An Efficient FPGA-based Depthwise Separable Convolutional Neural Network Accelerator with Hardware Pruning

ZHENGYAN LIU, QIANG LIU, and SHUN YAN, School of Microelectronics, Tianjin University, China
RAY C. C. CHEUNG, Department of Electrical Engineering, City University of Hong Kong

Convolutional neural networks (CNNs) have been widely deployed in computer vision tasks. However, the computation and resource intensive characteristics of CNN bring obstacles to its application on embedded systems. This article proposes an efficient inference accelerator on Field Programmable Gate Array (FPGA) for CNNs with depthwise separable convolutions. To improve the accelerator efficiency, we make four contributions: (1) an efficient convolution engine with multiple strategies for exploiting parallelism and a configurable adder tree are designed to support three types of convolution operations; (2) a dedicated architecture combined with input buffers is designed for the bottleneck network structure to reduce data transmission time; (3) a hardware padding scheme to eliminate invalid padding operations is proposed; and (4) a hardware-assisted pruning method is developed to support online tradeoff between model accuracy and power consumption. Experimental results show that for MobileNetV2 the accelerator achieves 10× and 6× energy efficiency improvement over the CPU and GPU implementation, and 302.3 frames per second and 181.8 GOPS performance that is the best among several existing single-engine accelerators on FPGAs. The proposed hardware-assisted pruning method can effectively reduce 59.7% power consumption at the accuracy loss within 5%.

CCS Concepts: • **Hardware → Hardware accelerators;**

Additional Key Words and Phrases: CNN accelerator, depthwise-seperable convolution, bottleneck, model compression

ACM Reference format:
Zhengyan Liu, Qiang Liu, Shun Yan, and Ray C. C. Cheung. 2024. An Efficient FPGA-based Depthwise Separable Convolutional Neural Network Accelerator with Hardware Pruning. *ACM Trans. Reconfig. Technol. Syst.* 17, 1, Article 15 (February 2024), 20 pages.
https://doi.org/10.1145/3615661

1 INTRODUCTION

In recent years, **convolutional neural networks (CNNs)** have been widely used in computer vision tasks such as image classification [1] and target detection [2, 3]. However, CNN is a

This research was supported in part by the National Natural Science Foundation of China under Grant U21B2031.
Authors' addresses: Z. Liu, Q. Liu (Corresponding author), and S. Yan, School of Microelectronics, Tianjin University, 92nd Rd, Weijin, Tianjin, Nankai, China, 300072; e-mails: liuzhengyan@tju.edu.cn, qiangliu@tju.edu.cn, yanshun@tju.edu.cn; R. C. C. Cheung, Department of Electrical Engineering, City University of Hong Kong; e-mail: eungz@cityu.edu.cn.

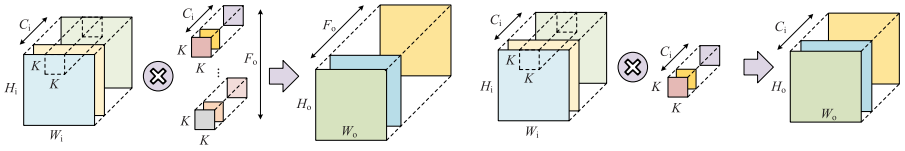
1936-7406/2024/02-ART15 $15.00
https://doi.org/10.1145/3615661

computational-intensive model because of its complex structures and complicated operations. To meet the requirements in real-time processing and memory resources in embedded systems, there are growing interests in neural network compression and dedicated hardware accelerators [4]. Neural network compression approaches include data quantization [5–7], network sparsification [8–10], compact models [11–13], and so on. Compact models aim at simplifying model architectures with acceptable accuracy loss. The existing typical compact models adopt **depthwise separable convolutions (DSCs)** instead of standard convolutions to achieve model simplification, such as MobileNets [20, 21].

Field Programmable Gate Array (FPGA) has become the ideal platform for CNN acceleration recently [22]. FPGA can effectively shorten the development time and facilitate future network optimization due to their reconfigurable substrate. Recent **DSC neural network (DSCNN)** accelerators based on FPGA can be classified into two categories: (a) dedicated computation engines for different types of convolution [23] and (b) a general convolution engine [24–28]. In Reference [23] a dedicated DSC engine is designed and operates in a pipeline with the standard convolution engine. Several implementations use a single convolution engine compatible with the standard, depthwise, and pointwise convolutions to obtain versatility and flexibility. MobileNet is compressed into **Redundancy-Reduced MobileNet (RR-MobileNet)** at model level and data level in a single convolution engine design [24]. A real-time object detection accelerator is implemented for **Single Shot MultiBox Detector Lite-MobileNetV2 (SSDLiteM2)**, which repeatedly runs the single computational engine to process the network [25]. A scalable DSC accelerator is proposed in Reference [26], where a matrix multiplication engine with the configurable memory system is designed to deal with the different convolutions. Furthermore, the CNN accelerator [27] can accelerate both the standard and depthwise separable convolutions with high hardware resource efficiency based on a roofline model. A compiler is used to optimize the FPGA implementation of MobileNet [28], where the standard convolution layers are selectively replaced with the functionally identical depthwise separable convolution layers.

However, after investigation we find that there are still rooms in optimizing DSCNN accelerator design. On the one hand, there is space for hardware design improvement with respect to the structure of DSCNNs, such as the shortcut connection and spindle-shaped inverted residual block, which is called as bottleneck block. On the other hand, there is also design improvement opportunities with respect to the general CNN operations such as padding. Our previous work [14] proposes a high-performance inference accelerator for MobileNet based on FPGA. This work extends the accelerator design in the following ways: (a) the accelerator can support various CNNs such MobileNets and ResNets with a multi-network mapping tool and (b) the accelerator can support hardware-assisted pruning, allowing tradeoff between model accuracy and power consumption online. The contributions of this article are as follows:

(1) A single-engine accelerator compatible with the standard convolution and the depthwise separable convolution is proposed. Five parallelization strategies are exploited for the different convolutions to increase the performance and efficiency of the engine. In addition, a configurable adder tree is proposed to support different ways of accumulations involved in the different convolutions to save computing resources.
(2) A dedicated hardware architecture for the bottleneck structure is proposed. The architecture can make full use of on-chip storage resources and reduce the time consumption of data transmission.
(3) An operation-separate padding scheme is proposed in which zero-padding operations are fused into different operations of the reshape unit. The scheme eliminates invalid padding operations and improves the efficiency of the accelerator.



(a) Standard convolution.

(b) Depthwise convolution.

(c) Pointwise convolution.

Fig. 1. Three types of convolution.

(4) An hardware-assisted data pruning method is proposed that can tune the sparsity modes of the accelerator to meet the accuracy and power consumption requirements of various application scenarios.

2 BACKGROUND AND PROBLEM STATEMENT

This section introduces the elementary knowledge of DSC, bottleneck structure, and model pruning. At the same time, the problems addressed by this work are discussed.

2.1 Depthwise Separable Convolutions

The DSC factorizes a standard convolution into a depthwise convolution and a pointwise convolution, which reduces the amount of arithmetic operations and weights effectively [20]. Figure 1 demonstrates the principles of standard, depthwise, and pointwise convolutions. The input feature map contains $H_\mathrm{i} \times W_\mathrm{i} \times C_\mathrm{i}$ pixels, where H_i, W_i, and C_i are the height, width, and the number of channels of the input feature map, respectively. The size of each kernel is $K \times K$. The standard convolution applies F_o sets of kernels to the input feature map to produce an output feature map with $H_\mathrm{o} \times W_\mathrm{o} \times F_\mathrm{o}$ pixels (Figure 1(a)), where H_o, W_o, and F_o are the height, width, and the number of channels of the output feature map. Different from the standard convolution, the depthwise convolution only applies one set of kernels to each channel of the input feature map and produces an output feature map with the size of $H_\mathrm{o} \times W_\mathrm{o} \times C_\mathrm{i}$ (Figure 1(b)). The pointwise convolution can be regarded as a standard convolution with $K = 1$ (Figure 1(c)). The classical DSCNN models such as MobileNets are built on the three types of convolutions.

Problem 1. *As shown in Figure 1, the thee types of convolutions show different parallelism inherent in the operations. A convolution engine with appropriate parallelization strategies is needed to maximize performance and make full use of the engine.*

2.2 Bottleneck Structures

To solve the degradation problem in deep networks, some layers of a neural network can artificially skip the connection of neurons in the next layer and weaken the strong connection between each layer. Such a structure is called a bottleneck structure with residuals. The bottleneck structures are widely seen in CNNs such as MobileNets and ResNets.

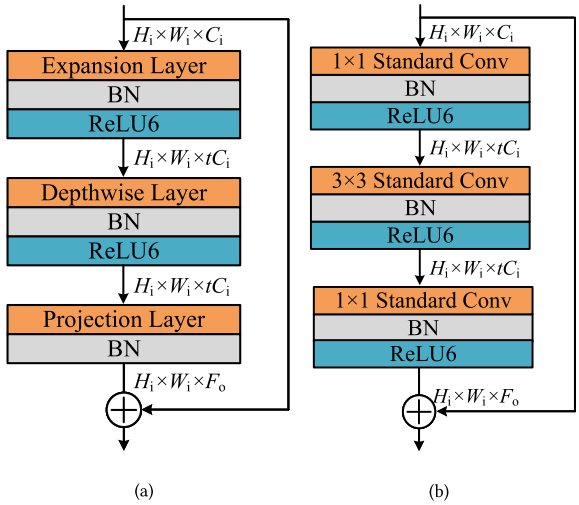


Fig. 2. Bottleneck blocks in (a) MobileNets and (b) ResNets.

As shown in Figure 2(a), the bottleneck block of MobileNetV2 consists of three layers, namely the expansion layer, depthwise layer, and projection layer. Among them, the expansion layer and the projection layer both perform pointwise convolutions and use expansion factor t to control the dimensions of the feature maps. The block applies batch normalization after every layer and uses ReLU6 activation function in the expansion layer and depthwise layer to add non-linearity. A shortcut connection is between the expansion layer and the projection layer. When stride is 1, the input and output of the bottleneck block are added to form a residual structure; otherwise, no shortcut connection is added.

Figure 2(b) shows the bottleneck structure of ResNet networks. This bottleneck structure is used for ResNet-50/101/152. The middle 3×3 convolution layer in the structure first reduces the computation under a dimension-reduced 1×1 convolution layer and then restores under another 1×1 convolution layer, both maintaining network accuracy and reducing the amount of computations.

Problem 2. *One of the issues coming from the shortcut connection is that the input feature map of the expansion layer needs to be transmitted to the on-chip memory in the projection layer, resulting in extra data transmission time.*

2.3 Model Compression

To overcome the problem of large amount of computation and storage caused by the numerous parameters of CNNs, in addition to compact model designs, model compression methods are proposed, including data quantization and model pruning.

Quantization attempts to reduce the bit-width of data representation in neural network models [19], which saves memory resources and simplifies computational operations. The core challenge of quantization is how to reduce the bit-width without significantly reducing representation accuracy, which is a tradeoff between compression rate and accuracy.

Model pruning is a method removing unimportant weights, whose value is less than a threshold, during training to reduce the size of models [17]. The number of the pruned weights over the total number of weights indicates the weight sparsity. The central problem of model pruning is how the weights are effectively selected and removed as much as possible while the loss of model accuracy is minimized [18].

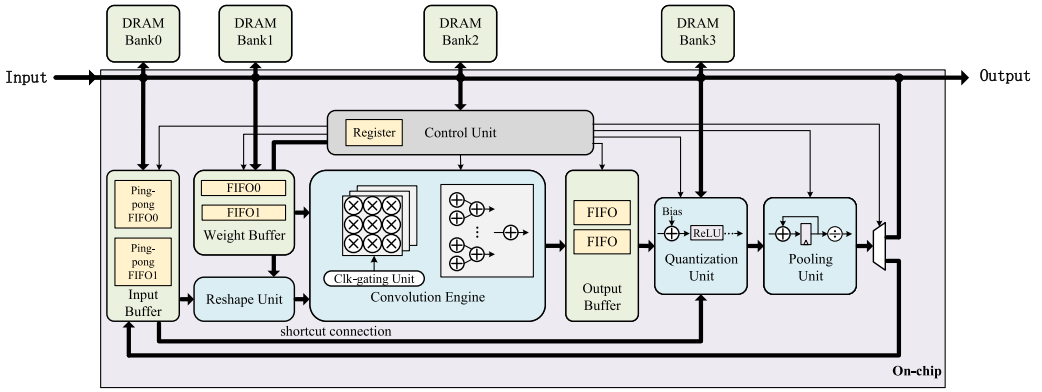


Fig. 3. Design overview of the accelerator.

Problem 3. *After quantization, the accuracy of current accelerators for MobileNet drops 4% compared with the single-precision floating-point implementation [23]. In addition, it is not always convenient to retrain a model with different weight sparsity requirements for deploying the model in various application scenarios.*

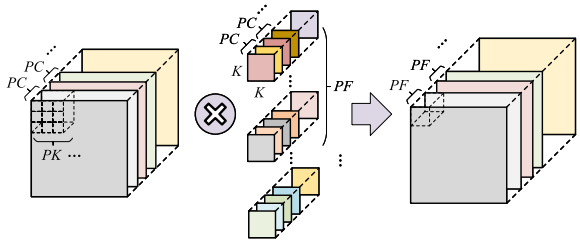
3 ACCELERATOR DESIGN

3.1 Design Overview

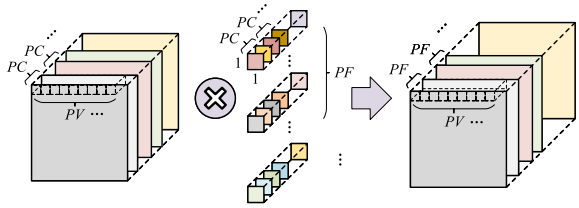
The proposed accelerator adopts a single convolution engine architecture, as shown in Figure 3. The single-engine architecture is versatile and easy to migrate to other models compared with the separate dedicated convolution engines. The images, weights, controlling instructions, and quantization parameters are loaded from the input port to the DRAM Bank0 Bank3, respectively. The processing results of the accelerator are outputted from the output port. The data in each bank of the DRAM are transferred to the on-chip at a rate of 9.2 GB/s. The bandwidth of the input and output ports is 8 GB/s. The input buffer and the weight buffer support the ping-pong buffering of data. The reshape unit transforms the input feature maps and weights to the forms required by different types of convolutions. The reshape unit provides the accelerator with good versatility and also reduces communications with the control processor (not shown in Figure 3 for simplicity) compared to leaving the reshape operations to the control processor. The convolution engine performs convolution operations, controlled by the instructions loaded from DRAM bank2, on the input matrices and weights. The results of convolution are accumulated along the channel dimension in the output buffer and then transmitted to the quantization unit and the pooling unit. The quantization unit loads quantization parameters for each layer from DRAM bank3. The shortcut connection is added to convolution results in the quantization unit. The output feature maps of a layer, which will be used as the input feature maps for the next layer, could be written to DRAM bank0 or to the input buffer, depending on the on-chip storage space. To improve the system performance, the accelerator works in a pipeline mode to overlap the data processing time and transmission time.

3.2 Parallelization Strategies

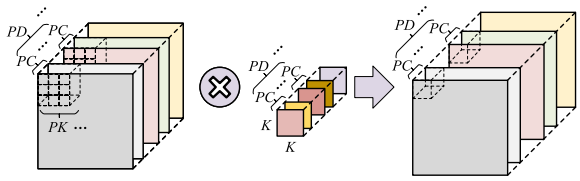
According to the dimensions of the convolution regarded as 6-level loop [29] in CNNs, the proposed accelerator adopts five parallelization strategies, including *PK* (parallelism on kernel dimension), *PV* (parallelism on vector dimension), *PC* (parallelism on input channel dimension), *PF* (parallelism on output channel dimension), and *PD* (parallelism for depthwise convolution).



(a) Parallelization of standard convolution.



(b) Parallelization of pointwise convolution.

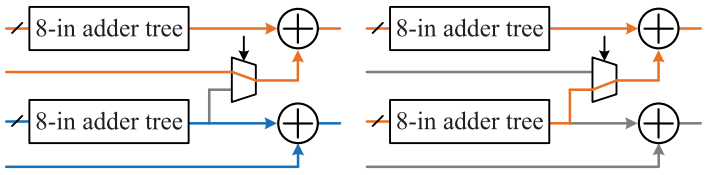


(c) Parallelization of depthwise convolution.

Fig. 4. Parallelization strategies for three types of convolutions.

In the standard convolution, the *PK* strategy aims to perform $K \times K$ multiplications of the elementary convolution operation in parallel. Meanwhile, the convolutions between *PC* channels and *PF* sets of kernels can be processed simultaneously as shown in Figure 4(a). As a result, the convolution engine should contain $PK \times PC \times PF$ multipliers to support the parallelism. The parallelization of the pointwise convolution is similar to the standard convolution but uses the *PV* strategy is adopted instead of *PK* as shown in Figure 4(b). The *PV* strategy process *PK* pixels of the input feature map at the same time. In this way, the convolution engine can remain full workload running. As for the depthwise convolution, it has only one set of kernels. That is, only $PK \times PC$ multipliers are activated in the convolution engine. To increase the utilization of the engine, this article proposes the *PD* parallelization strategy for the depthwise convolution, which is an extension of the *PC* strategy. As shown in Figure 4(c), in the *PD* strategy, the convolution engine processes $PK \times PD \times PC$ pixels of the input feature map in parallel. The utilization ratio of the convolution engine is increased from $1/PF$ to PD/PF, which effectively reduces the processing time of the depthwise convolution.

When implementing the proposed accelerator in this work, the values of *PK*, *PV PC*, *PF*, and *PD* are determined according to the DSCNN model structures and the available hardware resources. Since the common size of the convolution kernel in the typical DSCNN models such MobileNet is 3×3, *K* takes 3 in this work, i.e., 9 multipliers are needed for each convolution core. For the convolution layers containing convolution kernels with different sizes, the convolution operations can be transformed to 3×3 convolutions. Also, we set *PC* and *PF* to 16, and *PD* to 6, with respect



(a) 2-way 9-in mode (b) 1-way 16-in mode

Fig. 5. Basic unit of configurable adder tree.

to the number of input and output channels of the typical CNN models such as MobileNet and ResNet and the used FPGA platform.

3.3 Configurable Adder Tree

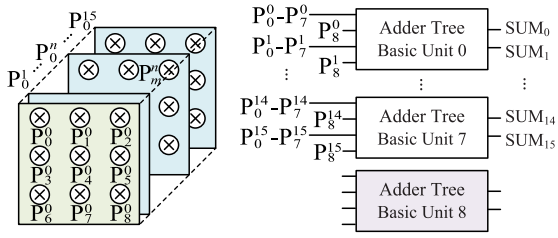
The three convolutions involve different ways of accumulations, bringing the design obstacle to the adder tree in the accelerator. Designing separate adder trees for the three types of convolutions will consume a large amount of computation resources. To address the issue, we design a configurable adder tree, which can be configured online to perform different formats of accumulation required by the three types of convolutions in a way of time division multiplexing. As mentioned in Section 3.2, the parallelism on the channel dimension PC in this article is 16. The depthwise convolution sums 9 products per channel, which means that the adder tree needs to accomplish 16-way 9-in addition. x-way y-in means the adder tree can process x additions, each adding y input data, at the same time. For the pointwise convolution, the adder tree sums the products on the channel dimension. That is, the adder tree needs to accomplish 9-way 16-in addition. Similarly, the standard convolution needs 16-way 9-in addition and 1-way 16-in addition.

The basic unit of the configurable adder tree is illustrated in Figure 5, which is inspired by the storage resource reuse mode of the CNN accelerator [26]. The switching of the operation modes of the configurable adder tree is mainly realized through the multiplexer. Figure 5(a) shows the 2-way 9-in mode while Figure 5(b) shows the 1-way 16-in mode.

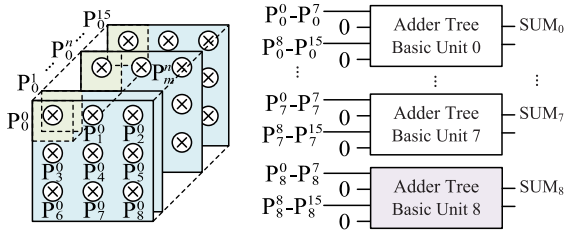
The configurable adder tree instantiates a total of 9 basic units, as shown in Figure 6. Basic Unit 0-Basic Unit 7 are used to accomplish the accumulation of the depthwise and pointwise convolutions. Basic Unit 8 either works as Basic Units 0-7 or sums the results on the channel dimension for the standard convolution. P_m^n is denoted as the mth product in the nth channel. As shown in Figure 6(a), the depthwise convolution only uses 8 basic units of the adder tree in the 2-way 9-in mode. Each basic unit processes the accumulation of 2 channels separately. For example, Basic Unit 0 accumulates the 9 products in the 0th and the 1st channels simultaneously. For the pointwise convolution, all the basic units adopt 1-way 16-in addition mode to sum along the channel dimension (Figure 6(b)). For instance, Basic Unit 0 accumulates P_0^0-P_0^{15} in this case. As for the standard convolution, Basic Units 0-7 adopt the 2-way 9-in mode as the depthwise convolution. In addition, Basic Unit 8 sums the results of the channel dimension, using the 1-way 16-in addition mode (Figure 6(c)).

3.4 Architecture for Bottleneck Block

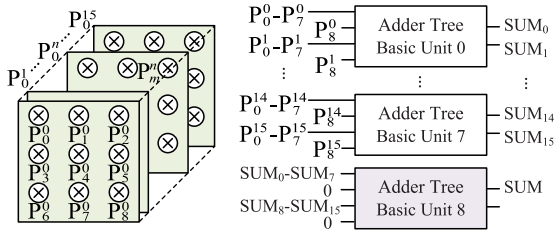
For the single-engine architecture, one of the issues coming from the shortcut connection is that the input feature map of the expansion layer needs to be transmitted to the on-chip memory in the projection layer, resulting in extra data transmission time. Therefore, this article proposes a dedicated architecture combined with the input buffer for the bottleneck block. RAM-S and RAM-M_0 ~ RAM-M_{N-1} are basic buffer units of the input buffer (N=PD). Each basic buffer unit has



(a) Depthwise convolution.



(b) Pointwise convolution.

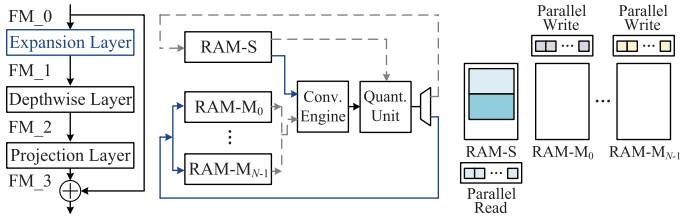


(c) Standard convolution.

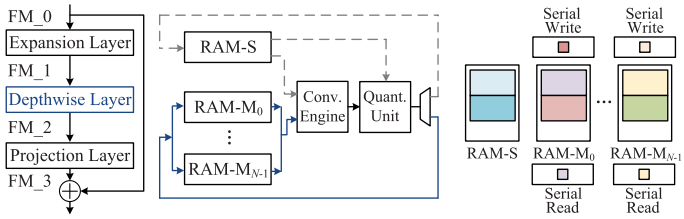
Fig. 6. Configuration of configurable adder tree for three types of convolutions.

a serial read/write mode (*PC* pixels along the channel dimension) and a parallel read/write mode (*PV*×*PC* pixels). The hardware architecture and dataflow are illustrated in Figure 7. Step 1 performs the operations of the expansion layer, which contains pointwise convolution (Figure 7(a)). The input feature map 0 (FM_0) is buffered in RAM-S, and the parallel read mode is enabled. The results are parallelly written to RAM-Ms cyclically in the order of RAM-M_0 ~ RAM-M_{N-1}. As shown in Figure 7(b), step 2 performs the depthwise convolution. To implement the *PD* parallelization strategy, all RAM-Ms are strobed for serial output. The output feature map (FM_2) is written to the position where the input feature map (FM_1) was read. As shown in Figure 7(c), step 3 performs the operations of the projection layer (pointwise convolution) and the shortcut connection. RAM-Ms are strobed in sequence and FM_2 is read into the convolution engine in parallel. Meanwhile, FM_0 that has been stored in RAM-S is read as the input of shortcut connection. The result (FM_3) is written back to RAM-S using the parallel write mode.

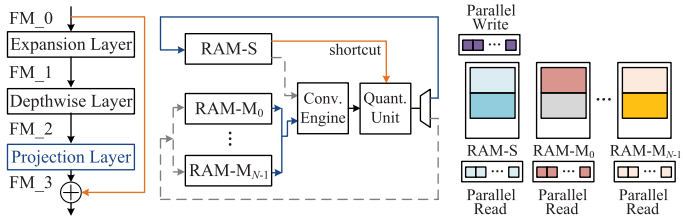
The three steps mentioned above form a closed loop. As discussed in Reference [25], the bottleneck block is memory-bound by the limited memory bandwidth. For this issue, the proposed architecture caches the intermediate results in the bottleneck block to eliminate the need for data transmission between on-chip and off-chip memory. Besides, the input feature map of the



(a) Step 1. Expansion Layer



(b) Step 2. Depthwise Layer



(c) Step 3. Projection Layer

Fig. 7. Architecture and operation flow for the bottleneck block and shortcut connection.

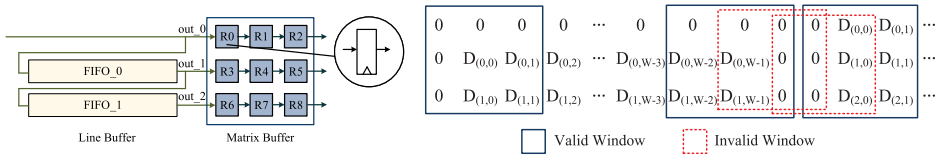
expansion layer is reused in the projection layer, which avoids extra data transmission of the shortcut connection. The data transmission time per bottleneck block saved by the proposed architecture can be evaluated by

$$T_{\text{extra}} = \frac{H_{\text{i}} \times W_{\text{i}} \times \lceil C_{\text{i}}/PC \rceil \times B_{\text{data}}}{Bandwith_{\text{DMA}}}, \tag{1}$$

where B_{data} is the bit-width of a pixel of the input feature map and $Bandwith_{\text{DMA}}$ is the bandwidth of the **direct memory access (DMA)** interface between the on-chip and off-chip memories. It can be observed from Equation (1) that the proposed architecture for bottleneck significantly reduces transmission overhead, which is proportional to the size and number of channels of the input feature map. It is beneficial to relieve the transmission pressure caused by the limited bandwidth of the off-chip memory. In addition, the combination of RAM-S and RAM-Ms essentially form a ping-pong buffer, which contributes to the system performance improvement.

3.5 Operation-separate Padding

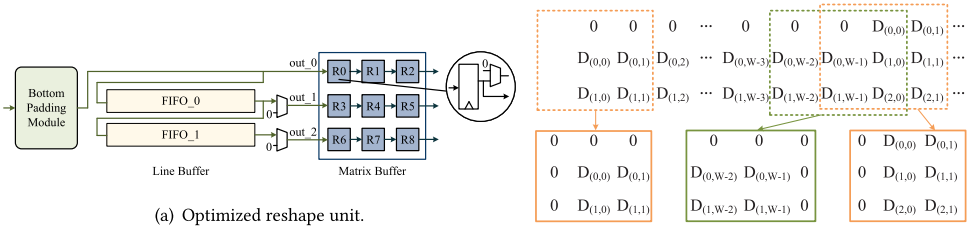
Padding is a process of adding zeros on the boundaries of input feature maps to avoid obliterating information. The direct padding scheme, which inserts zeros directly on the input feature maps before the reshape unit, is the widely used padding scheme [25, 30]. Zeros are inserted during the



(a) Reshape unit.

(b) Data windows under the direct padding scheme.

Fig. 8. Architecture of the reshape unit and the direct padding scheme.



(a) Optimized reshape unit.

(b) Data windows under the proposed padding scheme.

Fig. 9. Architecture of the optimized reshape unit and the proposed padding scheme.

data-loading process in Reference [30], and padding operations are completed before the feature maps are reshaped into 3×3 matrices through a dedicated padding module in Reference [25].

The direct padding scheme has the efficiency issue. As shown in Figure 8(a), the reshape unit is composed of a line buffer and a matrix buffer. The line buffer inputs the pixels one by one and outputs the data line by line. After that, the matrix buffer receives the data from the line buffer and produces 3×3 input matrices to the convolution engine. Figure 8(b) illustrates the data windows of the matrix buffer under the direct padding scheme. $D_{(m,n)}$ is denoted as the pixel in the mth row and nth column in the feature map. The pixels are arranged in the order of entering the matrix buffer from the line buffer. The blue solid frames and the red dotted frames represent the valid and invalid data windows generated by the matrix buffer, respectively. It can be observed from Figure 8(b) that the direct padding scheme inevitably generates invalid windows during the transitions between every two rows. The invalid windows increase the preprocessing time and waste computing resources.

To avoid the invalid windows existing in the direct padding scheme [25, 30], this article proposes an operation-separate padding scheme in which the padding is separated into different operations and realized in different parts of the reshape unit. Figure 9(a) illustrates the optimized architecture of the reshape unit. The bottom padding module inserts zeros at the bottom of the feature map. Padding at the top is accomplished through the multiplexers when the padded data enter the line buffer. As shown in Figure 9(b), an invalid window (green dotted frame) is generated when the data in the 0th column enters the matrix buffer. The outputs of R0, R3, and R6 in the matrix buffer are set to zeros to complete padding operations on the right side of the feature map (green solid frame). Similarly, the outputs of R2, R5, and R8 are set to zeros when the data in the 1st column enters the matrix buffer to pad on the left side (orange frames). Overall, the padding operations of different parts of the feature map are distributed to different modules of the reshape unit.

The padding scheme proposed in this article eliminates the invalid windows during transitions of rows and reduces the preprocessing time before convolution. Take the $H_i \times W_i$ input feature map with padding number of 1 as an example. The direct padding scheme needs $(H_i + 2)(W_i + 2)$ cycles

every PC channels due to adding zeros on the boundaries. The optimized scheme only inserts zeros at the bottom of the feature map, so it consumes $(H_\mathrm{i}+1)W_\mathrm{i}$ cycles every PC channels. The acceleration obtained by the proposed scheme is evaluated by

$$h = \frac{(H_\mathrm{i}+2)(W_\mathrm{i}+2)}{(H_\mathrm{i}+1)W_\mathrm{i}} = \left(1+\frac{2H_\mathrm{i}+W_\mathrm{i}+4}{H_\mathrm{i}W_\mathrm{i}+W_\mathrm{i}}\right)\times 100\%. \tag{2}$$

It can be observed from Equation (2) that the smaller the size of feature map is, the better acceleration the proposed scheme achieves. In MobileNetV2, the scheme can achieve up to 1.45× performance improvement in the padding process.

4 AFFINE QUANTIZATION

As discussed in Section 2, the accuracy of current accelerators for MobileNet drops significantly compared with the single-precision floating-point implementation. To reduce the loss of accuracy, this article uses the affine quantization scheme [31] to convert floating-point numbers to 8-bit integers. The affine quantization scheme is suitable for models that are already efficient at trading off latency with accuracy (e.g., MobileNet), and is hardware-friendly [41]. The data format of 8-bit integer is adopted, because it can reduce the model size while ensuring the model accuracy according to Reference [42]. In affine quantization, a real value r is quantized to an integer q as (3),

$$q = r/S + Z, \tag{3}$$

where S and Z are quantization parameters. The affine quantization is applied to the inference process of DSCNN. To reduce the hardware implementation complexity of the quantization scheme, this work fuses linear activation, ReLU6 and shortcut connection involved in the CNN structure into the quantization scheme. The quantization scheme saves computing and storage resources and improves system performance on the premise that the classification accuracy of the model is only a small loss. The scheme is implemented as a dedicated quantization unit shown in Figure 10.

Assume that the number of multiplication operations required for the convolution is N. $r^j_{d/w/o}$, $q^j_{d/w/o}$, $S_{d/w/o}$, and $Z_{d/w/o}$ indicate the floating-point value, quantized integer value and quantization parameters of the input feature map, the weights, and the output feature map, respectively. Given the convolution with the linear activation in Equation (4),

$$r_\mathrm{o} = \sum_{j=1}^{N} r^j_\mathrm{d} r^j_\mathrm{w} + r_\mathrm{b}, \tag{4}$$

the affine quantization of it is

$$q_o = Z_\mathrm{o} + M\times\left(q_{\mathrm{b\_f}} - Z_\mathrm{w}\sum_{j=1}^{N} q^j_\mathrm{d} + \sum_{j=1}^{N} q^j_\mathrm{w} q^j_\mathrm{d}\right), \tag{5}$$

where M is equal to $S_d S_w/S_o$ and $q_{\mathrm{b\_f}}$ is defined in Equation (6),

$$q_{\mathrm{b\_f}} = q_\mathrm{b} + NZ_\mathrm{d}Z_\mathrm{w} - Z_\mathrm{d}\sum_{j=1}^{N} q^j_\mathrm{w}. \tag{6}$$

Similarly, given the convolution with the ReLU6 activation function in Equation (4),

$$r_\mathrm{o} = \mathrm{ReLU}\,6\left(\sum_{j=1}^{N} r^j_\mathrm{d} r^j_\mathrm{w} + r_\mathrm{b}\right). \tag{7}$$

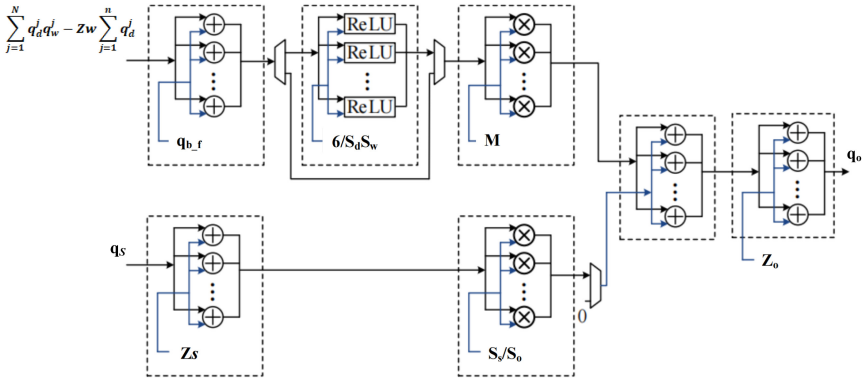


Fig. 10. Quantization unit.

The affine quantization transformation of Equation (4) becomes Equation (8),

$$q_{\mathrm{o}} = Z_{\mathrm{o}} + M \times \mathrm{ReLU}\left[\frac{6}{S_{\mathrm{d}}S_{\mathrm{w}}}\right]\left(q_{\mathrm{b\_f}} - Z_{\mathrm{w}}\sum_{j=1}^{M} q_{\mathrm{d}}^{j} + \sum_{j=1}^{N} q_{\mathrm{w}}^{j} q_{\mathrm{d}}^{j}\right), \tag{8}$$

where the ReLU6 activation function is transformed to

$$\mathrm{ReLU}\left[\frac{6}{S_{\mathrm{d}}S_{\mathrm{w}}}\right](x) = \begin{cases} \frac{6}{S_{\mathrm{d}}S_{\mathrm{w}}} & \left(x \geq \frac{6}{S_{\mathrm{d}}S_{\mathrm{w}}}\right) \\ x & \left(0 \leq x < \frac{6}{S_{\mathrm{d}}S_{\mathrm{w}}}\right) \\ 0 & (x < 0) \end{cases}. \tag{9}$$

As shown in Figure 2, the shortcut involves data transmission across layers. Assuming the cross-layer input feature map in the shortcut connection operation is r_s. The output of the last layer in the bottleneck structure plus r_s is represented as follows:

$$r_{\mathrm{o}} = \sum_{j=1}^{N} r_{\mathrm{d}}^{j} r_{\mathrm{w}}^{j} + r_{\mathrm{b}} + r_{\mathrm{s}}. \tag{10}$$

Similarly, applying the affine quantization to Equation (10) leads to

$$q_{\mathrm{o}} = Z_{\mathrm{o}} + M \times \left(q_{\mathrm{b\_f}} - Z_{\mathrm{w}}\sum_{j=1}^{N} q_{\mathrm{d}}^{j} + \sum_{j=1}^{N} q_{\mathrm{w}}^{j} q_{\mathrm{d}}^{j}\right) + \frac{S_{\mathrm{s}}}{S_{\mathrm{o}}}\left(q_{\mathrm{s}} - Z_{\mathrm{s}}\right) \tag{11}$$

Overall, the quantization unit performs the computations in Equations (5), (8), and (11), respectively, according to the linear activation, ReLU6 activation, and shortcut operations.

In the equations, quantization parameters can be obtained offline. $Z_{\mathrm{w}}\sum_{j=1}^{N} q_{\mathrm{d}}^{j}$ and $\sum_{j=1}^{N} q_{\mathrm{d}}^{j} q_{\mathrm{w}}^{j}$ are calculated in the convolution unit. The other computations need to be calculated in the quantization unit. Specifically, in Figure 10, the upper part calculates the bias and activation functions and multiplies the result with M; the lower part calculates quantization result of the cross-layer input feature map (i.e., the third term of Equation (11)). Then, if the shortcut exists, then the upper part adds the lower part; otherwise, it adds zero. Finally, Z_0 is added to the previous sum. Note that three FIFOs are used. The output feature map has a different bias per channel, so FIFO_1 stores all the bias $q_{\mathrm{b\_f}}$ in the current layer. FIFO_2 stores the upper bound $\frac{6}{S_{\mathrm{d}}S_{\mathrm{w}}}$ of the ReLU6 function, which is calculated offline. Similarly, FIFO_3 stores M.

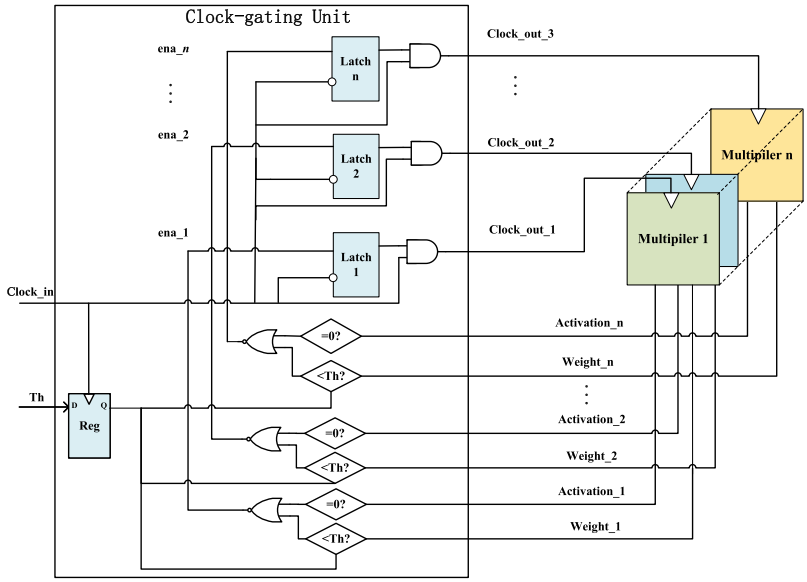


Fig. 11. Pruning unit.

The quantization unit results in 8-bit unsigned integers. The outputs are transferred to the off-chip memory or the on-chip input buffer as the inputs to the next convolution layer. With the 8-bit affine quantization scheme, the designed inference accelerator achieves better tradeoff between performance and accuracy than the existing FPGA-based accelerator designs, which will be presented in Section 7.

5 HARDWARE-ASSISTED DATA PRUNING

The post-training pruning method is promoting from a practical perspective, because model retraining is avoided [35]. This means that the CNN model to be pruned could come from any frameworks and access to the original training platform and training dataset is not required. For a step further, we design a hardware-assisted pruning method, which can prune the model online on the accelerator. This allows the on-site tuning between the model accuracy and power consumption.

To realize the hardware-assisted pruning method, we design a pruning unit shown in Figure 11, which is integrated into the convolution engine. In Figure 11, we add a clock-gating unit to control the clock input of each multiplier in the convolution engine. The enabling signal ena_i is generated by monitoring the values of weight_i and activation_i. Either weight_i is less than a threshold Th or activation_i is equal to zero, ena_i is zero. Then, the corresponding multiplier_i is clock-gated off. When clock-gated off, the input of the adder tree connected to multiplier_i is switched to zero with a 2-to-1 MUX, which is controlled by ena_i. In this way, the power consumption is reduced. Therefore, by setting different thresholds Th we fulfill the weight pruning with different sparsities.

Now the question is how Th is determined given a sparsity without large accuracy drop. We propose a method for determining the threshold Th_l for each layer l offline. Given a CNN model, a dataset (such as ImageNet), and the targeted model sparsity S, we adopt the heuristic approach [35] to obtain the layerwise sparsity S_l and the corresponding Th_l, together with the model accuracy. Then the model is deployed on the accelerator, and its power consumption is measured while each layer of the model is pruned according to Th_l. We repeat this procedure for different S^k such as from 0% to 80%, to obtain $Th^k = \{Th_1^k, Th_2^k, \ldots, Th_l^k, \ldots\}$, the corresponding model accuracy A^k

and power consumption P^k. In this way, we build a lookup table $\mathbf{T}(S^k, Th^k, A^k, P^k)$. When the accelerator is deployed on-site for a certain application with specific model accuracy and power consumption requirements, the threshold Th^k can be determined by looking up $\mathbf{T}$. As a result, the accelerator can switch the pruning modes on-line, allowing convenient deployment according to the application scenarios.

6 MAPPING TOOL

The proposed accelerator design supports various CNNs with the standard convolution, DSCs and bottleneck structure. This section presents a mapping tool, which can map the CNN models onto the accelerator by generating control instructions. The tool performs three steps, slicing, allocating, and scheduling, according to the parameters extracted from the machine learning framework such as TensorFlow.

Slicing. The three-dimensional (3D) input feature tensor and 4D weights tensor need to be sliced to be accommodated in on-chip buffers and support the parallel computations. As described in Section 3, the proposed accelerator adopts five parallelization strategies. For the standard convolution, the input feature map is sliced along height, width. and input channel to generate $K \times K \times PC$ blocks. The weights are sliced along input and output channels to generate $K \times K \times PC \times PF$ blocks. The input feature map and weights are sliced similarly for the depthwise convolution and pointwise convolution.

Allocating. After slicing, the tool allocates the computation and storage resources for performing the convolution of the input feature map block and the weight block. With respect to the bottleneck structure, as discussed in Section 3, the on-chip RAM-S and RAM-M_s are allocated to store the feature maps of the first and last layers, respectively.

Scheduling. Given the single convolution engine design, different layers of a CNN model are mapped onto the accelerator one by one. The process of each layer includes inputting the feature map blocks and weight blocks, performing convolution, outputting result blocks, quantization, and pooling. These operations are scheduled in a pipeline shown in Figure 12. After inputting the feature map and weights of layer L0, the convolution engine starts to perform convolution block by block. The mentioned five parallelization strategies are deployed in the convolution engine. At the same time, the weights of Layer L1 are loaded. After each block is computed, it is transferred to the output buffer. When the convolution of layer L0 is completed, the quantization and pooling units start operation and the results are outputted to the input buffer used as the input feature map of layer L1. The operation flow for the bottleneck block and the shortcut connection is scheduled at this step. Afterwards, the convolution of layer L1 starts.

After three steps, the tool can generate the control instructions, which will control the execution of the proposed accelerator. As shown in Figure 3, the instructions are transferred to the control unit, and the control unit configures each unit to execute different tasks without recompiling FPGA.

7 EXPERIMENTAL RESULTS

The proposed accelerator is designed in the RTL level using Verilog. The accelerator is implemented in an FPGA platform with the Xilinx Virtex-7 XC7V690t FPGA device, which is available to our experiments. Other FPGA platforms can also be used as long as meeting the resource requirements of the accelerator. After placement and routing, the accelerator runs at 150 MHz clock frequency. To validate the accelerator, the typical DSCNN models (MobileNetV1, MobileNetV2) and the CNN models with bottleneck (ResNet-50, ResNet-152) are mapped onto the inference accelerator. The ImageNet dataset is used. The experimental results include four parts. The first part evaluates the resource usage of the accelerator. The second part evaluates the performance

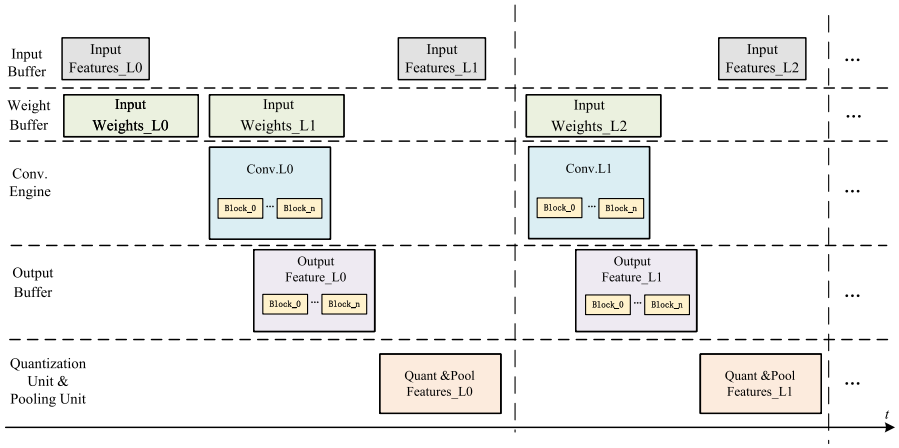


Fig. 12. Pipeline of the accelerator.

Table 1. Resource Usage of the Proposed Accelerator

Resource	Utilization	Available	Percentage
LUT	308,449	433,200	71.20%
BRAM	941.5	1,470	64.05%
DSP	2,160	3,600	60.00%
FF	278,926	866,400	32.19%

of MobileNets on the accelerator compared to their implementations on embedded CPU, desktop CPU, desktop GPU, and other FPGA accelerators. The third part makes the similar comparisons on the ResNet models. The last part shows the effect of the hardware-assisted pruning method.

7.1 Resource usage

Table 1 shows the resource usage of the proposed accelerator. As mentioned in Section 3.2, the accelerator exploits five strategies for the parallelism inherent in the convolution operations. Given $PK = 9$, $PV = 9$, $PC = 16$, $PF = 16$, and $PD = 6$, there are 2,304 multipliers and nine adder tree units in the accelerator. To balance resource usage and performance of the whole accelerator design, we implement the multipliers using the DSP blocks embedded in the FPGA and the adder tree units in the **lookup tables (LUTs)**. Each multiplier is implemented with three pipeline stages and each adder with two pipeline stages. Considering the actual resources on FPGA, the accelerator consumes 2,160 DSP blocks. The input buffer, weight buffer, and output buffer as the accelerator's on-chip memory are implemented by 941 32k-bit BRAM blocks and one 16k-bit BRAM block. The most consumed resource is LUT, which is 71.2% of the available LUTs in the FPGA. This means that smaller FPGAs can be used. In addition, because some existing FPGA-based inference accelerators related to our design use different FPGA devices and platforms, we evaluate them in terms of resource utilization efficiency (GOPS per resource) in Section 7.3. High resource utilization efficiency means that higher performance is achieved with less resource usage.

7.2 MobileNets Acceleration

MobileNetV1 and MobileNetV2 models are mapped onto the accelerator. This part compares the performance and energy efficiency of the accelerator to the implementations on embedded CPU,

Table 2. Comparison of CPU, GPU, and the Proposed Accelerator Implementations of MobileNets

Network	MobileNetV1		MobileNetV2			
	Embedded CPU	Our Work	Embedded CPU	Desktop CPU	Desktop GPU	Our Work
Platform	Qualcomm Snapdragon 821	Xilinx Virtex-7 XC7V690t	Qualcomm Snapdragon 821	Intel Core i7-6700HQ	NVIDIA GeForce GTX 960M	Xilinx Virtex-7 XC7V690t
Frequency (MHz)	2,400	150	2,400	2,600	1,096	150
Processing Time per Frame (ms)	113	3.73	75	12.8	4.55	3.31
FPS	8.84	267.8	13.3	78.2	219.7	302.3
Performance (GOPS)	9.26	305.2	8.0	47.1	132.2	181.8
Power (W)	11	11.24	11	29.88	46.67	11.35
Energy Efficiency (GOPS/W)	0.84	27.15	0.73	1.58	2.83	16.02

desktop CPU, desktop GPU, and FPGAs. The results are shown in Table 2. The embedded CPU is Qualcomm Snapdragon 821 and the reported results are from Reference [21], where the power consumption data were not shown. We obtain the nominal power consumption data of Qualcomm Snapdragon 821 from Reference [43] as a reference. The desktop CPU and GPU are Intel Core i7-6700HQ with clock frequency of 2.40 GHz and NVIDIA GeForce GTX 960M with clock frequency of 1.10 GHz, respectively. The implementation in Intel Core i7 CPU uses the SIMD instructions. We implement MobileNets on the CPU and GPU platforms based on TensorFlow using 32-bit single-precision floating-point data, because both i7 CPU and GTX 960M GPU contain 32-bit processing cores. The reported power consumption values of the i7 CPU, the GTX 960M GPU, and the FPGA are the board-level power measured by a power meter.

When MobileNetV1 is mapped onto the accelerator, it achieves 305.2 GOPS and 267.8 **frames per secons (FPS)**. Compared with the Snapdragon 821 CPU, it achieves 32.9× speedup. The MobileNetV2 implemented on the accelerator reaches 181.8 GOPS and 302.3 FPS. Compared with the Snapdragon 821 CPU, it achieves 22.7× speedup. Compared with the i7-6700HQ CPU and GTX 960M GPU, the accelerator achieves 3.9× and 1.4× acceleration, respectively. In terms of energy efficiency, the MobileNetV2 accelerator reaches 10.1× and 5.7× of i7-6700HQ and GTX 960M, respectively. These results demonstrate that the proposed accelerator has advantages of high performance and high energy efficiency, and is suitable for mobile devices with a real-time requirement.

To further evaluate the performance of the proposed accelerator, we also compare it with the related FPGA accelerators of the MobileNet models, and the results are shown in Table 3. The software implementations [20, 21] on CPU are used as the baseline. In terms of architecture, all accelerators except for Reference [23] use the single-engine architecture. In terms of performance, the accelerator proposed in this article shows the highest performance among the accelerators with the single-engine architecture. The performance advantages come from the efficient convolution parallelization, efficient padding operation, and the dedicated design for the bottleneck structure. The performance of the accelerator in Reference [23] is higher than that of the rest accelerators listed in Table 3, mainly due to its higher clock frequency (333 MHz) and two dedicated convolution

Table 3. Comparison of FPGA-based MobileNets Accelerators

Design	Network	Platform	Frequency (MHz)	FPS	Performance (GOPS)	Precision	Top-1 Accuracy
[20]	MobileNetV1	CPU	2,400	8.84		float32	70.6%
[21]	MobileNetV2	CPU	2,400	13.3	8.0	float32	72.0%
[28]	MobileNetV1	Stratix-V	150	231.2	–	int8	–
[24]	RR-MobileNet	ZU9EG	150	127.4	91.2	int8/4	64.6%
[25]	SSDLiteM2	7Z045	100	64.8	–	int8+float32	–
[23]	MobileNetV2	ZU2EG	433	205.3	123.5	int8	68.1%
[23]	MobileNetV2	ZU9EG	333	809.8	487.1	int8	68.1%
[26]	MobileNetV2	Arria-10	133	266.2	160.1	int16	–
Our work	MobileNetV1	Virtex-7	150	267.8	305.2	int8	69.3%
Our work	MobileNetV2	Virtex-7	150	302.3	181.8	int8	70.8%

engines. At the quantization level, most accelerators use the integer 8-bit numerical representation with more than 3.9% loss in Top-1 accuracy. The MobileNetV2 accelerator in this work using the affine quantization scheme reaches 70.8% Top-1 accuracy, which achieves the highest classification accuracy among the accelerators listed in Table 3.

7.3 ResNets Acceleration

ResNets models use standard convolutions and contains bottleneck structure. We also map the ResNet-50 and ResNet-152 onto the accelerator to show its performance and versatility. Similarly, the performance and energy efficiency are evaluated compared to ResNets' implementations on CPU and FPGA platforms. The results are shown in Table 4. The CPU is Intel Core i7-11700 with clock frequency of 2.50 GHz. ResNets runs on the CPU platform based on TensorFlow, using 32-bit single-precision floating-point representation.

When ResNet-50 is implemented on the accelerator, it reaches 252.63 GOPS, which is 1.3× higher than the CPU implementation, 1.9× higher than the FPGA implementation [40] and similarly to Reference [38]. In terms of resource utilization efficiency, our accelerator shows 2.6× improvement over Reference [38] and similar to that in Reference [40].

Similar results can also be observed on ResNet-152. The accelerator achieves 281.17 GOPS and 12.43 FPS, which is superior to CPU implementation and the FPGA implementation [37]. Also with plenty of DSPs, the FPGA implementation [39] shows great performance and has the similar resource utilization efficiency to our accelerator. Overall, the proposed accelerator shows good performance and efficiency for accelerating ResNet-50 and ResNet-152.

7.4 Hardware-assisted Data Pruning Results

The proposed hardware-assisted pruning method allows online tradeoff between model accuracy and power consumption. In this part, we use MobileNetV2 model to evaluate the method. Figure 13 shows the Top-1 accuracy and power consumption over different weight sparsities when the model is mapped onto the accelerator.

As shown in the figure, when the sparsity of the weights increase from 0% to 75% by tuning the threshold, the power consumption of the accelerator decreases from 11.35 to 4.57 W linearly. At the same time, the model Top-1 accuracy drops from 70.8% to 66.65%. Overall, the power consumption of the accelerator reduces about 59.7% at the accuracy loss under 5%. This demonstrates that the accelerator allows model pruning online to meet different requirements of power consumption and model accuracy. With this function, the model does not require retaining and is easily to be deployed on the accelerator in different application scenarios.

Table 4. Comparison of CPU and FPGA Implementations of ResNets

Network	ResNet-50				ResNet-152			
	Desktop CPU	FPGA [40]	FPGA [38]	Our Work	Desktop CPU	FPGA [37]	FPGA [39]	Our Work
Platform	Intel Core i7-11700	Intel Zynq 7Z045	Xilinx Virtex US+ VU9P	Xilinx Virtex-7 XC7V690t	Intel Core i7-11700	Intel Stratix V GXA7	Xilinx VU9P	Xilinx Virtex-7 XC7V690t
DSP Usage	–	900	6,005	2,160	–	–	5,632	2,160
Frequency (MHz)	2500	166	125	150	2500	150	180	150
Processing Time per Frame (ms)	40	–	28.9	30.64	129.36	81.19	15.24	80.45
FPS	25	–	34.6	32.64	7.73	12.31	65.5	12.43
Performance (GOPS)	193.5	130.4	268.5	252.63	174.85	278.67	805.27	281.17
Density (GOPS/DSP)	–	0.14	0.05	0.12	–	–	0.14	0.13
Power (W)	48.39	9.4	–	10.87	50.27	–	–	10.65
Energy Efficiency (GOPS/W)	3.99	13.87	–	23.24	2.83	–	–	26.40

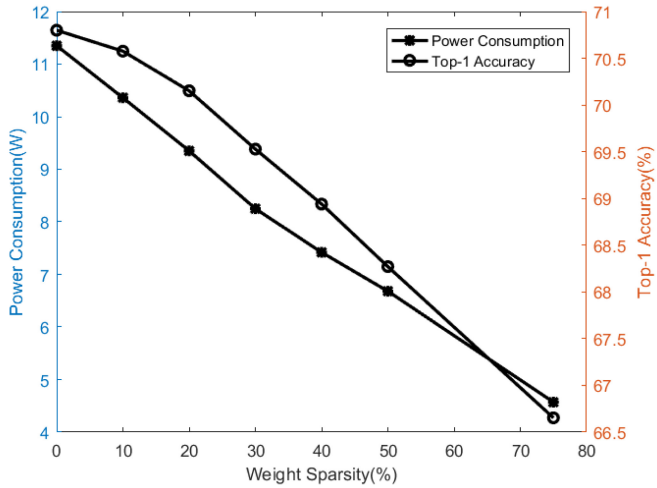


Fig. 13. Evaluation of the hardware-assisted data pruning function of the accelerator implementing MobileNetV2.

8 CONCLUSION

This article designs an efficient inference accelerator for CNNs with DSCs and bottleneck structures on FPGA. The accelerator possesses four structural features: a convolution engine supporting DSCs and hardware pruning, an input buffer facilitating the bottleneck, a reshape unit optimizing zero-padding, and a quantization unit realizing affine quantization. With these features, the accelerator can support various CNNs with standard convolution, DSC and bottleneck. Experimental

results show that for MobileNetV2 the accelerator achieves 10× and 6× energy efficiency improvement over the CPU and GPU implementation, and 302.3 FPS and 181.8 GOPS performance that is the best among several existing single-engine accelerators on FPGAs. The proposed hardware-assisted pruning can effectively trade model accuracy for power consumption online.

In the future, the proposed accelerator is expected to support more models such as MobileNetV3 [32] and EfficientNet [33] by adding 5×5 bottleneck and activation functions such as h-swish and swish. In addition, existing CNNs contain a large proportion of zeros in both weights and activation. The accelerator design will be optimized to support compressed storage, transfer, and processing of weights and activations to further improve performance, storage, and energy consumption. Finally, considering all design parameters, the design space is huge. We will formulate and automate the design space exploration.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 779–788.

[3] J. Redmon and A. Farhadi. 2017. YOLO9000: Better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 7263–7271.

[4] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie. 2020. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proc. IEEE* 108, 4 (2020), 485–532.

[5] Y. Gong, L. Liu, M. Yang, and L. Bourdev. 2014. Compressing deep convolutional networks using vector quantization. arXiv:1412.6115. Retrieved from https://arxiv.org/abs/1412.6115

[6] F. Li, B. Zhang, and B. Liu. 2016. Ternary weight networks. arXiv:1605.04711. Retrieved from https://arxiv.org/abs/1605.04711

[7] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. 2016. Binarized neural networks. In *Advances in Neural Information Processing Systems*. 4107–4115.

[8] S. Han, H. Mao, and W. J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained huffman coding. arXiv:1510.00149. Retrieved from https://arxiv.org/abs/1510.00149

[9] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. arXiv:1712.01887. Retrieved from https://arxiv.org/abs/1712.01887

[10] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. 2017. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*. 2736–2744.

[11] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. 2016. SqueezeNet: Alexnet-level accuracy with 50× fewer parameters and <0.5 MB model size. arXiv:1602.07360. Retrieved from https://arxiv.org/abs/1602.07360

[12] X. Zhang, X. Zhou, M. Lin, and J. Sun. 2018. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6848–6856.

[13] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. 2018. ShuffleNet V2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*. 116–131.

[14] S. Yan et al. 2021. An FPGA-based MobileNet accelerator considering network structure characteristics. In *Proceedings of the 31st International Conference on Field-Programmable Logic and Applications (FPL'21)*. 17–23.

[15] S.-F. Hsiao and B.-C. Tsai. 2021. Efficient computation of depthwise separable convolution in MoblieNet deep neural network models. In *Proceedings of the IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW'21)*.

[16] B. Li et al. 2021. Dynamic dataflow scheduling and computation mapping techniques for efficient depthwise separable convolution acceleration. *IEEE Trans. Circ. Syst. I: Regul. Pap.* 68, 8 (2021), 3279–3292.

[17] L. Xiaolin, R. C. Panicker, B. Cardiff, and D. John. 2012. Multistage pruning of CNN based ECG classifiers for edge devices. In *Proceedings of the 43rd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC'21)*. 1965–1968.

[18] S. Sarkar, M. Agarwalla, S. Agarwal, and M. P. Sarma. 2020. An incremental pruning strategy for fast training of CNN models. In *Proceeeings of the International Conference on Computational Performance Evaluation (ComPE'20)*. 371–375.

[19] S. Kim and H. Kim. 2021. Linear domain-aware log-scale post-training quantization. In *Proceedings of the IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia'21)*. 1–3.

[20] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861. Retrieved from https://arxiv.org/abs/1704.04861

[21] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4510–4520.
[22] G. Wei, Y. Hou, Z. Zhao, Q. Cui, G. Deng, and X. Tao. 2018. Demo: FPGA-cloud architecture for CNN. In *Proceedings of the 24th Asia-Pacific Conference on Communications (APCC'18)*. 7–8.
[23] D. Wu, Y. Zhang, X. Jia, L. Tian, T. Li, L. Sui, D. Xie, and Y. Shan. 2019. A high-performance CNN processor based on FPGA for MobileNets. In *Proceedings of the 29th International Conference on Field Programmable Logic and Applications (FPL'19)*. IEEE, 136–143.
[24] J. Su, J. Faraone, J. Liu, Y. Zhao, D. B. Thomas, P. H. Leong, and P. Y. Cheung. 2018. Redundancy-reduced mobilenet acceleration on reconfigurable logic for imagenet classification. In *International Symposium on Applied Reconfigurable Computing*. Springer, 16–28.
[25] H. Fan, S. Liu, M. Ferianc, H.-C. Ng, Z. Que, S. Liu, X. Niu, and W. Luk. 2018. A real-time object detection accelerator with compressed SSDLite on FPGA. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'18)*. IEEE, 14–21.
[26] L. Bai, Y. Zhao, and X. Huang. 2018. A CNN accelerator on FPGA using depthwise separable convolution. *IEEE Trans. Circ. Syst. II: Expr. Briefs* 65, 10 (2018), 1415–1419.
[27] B. Liu, D. Zou, L. Feng, S. Feng, P. Fu, and J. Li. 2019. An FPGA-Based CNN accelerator integrating depthwise separable convolution. *Electronics* 8, 3 (2019), 281.
[28] R. Zhao, X. Niu, and W. Luk. 2018. Automatic optimising CNN with depthwise separable convolution on FPGA: (abstact only). In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 285–285.
[29] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 161–170.
[30] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy. 2018. VTA: An open hardware-software stack for deep learning. arXiv:1807.04188. Retrieved from https://arxiv.org/abs/1807.04188
[31] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. 2018. Training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2704–2713.
[32] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, and V. Vasudevan, et al. 2019. Searching for MobileNetV3. In *Proceedings of the IEEE International Conference on Computer Vision*. 1314–1324.
[33] M. Tan and Q. Le. 2019. EfficientNet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*. PMLR, 6105–6114.
[34] T. Gale, E. Elsen, and S. Hooker. 2019. The state of sparsity in deep neural networks. arXiv:1902.09574. Retrieved from https://arxiv.org/abs/1902.09574
[35] I. Lazarevich, A. Kozlov, and N. Malinin. 2021. Post-training deep neural network pruning via layer-wise calibration. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops (ICCVW'21)*. 798–805.
[36] M. S. Abdelfattah et al. 2018. DLA: Compiler and FPGA overlay for neural network inference acceleration. In *Proceedings of the 28th International Conference on Field Programmable Logic and Applications (FPL'18)*. 411–4117.
[37] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo. 2017. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. 1–8.
[38] R. Kuramochi and H. Nakahara. 2020. An FPGA-based low-latency accelerator for randomly wired neural networks. *Proceedings of the 30th International Conference on Field-Programmable Logic and Applications (FPL'20)*. 298–303.
[39] X. Wei, Y. Liang, and J. Cong. 2019. Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management. In *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC'19)*. 1–6.
[40] Y. Liang et al. 2020. Evaluating fast algorithms for convolutional neural networks on fpgas. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 39, 4 (2020), 857–870.
[41] Chen Xiaobo et al. 2011. Real-time affine invariant patch matching using DCT descriptor and affine space quantization. In *Proceedings of the IEEE International Conference on Image Processing IEEE*.
[42] W. Pang et al. 2020. 8-bit convolutional neural network accelerator for face recognition. In *Proceedings of the 11th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON'20)*.
[43] Description of Qualcomm Snapdragon 821 MSM8996 Pro. Retrieved from https://rankquality.com/en/qualcomm-snapdragon-821-msm8996-pro/

Received 2 June 2022; revised 23 May 2023; accepted 4 August 2023