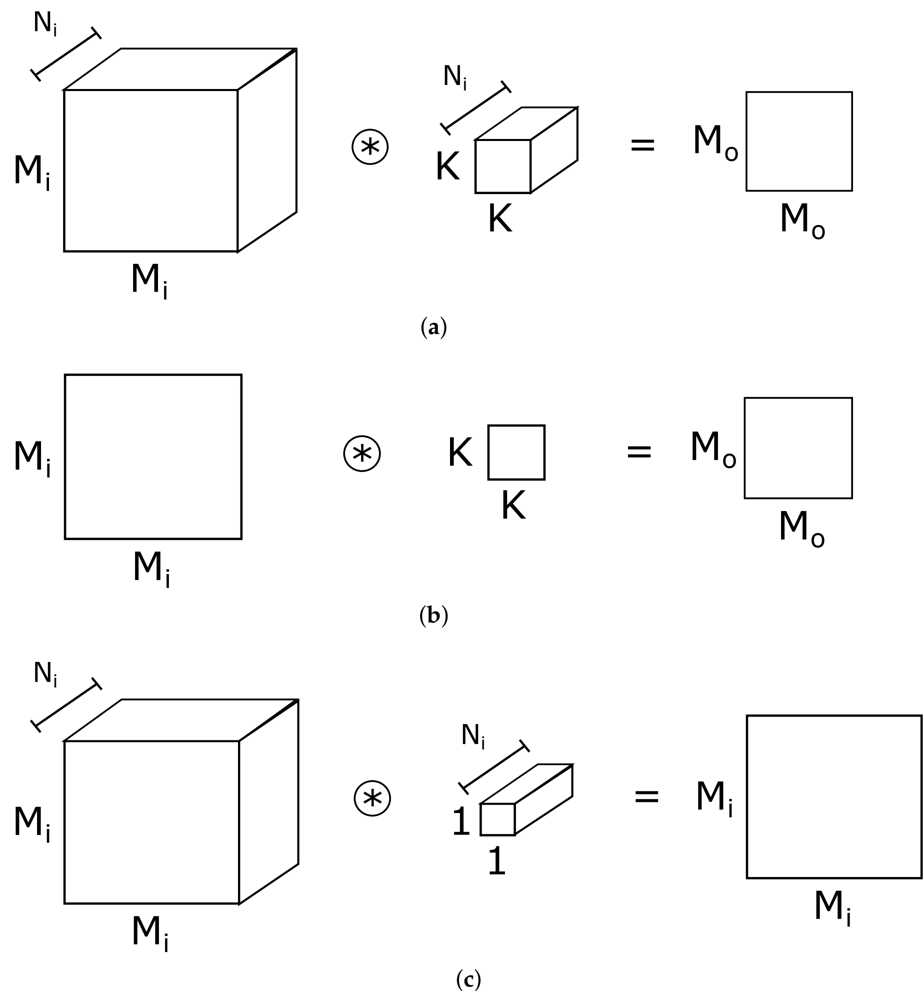

Figure 1. Convolution types in MobileNet V2. (**a**) standard convolution; (**b**) depthwise convolution; (**c**) expansion/projection convolution.

MobileNet V2 uses batch normalization [55] to improve the speed, performance, and stability of training and inference at convolutional layers. Equation (1) shows the batch normalization:

$$y_{norm} = \frac{y_{conv} - \mathrm{E}[y_{conv}]}{\sqrt{\mathrm{Var}[y_{conv}] + \epsilon}} \times \gamma + \beta \tag{1}$$

where y_{conv} and y_{norm} are the input and output feature maps, $\mathrm{E}[y_{conv}]$ and $\mathrm{Var}[y_{conv}]$ are the mean and variance of y_{conv}, γ and β are multiplicative and additive weights, and ϵ is the stability coefficient. During inference, $\mathrm{E}[y_{conv}]$ and $\mathrm{Var}[y_{conv}]$ are constant values, which are computed during training. As an activation function, MobileNet V2 uses ReLU6(), which saturates with input values less than zero and greater than six, and helps to maintain CNN stability.

MobileNet V2 combines convolution, normalization, and activation stages into bottleneck blocks, which are shown in Figure 2. Bottleneck blocks increase the number of channels with expansion convolutions, extract features with depthwise convolutions, and reduce the output depth with projection convolutions. Figure 2b shows a variant of a bottleneck block that uses residual layers [15]. Residuals allow the CNN to increase the number of layers, improving inference precision. These layers add the input feature map x_{conv} and the output of projection convolution y_{proj} to compute the output feature map y_{res}.

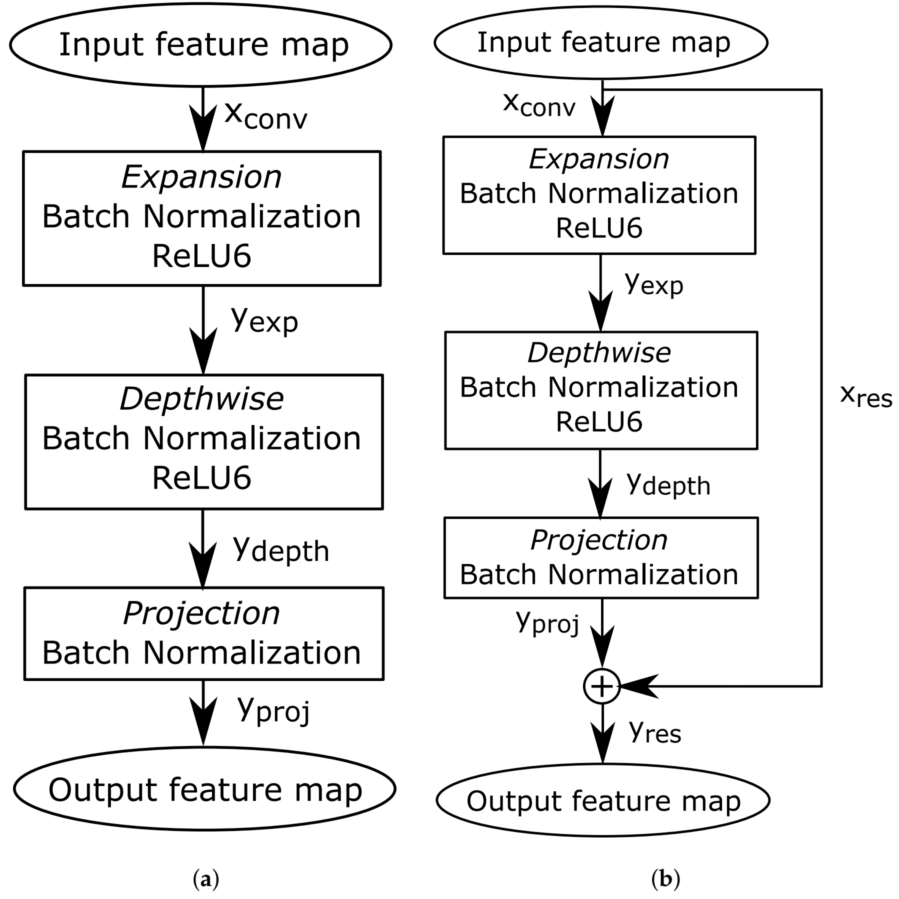

Figure 2. Bottleneck blocks. (**a**) bottleneck block without residual layer; (**b**) bottleneck block with residual layer.

3.1.2. Classification Layers

MobileNet V2 uses an average pooling function to transform three-dimensional feature maps to a one-dimensional array. The pooling function averages the pixels of each channel of the feature map to generate a vector of size N_o, which is the depth of the output feature

map of the last convolution layer. Then, the CNN applies a fully-connected layer to classify the features in the array. Equation (2) shows the computation of the fully connected layer:

$$y_{fc}[i] = b_{fc}[j] + \sum_{j=0}^{N_o-1} W_{fc}[i,j] \times y_{avg}[j] \quad \text{with} \quad 0 \leq i \leq N_{class} \tag{2}$$

The CNN linearly associates each component of the array y_{avg} with each other by adding the product between each component and weight W_{fc} to the bias b_{fc}. The output array y_{fc} represents the classification probabilities of the N_{class} categories in the input image.

3.1.3. MobileNet V2 Model

Table 2 shows the MobileNet V2 model. It uses a standard convolution in the first layer to combine the RGB channels, 18 bottleneck blocks to extract features, and an average pooling and a fully connected layer to classify the features. The neural network has 3.4 million parameters.

Table 2. MobileNet V2 model.

Input ($MxMxN_i$)	Output ($MxMxN_o$)	Layer	Repeat Time	Expansion Factor	Stride	Residual	Thousands of Parameters
$224 \times 224 \times 3$	$112 \times 112 \times 32$	Standard conv	1	-	2	No	0.8
$112 \times 112 \times 32$	$112 \times 112 \times 16$	Bottleneck	1	1	1	No	0.7
$112 \times 112 \times 16$	$56 \times 56 \times 24$	Bottleneck	2	6	2	Yes	12.9
$56 \times 56 \times 24$	$28 \times 28 \times 32$	Bottleneck	3	6	2	Yes	37.3
$28 \times 28 \times 32$	$14 \times 14 \times 64$	Bottleneck	4	6	2	Yes	177.9
$14 \times 14 \times 64$	$14 \times 14 \times 96$	Bottleneck	3	6	1	Yes	296.0
$14 \times 14 \times 96$	$7 \times 7 \times 160$	Bottleneck	3	6	2	Yes	784.2
$7 \times 7 \times 160$	$7 \times 7 \times 320$	Bottleneck	1	6	1	No	469.4
$7 \times 7 \times 320$	$7 \times 7 \times 1280$	Expansion conv	1	-	1	No	409.6
$7 \times 7 \times 1280$	$1 \times 1 \times 1280$	Avg pooling	1	-	-	-	0.0
$1 \times 1 \times 1280$	$1 \times 1 \times 1000$	FC	1	-	-	-	1280.0

3.2. *Imagenet Dataset*

To test their performance on object recognition, CNNs typically use the ImageNet dataset [52]. ImageNet has about 1.3 million high-definition RGB images divided into 1000 categories. The dataset has a training group with 1.3 million images, a validation group with 50,000 images , and a test group with 100,000 unlabeled images , which are used to compute the accuracy of the network.

3.3. *Complexity-Reduction Techniques*

Following the experience published in the literature [36,43,46,56], we applied different techniques to MobileNet V2 to reduce its complexity before designing the architecture of the hardware accelerator. We merged batch normalization into convolutions, divided the activations and weights using loop tiling, and used pruning and quantization techniques to reduce the size of the network.

3.3.1. Batch Normalization

Batch normalization is complex to implement on FPGA hardware because it requires computing a division and a square root. These operations are resource-intensive and add significant latency. To simplify the FPGA implementation, we merged batch normalization with convolutions [56], by modifying the convolutional masks and adding a bias. We factored Equation (1) into the computation of y_{conv} to obtain Equation (3):

$$y_{norm} = \frac{\gamma}{\sqrt{\text{Var}[y_{conv}] + \epsilon}} \times y_{conv} + (\beta - \frac{\text{E}[y_{conv}]}{\sqrt{\text{Var}[y_{conv}] + \epsilon}} \times \gamma) = W_{norm} \times y_{conv} + b_{norm} \quad (3)$$

where $W_{norm} = \frac{\gamma}{\sqrt{\text{Var}[y_{conv}]+\epsilon}}$ are the weights and $b_{norm} = (\beta - \frac{\text{E}[y_{conv}]}{\sqrt{\text{Var}[y_{conv}]+\epsilon}} \times \gamma)$ is the bias of the normalization. Then, we combined Equation (3) with the convolution $y_{conv} = x_{conv} \times W_{conv}$ to obtain Equation (4):

$$\begin{aligned} y_{norm} &= (W_{norm} \times W_{conv}) \times x_{conv} + b_{norm} \\ y_{norm} &= W_{cn} \times x_{conv} + b_{norm} \end{aligned} \quad (4)$$

where x_{conv} and y_{conv} are the input and output maps, respectively, and W_{conv} are the convolutional weights. Equation (4) shows the batch normalization process folding into convolutions, with W_{cn} and b_{norm} being the new convolutional weights and bias, respectively.

3.3.2. Loop Tiling

We used loop tiling to reuse the data stored on the limited on-chip memory in an FPGA. Figure 3 shows the division used on the maps and parameters to apply the technique. We separated the activations and weights into blocks of size $T_M \times T_M \times T_N$ and $T_N \times T_N$, where T_M and T_N are tiling factors used on the map size and layer depth, respectively. Each iteration of the inference computes one block at a time, reducing the number of on-chip memory blocks used in this part of the architecture.

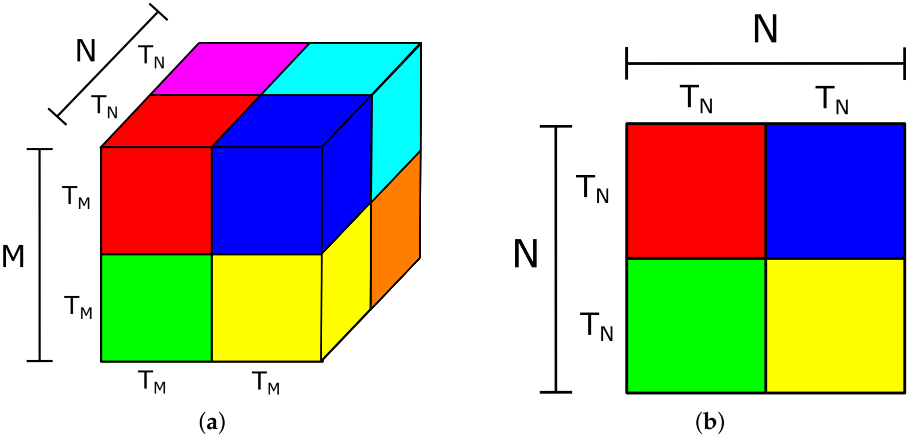

Figure 3. (**a**) Loop tiling in feature maps; (**b**) loop tiling in weights.

3.3.3. Pruning

Although pruning allows us to reduce the number of weights and arithmetic operations, it must be applied carefully so that each PE processes a similar amount of data. Otherwise, load unbalance will negatively affect the performance of the accelerator. We applied the bank-balanced pruning technique proposed in [46], which groups data in blocks and uses pruning to remove the same number of parameters in each block. Figure 4 shows the bank-balanced pruning that we used on MobileNet V2. We divided the weights into blocks of size T_N, where T_N is the tiling factor. Then, we removed the data with the lowest absolute value and retrained the CNN to improve accuracy. We repeated the process until we reached an acceptable pruning factor and inference accuracy.

Because the new post-pruning weight matrix is sparse, we used the index system presented in [44], where each non-zero weight is associated with an index that stores the distance with the next non-zero weight. Using this technique, we store only non-zero parameters in on-chip memory.

In our application, we did not apply pruning to depthwise convolution because their weights only represent 3% of the total number of MobileNet V2 parameters. Moreover, because depthwise convolution extracts features in the CNN, removing weights from

that layer negatively affects accuracy. Specifically, applying a reduction rate of 0.1 in depthwise convolution layers only eliminates 0.19% of the total weights of the network, but top-1 accuracy reduces by 16.67%. Conversely, when we apply the same reduction rate to expansion/projection the convolution layers, we can eliminate 6.25% of the total parameters, and top-1 accuracy is reduced only by 0.84%.

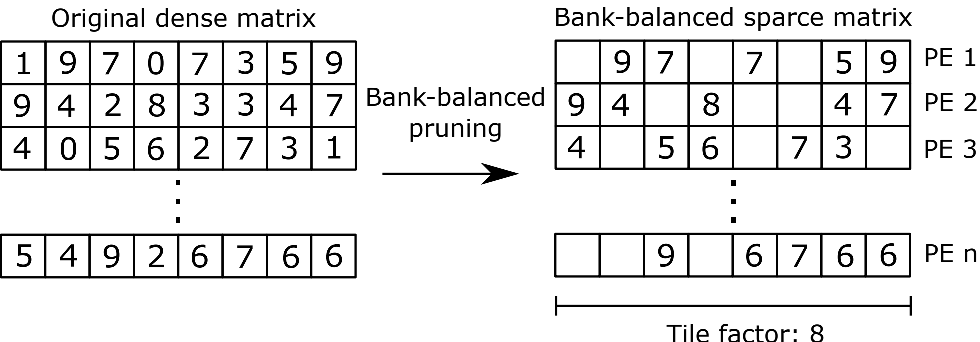

Figure 4. Bank-balanced pruning in MobileNet V2.

3.3.4. Quantization

We used dynamic quantization [43] to reduce the number of bits in the activations and weights. Equation (5) shows the quantization strategy that we applied to MobileNet V2. We divided the floating-point data y_{float} by a scaling factor Δ and rounded the result to obtain the quantized fixed-point value y_{quant}. The scaling factor Δ is computed as the difference between the maximum and minimum floating-point data divided by number of values that can be represented by y_{quant}:

$$y_{quant} = \text{round}(\frac{y_{float}}{\Delta}) \quad \text{with} \quad \Delta = \frac{\max(y_{float}) - \min(y_{float})}{2^{bits}} \tag{5}$$

4. Hardware Architecture

4.1. MobileNet V2 Accelerator Architecture

4.1.1. General Architecture

Figure 5 shows the architecture of our MobileNet V2 accelerator. The heterogeneous architecture is divided into a processing system (PS) and programmable logic (PL). The PS integrates a programmable processor (CPU) that acts as a controller and off-chip memory that stores the activations and weights of the CNN. The PL uses reconfigurable logic to implement the PEs that compute each layer of the network. In our current implementation, the PL has four PEs that operate in parallel. Inference in the architecture operates as follows: (1) the CPU writes the off-chip memory address of weights and control data onto the PL using an AXI Master protocol, (2) the CPU sends feature and residual maps from external memory to each PE as a data stream, using a direct-memory access (DMA) controller, (3) each PE stores the blocks of weights and control data in on-chip buffers, and (4) the PEs compute each layer, reading and writing the input and output activations from/to off-chip memory as a data streams using DMA. The accelerator iterates over the data until it completes the inference.

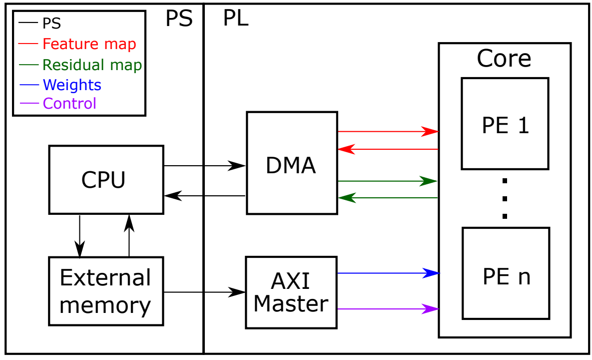

Figure 5. Architecture of the MobileNet V2 accelerator.

4.1.2. Processing Elements

Each PE contains functional units (FUs) that are custom-designed to compute each type of MobileNet V2 layer. A multiplexer selects the correct FU at each point in the computation.

Standard and depthwise FUs: Figure 6 shows the FUs that computes the standard and depthwise convolutions. The input maps are stored in line buffers. The standard and depthwise FUs implement Equation (4) to compute the convolutions. The FUs also use a multiplexer to implement ReLU6 function. The PEs write out each output map as a data stream onto external memory.

(**a**)

(**b**)

Figure 6. (**a**) Standard convolution FU; (**b**) depthwise convolution FU.

Standard and depthwise convolutions use values in a 3×3 neighborhood to compute their results. Because the PEs receive the pixels as a data stream that traverses consecutive rows of the tiling map, the FUs use two line buffers and a 9-register array to compute the 3×3 window of the input map x_{conv}, and nine registers to store the 3×3 window of weights W_{cn}. The accelerator implements each line buffer as a First-In-First-Out (FIFO) queue using on-chip memory. Each line buffer has a size of T_M words to store a row of the tiling map.

The second stage of the standard and depthwise FUs computes the multiplications of convolutions using nine parallel multipliers and a pipelined adder tree. The standard convolution uses an additional stage to add the convolution of the current input channel with the partial sum of the previous input channels to compute the output channel. The architecture stores the partial sum in a buffer of size $T_M \times T_M$.

The final two stages add the batch normalization bias b_{norm} of Equation (4) and compute the activation function. A multiplexer implements the ReLU6 function by saturating the output to zero or six. The PEs send their outputs to external memory as a data stream.

Expansion/projection FU: Figure 7 shows the expansion/projection FU. Because expansion/projection convolutions reuse the input maps and use pruning to compute Equation (4), and because the CPU sends the activations only once, the FU must store the inputs. For this reason, the input stage of the FU uses buffers of size $T_{Ni} \times T_M \times T_M$ and $T_{Ni} \times T_{No}$ to store an input map block x_{conv} and a weight block W_{cn}, respectively.

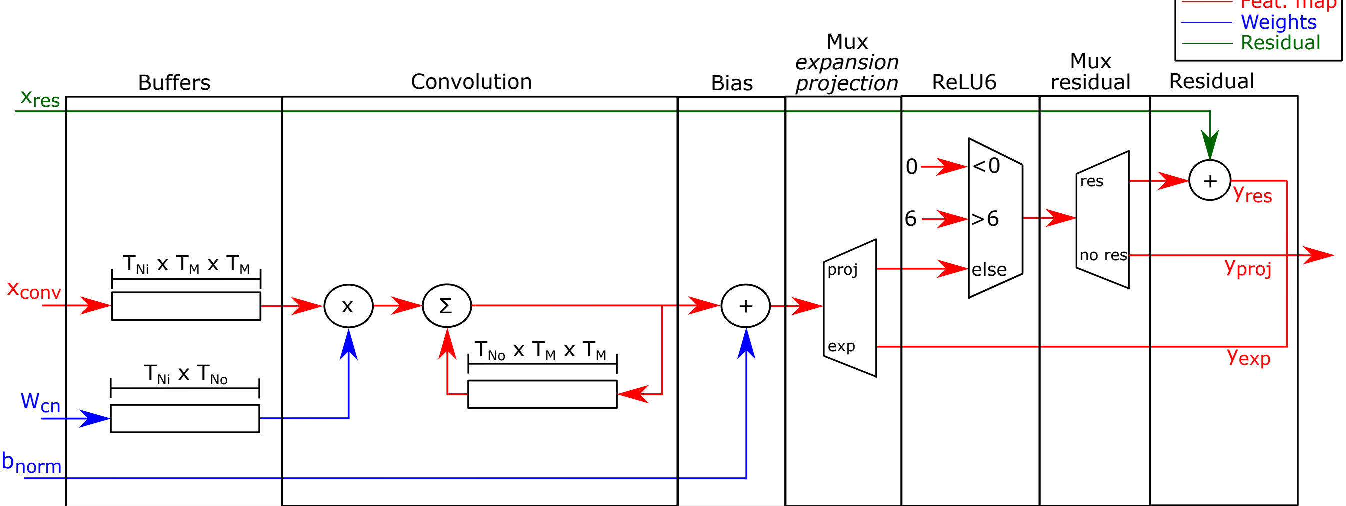

Figure 7. Expansion/projection FU.

The second stage of the FU computes the convolution using the weights W_{cn} and the input maps x_{conv} read from the buffer. A single scalar multiplier compute the 1×1 convolutions used by these layers. Then, the FU adds the convolution of the current input channel to the partial sum of the previous channel to compute the output channel. The accelerator uses a buffer of size $T_{No} \times T_M \times T_M$ to store the partial sum.

The third stage adds the batch normalization bias b_{norm} of Equation (4). If the layer is a projection convolution, the PE computes the ReLU6 activation function and adds the residual map x_{res} if the layer uses it. Finally, the expansion/projection accelerator sends the output map y_{layer} to external memory as a data stream.

Average pooling FU: Figure 8 shows the average pooling FU. The PE receives the input map block as a data stream. Because the CPU sends each input channel consecutively, the accelerator does not need to store the activations in buffers. The PE adds all input pixels of the current channel and stores the intermediate results in a register. Then, the accelerator divides the sum by the number of pixels of the channel to compute the average value. Because the last convolutional map is of size 7×7, the accelerator divides the sum by the

constant value 49. To simplify the design, we used a lookup table (LUT) to implement the division. Finally, the FU streams out the output y_{avg} to external memory.

Fully-connected layer FU: Figure 9 shows the architecture of the fully-connected layer FU. An input stage stores the input array and weights in buffers. The PE computes Equation (2) and streams the output array to the CPU using DMA.

Like the average pooling FU, the fully-connected layer reuses the input array block read from the CPU N_o times to compute Equation (2). Therefore, each PE stores a full tile of the input in a buffer of size T_{Ni}. Similarly, the accelerator stores the weight blocks W_{fc} in a buffer of size $T_{Ni} \times T_{No}$ to apply the pruning indices.

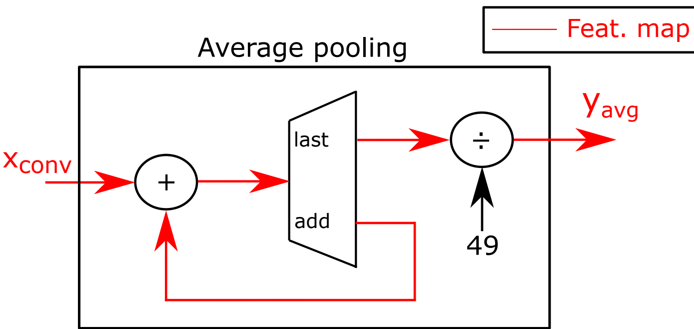

Figure 8. Average pooling FU.

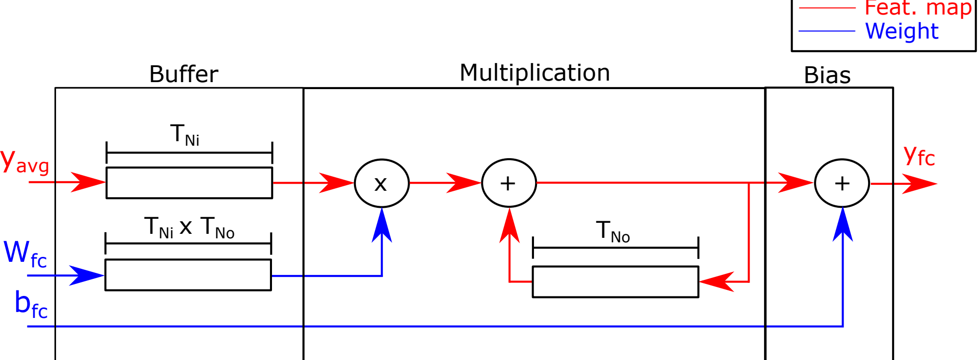

Figure 9. Fully-connected layer FU.

The second stage computes Equation (2) using the input array y_{avg}, the weights W_{fc}, and the bias b_{fc}. The PE multiplies the input y_{avg} by the weight W_{fc} and adds the output to the partial sum of the previous results of the current component of the output array y_{fc}. The architecture uses a buffer of size T_{No} to store the partial results. Finally, when the FU has added the N_o products, it adds the bias b_{fc} and streams the output to the CPU, which writes it out as the inference result.

4.1.3. Parallel Map Processing

Our architecture uses $n = 4$ PEs in parallel to speed up the inference. Figure 10 shows the spatial division of feature maps, where each partition is computed in a separate PE. We partition the maps across the channels to keep the bank-balanced pruning intact. Each PE processes N/n channels, subdivided into blocks of size T_N for each iteration of the layer.

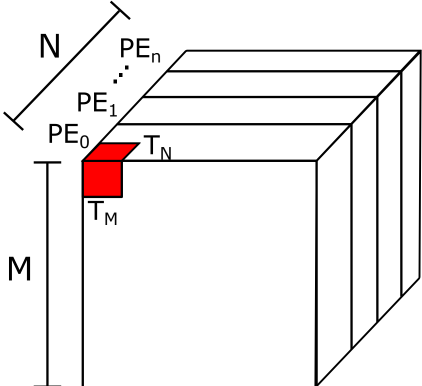

Figure 10. Partition of feature maps for parallel PE processing.

4.2. Design Space Exploration

In this section, we show how we applied loop tiling, pipelining, loop unrolling, and array partitioning to improve the performance of the accelerator.

4.2.1. Loop Tiling Factor

As discussed in Section 3.3.2, we used loop tiling to the depth and size of the data. We considered different loop tiling factors T_N and T_M to reduce inference time. Table 3 shows the inference time for different loop tiling factors applied to activations and weights. The times were measured on the implementation of the accelerator described in Section 5. As the table shows, inference time decreases with the size of the loop tiling factor because the accelerator reuses more data, decreasing access to off-chip memory. However, larger tiling factors require more on-chip memory. We used loop tiling factors of 32 and 28 for the depth and size, respectively, which provide and adequate trade-off between performance and memory usage adequate for limited-resource devices.

Table 3. Inference time with different loop tiling factors.

Loop Tiling Factor		Inference Time (ms)
Activations	Weights	
14	16	252.8
14	32	239.7
14	64	241.6
28	16	230.2
28	32	220.5

4.2.2. Pipelining

The accelerator combines the spatial parallelism of multiple PEs with deep pipelining within the architecture of each PE. A pipelined architecture executes multiple stages of the computation in parallel on different data, using synchronized registers to decouple the stages. Although latency is not decreased and can even increase in a pipelined architecture, throughput and clock rate increase, boosting performance. We use pipelining in the design of every FU in the PEs.

4.2.3. Loop Unrolling Factor

The loop unrolling technique eliminates bubbles caused by control dependencies in the pipeline and exposes additional parallelism in the algorithm. Although the data stream provided by the DMA controller restricts parallelism because the PEs can only access one data element at a time, in the expansion/projection accelerator, we can parallelize its operation when the PE reads the data from the activation buffer. For this reason, we used loop unrolling on the inner loop that traverses the feature map block. Table 4

shows the number of clock cycles needed by the accelerator to execute the inner loop of the expansion/projection convolution, for different loop unrolling factors. As the table shows, processing time decreases with the unrolling factor because it increases the data parallelism available to the PE. We used a loop unrolling factor of 28 in the inner loop of expansion/projection convolution FU. Larger unrolling factors have no effect because performance is limited by the loop tiling factor.

Table 4. Processing time on expansion/projection FU with different loop unrolling factors.

Loop Unrolling Factor	Processing Cycles	
	Min	Max
Without unrolling	53	788
Unrolling ×7	30	114
Unrolling ×14	15	57
Unrolling ×28	8	29

4.2.4. Array Partitioning

HLS allows using an array partitioning pragma to divide an array into blocks and synthesize it in independent on-chip memories. Using this feature, the architecture can access more data at the same time, increasing parallelism. We used a partitioning factor of 28 in the map-size component of the activation buffer of the expansion/projection accelerator to access 28 pixels in parallel, which are then processed by the parallel units created through loop unrolling.

5. Results

5.1. Classification Performance

As discussed in Section 3.3, we used pruning and quantization to obtain a reduced version of MobileNet V2, and retrained the network after applying these techniques using the original parameters as initial values. Table 5 shows the MobileNet V2 hyperparameter values used for retraining and inference. We configured the network for input images of 224×224 pixels with a width multiplier of 1, which preserves the number of channels in the default configuration. The CNN has 21 layers, composed of one standard convolution layer and 18 bottleneck blocks to extract features, plus one average pooling and one fully connected layer to classify the features. For bank-balanced pruning, we used a reduction rate of 0.3 for the expansion/projection convolutions and 0.7 for the fully-connected layer, and retrained the CNN with a learning rate of 0.001, a momentum of 0.9 and 30 epochs. In dynamic quantization, we used 12 and 10 bits for maps and parameters in the convolutional layers, respectively, and 12 and 6 bits for activations and weights in the fully-connected layer. With these modifications, our reduced version of MobileNet V2 achieves top-1 and top-5 accuracy of 65.62% and 87.03% on ImageNet. Compared to the unmodified network, top-1 and top-5 accuracy is reduced by 6.26% and 3.26%, respectively. Table 6 shows examples of inference results in our reduced CNN. Columns 2 and 3 of the table show images that the network correct classifies in the top and within the top 5 probabilities, respectively, while column 4 shows an incorrectly classified image.

Table 5. MobileNet V2 hyperparameters used for retraining and inference (N/A: not applicable).

Hyper Parameter	Retraining	Inference
Batch size	32	1
Input image size	224 × 224	224 × 224
Width multiplier	1	1
Number of layers	21	21
Learning rate	0.001	N/A
Momentum	0.9	N/A
Number of epochs	30	N/A

Table 6. Inference results in our reduced MobileNet V2 network (Images taken from the ImageNet 2012 validation dataset [52]).

Image id	00041633	00031834	00023151
Correct label	mink	walker hound	combination lock
Results	mink: 71.63% weasel: 11.72% polecat: 6.79% hamster: 2.12% wombat: 1.91%	beagle: 28.18% walker hound: 14.47% basset: 6.34% basenji: 5.90% pembroke: 4.81%	syringe: 7.65% reel: 6.17% fountain pen: 6.00% corkscrew: 3.91% wine bottle: 3.34%

5.2. Performance and Resource Utilization

We designed the accelerator architecture using the Xilinx HLS environment. We used Xilinx Vivado HLS 2019.2 to compile the HLS code to a register transfer level (RTL) description, and Xilinx Vitis IDE 2019.2 to manage the CPU and processor-logic communication. We synthesized the design and analyzed its performance using the Xilinx Vivado 2019.2 design suite. We implemented the accelerator on the reconfigurable logic of a Xilinx Zynq UltraScale+ XCZU7EV FPGA, which also has an ARM Cortex-A53 processor. Our architecture uses four parallel PEs implemented on the FPGA, which communicates with the ARM processor using a direct memory access (DMA) block.

Table 7 shows the resource utilization of each PE in our implementation. The table separates the resources used for standard convolutions, depthwise convolutions, expansion/projection convolution, average pooling, and fully-connected FUs, the read-control information section, and the communication protocols of input and output signals on the PE. As the table shows, each PE uses 125 BRAMs and 85 DSP slices. The expansion/projection architecture uses most of the BRAMs because these layers store a full input map block. In addition, standard convolution and fully-connected FUs use BRAMs to store their partial sums. The control information section uses the rest of the BRAMs as a buffer to store the PE control signals. Because each map block is of size 28 × 28, standard and depthwise architectures use LUTs and registers to implement the line buffers. The expansion/projection and fully-connected FUs use 72-bit wide URAM memory blocks to store the prune weights.

Table 8 shows the resource utilization of the complete accelerator on the FPGA, using four PEs in parallel. As the table shows, this implementation of our architecture uses 532 BRAMs, 24 URAMs, and 340 DSP slices. The PEs use the URAM and DSP slices to compute the inference, while the DMA block uses 32 BRAMs as buffers to store the input and output data.

Table 7. Resource utilization of each PE.

Module	Slice LUT	Slice Registers	BRAM	URAM	DSP
Standard FU	2378	1737	6	0	21
Depthwise FU	1904	1747	0	0	18
Expansion/projection FU	3668	1335	112	3	40
Average pooling FU	363	209	0	0	1
Fully-connected FU	775	578	1	3	4
Read control information	3215	2211	6	0	1
Communication protocols	3069	5514	0	0	0
Total	15,372	13,331	125	6	85
Percent	7.36%	3.06%	19.71%	6.25%	4.92%

Table 8. Resource utilization of the accelerator.

Module	Slice LUT	Slice Registers	BRAM	URAM	DSP
DMA	57,072	75,322	32	0	0
PEs	61,161	53,292	500	24	340
Total	118,233	128,614	532	24	340
Percent	51.32%	27.91%	85.26%	25.00%	19.68%

Our implementation runs at a maximum clock frequency of 200 MHz. The critical path that limits the clock frequency runs from the output of a DSP slice that computes the multiplication between the input maps and the weights, to the input of a register that stores the result in the standard convolution FU.

At the maximum clock frequency, Xilinx Vivado estimates the power consumption of the accelerator as 7.35 W, with 0.73 W and 6.62 W of static and dynamic power respectively. The AMR processor consumes 2.79 W (0.12 W of static power and 2.67 W of dynamic power), while the FPGA logic section consumes 4.56 W (0.62 W of static power and 3.94 W of dynamic power). Each PE consumes 0.56 W, while the DMA block and the clock distribution network consume the remaining 2.32 W.

Our implementation processes an image of 224×224 pixels at 4.54 fps (220.5 ms), performing 2.76 GOPS. The processor uses 57.4 ms to manage and send the data to the accelerator, which computes the inference in 163.1 ms. Table 9 shows the execution time of each layer of the MobileNet V2 network. As the table shows, the PS processing time is larger in the first layers of the CNN because the processor must manage and send more blocks to the accelerator when the size of the feature maps is larger than 28. The accelerator processing time increases with the size (in the first layers) or depth (in the last layers) of the activation because the number of blocks that each PE processes per iteration increases.

Table 9. Execution time of each layer of MobileNet V2 on the accelerator.

Input ($MxMxN_i$)	Output ($MxMxN_o$)	Layer	Processor [ms]	Accelerator [ms]	Total [ms]
224 × 224 × 3	112 × 112 × 32	Standard conv	1.5	18.1	19.6
112 × 112 × 32	112 × 112 × 16	Bottleneck	5.7	7.2	12.9
112 × 112 × 16	56 × 56 × 24	Bottleneck	29.0	29.9	58.9
56 × 56 × 24	28 × 28 × 32	Bottleneck	9.2	12.7	21.9
28 × 28 × 32	14 × 14 × 64	Bottleneck	4.0	12.9	16.9
14 × 14 × 64	14 × 14 × 96	Bottleneck	3.4	18.7	22.1
14 × 14 × 96	7 × 7 × 160	Bottleneck	3.3	30.1	33.4
7 × 7 × 160	7 × 7 × 320	Bottleneck	0.7	14.6	15.3
7 × 7 × 320	7 × 7 × 1280	E × pansion conv	0.2	12.9	13.1
7 × 7 × 1280	1 × 1 × 1280	Avg pooling	0.2	0.3	0.5
1 × 1 × 1280	1 × 1 × 1000	Fc	0.2	5.7	5.9
	MobileNet V2		57.4	163.1	220.5

5.3. Scalability

When we synthesize the architecture onto a larger device, we can increase the number of PEs and the amount of activation and weight data that can be stored on chip. This allows the architecture to scale its performance at the cost of increased resource utilization. Table 10 shows resource utilization, inference time, and maximum fps achieved by multiple versions of the accelerator configured with 4, 8, and 12 PEs. We obtained these results by synthesizing the accelerator onto a XCZU19EG FPGA, which features with 2.26 times more LUTs and registers, 3.15 times more BRAMs, 1.33 times more URAMs and 1.14 times more DSP than our XCZU7EV FPGA. As the table shows, LUTs and BRAMs limit the number of PEs that can be implemented on the chip because the architecture requires more finite-state machines to control the inference, and stores more data in on-chip memories, respectively. With 12 PEs, the accelerator computes inference on an image in 40.65 ms, reaching a throughput of 24.6 fps.

Table 10. Resource utilization and inference time on an XCZU19EG FPGA for different numbers of PEs.

Number of PEs	Slice LUT	Slice Registers	BRAM	URAM	DSP	Time [ms]	fps
4	121,689	127,020	524	24	340	126.91	7.87
8	240,660	250,278	1047	48	680	63.45	15.76
12	368,936	391,517	1572	72	1020	40.65	24.60

5.4. Discussion

Table 11 summarizes the key parameters of the FPGA accelerators of Table 1 and compares them to the accelerator presented in this paper. As the table shows, our design is closest to [26,27], which implement versions 1 and 2 of MobileNet, respectively. Compared to our implementation, these designs achieve significantly higher throughput, mainly because they use large FPGA devices that allow them to store all the map data in on-chip memory. This has several benefits: it allows them to increase parallelism through more PEs, it eliminates the need to access comparatively slow off-chip memory, and it eliminates the computation and communication latency introduced by the processor. The cost is increased resource utilization: indeed, our design uses 23–30% of the arithmetic and memory resources used by these solutions, allowing us to map the accelerator onto a wider range of devices. Even when using 12 PEs in parallel, our accelerator exhibits lower resource usage because we process only one channel at a time in each PE to save hardware resources and power, while the other solutions process 32 simultaneous channels per PE.

Compared to the VGG [32,39] accelerators, our implementation achieves similar inference time and top-5 accuracy, but lower resource utilization. The accelerator in [40] uses the CNN DiracDeltaNet architecture, which has been custom-designed for FPGAs, replacing multiplications with shift operations and using aggressive quantization. They report better resource utilization than our implementation, with slightly better accuracy. The AlexNet accelerator [50] computes inference 2.14 times faster than our accelerator but uses a clock frequency 1.5 times faster. In comparison, our accelerator achieves better accuracy, uses less resources, and consumes 2.4 times less power. The SqueezeNet [51] CNN has fewer parameters and layers than MobileNet V2, and their accelerator is faster than our design, but with lower top-1 and top-5 accuracy.

Table 11. Comparison to other CNN accelerators (N/A: Not available).

Implementation	CNN	Top-1 (%)	Top-5 (%)	Power (W)	fps	LUT	BRAM	FF	DSP
Qiu et al. [32]	VGG-16	N/A	86.7	9.63	4.45	182,616	486	127,653	780
Guo et al. [39]	VGG-16	67.7	88.1	3.5	2.75	29,867	85.5	35,489	190
Su et al. [26]	MobileNet	64.6	84.5	N/A	127	139,000	1729	550,00	1452
Bai et al. [27]	MobileNet V2	71.8	91.0	N/A	266	163,506	1844	N/A	1278
Yang et al. [40]	DiracDeltaNet	70.1	88.2	5.5	58.7	24,130	170	29,867	37
Zhang et al. [50]	AlexNet	56.0	N/A	17.67	9.73	101,953	198.5	127,577	696
Panagiotis et al. [51]	SqueezeNet	56.9	79.9	N/A	14.2	34,489	97.5	25,036	172
This work	MobileNet V2	65.6	87.0	7.35	4.54	118,233	532	128,614	340

We also compared the performance of our accelerator to the Nvidia Jetson AGX Xavier benchmarks [57] and the Google Coral Dev Board [25], which are commercially available platforms. In the case of the Nvidia Jetson platform, the processor and GPU cores on this device run at a clock frequency 2.27 GHz and 1.38 GHz, respectively. Using the ResNet-50 [15] and VGG19 CNN architectures, the GPU performs an inference between 6.8–11.3 times faster than our accelerator. However, its power consumption is 4.2–5.2 times higher, limiting its use in power-constrained edge devices. The Coral Dev Board uses a Tensor Processing Unit (TPU), a custom-built accelerator for the TensorFlow framework. Implementing MobileNet V2, the Coral hardware performs inference at 385 fps and consumes 5.4 W. Indeed, Coral achieves better performance and power than our accelerator, but its special-purpose architecture limits its application to neural-network computation. In comparison, our accelerator was designed for programmable hardware platforms, which can easily assign their logic resources to other tasks, such as image processing and machine vision algorithms, through dynamic reconfiguration.

We also compared the performance of the accelerator to desktop-class hardware. We computed MobileNet V2 inference in software using the PyTorch framework with CUDA 10.2 and tested the software on an Nvidia RTX 2080 GPU and an Intel i9-9900K CPU. The desktop GPU is one order of magnitude faster than our accelerator (21.7 ms inference), but with 30 times the power consumption (215 W). The CPU achieves approximately the same inference time (222 ms) with more than one order of magnitude higher power (95 W).

Although other accelerator architectures achieve faster inference than our design, our accelerator achieves a good balance between accuracy, performance, and power consumption. This makes our architecture attractive for embedded and portable devices that process video with limited hardware resources and power budget. For example, our accelerator could be embedded on a smart camera architecture, allowing it to classify objects at a frame rate of 4 to 25 fps, depending on the device used in the implementation. For applications such as face recognition in video analytics, this performance can be sufficient in most scenarios [29].

6. Conclusions

In this paper, we proposed the architecture of a hardware accelerator for MobileNet V2 inference. Our architecture was designed for reconfigurable logic devices, and features

good scalability, as well as lower resource utilization and power consumption compared to other published and commercial accelerators that run on reconfigurable or programmable hardware. This allows our architecture to be implemented on edge devices for real-time image classification that are resource- and power-constrained. The accelerator uses loop tiling, bank-balanced pruning, dynamic quantization, and off-chip storage to increase performance and reduce hardware resource utilization and power consumption. Our implementation uses an embedded processor and external memory to control the execution of the algorithm and to store parameters, respectively. The accelerator exploits the parallelism available in the various MobileNet V2 layers using pipelining and multiple processing elements implemented on reconfigurable hardware. By configuring the number of processing elements, our architecture can trade off inference speed for power and resource utilization. This allows us to target the implementation to a wide range of devices, or to share hardware resources with other algorithms on the same device.

We implemented a prototype of the architecture on a Xilinx Zynq Ultrascale+ XCZU7EV FPGA with 2 GB of DDR4 external memory. Our accelerator performs inference on 224×224-pixel images at 4.53 fps, consumes 7.35 W, and achieves a top-1 and top-5 accuracy of 65.62% and 87.03%, respectively. Our implementation uses 20% and 85% of the DSP slices and on-chip memory blocks available on the FPGA. Implemented on a larger XCZU19EG FPGA, our design reaches 24.6 fps with 12 PEs.

We are currently working on improving the communication between the processor and the accelerator using a faster DMA-SG interface, which stores transfer instructions in on-chip memory instead of the CPU cache. We are also working on quantizing MobileNet V2 with PACT functions to further reduce the number of bits used to represent weights and activations, allowing us to store more data in the same amount of on-chip memory.

Author Contributions: Conceptualization, I.P. and M.F.; Methodology, I.P. and M.F.; Software, I.P.; Supervision, M.F.; Writing—original draft, I.P. and M.F. Both authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Agency for Research and Development (ANID) through graduate scholarship folio 22180733 and FONDECYT Regular Grant No. 1180995.

Data Availability Statement: The experiments presented in this paper were performed using images from the ImageNet database available at http://www.image-net.org/index accessed on 28 December 2020.

Acknowledgments: The authors would like to thank Gonzalo Carvajal for his help with providing information on the performance of dedicated hardware for CNN inference.

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

Abbreviations

The following abbreviations are used in this manuscript:

BRAM	Block Random Access Memory
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
fps	frames per second
FU	Functional Unit
GOPS	Giga Operations per Second
GPU	Graphics Processing Unit
HLS	High-Level Synthesis
KNN	K-Nearest Neighbor

LBP	Local Binary Patterns
LUT	Lookup Table
PE	Processing Element
PL	Programmable Logic
PS	Processing System
SG	Scatter Gather
SIFT	Scale Invariant Feature Transform
SVD	Singular Value Decomposition
SVM	Support Vector Machines
RTL	Register Transfer Level
TPU	Tensor Processing Unit
URAM	Ultra Random Access Memory

References

1. LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324. [CrossRef]
2. Shaha, M.; Pawar, M. Transfer Learning for Image Classification. In Proceedings of the 2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 29–31 March 2018; pp. 656–660.
3. Makkar, T.; Kumar, Y.; Dubey, A.K.; Rocha, A.; Goyal, A. Analogizing time complexity of KNN and CNN in recognizing handwritten digits. In Proceedings of the 2017 Fourth International Conference on Image Information Processing (ICIIP), Shimla, India, 21–23 December 2017; pp. 1–6.
4. Chaganti, S.Y.; Nanda, I.; Pandi, K.R.; Prudhvith, T.G.N.R.S.N.; Kumar, N. Image Classification using SVM and CNN. In Proceedings of the 2020 International Conference on Computer Science, Engineering and Applications (ICCSEA), Sydney, Australia, 19–20 December 2020; pp. 1–5.
5. Pérez-Hernández, F.; Tabik, S.; Lamas, A.; Olmos, R.; Fujita, H.; Herrera, F. Object Detection Binary Classifiers methodology based on deep learning to identify small objects handled similarly: Application in video surveillance. *Knowl.-Based Syst.* **2020**, *194*, 105590. [CrossRef]
6. Feng, D.; Haase-Schütz, C.; Rosenbaum, L.; Hertlein, H.; Gläser, C.; Timm, F.; Wiesbeck, W.; Dietmayer, K. Deep Multi-Modal Object Detection and Semantic Segmentation for Autonomous Driving: Datasets, Methods, and Challenges. *IEEE Trans. Intell. Transp. Syst.* **2021**, *22*, 1341–1360. [CrossRef]
7. Afif, M.; Ayachi, R.; Said, Y.; Pissaloux, E.; Atri, M. An evaluation of retinanet on indoor object detection for blind and visually impaired persons assistance navigation. *Neural Process. Lett.* **2020**, *51*, 2265–2279. [CrossRef]
8. Jiang, Q.; Tan, D.; Li, Y.; Ji, S.; Cai, C.; Zheng, Q. Object detection and classification of metal polishing shaft surface defects based on convolutional neural network deep learning. *Appl. Sci.* **2020**, *10*, 87. [CrossRef]
9. Lyra, S.; Mayer, L.; Ou, L.; Chen, D.; Timms, P.; Tay, A.; Chan, P.Y.; Ganse, B.; Leonhardt, S.; Hoog Antink, C. A Deep Learning-Based Camera Approach for Vital Sign Monitoring Using Thermography Images for ICU Patients. *Sensors* **2021**, *21*, 1495. [CrossRef] [PubMed]
10. Shibata, T.; Teramoto, A.; Yamada, H.; Ohmiya, N.; Saito, K.; Fujita, H. Automated Detection and Segmentation of Early Gastric Cancer from Endoscopic Images Using Mask R-CNN. *Appl. Sci.* **2020**, *10*, 3842. [CrossRef]
11. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv* **2017**, arXiv:1704.04861.
12. Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L.C. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–23 June 2018.
13. Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In Proceedings of the 3rd International Conference on Learning Representations, San Diego, CA, USA, 7–9 May 2015.
14. Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; Rabinovich, A. Going Deeper With Convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA, 7–12 June 2015.
15. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016.
16. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*; Curran Associates, Inc.: Red Hook, NY, USA 2012; pp. 1097–1105.
17. Iandola, F.N.; Moskewicz, M.W.; Ashraf, K.; Han, S.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1 MB model size. *arXiv* **2016**, arXiv:1602.07360.
18. Teichmann, M.; Weber, M.; Zöllner, M.; Cipolla, R.; Urtasun, R. MultiNet: Real-time Joint Semantic Reasoning for Autonomous Driving. In Proceedings of the 2018 IEEE Intelligent Vehicles Symposium (IV), Suzhou, China, 26 June–1 July 2018; pp. 1013–1020.

19. Strigl, D.; Kofler, K.; Podlipnig, S. Performance and Scalability of GPU-Based Convolutional Neural Networks. In Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, Pisa, Italy, 17–19 February 2010; pp. 317–324.
20. Kim, H.; Nam, H.; Jung, W.; Lee, J. Performance analysis of CNN frameworks for GPUs. In Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Santa Rosa, CA , USA, 24–25 April 2017; pp. 55–64.
21. Li, D.; Chen, X.; Becchi, M.; Zong, Z. Evaluating the Energy Efficiency of Deep Convolutional Neural Networks on CPUs and GPUs. In Proceedings of the 2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom), Atlanta, GA, USA, 8–10 October 2016; pp. 477–484.
22. Zhu, Y.; Samajdar, A.; Mattina, M.; Whatmough, P.N. Euphrates: Algorithm-SoC Co-Design for Low-Power Mobile Continuous Vision. *arXiv* **2018**, arXiv:1803.11232.
23. Haut, J.M.; Bernabé, S.; Paoletti, M.E.; Fernandez-Beltran, R.; Plaza, A.; Plaza, J. Low–High-Power Consumption Architectures for Deep-Learning Models Applied to Hyperspectral Image Classification. *IEEE Geosci. Remote Sens. Lett.* **2019**, *16*, 776–780. [CrossRef]
24. Caba, J.; Díaz, M.; Barba, J.; Guerra, R.; López, J.A. Fpga-based on-board hyperspectral imaging compression: Benchmarking performance and energy efficiency against gpu implementations. *Remote Sens.* **2020**, *12*, 3741. [CrossRef]
25. Kang, P.; Jo, J. Benchmarking Modern Edge Devices for AI Applications. *IEICE Trans. Inf. Syst.* **2021**, *104*, 394–403. [CrossRef]
26. Su, J.; Faraone, J.; Liu, J.; Zhao, Y.; Thomas, D.B.; Leong, P.H.; Cheung, P.Y. Redundancy-reduced mobilenet acceleration on reconfigurable logic for ImageNet classification. In Proceedings of the Applied Reconfigurable Computing. Architectures, Tools, and Applications: 14th International Symposium, ARC 2018, Santorini, Greece, 2–4 May 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 16–28.
27. Bai, L.; Zhao, Y.; Huang, X. A CNN Accelerator on FPGA Using Depthwise Separable Convolution. *IEEE Trans. Circuits Syst. II Express Briefs* **2018**, *65*, 1415–1419. [CrossRef]
28. Hareth, S.; Mostafa, H.; Shehata, K.A. Low power CNN hardware FPGA implementation. In Proceedings of the 2019 31st International Conference on Microelectronics (ICM), Cairo, Egypt, 15–18 December 2019; pp. 162–165.
29. Kim, S.; Lee, J.; Kang, S.; Lee, J.; Yoo, H. A Power-Efficient CNN Accelerator With Similar Feature Skipping for Face Recognition in Mobile Devices. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2020**, *67*, 1181–1193. [CrossRef]
30. Bahl, G.; Daniel, L.; Moretti, M.; Lafarge, F. Low-Power Neural Networks for Semantic Segmentation of Satellite Images. In Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops, Seoul, Korea, 27 October–3 November 2019.
31. Yih, M.; Ota, J.M.; Owens, J.D.; Muyan-Özçelik, P. FPGA versus GPU for Speed-Limit-Sign Recognition. In Proceedings of the 2018 21st International Conference on Intelligent Transportation Systems (ITSC), Maui, HI, USA, 4–7 November 2018; pp. 843–850.
32. Qiu, J.; Wang, J.; Yao, S.; Guo, K.; Li, B.; Zhou, E.; Yu, J.; Tang, T.; Xu, N.; Song, S.; et al. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 21–23 February 2016; pp. 26–35.
33. Fowers, J.; Ovtcharov, K.; Strauss, K.; Chung, E.S.; Stitt, G. A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication. In Proceedings of the 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines, Boston, MA, USA, 11–13 May 2014; pp. 36–43.
34. Colleman, S.; Verhelst, M. High-Utilization, High-Flexibility Depth-First CNN Coprocessor for Image Pixel Processing on FPGA. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* **2021**, *29*, 461–471. [CrossRef]
35. Jin, Z.; Finkel, H. Population Count on Intel® CPU, GPU and FPGA. In Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), New Orleans, LA, USA, 18–22 May 2020; pp. 432–439.
36. Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2015; pp. 161–170.
37. Ni, Y.; Chen, W.; Cui, W.; Zhou, Y.; Qiu, K. Power optimization through peripheral circuit reusing integrated with loop tiling for RRAM crossbar-based CNN. In Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 19–23 March 2018; pp. 1183–1186.
38. Abdelouahab, K.; Pelcat, M.; Sérot, J.; Berry, F. Accelerating CNN inference on FPGAs: A Survey. *arXiv* **2018**, arXiv:1806.01683.
39. 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.* **2018**, *37*, 35–47. [CrossRef]
40. Yang, Y.; Huang, Q.; Wu, B.; Zhang, T.; Ma, L.; Gambardella, G.; Blott, M.; Lavagno, L.; Vissers, K.; Wawrzynek, J.; et al. Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, 24–26 February 2019; ACM: New York, NY, USA, 2019; pp. 23–32.
41. Zhou, A.; Yao, A.; Guo, Y.; Xu, L.; Chen, Y. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. *arXiv* **2017**, arXiv:1702.03044.

42. Banner, R.; Nahshan, Y.; Soudry, D. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Advances in Neural Information Processing Systems 32*; Curran Associates, Inc.: Red Hook, NY, USA 2019; pp. 7950–7958.
43. Mathew, M.; Desappan, K.; Kumar Swami, P.; Nagori, S. Sparse, Quantized, Full Frame CNN for Low Power Embedded Devices. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops, Honolulu, HI, USA, 22–25 July 2017.
44. Han, S.; Mao, H.; Dally, W.J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. Presented at the 4th International Conference on Learning Representations (ICLR), San Juan, Puerto Rico, 2–4 May 2016.
45. Narang, S.; Undersander, E.; Diamos, G. Block-sparse recurrent neural networks. *arXiv* **2017**, arXiv:1711.02782.
46. Cao, S.; Zhang, C.; Yao, Z.; Xiao, W.; Nie, L.; Zhan, D.; Liu, Y.; Wu, M.; Zhang, L. Efficient and effective sparse LSTM on FPGA with Bank-Balanced Sparsity. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, 24–26 February 2019; ACM: New York, NY, USA, 2019; pp. 63–72.
47. Lin, S.; Ji, R.; Yan, C.; Zhang, B.; Cao, L.; Ye, Q.; Huang, F.; Doermann, D. Towards Optimal Structured CNN Pruning via Generative Adversarial Learning. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, CA, USA, 15–21 June 2019.
48. Luo, J.; Wu, J. An Entropy-based Pruning Method for CNN Compression. *arXiv* **2017**, arXiv:1706.05791.
49. Zhang, X.; Zhou, X.; Lin, M.; Sun, J. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–23 June 2018.
50. Zhang, M.; Li, L.; Wang, H.; Liu, Y.; Qin, H.; Zhao, W. Optimized Compression for Implementing Convolutional Neural Networks on FPGA. *Electronics* **2019**, *8*, 295. [CrossRef]
51. Mousouliotis, P.G.; Petrou, L.P. CNN-Grinder: From Algorithmic to High-Level Synthesis descriptions of CNNs for Low-end-low-cost FPGA SoCs. *Microprocess. Microsyst.* **2020**, *73*, 102990. [CrossRef]
52. Deng, J.; Dong, W.; Socher, R.; Li, L.; Li, K.; Li, F.-F. ImageNet: A large-scale hierarchical image database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009; pp. 248–255.
53. Qasaimeh, M.; Sagahyroon, A.; Shanableh, T. FPGA-Based Parallel Hardware Architecture for Real-Time Image Classification. *IEEE Trans. Comput. Imaging* **2015**, *1*, 56–70. [CrossRef]
54. Afifi, S.; GholamHosseini, H.; Sinha, R. SVM classifier on chip for melanoma detection. In Proceedings of the 2017 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), Jeju Island, Korea, 11–15 July 2017; pp. 270–274.
55. Ioffe, S.; Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 6–11 July 2015; Volume 37, pp. 448–456.
56. Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv* **2018**, arXiv:1806.08342.
57. Nvidia Corporation. Jetson AGX Xavier: Deep Learning Inference Benchmarks. Available online: https://developer.nvidia.com/embedded/jetson-agx-xavier-dl-inference-benchmarks (accessed on 28 December 2020).