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

LIU, Zhengyan; LIU, Qiang; YAN, Shun; CHEUNG, Ray C. C.

Published in:
ACM Transactions on Reconfigurable Technology and Systems

Published: 01/03/2024

Document Version:
Post-print, also known as Accepted Author Manuscript, Peer-reviewed or Author Final version

Published version (DOI):
10.1145/3615661

Publication details:
LIU, Z., LIU, Q., YAN, S., & CHEUNG, R. C. C. (2024). An Efficient FPGA-based Depthwise Separable Convolutional Neural Network Accelerator with Hardware Pruning. *ACM Transactions on Reconfigurable Technology and Systems*, *17*(1), Article 15. https://doi.org/10.1145/3615661

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 paper proposes an efficient inference accelerator on FPGA for CNNs with depthwise separable convolutions (DSCs). To improve the accelerator efficiency, this work: (1) designs an efficient convolution engine with multiple parallel strategies and a configurable adder tree to support three types of convolution operations; (2) designs a dedicated architecture with input buffers for bottleneck network structure to reduce data transmission time; (3) proposes a hardware padding scheme to eliminate invalid padding operations; and (4) develops hardware-assisted pruning to make trade-off between model accuracy and power consumption on-line. Experimental results show that for MobileNetV2 the accelerator achieves 10x and 6x energy efficiency improvement over the CPU and GPU implementation, and 302.3 FPS and 181.8 GOPS performance which is the best among several existing single-engine accelerators on FPGAs. The proposed hardware-assisted pruning can effectively reduce 59.7% power consumption at the accuracy loss within 5%.

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. 2022. An Efficient FPGA-based Depthwise Separable Convolutional Neural Network Accelerator with Hardware Pruning. 1, 1 (June 2022), 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 computational-intensive model because of its complex structures and complicated operations. To meet the requirement 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], etc. Compact models aim at designing more efficient and smaller model architectures with acceptable accuracy. The MobileNets network is a typical representative among compact models [20, 21] which adopts depthwise separable convolutions (DSCs) instead of standard convolutions.

Field Programmable Gate Arrays (FPGAs) become ideal platforms for CNN acceleration recently[22]. FPGAs can effectively shorten the development time and facilitate future network optimization due to their reconfigurable substrate.

*Corresponding author.

Authors' addresses: Zhengyan Liu, liuzhengyan@tju.edu.cn; Qiang Liu, qiangliu@tju.edu.cn; Shun Yan, yanshun@tju.edu.cn, School of Microelectronics, Tianjin University, 92nd Rd, Weijin, Tianjin, Nankai, CHINA, 300072, yanshun@thu.edu.cn; Ray C.C. Cheung, Department of Electrical Engineering, City University of Hong Kong.

Recent DSC neural network accelerators based on FPGA can be classified into two categories: (a) a general convolution engine; (b) dedicated computation engines for different types of convolution. A layer pipeline scheme is used to make a convolution engine and a dedicated depthwise convolution engine work in parallel[23]. More implementations use a single convolution engine compatible with 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[24]. A real-time object detection accelerator is implemented for SSDLite-MobileNetV2 (SSDLiteM2) with hardware and software optimizations[25]. A scalable DSC accelerator is proposed in [26]where all layers are processed in a computing unit named matrix multiplication engine (MME). The depthwise and pointwise convolutions are combined to get a higher performance in [15]. A adaptive dataflow scheduling is proposed to achieve a trade-off between hardware efficiency and performance in[16]. Furthermore, roofline model [27] and compiler [28] were used to optimize the FPGA implementations of MobileNet.

However, there is still room in accelerating DSCNN (DSC nerual networks). On the one hand, there is space for hardware design improvement with respect to the structure of network, especially the bottleneck block and shortcut connection. 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 DSCNNs such MobileNets and ResNets with a multi-network mapping tool, and (b) the accelerator is optimized to support hardware-assisted pruning, allowing trade-off between model accuracy and power consumption on-line. The contributions of this paper are as follows:

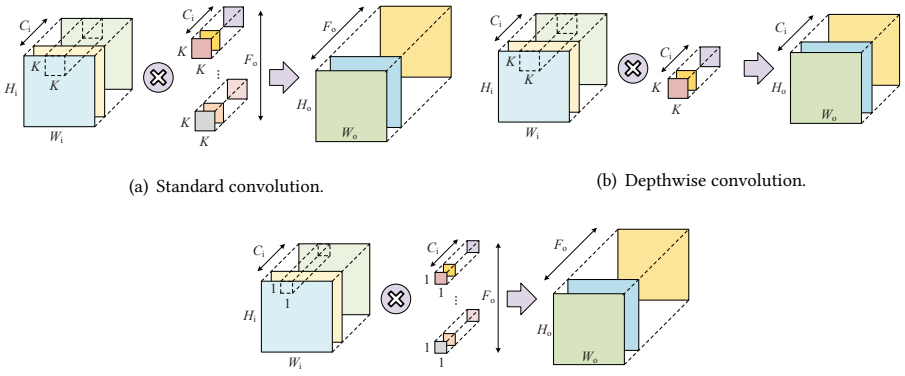
(1) A single-engine accelerator compatible with standard convolution and depthwise separable convolution is proposed. Five parallel strategies are exploited for different convolution operations to increase the utilization of the engine. In addition, a configurable adder tree is proposed to support different ways of accumulations involved in different convolutions, in order to save computing resources.
(2) A dedicated hardware architecture considering 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 the padding is separated into different operations of the reshape unit. The scheme eliminates invalid padding operations and improves efficiency.
(4) An hardware-assisted data pruning method is proposed, which provides different sparsity modes to meet with low power application scenarios with tolerable accuracy loss.

2 BACKGROUND AND PROBLEM STATEMENT

This section introduces the elementary knowledge of DSC, bottleneck structure and the pruning methods. 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 operations and weights effectively [20]. Fig.1 demonstrates the principles of standard, depthwise and pointwise convolutions. The standard convolution takes an $H_\mathrm{i} \times W_\mathrm{i} \times C_\mathrm{i}$ input tensor and applies $K \times K \times C_\mathrm{i} \times F_\mathrm{o}$ convolution weights to produce an $H_\mathrm{o} \times W_\mathrm{o} \times F_\mathrm{o}$ output tensor (Fig.1(a)). Different from the standard convolution, the depthwise convolution only applies one single kernel to each input channel (Fig.1(b)). The pointwise convolution can



(a) Standard convolution.

(b) Depthwise convolution.

(c) Pointwise convolution.

Fig. 1. Different types of convolution.

be regarded as a standard convolution with 1×1 kernels (Fig.1(c)). The network models MobileNets and ResNets are built on the three types of convolutions.

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

2.2 Bottleneck Structures

MobileNetV2 is based on an inverted residual structure that can be regarded as a bottleneck DSC with residuals. As shown in Fig.2(a), a bottleneck block consists of three layers, namely expansion layer, depthwise layer and projection layer. Among them, the expansion layer and the projection layer both perform pointwise convolution and use expansion factor t to control the dimensions of the feature maps. The block applies batch normalization (BN) 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.

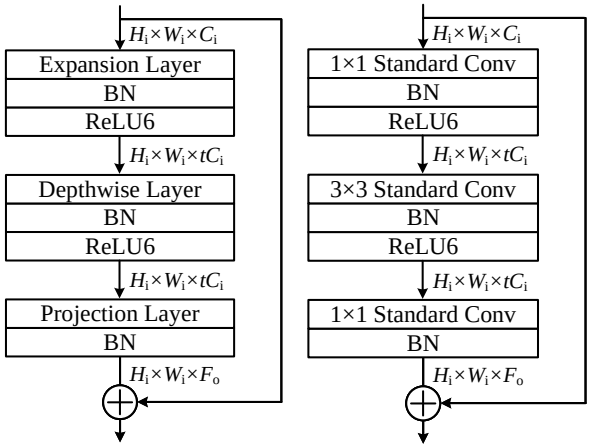
In order to solve the degradation problem in deep networks, some layers of the neural network can artificially skip the connection of neurons in the next layer, and weaken the strong connection between each layer. Such a neural network is called a residual network (ResNets). Fig. 2(b) shows the bottleneck structure of ResNet networks.

This bottleneck structure was used for ResNet-50/101/152. The middle 3x3 convolution layer in the structure first reduces the computation under a dimension-reduced 1x1 convolution layer and then restores under another 1x1 convolution layer, both maintaining the accuracy and reducing the computation.

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, the model compression methods are proposed including quantification and pruning [17].



(a) MobileNet block. (b) ResNet block.

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

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

Network pruning is a method removing unimportant weights, whose value is less than a threshold, during training to reduce the size of model. The central problem of model pruning is how to effectively crop the model weights and minimize the loss of accuracy [18].

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 the model with different pruning sparsity for deploying the model in various low-power scenarios.

3 ACCELERATOR DESIGN

3.1 Design Overview

The proposed accelerator adopts a single convolution engine architecture, as shown in Fig.3. The single-engine architecture is versatile and easy to migrate to other models compared with the separate dedicated convolution engines. The input feature maps, weights and quantization parameters are stored in the external memory. The weight buffer caches the weights on chip and reshapes them into the forms required by different types of convolutions. The input buffer caches the input feature maps on chip and reshapes them into input matrices of 3×3 through the reshape unit. The convolution engine performs convolution operations 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 shortcut connection is added to convolution results in the quantization unit. The output feature maps are written to the off-chip memory or back to the input buffer according to the on-chip storage space. To improve the system performance, the accelerator uses pipeline to overlap the processing time and transmission time.

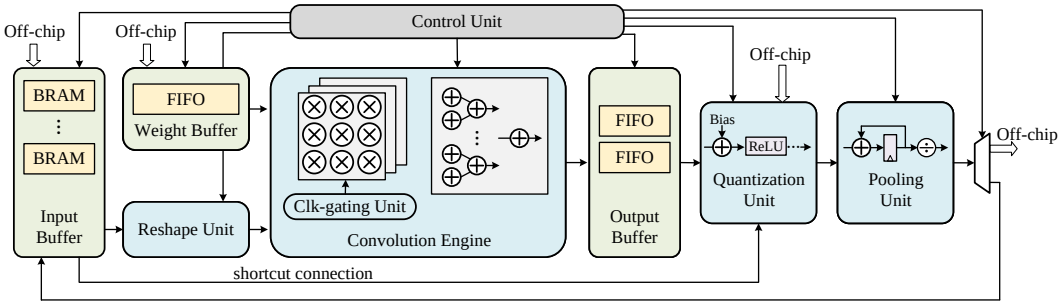


Fig. 3. Design overview of the accelerator.

3.2 Parallel Strategies

According to the dimensions of the convolution in CNNs which is regarded as 6-layer loop [29], this paper adopts parallel strategies including *PK* (parallelism on kernel dimension), *PV* (parallelism on vector dimension), *PC* (parallelism on channel dimension), *PF* (parallelism on filter dimension), and *PD* (parallelism for depthwise convolution).

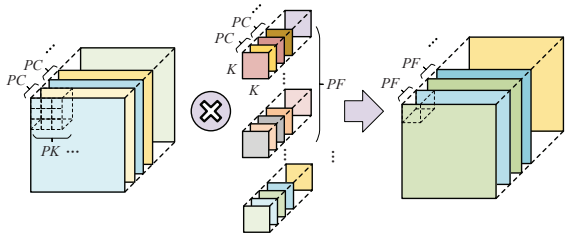
In the standard convolution, the *PK* strategy aims to compute the $K \times K$ convolution in parallel. Meanwhile, the convolution engine can process *PC* channels and *PF* filters as shown in Fig.4(a). In the pointwise convolution, the *PV* strategy is adopted instead of *PK* in order to process 9 pixels in parallel (Fig.4(b)). As for the depthwise convolution, it performs convolution between the single channel of input feature map and the kernel. That is to say, only $PK \times PC$ multiplications are activated in the convolution engine. To increase the utilization of the engine, this paper proposes the *PD* parallel strategy for the depthwise convolution, which is an extension of the *PC* strategy. As shown in Fig.4(c), in the *PD* strategy, the convolution engine processes $PK \times PD \times PC$ pixels of 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. This paper sets *PK*, *PV*, *PC*, *PF*, and *PD* as 9, 9, 16, 16, and 6, respectively.

3.3 Configurable Adder Tree

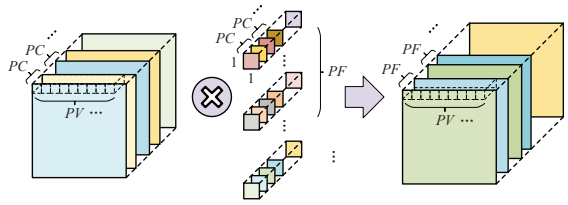
The three convolutions involve different ways of accumulations, which brings obstacles to the design of the adder tree in MobileNet. A general-purpose adder tree saves more computing resources than designing adder trees separately for every type of convolution. Therefore, this paper designs a configurable adder tree compatible with the depthwise, pointwise and standard convolutions. As mentioned in Section 3.2, the parallelism on the channel dimension *PC* in this paper 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. The standard convolution needs 16-way 9-in addition and 1-way 16-in addition.

The basic unit of the configurable adder tree inspired by [26] is illustrated in Fig.5. The switching of functions is mainly realized through the multiplexer. Fig.5(a) shows the 2-way 9-in mode while Fig.5(b) shows the 1-way 16-in mode.

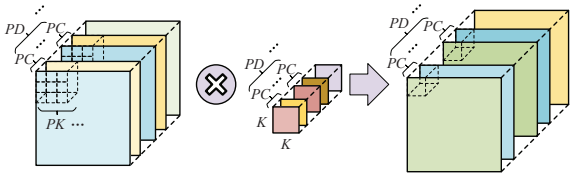
The configurable adder tree instantiates a total of 9 basic units, as Fig. 6 shows. Basic Unit 0~Basic Unit 7 are used to accomplish the accumulation of the depthwise and pointwise convolutions. Basic Unit 8 either works as Basic Unit 0~7



(a) Parallelism on standard convolution.

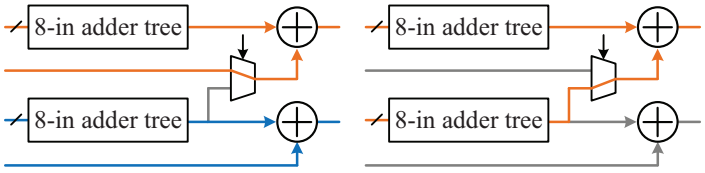


(b) Parallelism on pointwise convolution.



(c) Parallelism on depthwise convolution.

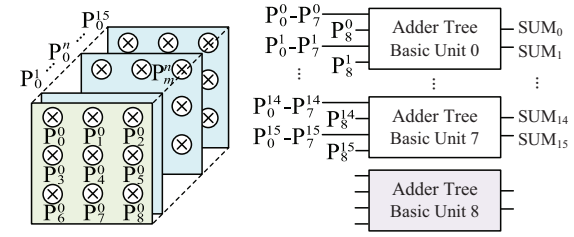
Fig. 4. Parallel strategies.



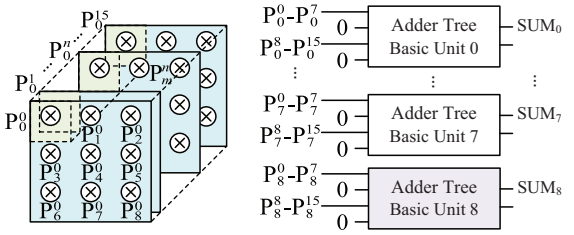
(a) 2-way 9-in mode (b) 1-way 16-in mode

Fig. 5. Basic unit of configurable adder tree.

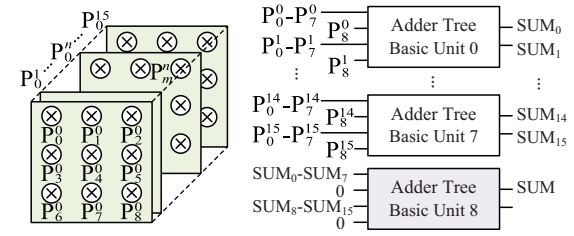
or sums the results on the channel dimension for the standard convolution. P_m^n is denoted as the m-th product in the n-th channel. As shown in Fig.6(a), the depthwise convolution only uses 8 basic units of the adder tree in the 2-way 9-in mode. Each basic unit processes the accumulation of 2 channels separately. For example, Basic Unit 0 accumulates the 9 products in the 0-th and the 1-st channel simultaneously. In the pointwise convolution, all the basic units adopt 1-way 16-in addition mode to sum along the channel dimension (Fig.6(b)). For instance, Basic Unit 0 accumulates P_0^0-P_0^{15} in



(a) Depthwise convolution.



(b) Pointwise convolution.



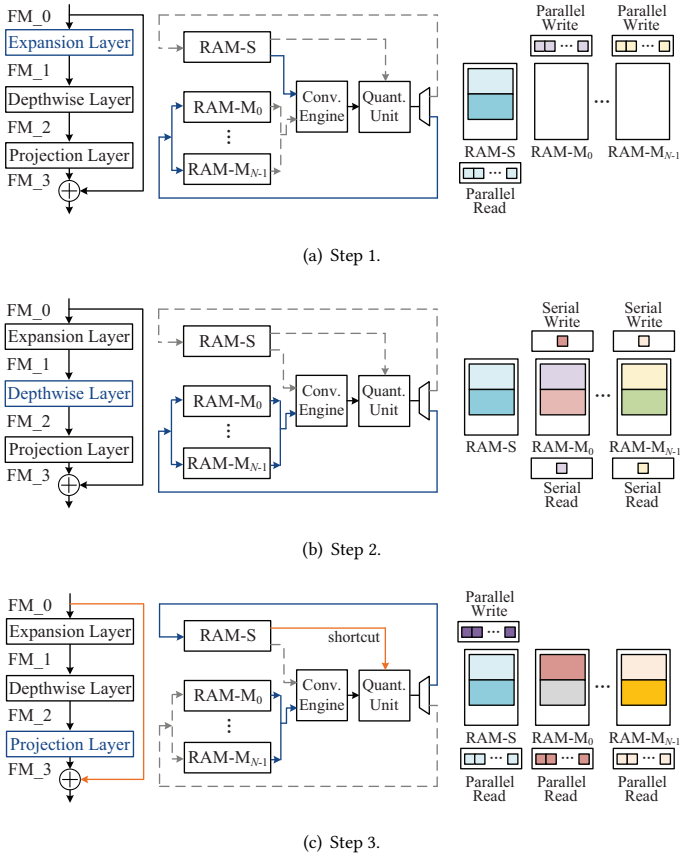
(c) Standard convolution.

Fig. 6. Configurable adder tree in different convolutions.

this case. As for the standard convolution, Basic Unit 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 (Fig.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 paper 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 serial read/write mode (PC pixels along the channel dimension) and a parallel read/write mode ($PV \times PC$ pixels). The hardware architecture and data flow are illustrated in Fig.7. Step 1 performs the operations of the expansion layer, which contains pointwise convolution (Fig.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 in RAM-Ms, cyclically in the order of



(a) Step 1.

(b) Step 2.

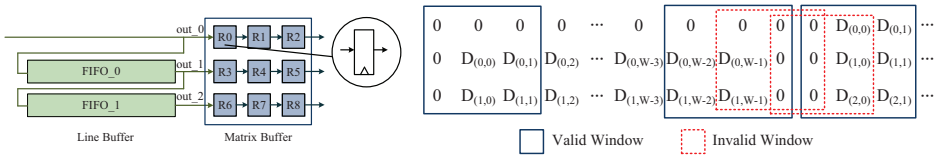
(c) Step 3.

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

RAM-M_0 ~ RAM-M_{N-1}. As shown in Fig.7(b), step 2 performs the depthwise convolution. In order to implement the *PD* parallel strategy, all RAM-Ms are strobed for serial output. The output feature map (FM_2) is overwritten to the position where the input feature map (FM_1) was read. As shown in Fig.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 parallelly 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 forms a closed loop. As discussed in [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 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

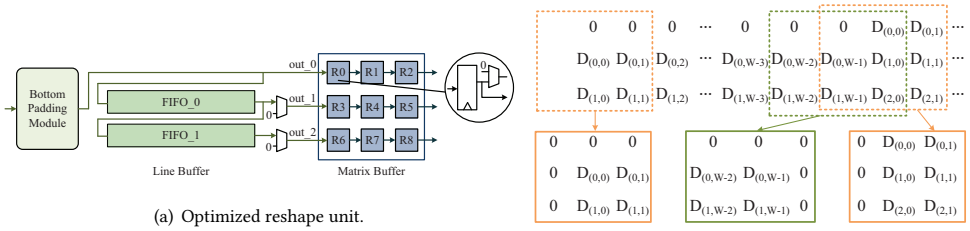
$$T_{\text{extra}} = \frac{H_{\text{i}} \times W_{\text{i}} \times \lceil C_{\text{i}}/PC \rceil \times B_{\text{data}}}{Bandwith_{\text{DMA}}} \tag{1}$$



(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.

Among them, B_{data} is the bit-width of the feature maps and $Bandwith_{\text{DMA}}$ is the bandwidth of DMA. It can be observed by (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 forms 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. [30] inserted during the data-loading process while [25] completed padding operations before the feature maps are reshaped into 3×3 matrices through a dedicated padding module.

However, the direct padding scheme has the efficiency issue. As shown in Fig.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. Fig.8(b) illustrates the data windows of the matrix buffer under the direct padding scheme. $\mathrm{D}_{(m,n)}$ is denoted as the pixel in the m-th row and n-th 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 Fig.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.

For the issue of invalid windows, this paper proposes an operation-separate padding scheme in which the padding is separated into different operations and realized in different parts of the reshape unit. Fig.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 enters the line buffer. As shown in Fig.9(b), an invalid window (green dotted frame) is generated when the data in the 0-th column enters the matrix buffer. The output of R0, R3, and R6 in the matrix buffer are set to zero in order to complete padding operations on the right side of the feature map (green solid frame). Similarly, the output of R2, R5, and R8 are set to zero when the data in the 1-st 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 paper eliminates the invalid windows during transitions of rows and reduces the preprocessing time before convolution. Take the $H_\mathrm{i} \times W_\mathrm{i}$ input feature map with padding number of 1 as an example. The direct padding scheme needs $(H_\mathrm{i}+2)(W_\mathrm{i}+2)$ 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

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

It can be observed from (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 1.45x 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 paper 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). 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. In order 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 Fig.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 $Z_{d/w/o}$ are the floating-point value, quantified 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 (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 :

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

where M is equal to $S_d S_w / S_o$ and q_{b_f} is defined in (6)

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

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

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

The affine quantization transformation of (4) becomes (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 Fig.2, the shortcut involves data transmission across layers. Let the cross-layer input feature map in the shortcut connection operation be r_s. The output of the last layer in the bottleneck structure plus r_s is represented as below

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

Similarly, applying the affine quantization to (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 (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 Fig.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 (11)). Then, if the shortcut exists, the upper part adds the lower part; otherwise 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_{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.

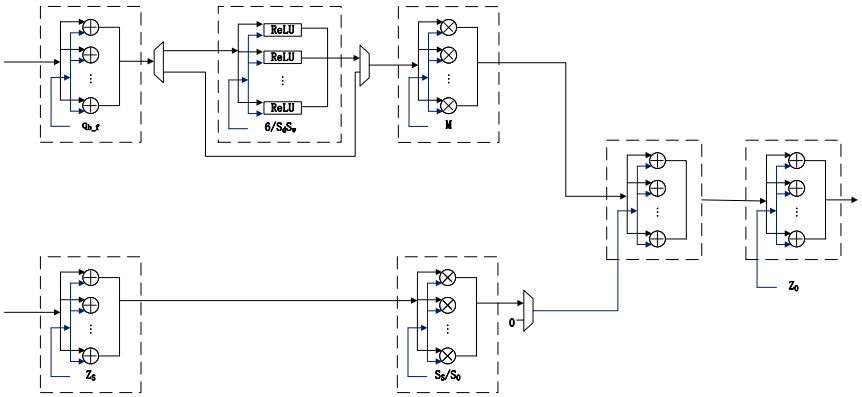


Fig. 10. Quantization Unit.

The quantization unit results in an 8-bit unsigned integer. The output is transferred to the off-chip memory or the on-chip input cache to continue the convolution operation as an input to the next layer.

5 HARDWARE-ASSISTED DATA PRUNING

The post-training pruning method is promoting from a practical perspective, because model re-training 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 on-line 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 Fig.11, which is integrated into the convolution engine. In Fig.11, we add a clock-gating unit to control the clock input of each multiplier in the convolution engine. The enable 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. In this way, the power consumption is reduced. Therefore, by setting different thresholds Th we fulfill the weight pruning with different sparsity.

Now, the question is how to determine Th for a given sparsity without large accuracy drop. The post-training method [35] uses a heuristic approach to select the layer-wise sparsity given a global sparsity constraint. We adopt the heuristic approach to tune the layer-wise Th_l with respect to model accuracy and power consumption. Specifically, during each iteration of the heuristic procedure, the model accuracy and the power consumption of the accelerator are measured. The measured accuracy and power will guide the update of Th_l. This procedure iterates until a trade-off between the model accuracy and the power consumption is made. This on-line (in-circuit) tuning is fast and achieves real performance directly.

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 configuration

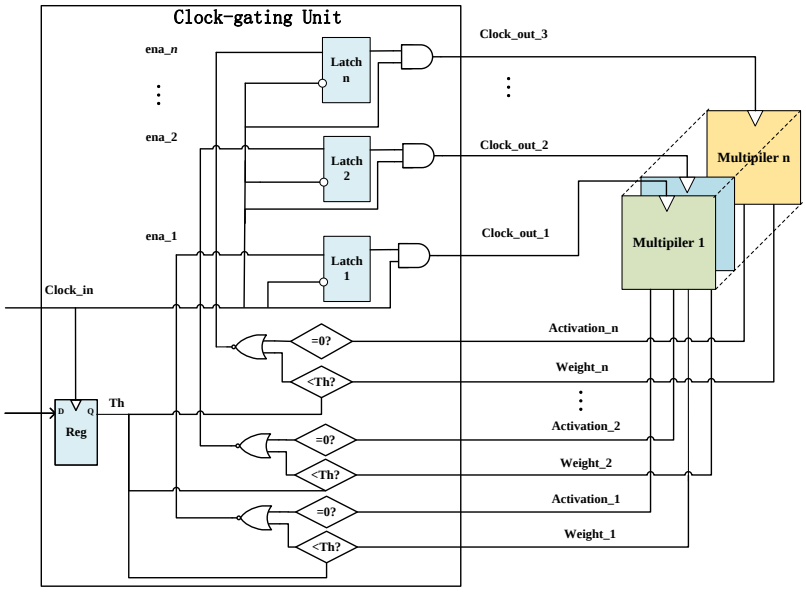


Fig. 11. Pruning Unit.

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 3-D input feature tensor and 4-D weights tensor need to be sliced, in order to be accommodated in on-chip buffers and support the parallel computations. As the described in Section 3, the proposed accelerator adopts five parallel strategies. For the standard convolution, the input feature map is sliced along height, width and depth to generate $K \times K \times PC$ blocks. The weights are sliced along depth and filter 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 3, the on-chip RAM-s and RAM-Ms are allocated to store the feature map of the first layer and the last layers, respectively.

Scheduling The main tasks of the proposed accelerator includes inputting the feature map blocks and weight blocks, performing convolution and outputting result blocks. Because the input, weight and output buffers use the ping-pong scheme, the tool schedules the tasks in a pipeline to reduce the processing time. The scheduling of the four tasks is shown in Fig.12. In addition, the operation flow for the bottleneck block and the shortcut connection is scheduled at this step.

After three steps, the tool can generate the configuration instructions, which will control the execution of the proposed accelerator. These instructions will transfer to the control unit as shown in Fig.3.

7 EXPERIMENT RESULTS

To evaluate the performance of the proposed accelerator, the typical DSCNN models (MobileNetV1, MobileNetV2) and the DNN models with bottleneck (ResNet-50, ResNet-152) are mapped onto the inference accelerator. The ImageNet dataset is used. The proposed accelerator is implemented in a Xilinx Virtex-7 XC7V690t FPGA. After placement and

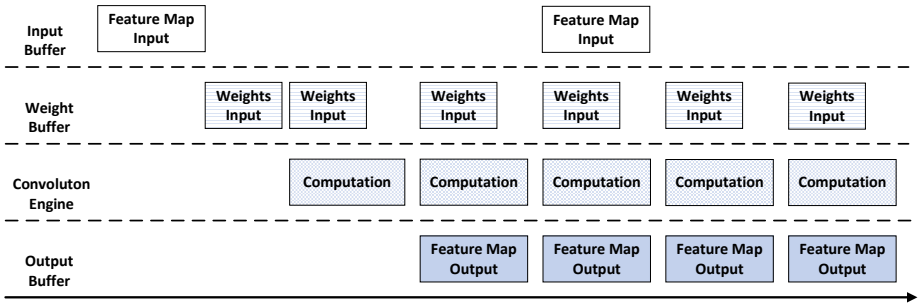


Fig. 12. Task pipeline of the accelerator.

Table 1. Resource usage of the proposed accelerator.

Resource	Utilization	Available	Percentage
LUT	308449	433200	71.20%
BRAM	941.5	1470	64.05%
DSP	2160	3600	60.00%
FF	278926	866400	32.19%

routing, the accelerator runs at 150 MHz clock frequency. The experimental results includes four parts. The first part evaluates the resource usage of the accelerator. The second part evaluates the performance of MobileNets on the accelerator. It is compared to the MobileNets 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 levels of parallelism inherent in the convolution operations. Given PK=9, PV=9, PC=16, PF=16 and PD=6, there are 2304 multipliers and 9 adder tree units in the accelerator, consuming 2160 DSP blocks embedded in the FPGA. 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 lookup tables (LUTs). As will be discussed below, the accelerator design achieves high resource utilization efficiency.

7.2 MobileNets acceleration

MobileNetV1 and MobileNetV2 models are mapped onto the accelerator. This part compares its performance and energy efficiency to several MobileNets implementations on embedded CPU, 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 [21]. 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. We implement MobileNets on the CPU and GPU platforms based on TensorFlow, using 32-bit single-precision floating-point data. The reported power consumption values are the board-level power measured by a power meter.

Table 2. Comparison of CPU, GPU and FPGA 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)	2400	150	2400	2600	1096	150
Processing Time per Frame (ms)	123	3.73	75	12.8	4.55	3.31
Frame per Second (FPS)	8.13	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)	-	11.24	-	29.88	46.67	11.35
Energy Efficiency (GOPS/W)	-	27.15	-	1.58	2.83	16.02

When MobileNetV1 is mapped onto the accelerator, it achieves 305.2 GOPS and 267.8 FPS. Compared with the Snapdragon 821 CPU, it achieves 32.9x speedup. The MobileNetV2 implemented on the accelerator reaches 181.8 GOPS and 302.3 FPS. Compared with the Snapdragon 821 CPU, it achieves 22.7x speedup. Compared with the i7-6700HQ CPU and GTX 960M GPU, the accelerator achieves 3.9x and 1.4x acceleration, respectively. In terms of energy efficiency, the MobileNetV2 accelerator reaches 10.1x and 5.7x 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 related FPGA accelerators of the MobileNet models. Table 3 shows the results. The software implementations [20] and [21] on CPU are used as the baseline. In terms of architecture, all accelerators except for [23] use the single-engine architecture. In terms of performance, the accelerators proposed in this paper shows the highest performance among the accelerators with the single-engine architecture. The performance advantages come from the efficient parallelization computation and padding operation. The performance of the accelerator in [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 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 an 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 among ResNets implementations on CPU and FPGA platforms. The results are shown in Table 4. The CPU is

Table 3. Comparison of FPGA-based MobileNets accelerators.

Design	Network	Platform	Frequency (MHz)	Frame per Second (FPS)	Performance (GOPS)	Precision	Top-1 Accuracy
[20]	MobileNetV1	CPU	2400	8.13		float32	70.6%
[21]	MobileNetV2	CPU	2400	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%

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(R) Core(TM) i7-11700	Intel Zynq 7Z045	Xilinx Virtex US+ VU9P	Xilinx Virtex-7 XC7V690t	Intel(R) Core(TM) i7-11700	Intel Stratix V GXA7	Xilinx VU9P	Xilinx Virtex-7 XC7V690t
DSP Usage	-	900	6005	2160	-	-	5632	2160
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
Frame per Second (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

Intel(R) Core(TM) 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 and 32.64 FPS, which is 1.3x higher than the CPU and is similar to [37]. The performance of the accelerator [38] is higher than the rest accelerators listed in the table, mainly due to higher parallelism using a large amount of DSPs. In terms of density, our accelerator shows 2.6× improvement over [38].

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

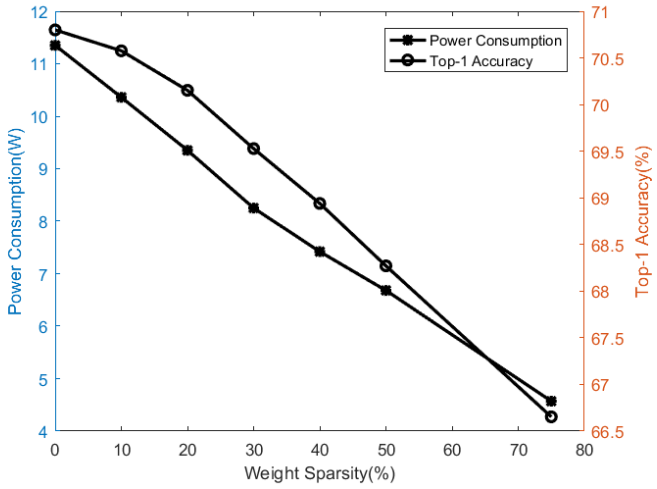


Fig. 13. Evaluation of the hardware-assisted data pruning function of the accelerator implementing MobileNetV2.

7.4 Hardware-assisted data pruning results

The proposed hardware-assisted pruning method allows on-line trade-off between model accuracy and power consumption. In this part, we use MobileNetV2 model to evaluate the method. Fig.13 shows the Top-1 accuracy and power consumption over different weight sparsity 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 W 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 on-line 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.

8 CONCLUSION

This paper designs an efficient inference accelerator for CNNs with DSCs and bottleneck structures on FPGA. The accelerator features four structural points: 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 10x and 6x energy efficiency improvement over the CPU and GPU implementation, and 302.3 FPS and 181.8 GOPS performance which is the best among several existing single-engine accelerators on FPGAs. The proposed hardware-assisted pruning can effectively trade model accuracy for power consumption on-line.

In the future, the proposed accelerator is expected to support more models such as MobileNetV3 [32] and EfficientNet [33] by adding 5×5 bottleneck, h-swish, and swish activation. In addition, sparse data flow will be used in the accelerator to achieve higher resource utilization.

9 ACKNOWLEDGMENT

This research was supported in part by the National Natural Science Foundation of China under Grant U21B2031.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 779–788.

[3] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 7263–7271.

[4] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[5] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing Deep Convolutional Networks using Vector Quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[6] F. Li, B. Zhang, and B. Liu, "Ternary Weight Networks," *arXiv preprint arXiv:1605.04711*, 2016.

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

[8] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv preprint arXiv:1510.00149*, 2015.

[9] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training," *arXiv preprint arXiv:1712.01887*, 2017.

[10] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning Efficient Convolutional Networks through Network Slimming," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 2736–2744.

[11] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: Alexnet-Level Accuracy with 50x Fewer Parameters and <0.5 MB Model Size," *arXiv preprint arXiv:1602.07360*, 2016.

[12] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6848–6856.

[13] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 116–131.

[14] S. Yan et al., "An FPGA-based MobileNet Accelerator Considering Network Structure Characteristics," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 17-23.

[15] S. -F. Hsiao and B. -C. Tsai, "Efficient Computation of Depthwise Separable Convolution in MoblieNet Deep Neural Network Models," in *2021 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW)*, 2021.

[16] B. Li et al., "Dynamic Dataflow Scheduling and Computation Mapping Techniques for Efficient Depthwise Separable Convolution Acceleration," in *IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 68, no. 8*, 2021, pp. 3279-3292.

[17] L. Xiaolin, R. C. Panicker, B. Cardiff and D. John, "Multistage Pruning of CNN Based ECG Classifiers for Edge Devices," in *2021 43rd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, 2021, pp. 1965-1968.

[18] S. Sarkar, M. Agarwalla, S. Agarwal and M. P. Sarma, "An Incremental Pruning Strategy for Fast Training of CNN Models," in *2020 International Conference on Computational Performance Evaluation (ComPE)*, 2020, pp. 371-375.

[19] S. Kim and H. Kim, "Linear Domain-aware Log-scale Post-training Quantization," in *2021 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia), 2021*, pp. 1-3.

[20] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.

[21] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[22] G. Wei, Y. Hou, Z. Zhao, Q. Cui, G. Deng and X. Tao, "Demo: FPGA-Cloud Architecture For CNN," in *2018 24th Asia-Pacific Conference on Communications (APCC)*, 2018, pp. 7-8.

[23] D. Wu, Y. Zhang, X. Jia, L. Tian, T. Li, L. Sui, D. Xie, and Y. Shan, "A High-Performance CNN Processor Based on FPGA for MobileNets," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 136–143.

[24] J. Su, J. Faraone, J. Liu, Y. Zhao, D. B. Thomas, P. H. Leong, and P. Y. Cheung, "Redundancy-Reduced MobileNet Acceleration on Reconfigurable Logic for ImageNet Classification," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2018, pp. 16–28.

[25] H. Fan, S. Liu, M. Ferianc, H.-C. Ng, Z. Que, S. Liu, X. Niu, and W. Luk, "A Real-Time Object Detection Accelerator with Compressed SSDLite on FPGA," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 14–21.

[26] L. Bai, Y. Zhao, and X. Huang, "A CNN Accelerator on FPGA using Depthwise Separable Convolution," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 10, pp. 1415–1419, 2018.

[27] B. Liu, D. Zou, L. Feng, S. Feng, P. Fu, and J. Li, "An FPGA-Based CNN Accelerator Integrating Depthwise Separable Convolution," *Electronics*, vol. 8, no. 3, p. 281, 2019.

[28] R. Zhao, X. Niu, and W. Luk, "Automatic Optimising CNN with Depthwise Separable Convolution on FPGA: (abstact only)," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 285–285.

[29] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[30] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "VTA: An Open Hardware-Software Stack for Deep Learning," *arXiv preprint arXiv:1807.04188*, 2018.

[31] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[32] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for MobileNetV3," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 1314–1324.

[33] M. Tan and Q. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6105–6114.

[34] T. Gale, E. Elsen, S. Hooker," The State of Sparsity in Deep Neural Networks." *arXiv preprint arXiv:1902.09574*, 2019

[35] I. Lazarevich, A. Kozlov and N. Malinin, "Post-training deep neural network pruning via layer-wise calibration," in *2021 IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*, 2021, pp. 798-805.

[36] M. S. Abdelfattah et al., "DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 411-4117.

[37] Y. Ma, Y. Cao, S. Vrudhula and J. -s. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1-8..

[38] R. Kuramochi and H. Nakahara, "An FPGA-Based Low-Latency Accelerator for Randomly Wired Neural Networks," 2020 30th International Conference on Field-Programmable Logic and Applications (FPL), 2020, pp. 298-303.

[39] X. Wei, Y. Liang and J. Cong, "Overcoming Data Transfer Bottlenecks in FPGA-based DNN Accelerators via Layer Conscious Memory Management," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1-6.

[40] Y. Liang et al., "Evaluating fast algorithms for convolutional neural networks on fpgas," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*,2020, vol. 39, no. 4, pp. 857–870.