2018 International Conference on Field-Programmable Technology (FPT)

A Real-Time Object Detection Accelerator with Compressed SSDLite on FPGA

Hongxiang Fan*§†, Shuanglong Liu§†, Martin Ferianc†, Ho-Cheung Ng†, Zhiqiang Que†, Shen Liu†, Xinyu Niu‡, Wayne Luk†

† Dept. of Computing, School of Engineering, Imperial College London, UK

{*h.fan17, s.liu13, m.ferianc15, h.ng16, z.que, w.luk*}*@imperial.ac.uk*

‡ Corerain Technologies Ltd., Shenzhen, China, *xinyu.niu@corerain.com*

***Abstract*—Convolutional neural network (CNN)-based object detection has been widely employed in various applications such as autonomous driving and intelligent video surveillance. However, the computational complexity of conventional convolution hinders its application in embedded systems. Recently, a mobile-friendly CNN model *SSDLite-MobileNetV2* (*SSDLiteM2*) has been proposed for object detection. This model consists of a novel layer called bottleneck residual block (BRB). Although *SSDLiteM2* contains far fewer parameters and computations than conventional CNN models, its performance on embedded devices still cannot meet the requirements of real-time processing. This paper proposes a novel FPGA-based architecture for *SSDLiteM2* in combination with hardware optimizations including fused BRB, processing element (PE) sharing and load-balanced channel pruning. Moreover, a novel quantization scheme called partial quantization has been developed, which partially quantizes *SSDLiteM2* to 8 bits with only 1.8% accuracy loss. Experiments show that the proposed design on a Xilinx ZC706 device can achieve up to 65 frames per second with 20.3 mean average precision on the COCO dataset.**

I. Introduction

Over the past few years, object detection has been employed in a wide range of applications such as autonomous driving, face detection and traffic monitoring. Whereas, traditional methods for object detection such as sliding window and region-based algorithms suffer from low accuracy [1]. The development of deep learning has led to significant advances in object detection tasks. Various convolutional neural networks (CNNs) such as *SSD* [2], *Faster R-CNN* [3], and *YOLO* [4] have been proposed for object detection with high accuracy. The success of these networks sparks great research interests in deploying them on embedded systems such as aerial drones and security cameras. However, the accuracy improvement of deep learning-based algorithms comes at a cost: the algorithm complexity imposes large overhead on the speed of these networks, which limits their deployments on embedded systems. For example, *Faster R-CNN* can only achieve single-digit frame rates even on a high-end graphics processing unit (GPU) [5].

Recently, a lightweight building layer called bottleneck residual block (BRB) [6] has been proposed, which is composed of point-wise [7] and depth-wise convolutions [8]. Based on BRB, the *SSDLite-MobileNetV2* (*SSDLiteM2*) can achieve nearly 8.3 times compression rate compared with the original *SSD* model while maintaining the same level of accuracy. Although the model size of *SSDLiteM2* is smaller than that of other high-accuracy models, it can only achieve 5 frames per second (fps) on a high-end embedded CPU [6], which still cannot meet the requirement of real-time processing. Therefore, there is a great demand for the hardware acceleration of *SSDLiteM2* for object detection.

Various hardware platforms including field programmable gate array (FPGA), application specific integrated circuit (ASIC) and graphics processing unit (GPU) can be used to accelerate deep learning-based object detection. Among these hardware platforms, FPGAs are gaining popularity because of their better reconfigurability and shorter turn-around time than ASICs and higher energy efficiency than GPUs [9] [10].

However, there are several challenges when accelerating *SSDLiteM2* on FPGA for object detection:

1) To support point-wise and depth-wise convolutions, one approach [11] deploys different computational engines for different convolutions separately, but it does not achieve high hardware efficiency.
2) To achieve real-time response, conventional compression techniques for lightweight models often sacrifice the accuracy and thus cannot meet the applications with high accuracy requirement.

To address the above challenges, we introduce a novel FPGA-based architecture together with the processing element (PE) sharing optimization to improve the hardware efficiency. Based on the roofline model of *SSDLiteM2*, a fused BRB is proposed to improve overall performance by caching all intermediate results on the on-chip memory. Furthermore, a load-balanced channel pruning is presented, which not only compresses the *SSDLiteM2* model but also improves the hardware efficiency. Several software optimizations including partial quantization and bias folding are proposed to decrease the amount of computation and parameters while maintaining the same level of accuracy. The optimized *SSDLiteM2* called *C-SSDLiteM2* is implemented on the proposed FPGA-based architecture, which achieves real-time performance for object detection task.

The main contributions of this work are the following:

- A novel FPGA-based architecture for *SSDLiteM2*, which

* The first author is financially supported by China Scholarship Council.
§ Corresponding author.

supports multiple types of convolutions with different kernel sizes (Section III).

- Several innovative hardware optimizations such as fused BRB, PE sharing and load-balanced channel pruning, which improve the overall performance as well as the hardware efficiency (Section III-B).
- Complementary software optimizations including partial quantization and bias folding, which reduce not only the computational complexity but also the amount of parameters (Section III-C).

II. Background

A. Depth-wise Convolution and Bottleneck Residual Block

Depth-wise convolution is a lightweight building layer in modern CNNs. Figure 1 illustrates standard convolution versus depth-wise convolution. Compared to the standard convolution, depth-wise convolution only applies one filter on each channel, which significantly decreases the amount of computation and parameters.

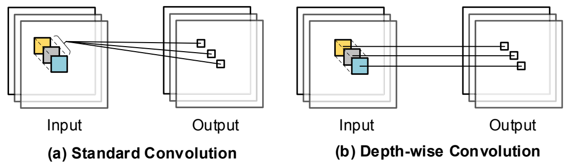


Fig. 1: The standard convolution and depth-wise convolution

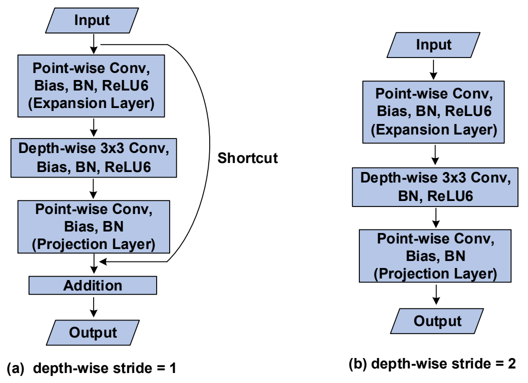

Input
Point-wise Conv, Bias, BN, ReLU6 (Expansion Layer)
Depth-wise 3x3 Conv, Bias, BN, ReLU6
Point-wise Conv, Bias, BN (Projection Layer)
Addition
Output
Shortcut
(a) depth-wise stride = 1
Input
Point-wise Conv, Bias, BN, ReLU6 (Expansion Layer)
Depth-wise 3x3 Conv, BN, ReLU6
Point-wise Conv, Bias, BN (Projection Layer)
Output
(b) depth-wise stride = 2

Fig. 2: The structure of bottleneck residual block

A bottleneck residual block (BRB) is mainly composed of three convolutional layers, namely expansion convolution, depth-wise convolution and projection convolution. The structure of BRB is illustrated in Figure 2, where the kernel size of depth-wise convolution is 3×3. The expansion convolution and projection convolution are the standard convolutions with the kernel size of 1×1 (point-wise convolution). Table I summaries the parameters utilized in this paper. There is a shortcut addition when the stride of depth-wise convolution is 1, which is utilized for residual learning [12]. Each convolution in BRB is followed by a batch normalization (BN) layer. The rectified linear unit with threshold being 6 (ReLU6) is only applied after the expansion convolution and the depth-wise convolution. Table II summaries the input and output tensors of each convolution, where the expansion factor t is utilized to expand the internal dimension in BRB.

TABLE I: Parameters used in the FPGA implementation of *SSDLiteM2* for object detection acceleration.

Parameter	Description
H	The height of input feature map
W	The width of input feature map
K_{dw}	The kernel size of depth-wise convolution
N_c	The number of channels
N_f	The number of filters
t	The expansion factor

TABLE II: The input and output tensors within BRB.

Input Tensor	Convolution	Output Tensor
$H \times W \times N_c$	Projection Convolution	$H \times W \times N_c \times t$
$H \times W \times N_c$	Depth-wise Convolution	$H \times W \times N_c \times t$
$H \times W \times N_c \times t$	Expansion Convolution	$H \times W \times N_f$

B. SSDLite-MobileNetV2

An object detection problem can be separated into two tasks: one is predicting the bounding boxes for the localization, the other is the associated object classification for the proposed bounding boxes. The *SSD* architecture [2] is a popular CNN framework for object detection, which consists of two components, feature extractor and bounding box predictor. The feature extractor, which is also called base network, is usually a truncated classification network such as *VGG-16* [13], followed by a set of auxiliary convolutional layers which enable features extraction at multiple scales and decrease the input size of each subsequent layer. The bounding box predictor is a group of small convolutional filters used to predict category scores and box offsets for a fixed set of default bounding boxes (anchor boxes [14]).

The *SSDLite* is a mobile-friendly variant of *SSD*, where the regular convolutions in bounding box predictor are replaced by the depth-wise convolutions. Based on the *SSDLite* framework, the *SSDLite-MobileNetV2* (*SSDLiteM2*) is proposed in [6], which utilizes *MobileNetV2* as the base network. Figure 3 illustrates the network structure of *SSDLiteM2*.

C. Related Work

Before the advent of deep learning, the state-of-the-art approach for object detection is based on the histogram of oriented gradients (HOG) algorithm [15]. Several FPGA-based accelerators [16], [17] have been proposed for HOG-based object detection. However, their low accuracy hinders their deployment in real-life applications [18].

As deep learning evolved, several CNN models have been proposed for object detection with high accuracy. *Faster R-CNN* [3] is one of the object detection models, which extends

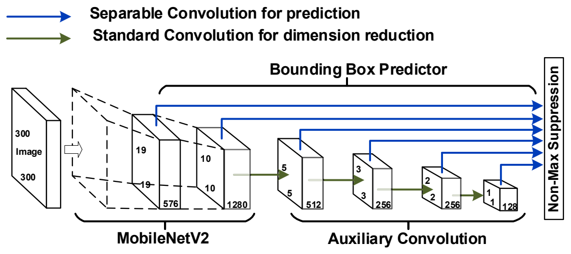


Fig. 3: The network architecture of *MobileNetV2-SSDLite*

Fast RCNN [14] with Region Proposal Network (RPN). Similarly, the *SSD* [2] utilizes a set of pre-defined boxes to perform the localization and classification at multiple scales. *YOLO* [4] re-frames object detection as a regression problem, where only a single convolutional network is utilized to simultaneously predict multiple bounding boxes and class probabilities. Various FPGA-based designs have been proposed to accelerate these models. Zhao et al. [1] accelerate the *YOLO* and *Faster R-CNN* on FPGA with several hardware optimizations. Ma et al. [19] thoroughly explore the design space of CNN-based algorithms on FPGA and propose a novel hardware architecture for CNN models. However, the speed of these designs still cannot meet the requirement of real-time processing. Although Nakahara et al. [20] propose an FPGA-based binarized neural networks (BNNs) accelerator which can achieve 40.81 frames per second in the object detection task, its accuracy is not evaluated on a large dataset. Therefore, there is a great demand for a hardware accelerator for object detection with real-time processing capability and high accuracy.

III. FPGA Accelerator Design

In this section, the overview of the proposed design is first presented. Then several hardware and software optimization techniques are proposed. Finally, the hardware architecture is illustrated in detail.

A. Design Overview

The overview of the proposed FPGA-based accelerator is illustrated in Figure 4, which is mainly composed of the computational engine, data stream controller, off-chip and on-chip memory. To improve the scalability, the single processing engine architecture [21] is used in our work, where the computational engine is designed to run one layer or block at one time, and the whole network is processed by repeatedly running the computational engine. Each processing element (PE) consists of several multipliers and a pipelined adder tree, which is used to perform the multiply-accumulation in convolutional layers. The data stream controller, which consists of several buffers, is dedicated to the data communication between the computational engine and on-chip memory.

B. Hardware Optimization

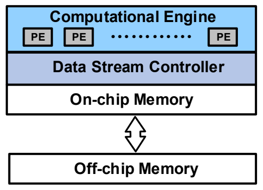


Fig. 4: Overview of FPGA-based accelerator

1) Fused BRB: As mentioned in Section II, the BRB is mainly composed of the projection convolution, depth-wise convolution and expansion convolution. To estimate the performance limitation of these three convolutional layers, the roofline model is used in this paper. The roofline model mainly consists of one roofline curve which characterizes the performance limitation, and several vertical lines that represent an algorithm intensity, where the roofline curve is given by the computational resources and off-chip bandwidth of the specific hardware, and a vertical line is determined by the number of operations in a algorithm. The interaction between the roofline curve and the line of algorithm intensity gives the theoretical peak performance point, which is either compute-bound or memory-bound.

Based on the methodology proposed in [22], the roofline model for the Xilinx Zynq ZC706 with 100 MHz clock frequency and 1.2 GB/s of DRAM bandwidth is developed [1] and shown in Figure 5. As can be seen from the figure, all the expansion, projection and depth-wise convolutions are memory-bound, and hence the overall performance of BRB will be also bounded by the limited memory bandwidth. To improve the overall performance, the fused BRB is proposed in our work. All the intermediate results of the BRB are cached in on-chip memory, which eliminates the need for data transfer between on-chip and off-chip memory for the BRB. After the fused BRB optimization, the BRB becomes compute-bound, which is given by the red line in Figure 5.

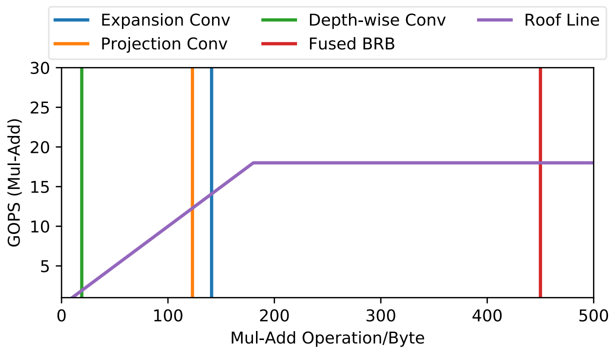


Fig. 5: Roofline model of *SSDLiteM2*

[1]Assuming that the operation refers to 32-bit multiply-accumulation, only DSP resource is utilized to perform the calculation and each operation costs 5 DSPs.

One issue comes with the fused BRB is that the on-chip memory is not enough to cache all the intermediate results of the BRB. To address this issue, the pixel blocking technique [23] is used in this paper, which divides the input feature maps into several equal blocks and caches each set of blocks on chip. The block sizes of each layer mainly depend on the network setting and the available on-chip memory resources.

2) PE Sharing: Since the design is based on the single processing engine architecture, the computational engine is only designed to run one BRB at one time. To fully pipeline the design, the computational resource allocation for the three convolution layers should be proportional to their amount of computation. However, the computation and corresponding proportions of the three convolutional layers vary in different BRBs, it is impossible to fully pipeline the design while running the whole network. To address this design challenge, the PE sharing strategy is proposed in this paper, which eliminates the need of resource allocation by sharing the PEs of depth-wise convolution with the point-wise convolution.

Since the kernel sizes of depth-wise and point-wise convolutions are different, the PEs designed for depth-wise convolution cannot be directly used for point-wise convolution. To address this issue, loop unrolling and loop interchanging are considered in our work. The pseudo code of the original point-wise convolution is illustrated in Algorithm 1. Loop unrolling is first applied to the loop with the variable $channels$, which splits the accumulation into $\lceil \frac{N_c}{K_{dw} \times K_{dw}} \rceil$ groups of $K_{dw} \times K_{dw}$, where the $\lceil \rceil$ is the ceiling function. Then, the unrolled loop with parameters $K_{dw} \times K_{dw}$ is interchanged as the innermost loop, which makes point-wise convolution possess the same computational pattern as the depth-wise convolution. The optimized point-wise convolution is illustrated in Algorithm 2. After the loop unrolling and interchanging, the PEs designed for $K_{dw} \times K_{dw}$ depth-wise convolution can be reused to perform the multiply-accumulation operations in point-wise convolution, which eliminates the need of resource allocation for these two different convolutions.

Algorithm 1 Original Point-wise Convolution

```
1: for $filters = 0$ to $N_f$ do
2:     for $channels = 0$ to $N_c$ do
3:         for $height = 0$ to $H$ do
4:             for $width = 0$ to $W$ do
5:                 output_fm[$filters$][$height$][$width$]+=
6:                 coef[$filters$][$channels$] $\times$
7:                 input_fm[$channels$][$height$][$width$];
```

Algorithm 2 Optimized Point-wise Convolution

```
1: for $filters = 0$ to $N_f$ do
2:     for $channels = 0$ to $\lceil \frac{N_c}{3*3} \rceil$ do          ▷ Loop Unrolling
3:         for $height = 0$ to $H$ do
4:             for $width = 0$ to $W$ do
5:                 for $i = 0$ to $3^2$ do                          ▷ Loop Interchange
6:                     output_fm[$filters$][$height$][$width$]+=
7:                     coef[$filters$][$channels + i$] $\times$
8:                     input_fm[$channels + i$][$height$][$width$];
```

3) Load-balanced channel pruning: Although the PE sharing can improve the resource efficiency, it may also cause sparse matrix multiplication when the channel number of point-wise convolution is not divisible by the kernel size of depth-wise convolution. Assume that the computational engine is composed of two PEs and each PE is designed for depth-wise convolution with the kernel size being 3×3. Figure 6a illustrates the case of sparse matrix multiplication when the channel number of point-wise convolution is 11. Since each PE can only process 9 channels in the input data, it requires 2 PEs to perform the point-wise convolution with 11 channels, which results in the unused computational resources in the second PE.

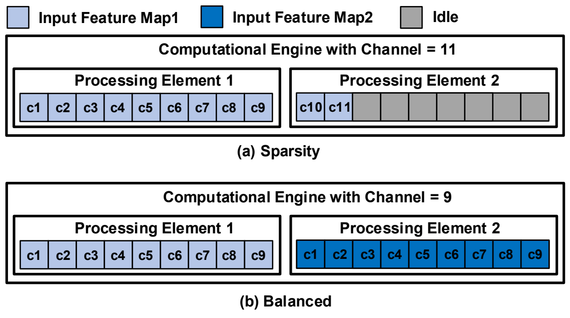

Input Feature Map1
Input Feature Map2
Idle
Computational Engine with Channel = 11
Processing Element 1
c1 c2 c3 c4 c5 c6 c7 c8 c9
Processing Element 2
c10 c11
(a) Sparsity
Computational Engine with Channel = 9
Processing Element 1
c1 c2 c3 c4 c5 c6 c7 c8 c9
Processing Element 2
c1 c2 c3 c4 c5 c6 c7 c8 c9
(b) Balanced

Fig. 6: Computational engine with and without sparsity

To avoid sparse matrix multiplication, the load-balance channel pruning is proposed. Channel pruning is the compression technique proposed in [24]. In this paper, we extend the channel pruning to load-balanced channel pruning, which not only aims at model compression but also considers the hardware efficiency. Instead of decreasing the channel number randomly, the load-balance channel pruning prunes the channel number of point-wise convolution to the number of $K_{dw} \times K_{dw}$. Because the pruned channel number is divisible by $K_{dw} \times K_{dw}$, every PE will be fully occupied. Figure 6b presents an example of balanced computational engine. Since the channel number is pruned to 9, each input feature map in point-wise convolution only requires one PE. In this case, the second PE can be fully utilized to process other input feature maps.

The benefits of load-balance channel pruning are twofold: Firstly, the sparse matrix multiplication can be avoided when the PEs of depth-wise convolution are used to perform the point-wise convolution. Secondly, the amount of parameters and calculation can be further reduced during the pruning process.

C. Software Optimization

1) Partial Quantization:
One of issues in the conventional quantization is that the accuracy loss varies in different models. In particular, our experiment shows that the accuracy drops significantly when

the quantization is applied to the light-weight models such as *SSDLiteM2*. To address this problem, partial quantization is proposed in our work, which partially quantizes the *SSDLiteM2* with negligible accuracy loss.

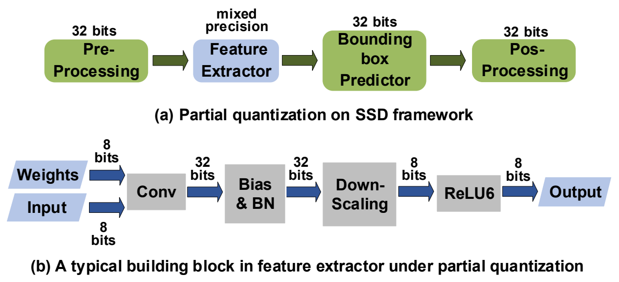


Fig. 7: Partial quantization

An SSD-based object detection model mainly consists of four components, namely, the pre-processing, the feature extractor, the bounding box predictor and the post-processing. To maintain the accuracy, the proposed partial quantization preserves the computation of the pre-processing, the bounding box predictor and the post-processing with 32-bit, which is illustrated in Figure 7(a). Note this will not introduce large overhead since nearly 85% of computation lies in the feature extractor. Furthermore, in the feature extractor with mixed precision, all the layers, except for the batch normalization and bias layers, are quantized to 8 bits according to the linear quantization scheme in [25]. A typical building block in feature extractor after partial quantization scheme is illustrated in Figure 7(b). Note that the precision in batch normalization and bias layers is 32bits to keep the accuracy.

2) Bias Folding:
Under the partial quantization scheme, the bitwidths of both bias and batch normalization layers are 32 bits. To decrease the amount of parameters and computation, the bias layer can be "folded" into the batch normalization layer.

During the inference stage, the batch normalization can be expressed as:

$$O_{bn} = \frac{O_{bias} - E}{\sqrt{V - \epsilon}}, \tag{1}$$

where the E, V, and ϵ are population statistics and O_{bias} is the result of the bias layers. Since E, V, and ϵ are constant values during the inference stage, equation (1) can be simplified as:

$$O_{bn} = n \times O_{bias} + m, \tag{2}$$

where $n = \frac{1}{\sqrt{V-\epsilon}}$ and $m = -\frac{E}{\sqrt{V-\epsilon}}$. By expanding the O_{bias}, the equation (2) can be formulated as:

$$O_{bn} = n \times (O_{conv} + bias) + m, \tag{3}$$

which becomes:

$$O_{bn} = n \times O_{conv} + l, \tag{4}$$

where O_{conv} is the output of convolutional layer and l is defined as:

$$l = n \times bias + m$$

In equation (4), both n and l are calculated before the inference, and cached in the on-chip memory. Therefore, only one multiplier and one adder are required for batch normalization and bias layers.

D. Hardware Design

1) Hardware Architecture: Based on the hardware and software optimization techniques presented above, the hardware architecture is shown in Figure 8, which is composed of the input cache, coefficient (Coef) buffer, computational engine, ReLU6 and reshape modules.

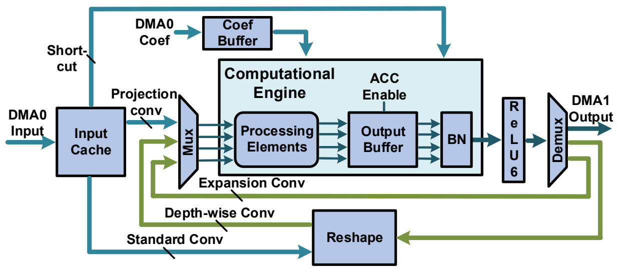
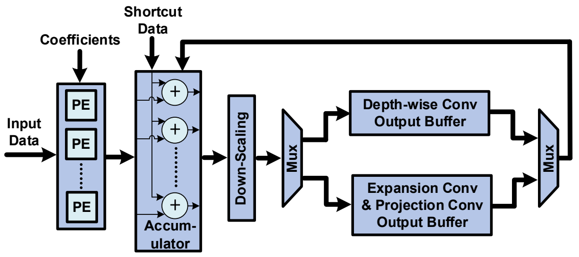


Fig. 8: Hardware Architecture

The processing of BRB starts with the projection convolution. The input feature maps of projection convolution are first stored in the input cache module for the data reuse. Then, the cached data flows into the computational engine to perform the projection convolution, bias and batch normalization sequentially. Note that both the bias and batch normalization have been integrated into the BN module due to the bias folding. Based on the fused BRB optimization, the results of projection convolution are cached in the output buffer module for the use of the next convolutional layer. After the ReLU6 and reshape modules, the cached results stream into the same computational engine again to perform the depth-wise convolution according to the PE sharing optimization. Since the depth-wise convolution is a channel-wise operation, the accumulator in the output buffer module is disabled at this stage. While processing the expansion convolution, the outputs of depth-wise convolution are fed back into the computational engine without going through the reshape module. Note that the ReLU6 is disabled in expansion convolution. In the last stage, the shortcut addition is enabled when the stride of depth-wise convolution is one, where the input data of projection convolution and the results of expansion convolution are added together by the accumulator in the output buffer module.

Note that the proposed hardware architecture also supports the standard convolution with the kernel size being 3×3. Similarly with the processing of BRB, the input feature maps are stored in the input cache module for the data reuse. Once the computation starts, the cached data will stream into the reshape module directly and then flow into the computational engine. The accumulator in the output buffer module is enabled for standard convolution. After the ReLU6 module, the results are transferred back to the off-chip memory.

2) Computational Engine: The computational engine is composed of PEs, accumulator, down-scaling and output buffer modules, which is illustrated in Figure 8.



Shortcut Data
Coefficients
Input Data
PE
PE
PE
Accum-ulator
Down-Scaling
Mux
Depth-wise Conv Output Buffer
Expansion Conv & Projection Conv Output Buffer
Mux

Fig. 9: Computational engine

Processing Elements — Each PE consists of $K_{dw} \times K_{dw}$ multipliers and a pipelined adder tree with $\log K_{dw}^2$ levels, where the bit-widths of input and output are 8-bit and 32-bit respectively. The number of PEs represents the parallelization level of the computational engine.

Accumulator and Down-scaling — The accumulator is composed of a group of adders, which will be enabled when *i)* the channel accumulation in standard convolution and *ii)* shortcut addition in BRB. The down-scaling module is used to map results from 32 bits to 8 bits after the accumulator.

Output buffer — There are only two buffers deployed on the output buffer module for three convolutions, where the output buffer of the projection convolution will be reused by the expansion convolution. The multiplexers (Mux-s) are used to control the datapath for different convolutions.

Reshape — The reshape module consists of a padding module and different types of buffers. The padding module is used to insert zeros into input feature maps. The line buffer takes the padded input data one by one, and produces the data line by line. Then, the matrix buffer receives the data from the line buffer and outputs $K \times K$ pixel matrix to the computation engine.

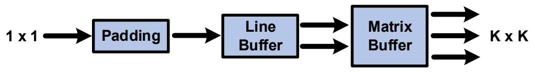


Fig. 10: Reshape module

IV. Experimental Results and Discussion

The proposed design is implemented on a Xilinx ZC706 platform which consists of a Kintex-7 FPGA and dual Arm Cortex-A9 processor. 1 GB DDR3 RAM is installed on the platform as an off-chip memory. The FPGA design is clocked at 100 MHz while the ARM processor runs at 1 GHz. Vivado 2016.2 is used for synthesis and implementation. The COCO dataset [26] which includes 2.5 million labeled instances of 91 object types in 328k images is used in the following experiments.

A. Implementation Detail

Figure 11 presents the implementation detail of the CPU+FPGA heterogeneous design. There are mainly three components in the system: FPGA design on the Processing Logic (PL), ARM CPU on the Processing System (PS), and DDR3 on the external memory. Since the design is based on the single processing engine architecture, various BRB layers with different parameters are repeatedly executed on the same FPGA design. The configuration parameters of each BRB are specified by PS side through the APB bus before the processing. Due to the limited on-chip memory, the results of each BRB will be cached in DDR3, and the DMA is utilized to transfer data and coefficients from DDR to PL.

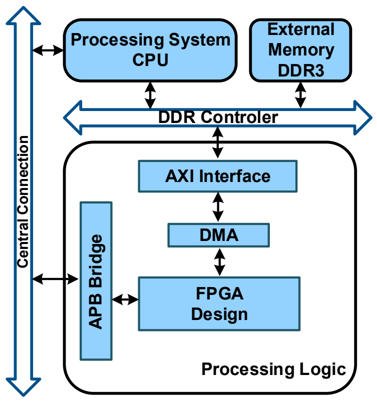


Fig. 11: System overview

Because the bounding box predictors are 32 bits under the partial quantization and its computation only accounts for less than 10% of the total amount of calculation, we put it on the PS side to overlap it with the feature extractor, which is illustrated in Figure 12.

B. Model Size and Accuracy

Two experiments are conducted in this section: the first experiment is to showcase the improvement of partial quantization, the second one presents the accuracy and model size of the fully optimized model that is utilized in our FPGA design.

TABLE III: The model size and accuracy of *SSDLiteM2* under different quantization methods.

SSDLiteM2	Accuracy (mAP)	Parameters	Compression Rate
Original	22.1	76.8Mb	1
Fully Quantized	11.8	19.2Mb	4
Partial Quantized	20.5	25.5Mb	3.01

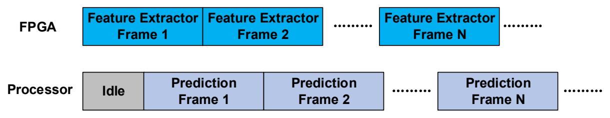

FPGA
Feature Extractor Frame 1
Feature Extractor Frame 2
Feature Extractor Frame N
Processor
Idle
Prediction Frame 1
Prediction Frame 2
Prediction Frame N

Fig. 12: Overlap of the bounding boxes predictor

TABLE IV: The network detail of *C-SSDLiteM2*

Component	Input	Operation	t	N_f	n	s	Params
Feature Extractor	$224^2 \times 3$	Conv 3×3	-	27	1	2	0.006 Mb
	$112^2 \times 27$	BRB	1	16	1	1	0.011 Mb
	$112^2 \times 16$	BRB	6	18	2	2	0.068 Mb
	$56^2 \times 18$	BRB	6	27	3	2	0.20 Mb
	$28^2 \times 27$	BRB	6	63	4	1	1.29 Mb
	$28^2 \times 63$	BRB	6	90	3	2	2.02Mb
	$14^2 \times 90$	BRB	6	144	3	2	4.91 Mb
	$7^2 \times 144$	BRB	6	288	1	1	2.9 Mb
	$7^2 \times 288$	Conv 1×1	-	1080	1	1	2.55 Mb
Bounding Box Predictor	-	-	-	-	-	-	8.4Mb
Total	-	-	-	-	-	-	22.35 Mb
Accuracy	-	-	-	-	-	-	20.3 mAP

To demonstrate the effectiveness of partial quantization, we compare the partially quantized *SSDLiteM2* with the original and fully quantized models, which is shown in Table III. As can be clearly seen from Table III, the partially quantized *SSDLiteM2* is 3.01 times smaller than the original model with 20.5% mean average precision (mAP) [2] on the COCO dataset. Compared with the fully quantization, the partial quantization improves the mAP by 8.5% with only 6.3Mb more parameters.

With all the optimizations applied to *SSDLiteM2*, we propose a compressed *SSDLiteM2* model named the *C-SSDLiteM2*. The network detail is presented in Table IV, where each line describe a sequence of identical layers with repeated number n and expansion factor t. The first layer of each sequence has a stride s and all others use stride 1. Because the load-balanced channel pruning is mainly utilized to eliminate the sparse matrix multiplication, it only prunes 5% to 10% numbers of channels to maintain the accuracy. Finally, the fully optimized *C-SSDLiteM2* is 3.43 times smaller than the original *SSDLiteM2* with only 1.8% mAP loss on the COCO dataset.

C. Performance Comparison

To compare the performance of proposed design on Xilinx ZC706 with other platforms, we implement the original SSDLite on Intel Xeon E5-2680 v2 CPU and NVIDIA TITAN X Pascal GPU based on Tensorflow framework [25]. The CuDNN libraries are used for optimizing the GPU solution, and the compilation flag $-Ofast$ is activated for the CPU implementation. Although there is another FPGA-based binarized NNs accelerator for object detection [20], the accuracy is not evaluated on the large dataset such as the COCO dataset, and we are unable to compare our results to their implementation. Table V shows the performance and power consumption on different platforms. Note that since the depth-wise convolution is not well optimized by CUDA and cudnn, the original *SSDLiteM2* on GPU does not show a high speed-up compared with the CPU implementation.

Our proposed design can achieve nearly 65 fps, which is considered to be sufficient for object detection in many real-life applications. Compared with the original *SSDLiteM2* on Xeon E5-2680 v2 CPU and TITAN X Pascal GPU, our FPGA design is 3.5 and 1.8 faster, and consumes 41.8 and 32 times less power respectively. To make a fair comparison, we also implement the *C-SSDLiteM2* on the CPU and GPU platforms, which are also presented in Table VI. However, we observe that the processing speed of *C-SSDLiteM2* is even slower than the original *SSDLiteM2* in CPU and GPU platforms. A potential reason may be the support of 8 bits integer arithmetic is not well optimized on CPU and GPU devices. Finally, the area cost of the final design based on Zynq ZC706 is shown in Table VI.

[2]The mAP in this paper refers to mAP@[.5:.95] with the intersection over union (IoU) threshold from 0.5 to 0.95

TABLE V: Performance comparison of the final FPGA design versus CPU and GPU.

	CPU		GPU		Our Work
Platform	Intel Xeon E5-2680 v2		TITAN X Pascal		Zynq ZC706
No. of cores	10		3584		–
Compiler	GCC 4.8.5		CUDA 8.0 cudnn 7.05		Vivado 2016.2
Flags	$-Ofast$		–		–
Frequency	2.8 GHz		1.53 GHz		100 MHz
Technology	22 nm		16 nm		28 nm
Power (W)	115		168		9.9
Model	*SSDLiteM2*	*C-SSDLiteM2*	*SSDLiteM2*	*C-SSDLiteM2*	*C-SSDLiteM2*
Precision	32 bit float	partially quantized	32 bit float	partially quantized	partially quantized
Processing Time per Frame (ms)	54.6	83.6	28.6	97.8	15.43
Frame per Second (fps)	18.3	11.9	34.9	10.2	64.8
Energy per Frame (J)	6.27	9.61	4.8	16.43	0.15

TABLE VI: Area cost of the final hardware on Zynq ZC706.

	LUTs	Registers	DSP48s	BRAMs
Available	218600	437200	900	545
Utilization	148484	191899	728	311
Percentage Used	67.9%	43.8%	80.8%	57%

V. Conclusion and Future Work

This work proposes a novel design that accelerates the compressed *SSDLiteM2* on FPGA for object detection. The FPGA-based design is optimized by the fused bottleneck residual block (BRB), processing element (PE) sharing and fused batch normalization (BN). To compress the *SSDLiteM2*, several compression techniques including partial quantization and load-balanced channel pruning are proposed, which not only aim at model compression, but also enhance the efficiency of the proposed hardware. Our FPGA-based object detection accelerator on a Xilinx ZC706 device can achieve nearly 65 frames per second for $224 \times 224 \times 3$ images with 20.3 mean average precision on the COCO dataset. Further work includes exploring *SSDLiteM2* for various applications, studying optimizations automated design merging [27] and custom precision arithmetic [28]–[30], extending the architecture to support deconvolution [31], automating its optimization for specific FPGA devices, and improving design portability with overlay such as QuickDough [32].

Acknowledgement

The support of Corerain Technologies, Lee Family Scholarship, the EU Horizon 2020 Research and Innovation Programme under grant agreement number 671653 and the UK EPSRC (EP/L00058X/1, EP/L016796/1, EP/N031768/1 and EP/P010040/1) is gratefully acknowledged.

References

[1] R. Zhao, X. Niu, Y. Wu, W. Luk, and Q. Liu, "Optimizing CNN-based Object Detection Algorithms on Embedded FPGA Platforms," in *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer, 2017, pp. 255–267.

[2] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single Shot MultiBox Detector," in *European Conference on Computer Vision*. Springer, 2016, pp. 21–37.

[3] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 91–99.

[4] 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.

[5] A. Wong, M. J. Shafiee, F. Li, and B. Chwyl, "Tiny SSD: A Tiny Single-shot Detection Deep Convolutional Neural Network for Real-time Embedded Object Detection," *arXiv preprint arXiv:1802.06488*, 2018.

[6] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation," *arXiv preprint arXiv:1801.04381*, 2018.

[7] 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.

[8] F. Chollet, "Xception: Deep learning with Depthwise Separable Convolutions," *arXiv preprint*, pp. 1610–02 357, 2017.

[9] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient CNN Implementation on a Deeply Pipelined FPGA Cluster," in *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, 2016, pp. 326–331.

[10] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-S. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA Accelerator for Large-scale Convolutional Neural Networks," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 16–25.

[11] 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 (ARC)*. Springer, 2018, pp. 16–28.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[13] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-scale Image Recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[14] R. Girshick, "Fast R-CNN," *arXiv preprint arXiv:1504.08083*, 2015.

[15] N. Dalal and B. Triggs, "Histograms of Oriented Gradients for Human Detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, vol. 1, 2005, pp. 886–893.

[16] K. Negi, K. Dohi, Y. Shibata, and K. Oguri, "Deep Pipelined One-chip FPGA Implementation of a Real-time Image-based Human Detection Algorithm," in *2011 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2011, pp. 1–8.

[17] X. Ma, W. A. Najjar, and A. K. Roy-Chowdhury, "Evaluation and Acceleration of High-throughput Fixed-point Object Detection on FPGAs," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 6, pp. 1051–1062, 2015.

[18] H. Li, J. J. Davis, J. Wickerson, and G. A. Constantinides, "ARCHITECT: Arbitrary-precision Constant-hardware Iterative Compute," in *International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017, pp. 73–79.

[19] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 45–54.

[20] H. Nakahara, H. Yonekawa, T. Fujii, and S. Sato, "A Lightweight YOLOv2: A Binarized CNN with A Parallel Support Vector Regression for an FPGA," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 31–40.

[21] J. Misra and I. Saha, "Artificial Neural Networks in Hardware: A Survey of Two Decades of Progress," *Neurocomputing*, vol. 74, no. 1-3, pp. 239–255, 2010.

[22] S. Muralidharan, K. O'Brien, and C. Lalanne, "A Semi-Automated Tool Flow for Roofline Anaylsis of OpenCL Kernels on Accelerators," in *Proc. 1st Intl. Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC15)*, 2015.

[23] H. Fan, X. Niu, Q. Liu, and W. Luk, "F-C3D: FPGA-based 3-dimensional Convolutional Neural Network," in *27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–4.

[24] Y. He, X. Zhang, and J. Sun, "Channel Pruning for Accelerating Very Deep Neural Networks," in *International Conference on Computer Vision (ICCV)*, vol. 2, no. 6, 2017.

[25] M. Abadi, A. Agarwal, P. Barham, and E. Brevdo, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[26] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," in *European Conference on Computer Vision*. Springer, 2014, pp. 740–755.

[27] H.-C. Ng, S. Liu, and W. Luk, "ADAM: Automated Design Analysis and Merging for Speeding up FPGA Development," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 189–198.

[28] S. Liu, G. Mingas, and C.-S. Bouganis, "An Unbiased MCMC FPGA-based Accelerator in the Land of Custom Precision Arithmetic," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 745–758, 2017.

[29] "An Exact MCMC Accelerator Under Custom Precision Regimes, author=Liu, Shuanglong and Mingas, Grigorios and Bouganis, Christos-Savvas," in *International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2015, pp. 120–127.

[30] H. Fan, H.-C. Ng, S. Liu, Z. Que, X. Niu, and W. Luk, "Reconfigurable Acceleration of 3D-CNNs for Human Action Recognition with Block Floating-Point Representation," in *28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 1–4.

[31] S. Liu *et al.*, "Optimizing CNN-based Segmentation with Deeply Customized Convolutional and Deconvolutional Architectures on FPGA," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2018.

[32] C. Liu, H.-C. Ng, and H. K. So, "QuickDough: A Rapid FPGA Loop Accelerator Design Framework Using Soft CGRA Overlay," in *International Conference on Field Programmable Technology (ICFPT)*, 2015, pp. 56–63.