Resource-constrained FPGA implementation of YOLOv2

Zhichao Zhang¹ · M. A. Parvez Mahmud¹ · Abbas Z. Kouzani¹

Received: 13 September 2021 / Accepted: 26 April 2022 / Published online: 29 May 2022 © The Author(s) 2022

Abstract

Progress is being made to deploy convolutional neural networks (CNNs) into the Internet of Things (IoT) edge devices for handling image analysis tasks locally. These tasks require low-latency and low-power computation on low-resource IoT edge devices. However, CNN-based algorithms, e.g. YOLOv2, typically contain millions of parameters. With the increase in the CNN’s depth, filters are increased by a power of two. A large number of filters and operations could lead to frequent off-chip memory access that affects the operation speed and power consumption of the device. Therefore, it is a challenge to map a deep CNN into a low-resource edge IoT platform. To address this challenge, we present a resource-constrained Field-Programmable Gate Array implementation of YOLOv2 with optimized data transfer and computing efficiency. Firstly, a scalable cross-layer dataflow strategy is proposed which allows on-chip data transfer between different types of layers, and offers flexible off-chip data transfer when the intermediate results are unaffordable on-chip. Next, a filter-level data-reuse dataflow strategy together with a filter-level parallel multiply-accumulate operation computing processing elements array is developed. Finally, multi-level sliding buffers are developed to optimize the convolutional computing loop and reuse the input feature maps and weights. Experiment results show that our implementation has achieved 4.8 W of low-power consumption for executing YOLOv2, an 8-bit deep CNN containing 50.6 MB weights, using low-resource of 8.3 Mbits on-chip memory. The throughput and power efficiency are 100.33 GOP/s and 20.90 GOP/s/W, respectively.

Keywords Field-programmable gate array (FPGA) · Convolutional neural network (CNN) · YOLOv2 · Low-resource edge platforms · Internet of things (IoT)

1 Introduction

Edge artificial intelligence (AI) is a system that involves sensors and machine learning algorithms for data collection and analysis on a local Internet of Things (IoT) edge device [1]. Recent edge AI solutions often involve Field-Programmable Gate Arrays (FPGAs) to implement and execute machine learning algorithms [2, 3]. However, most IoT edge devices, e.g. drones, smart cameras, smart glasses, and remote control vehicles, can only include low-resource FPGAs due to cost and power expenditure requirements [4, 5]. In this paper, we consider the low-resource of on-chip memory, i.e., no more than 11 M bits of block RAMs (BRAMs) [6].

On the other hand, a lot of edge devices employ image sensors that produce data containing complex scenes and multiple objects. The detection and classification of objects in data produced by image sensors often involve convolutional neural network (CNN). However, CNN typically involves millions of parameters and operations, thus is difficult to be mapped into a low-resource IoT edge device [6–8]. Currently, efficient data processing solutions on edge devices are under development. For instance, Diya et al. [9] explored the power efficiency of different I/O standards on FPGA-based data processing. Different from Diya et al.’s work, this paper focuses on improving the efficiency of resource utilization for FPGA-based CNN processing.

YOLO is a popular CNN-based system that has achieved outstanding performance in object detection [10, 11]. Darknet-19 is the main CNN structure of the YOLO [12]. By changing the Darknet structure, researchers have developed variants of YOLO, including multiple YOLO versions and tiny YOLO versions. Previous works
have utilized FPGAs to increase the efficiency of YOLO inference [13–15]. For instance, Sanchez et al. [13] developed an FPGA-based automated workflow to operate CNNs. In Sanchez et al.’s experiment, the tiny YOLOv2 is implemented on Xilinx XCZU9EG FPGA FPGA, using 1-bit weights and 3-bits activations. As a result, a high speed of 41.7 FPS has been achieved due to low-precision representation. Similarly, Almād et al. implemented tiny YOLO v3 Xilinx on Virtex 7 VC707 FPGA. However, these works mentioned above have mainly focused on implementing tiny YOLO versions on FPGAs. Compared with YOLOv2, tiny YOLO versions provide a lower accuracy with a relatively shallow CNN structure [16]. Therefore, this work focuses on the FPGA implementation of YOLOv2 for low-resource edge IoT devices.

By going through the related works that mainly focus on implementing CNNs on FPGAs, a common method has been noticed to focus on minimizing the CNN size and mapping the entire CNN into an FPGA with fully cross-layer computing dataflow, thus completely avoiding off-chip data access [13, 16–18]. Such fully cross-layers computing dataflow strategies are often found from the implementation of low-precision and binary CNNs [13, 16–18], which usually have two disadvantages: (i) the development of a low-precision CNN would involve exhausting training/optimization procedure and often experience a significant accuracy drop compared with 8-bit CNNs [16], and (ii) implementing the entire CNN on-chip would be limited to the on-chip memory resources [19]. For instance, Nguyen et al. [16] designed a layer-specific dataflow with the activations quantized in mixed precisions of 1-bit and 8-bit, which allows the entire CNN stored in the BRAMs of the Xilinx Virtex-7 VC707 FPGA, consuming around 22.4 M bits BRAMs. Their main purpose is to completely avoid off-chip access for feature maps and weights, which only applies to the case when all the weights can be stored on-chip [16]. However, the demand for high resource utilization could lead to high power consumption. Moreover, the low-resource IoT edge devices may not have enough on-chip memory.

For a low-resource IoT edge device, off-chip memory access could be unavoidable, even with optimization methods. For instance, Wang et. al. [20] introduced optimization methods of pruning and quantization before FPGA implementation, which optimize the CNN into an 8-bit integer format with a sparse structure, containing an approximate size of 80 M bites. The sparse CNN has been implemented on Intel Arria10 GX1150 FPGA, including more than 67 M bits of on-chip memory with 1518 DSPs, consuming 26 W of power [20]. However, a low-resource IoT edge device requires low-resource and low-power consumption. Although the optimized CNN reduces the weights by 20 times, the remaining weights are still too many to fit into a low-resource FPGA. Therefore, for a low-resource FPGA-based CNN implementation, it is necessary to develop new methods to handle the off-chip and on-chip data transfer.

Compared with Wang et al. [20], Li et. al. [21] focus on efficient FPGA implementation to handle the data transfer. Their main contribution is the pipelined multiply-accumulate operation (MAC) and dataflow that aims to remove loop dependency and improve parallelism, thus reducing off-chip and on-chip data access. However, the high resource utilization on the Intel Arria10 GX1150 FPGA leads to 27.2 W of power consumption, which could be unaffordable for a battery-powered low-resource IoT platform. Moreover, the developed pipeline structure of multiple assignments is expected to benefit batch processing; however, it is not suitable for IoT applications that require single batch processing [3]. To reduce the resource requirement, further improvement of the dataflow and buffer design is required to increase the efficiency of data reuse and transfer. Different from Li et. al.’s work [21], Gschwend et al. developed ZynqNet [22] on a medium FPGA, Xilinx Zynq7045, that reuses the convolutional core for operating max pooling, which saves the resources. Due to a significant number of filters increased with layers, the small and medium FPGA accelerators could only handle single layer operations [22]. For instance, the ZynqNet needs to fetch 13.6 M bits parameters to on-chip BRAMs to operate a single layer computation. To reuse the convolutional core, the ZynqNet has replaced the original max-pooling with a $3 \times 3$ convolution operation using a stride of 2. However, reusing the convolutional core would lead to a significant off-chip memory access involved for transferring the activations between the convolutional layer and the max-pooling layer.

From the literature discussed above, we have noticed that many previous works often consume high resource utilization, i.e. especially the on-chip memory, in FPGAs for YOLOv2 inference to possibly fully avoid off-chip data access [13, 16, 20, 21, 23]. For a low-resource IoT platform, off-chip data access would be unavoidable since it would not have enough on-chip resources. In such cases, it is necessary to develop an efficient system to deal with off-chip and on-chip data access. Therefore, in this work, we propose a resource-constrained FPGA implementation to support YOLOv2 inference for object detection consuming low-resource and low-power.

The novelty of this work is the developed resource-constrained FPGA implementation of YOLOv2, one of the most influential CNN-based object detection algorithms. The contributions of this work are as follows.

1. We propose a novel scalable cross-layer on-chip computing dataflow strategy. Our dataflow strategy

 Springer
aims to reduce the resource utilization of the host FPGA and offer a suitable partial cross-layer computing on-chip, as well as flexible off-chip data access when the intermediate results are unaffordable on-chip. The convolutional, ReLu, and max-pooling layers have been optimized and reorganized to the layers of convolutional, scale, and max-pooling. The partial cross-layer data transfer is achieved by high bandwidth FIFO between the optimized layers. Moreover, the developed dataflow strategy is scalable and it could be extended to map different sizes of CNNs to various scales of FPGAs.

(2) We propose a novel multi-level data reuse strategy for computing convolutional layers, including both input feature maps (IFms) reuse and filter reuse. A PE array is further developed to offer filter-level parallel convolutional computing with reused data. This work replaces the general loops of multiplications with hardware-friendly dataflow strategies. The developed PE array offers maximum parallelism with a minimum of data reloading based on the data reuse strategy. Furthermore, we optimize the max-pooling loop by dividing a $2 \times 2$ pool into two $1 \times 2$ pools. Channel-level parallel max-pooling together with high bandwidth is provided.

(3) We propose novel multi-level buffering strategies to provide storage for the dataflow and data reuse strategies. The sliding buffer and sliding cache are introduced to reuse input activations in multi-level, i.e., (i) reuse the IFms for multiplying with multiple filters, and (ii) reuse the overlapped activations between neighbor receptive fields. Moreover, to avoid re-organizing the activations produced by the convolutional layer, multi-level max_pooling buffers are developed after optimizing the max-pooling loop.

(4) We have developed a novel resource-constrained YOLOv2 overlay based on the cross-layer computing dataflow, data reuse strategies, and multi-level buffers. Compared with previous works [20, 21, 24], and [25], our implementation achieves low-resource and low-power consumption.

The paper is organized as follows. Section 2 gives the background, including the YOLOv2 structure and the 8-bit quantization method adopted in this paper. Section 3 presents our proposed scalable cross-layer dataflow strategy. Section 4 introduces the hardware implementation, i.e., overall architecture as well as the design methods, including hardware development of convolutional layers, scale layer, and max-pooling layer. The details of the data reuse strategy, multi-level buffers, and PE array are given in the same section. Section 5 discusses the experiment, results, and future work. Finally, Sect. 6 concludes the paper.

2 Background

2.1 YOLO

YOLO [12], developed by Redmon et al., is one of the most important object detection methods. This work focuses on hardware implementation of the YOLOv2 network as it offers higher detection accuracy than tiny YOLO. YOLOv2 is often executed on the cloud. However, further hardware design methods and implementation are necessary when it is operated on an IoT edge device with limited resource availability and low-power requirement [20, 20].

Table 1 Parameter configuration of the YOLOv2 in this paper [12]

| Type | Filters | Size/Stride | Output feature maps |
|------|---------|-------------|---------------------|
| C + B + L | 32 | $3 \times 3$ | $416 \times 416 \times 32$ |
| M | $2 \times 2/2$ | $208 \times 208 \times 32$ |
| C + B + L | 64 | $3 \times 3$ | $208 \times 208 \times 64$ |
| M | $2 \times 2/2$ | $104 \times 104 \times 64$ |
| C + B + L | 128 | $3 \times 3$ | $104 \times 104 \times 128$ |
| C + B + L | 64 | $1 \times 1$ | $104 \times 104 \times 64$ |
| C + B + L | 128 | $3 \times 3$ | $104 \times 104 \times 128$ |
| M | $2 \times 2/2$ | $52 \times 52 \times 128$ |
| C + B + L | 256 | $3 \times 3$ | $52 \times 52 \times 256$ |
| C + B + L | 128 | $1 \times 1$ | $52 \times 52 \times 128$ |
| C + B + L | 256 | $3 \times 3$ | $52 \times 52 \times 256$ |
| M | $2 \times 2/2$ | $26 \times 26 \times 256$ |
| C + B + L | 512 | $3 \times 3$ | $26 \times 26 \times 512$ |
| C + B + L | 256 | $1 \times 1$ | $26 \times 26 \times 256$ |
| C + B + L | 512 | $3 \times 3$ | $26 \times 26 \times 512$ |
| C + B + L | 256 | $1 \times 1$ | $26 \times 26 \times 256$ |
| C + B + L | 512 | $3 \times 3$ | $26 \times 26 \times 512$ |
| M | $2 \times 2/2$ | $13 \times 13 \times 512$ |
| C + B + L | 1024 | $3 \times 3$ | $13 \times 13 \times 1024$ |
| C + B + L | 512 | $1 \times 1$ | $13 \times 13 \times 512$ |
| C + B + L | 1024 | $3 \times 3$ | $13 \times 13 \times 1024$ |
| C + B + L | 512 | $1 \times 1$ | $13 \times 13 \times 512$ |
| C + B + L | 1024 | $3 \times 3$ | $13 \times 13 \times 1024$ |
| C + B + L | 512 | $3 \times 3$ | $13 \times 13 \times 1024$ |
| C + B + L | 1024 | $3 \times 3$ | $13 \times 13 \times 1024$ |
| C + B + L | 64 | $1 \times 1$ | $13 \times 13 \times 256$ |
| C + B + L | 1280 | $3 \times 3$ | $13 \times 13 \times 1280$ |
| C | 125 | $1 \times 1$ | $13 \times 13 \times 125$ |

C Convolutional; B Batch normalization; L LeakyReLU; M Max_pooling
YOLOv2 adopts Darknet-19 [12], a CNN structure shown in Table 1, representing object detection task as a regression problem. Darknet-19 is composed of the $3 \times 3$ convolutional layer, $1 \times 1$ convolutional layer, batch normalization layer, LeakyReLu layer, and max_pooling layer. The combination of a $3 \times 3$ convolutional layer and a $1 \times 1$ convolutional layer benefits the increase in the CNN depth, thus contributing to high detection accuracy. Besides, the $1 \times 1$ convolutional layer is memory friendly since it offers a significant reduction of the channels included in feature maps. Also, $2 \times 2$ max-pooling layer with stride $= 2$ is a hardware-friendly approach that reduces ifms size by half without involving complex computing. The main challenge is the operation of $3 \times 3$ convolutional layer which involves more complex computing and memory requirements with the increase in CNN depth.

The Darknet-19 includes a total number of 50,676,061 weights. Even with the lowest precision, i.e., 1-bit, this would require around 50 M bits of memory space. However, a low-resource FPGA usually has no more than 11 M bits of on-chip memory. The weights with such a large size could not be fit into the on-chip memory of a low-resource FPGA. During the CNN inference, millions of input and output activations as well as intermediate results are produced, which indicates that off-chip data transfer is unavoidable. Therefore, it is necessary to develop efficient dataflow strategies and multi-level buffers to reduce repeated off-chip data transfer and reuse the data on-chip maximally.

### 2.2 Quantization

As stated in Sect. 2.1, the weights and activations would be stored on the off-chip memory. However, the data transfer between on-chip and off-chip memory is the most energy-consuming step during the CNN inference. Reducing the size of the weights and activations would increase the bandwidth and reduce the latency of data transfer. To do so, quantization and pruning are the most common approaches. Pruning approaches often involve training and optimization of the CNN with reduced number of weights, e.g., reducing channels of Ifm. However, pruning approaches may not reduce the complexity of operating a single multiplication. Compared with pruning, quantization approaches focus on reducing the precision of the weights and/or activations and representing the values with lower precision. On the one hand, quantization offers the reduction of the storage requirement. On the other hand, the original 32-bit floating-point multiplications would be simplified by using the lower bit-width of the quantized weights and activations. Therefore, we prefer to employ a quantization approach [26] in this work.

The 8-bit data representation is currently a common preference due to its hardware-friendly feature during operations [26, 27]. Although the extreme low-bit precision could reduce on-chip memory usage, it often involves exhausting time and effort to train and optimize a 1-bit CNN with high accuracy [28]. Since this work focuses on developing dataflow strategies and multi-level buffers, we prefer a post-training quantization strategy introduced by Jacob et al. [26] to represent both weights and activations in 8-bit integers.

Suppose $r_i = S_i(q_i - Z_i)$ where $q_i$ represents the $i$th quantized matrix, $r_i$ represents the $i$th original matrix, and $S_i$ and $Z_i$ are corresponding constants. Jacob et al. [26] represented the multiplication between two matrices $r_3 = r_1r_2$ as:

$$S_3(q_3^{(k)} - Z_3) = \sum_{j=1}^{N} S_1(q_1^{(j)} - Z_1) S_2(q_2^{(j)} - Z_2)$$

Therefore,

$$q_3^{(i,k)} = Z_3 + M \sum_{j=1}^{N} \left( q_1^{(j)} - Z_1 \right) \left( q_2^{(j)} - Z_2 \right)$$

where,

$$M = \frac{S_1S_2}{S_3}$$

In Eqs. (2) and (3) [26], $M$ is the only non-integer value that is between 0 and 1. By introducing $M_0 \in [0.5, 1)$, the $M$ is calculated using Eq. (4) [26], where $M_0$ could be represented using fixed-point, and multiplying with $2^{-n}$ could be replaced by low-energy right shift operation.

$$M = 2^{-n}M_0$$

Finally, the floating-point 32-bit multiplication can be replaced by a simple 8-bit integer multiplication together with an efficient scale procedure that includes fix-point multiplication and bit-shift operation.

### 3 Scalable cross-layer dataflow strategy

We re-structure the YOLOv2 layers based on the post-quantization method [26], shown in Fig. 1. The normalization layers are attached to the convolutional layers after quantization. Besides, a scaling procedure that added the bias of $\beta - \gamma \mu / \sigma$ is applied after the MAC computing. We optimize the structure by separating the MAC computing from the convolutional layer and combining the rest of the modules, i.e., adding bias, bit-shift, and Leaky_ReLu together to a scale layer. The re-structured layers simplify the data transfer procedures and increase the parallelism of the algorithm.
Next, we propose a scalable cross-layer dataflow strategy that offers three key features: (i) reducing intermediate results transfer between off-chip and on-chip buffers, (ii) computing in parallel multi-layer pipeline, and (iii) mapping larger convolutional networks into various scales of FPGAs. Figure 2 shows the proposed dataflow strategy. The FIFO is utilized to directly transfer a sequence of output activations across the convolutional layers, scale layer, and max-pooling layer. Both weights and Ifms are partially stored in the BRAMs. Therefore, compared with output-stationary, the dataflow does not require storing output activations on-chip. Weight-stationary offers reusing filters on-chip, however not benefiting the reuse of Ifms. Input-stationary prefers to store and reuse Ifms on-chip; however, it would require large on-chip memory and does not offer reuse of the weights. Compared with weight-stationary and input-stationary, the dataflow reuses both the $fc$ channels of Ifms and $fn$ filters in maximum without repeated data reading, including Ifms, weights, partial sums (Psums), and output feature map (Ofm) from the off-chip memory. Row-stationary could reduce the repeated data reading; however, it requires re-organization of the data, i.e., Ifm, Ofm, and weights, and it causes discontinuous DRAM accessing, thus affecting the effectiveness of the bandwidth [29]. In comparison, our proposed dataflow does not involve data re-organization, and thus benefits high bandwidth for computing $fc$ channels of Ifms with $fn$ filters. Parallel computing is applied across the convolutional layers, scale layer, max-pooling layer, and the data loading of weights and Ifms. The dataflow can compute $fc$ channels of Ifms and stream out $fn$ channels of output activations in pipeline. The adjustable parameters of FN, FC, W, and Ifm are generated by the CNN, and that $fn$, $fc$, and PE_num depend on the FPGA. Since all the input parameters are adjustable, the dataflow offers the flexibility of mapping large deep neural networks on various FPGAs. The more resources the FPGA includes, the greater the $fn$ and $fc$ could be set; thus, more parallel computing the dataflow could offer.

Suppose a convolutional layer $l$ has $FN[l] = N$ filters, with each containing $FC[l] = CH$ channels. We design the on-chip buffer capable of storing a maximum of $fn$ filters with each one containing up to $fc$ channels. When $CH < fc$ and $N < fn$, the $fn$ output activations are streamed to the rest of the layers without sending back any intermediate results to DDR. We extend the bandwidth to $PE_{num} \times 8$ bits to transfer $PE_{num}$ output activations once at a time. The high bandwidth is applied for data transfer between layers and increases the efficiency of data reading and writing of the PE array. The scalable strategy is available for the case of $CH > fc$ and $N > fn$. When $CH > fc$, the MAC computing of the convolutional layer is accumulated for $CH/fc = rec$ times. Similarly, the convolutional layer is executed for $N/fn = ren$ times when $N > fn$. And the off-chip intermediate results transfer is involved only when $N > fn$ and $CH > fc$, with many intermediate results generated.

4 Overall architecture and design methodology of hardware implementation

We implement the scalable cross-layer dataflow strategy, buffer design, and the PE array on the FPGA. Figure 3 shows the top-level architecture. The input image is read through the CPU and pre-processed in the PS. The pre-processing aims to resize the image to (416,416,3). The image contains 3 channels and requires 24-bits data width, with each pixel represented in unsigned 8-bit format. The FIFO with 32-bit bandwidth is applied to stream pixels into BRAM.

One of the challenges on resource-constrained platforms is that the convolutional computation would result in repeated loading of the weights and Ifm. The repeated data transfer between DDR and BRAM causes the inefficiency of convolution computing. To address this issue, we implement multi-level buffers to keep the most needed weights and Ifm in BRAM, and reuse these data as much as possible, thus reducing the off-chip data transfer. We implement off-chip buffers in DDR to fully store the weights and Ifms, and then partially transfer the data to the on-chip buffers implemented in the BRAM. We implement an on-chip Ifm buffer to temporarily store a maximum of three rows of the Ifms, with each row containing a maximum of $fc$ channels. The on-chip Ifm buffer is designed to fetch the Ifm row by row, with each row of activations are stored in channel major. Then, PE array fetches every column of activations from the Ifm buffer to the local cache, with each column covering 3 rows × $fc$ channels of
activations. After the first output activation has been computed, the local cache would slide to the next column of the activations that have been stored in the Ifm buffer. Finally, the Ifm buffer stores three rows of activations for computing a row of output activations and then gets updated by sliding to the next row of the Ifms. The Ifm buffer is partitioned to reduce the latency of the data transfer. And the local cache in the PE array is partitioned with 9 ports for reading a channel of the receptive field from the Ifm buffer in one pass. At the same time, the array of weights is streamed by M_AXI port, with the size of $3 \times 3 \times fc \times fn$. On the one hand, the multi-level buffer allows the activations reused in maximum on-chip and avoids off-chip reloading. On the other hand, the most needed weights and input activations are transferred on time.

Another challenge is the large consumption of LUTs and DSPs, which could be caused by the fully parallel convolutional computing of $fn$ channels of output activations. Accordingly, we design a strategy that allows partial

---

Table: Scalable cross-layer dataflow strategy

| Input: Ifm, W, FN, FC, fn, fc, PE_num, Flag_conv, Flag_pad, Flag_pool, L_group |
| Output: the output of the CNN |

For $l$ in L_group:
- $ren = FN[l]/fn$
- $rec = FC[l]/fc$
- while (rec):
  - while (ren):
    - parallel compute following steps:
      - stream W[l] from DDR memory to BRAMs
      - stream Ifm[l] from DDR memory to BRAMs
    - if (Flag_pad):
      - apply padding
    - if (Flag_conv == 3)
      - operate MAC of the $3 \times 3$ convolutional computing
    - if (Flag_conv == 1)
      - operate MAC of the $1 \times 1$ convolutional computing
    - $ren = ren - 1$
    - if ($ren != 0$):
      - write MAC intermediate results to DDR
    - else:
      - if ($rec != 0$):
        - read previous MAC results from DDR
        - accumulate with previous MAC results
        - write MAC intermediate results to DDR
      - else:
        - stream MAC intermediate results to scale layer
        - if (Flag_pool):
          - stream output activations to Max_pooling layer
        - else:
          - write output activations to DDR
parallelism of computing $PE_{num} < fn$ channels. Moreover, we apply $PE_{num}$ ports for the on-chip weights buffer to support low latency data transfer while the parallel PE array computing.

Besides, our proposed architecture reduces the off-chip data transfer between layers. AXI-Stream is the key technique that is used to support the cross-layer dataflow implementation. We implement a $3 \times 3$ convolution layer, $1 \times 1$ convolution layer, scale layer, and max-pooling layer in the logic, and the intermediate results are allowed to flow in between these layers. The DSPs are utilized for computing MACs.

Moreover, to increase the scalability of the system, we use DDR to store the intermediate results from the convolutional computing that involves Ifms with more than $fc$ channels and weights more than $fn$ filters. The $M_{AXI}$ ports are applied to support the off-chip transfer of the intermediate results and output activations.

4.1 Convolutional layers

To support efficient convolutional computing on an FPGA with maximum data reuse and minimum data re-loading, we optimize the convolutional computing and propose the dataflow strategy, buffers, and PE array design accordingly. First of all, we have re-organized the loops of MAC computing to increase the reuse of the input activations. Secondly, we introduce two three-dimensional (3D) sliding windows that go across the Ifm to avoid reloading data from off-chip memory. Thirdly, we increase the bandwidth of outputting the Ofm activations.

4.1.1 Multi-level data reuse strategy

Figures 4 and 5 show the original and the optimized convolutional computing, respectively. Suppose the Ifms contains $fc$ channels, each includes an Ifm. And there are $fn$ filters, each containing $fc$ channels. As shown in Fig. 4, all the $fc$ channels of Ifms are reloaded $fn$ times to calculate $fn$ channels of Ofms. Besides, due to the overlaps of the receptive fields, the same input activation is used for multiple receptive fields to calculate the neighbour output activations. Reloading the input activations would reduce the time efficiency of convolutional computing. Besides, we have noticed that computing and transferring the activations could take a large partition of the time-consuming. Moreover, a convolutional layer would involve hundreds to thousands of filters, and each filter could contain hundreds to thousands of channels. Since there are limited resources available in the FPGA, it would be difficult to fit all the filters into the on-chip memory.

According to the above analysis, we re-organize the loops to reuse data in a multi-level form and accordingly design multi-level buffers to store Ifms and filters. Suppose there are $fn = rep_n \times PE_{num}$ filters loaded into the on-chip memory; the optimized loops are shown in Fig. 5.
Firstly, by moving the filters into the most inner loop, the same input activation is only read once to multiply with \( fn \) corresponding weights with each of them from the same location of the \( fn \) filters. Thus, \( fn - 1 \) times of re-loading the input activations are avoided. Secondly, to deal with the overlap between receptive fields, we introduce two 3D sliding windows to catch a receptive field, with each input activation being further reused for a maximum of \( 3 \times 3 = 9 \) times on-chip, thus avoiding the reading of the overlapping activations. Thirdly, parallel MAC computing is conducted between a receptive field and PE_num number of filters. Moreover, to improve the time efficiency, we increase the bit width of the output to transfer PE_num channels of activations at the same time, thus reducing the iterations of outputting the activations. High bandwidth is designed to transfer the extended bit width of output activations. Finally, we build multi-level buffers to fetch the filters from the off-chip memory, i.e. using the 3D sliding windows to cache the most needed activations and filters to the on-chip memory, thus improving the data transfer efficiency.

4.1.2 Multi-level buffers and filter-level parallel MAC computing

Multi-level buffers: We propose multi-level buffers to ensure each input activation is reused for a maximum of \( fn - 1 + 3 \times 3 \) times, which could significantly increase the data transfer efficiency. Suppose there are \( fc = 128 \) channels of the Ifms with the size of row \( \times \) col = 104 \( \times \) 104, which are required to compute with \( fn = 256 \) filters. Our multi-level buffers would avoid approximately \( 104 \times 104 \times 128 \times (256-1 + 3 \times 3) = 366,494,272 \) times of reloading of the input activations. To begin with, all the weights of the CNN are generated in the manner of filter-major and are stored in the off-chip memory, as shown in Fig. 6.

Then, the filters are divided into ren groups. The on-chip buffer fetches a group of filters once at a time, with each group containing \( fn \) number of filters. Besides, our dataflow strategy streams Ifms from the off-chip buffer into the on-chip buffer following the channel-major manner, as shown in Fig. 6. The designed on-chip Ifms buffer is capable of storing for 3 rows \( \times \) cl columns \( \times \) fc channels of activations, which is the designed first sliding window. Moreover, we design an on-chip cache, considered as a second sliding window, to store and update the receptive field. We use the sliding buffer and sliding cache to indicate the first sliding window and the second sliding window, respectively. The sliding buffer goes through the Ifms vertically to fetch Ifms row by row across all the fc channels, and that the sliding cache fetches from the sliding buffer column by column with all channels included. Therefore, by using the sliding buffer and the sliding cache over the fc channels of...
Ifms, fn channels of Ofm would be produced with fn filters without off-chip data reloading.

**Filter-level parallel MAC computing**: Once the sliding cache has updated a receptive field, the PE array would start MAC computing. The designed dataflow operates filter-level parallel MAC computing channel by channel. Figure 7 shows the dataflow of a single channel of MAC computing between a receptive field and fn filters. The $3 \times 3$ activations of channel $c_1$ are multiplied with the corresponding weights from fn filters, and the multiplication results are accumulated per filter. Therefore, a channel of MAC computation produces fn intermediate results that are stored in a small output on-chip cache. After that, the MAC computing is operated across fc channels of input activations, as shown in Fig. 6. Accordingly, the intermediate results get accumulated channel by channel, and that fn channels of output activations are generated.

After the first receptive field has been completely used for computing the fn channels of output activations, the sliding cache would keep the last two columns of activations and load the fourth column. By sliding the cache window through all the columns, we calculate a row of fn channels of output activations. Meanwhile, we update the Ifms sliding buffer by replacing the first row of $c_1 \times fc$ activations with a new row of activations. The sliding buffer and sliding cache ensure every single activation would be loaded once from off-chip memory to the on-chip buffer for calculating fn channels of output activations. Therefore, the most expensive data reloading between on-chip and off-chip memory can be avoided.

**Resource analysis**: The total on-chip storage $M$ required for $3 \times 3$ convolutional computing using multi-level buffers is represented by Eq. (5), where $Ifm_{buf}$ and $W_{buf}$ indicate the sliding buffer and the weight buffer, respectively.

$$M = Ifm_{buf} + Rec_{cache} + W_{buf} + M_{ports}$$

$$= 3 \cdot \text{max}_\text{col} \cdot fc + 3 \cdot 3 \cdot fc(1 + fn) + \beta$$

The $Rec_{cache}$ represents the sliding cache, implemented using FFs for the local cache of the PE array. The $M_{ports}$ represents the resources required for interfaces. The max_col indicates the maximum columns of the Ifms. The Ifm buffer $Ifm_{buf}$ is implemented using true dual-port RAM.
that supports both reading and writing on both ports. $I_{num\_ports} = 3\text{rows} \times \text{PE}_{num}$ number of ports are attached to the Ifmbuf to support parallel data transfer. Besides, the weight buffer $W_{buf}$ is implemented as a ping-pong buffer using $w_{num\_ports} = 3 \cdot 3 \cdot \frac{\text{PE}_{num}}{2}$ ports to provide parallel data loading and computation. Therefore, the total number of BRAMs $num\_bram$ is calculated, as shown in Eq. (6), where $T_{bram}$ indicates the total number of BRAMs available.

$$num\_bram = I_{num\_ports} + w_{num\_ports} + M_{ports}$$

$$= 3 \cdot \text{PE}_{num} + 3 \cdot 3 \cdot \frac{\text{PE}_{num}}{2} + \beta$$ (6)

s.t. $num\_bram < T_{bram}$

$$P_{bram} = \frac{I_{num\_ports}}{18\text{Kb}\text{/sbits}} \times 100\% = \frac{3 \cdot \max\_col \cdot fc}{3 \cdot \text{PE}_{num}} / 2250 \times 100\%$$ (7)

$$P_{bram\_w} = \frac{W_{buf}}{18\text{Kb}\text{/sbits}} \times 100\%$$

$$= \frac{3 \cdot 3 \cdot fc(1 + fn)}{3 \cdot 3 \cdot \frac{\text{PE}_{num}}{2}} / 2250 \times 100\%$$ (8)

Suppose the BRAM is integrated with 18 Kb block RAMs, with each block RAM is capable of storing 18 Kb / 8 bit = 2250 bytes. When partitioning the on-chip buffer and applying multiple ports to BRAMs, each BRAM may not be full-filled. To save the BRAMs, we keep the utilization of each block RAM as high as possible. Equations (7) and (8) represent the resource utilization percentage in each block RAM (RUPB) of the Ifm buffer and weights buffer.

By increasing RUPB, the total number of utilized BRAM blocks would be reduced. The PE$_{num}$ is mainly restricted to the number of DSP, i.e., $num\_DSP$, since DSPs are utilized to operate parallel MAC computing on an FPGA. Equation (9) represents the overall resource utilization percentage. By combining Eqs. (5–9), we can calculate the $fc$, $fn$, and PE$_{num}$ for maximum parallel computing with maximum RUPB.

$$P_{overall} = \min(\max(P_{bram}), \max(P_{bram\_w}))$$ (9)

s.t. $fn = 2 \cdot fc$, $fc = 32, 64, 128, 256 \ldots$

In this work, we aim to reduce the resource utilization of the FPGA and achieve low-power consumption for low-resource IoT platforms, which consider the BRAMs $\approx 11$ M bits. Suppose $fc = 128$, $fn = 256$, PE$_{num} = 32$ and max$_\_col = 418$, the RUPB of the Ifm buffer is $P_{bram\_I} = 74.31\%$. The max$_\_col$ could support up to 562 columns of the Ifm with 99.9% of RUPB. Moreover, a high RUPB $P_{bram\_w} = 91.38\%$ of the weight buffer is achieved.

4.1.3 PE array design

We propose an effective PE array to operate filter-level parallel MAC computing. Figure 8 shows an example of the MAC computation within the PE array. The array includes $3 \times 3 = 9$ PEs for a $3 \times 3$ convolutional window, with each of them containing PE$_{num}$ number of weights from the corresponding location of the PE$_{num}$ filters in parallel. Moreover, the entire PE array offers a channel of MAC computing each pass, where each channel involves $3 \times 3 = 9$ activations from the receptive field. A channel of intermediate results is produced from all the PEs, accumulated and stored temporarily in a small output cache implemented by FFs. After that, the weights from the corresponding location of the next group of PE$_{num}$ filters would be passed to the PE array to operate MAC without updating the input activations. Following the optimized loop, $fc$ channels of MAC are conducted in the pipeline to generate $fn$ channels of
output activations. In this way, the PE array avoids reloading the same input activations to compute with \( fn \) filters. Moreover, each input activation is read once and passed to neighbor PEs to reuse the overlapped activations between receptive fields, thus increasing the data transfer efficiency.

4.1.4 1 × 1 Convolutional layer

The optimized 1 × 1 convolutional computation is slightly different from the 3 × 3 convolutional computation. The only difference is that the dataflow prefers to compute the MAC between a receptive field and a filter, both have the size of 1 × 1 × \( fn \), where \( fn \) indicates the number of channels of the filter, the same \( fn \) number of 3 × 3 filters. The reason for designing \( fn \) channels for 1 × 1 convolution is to avoid re-organization of input activations that are calculated by the previous 3 × 3 convolutional layer. The 3 × 3 convolutional layer outputs \( fn \) channels of activations and is directly streamed to the 1 × 1 convolutional layer. By setting \( fn \) channels for 1 × 1 convolution computing, the activation stream produced by the 3 × 3 convolutional layer would keep the same manner.

Besides, the designed dataflow accumulates all the multiplication intermediate results across \( fn \) channels for multiplying with a single 1 × 1 filter and then passes the same \( fn \) channels of input activations to the rest of the \( fc-1 \) filters. The PE array computes PE_num channels of MAC in parallel. There is no need for output cache since there is one output activation generated at a pass; thus, it could be extended to a high bit-width representation. The high bandwidth stream is implemented to allow PE_num activations transferred at once.

4.2 Scale layer

The scale layer directly reads the output activations from the convolutional layer that completes MAC computation. Figure 9 shows the data loading and computing of the scale layer. Firstly, the corresponding bias is added to the activations, represented in int32 format. Then, the right bit shift is operated to obtain the int8 output activations. After that, the output is filtered by the Leaky ReLu. The scale layer offers high I/O bandwidth to read and write PE_num number of input and output activations in parallel.
4.3 Max pooling layer

Max pooling does not involve much computation as the convolutional layers; however, it requires efficient data transfer and storage due to overlap between pooling receptive fields, as shown in Fig. 10. Since a large partition of the FPGA resources has been utilized for convolutional computing, we design an efficient resource-constrained max pooling strategy.

We have designed several max pooling strategies to apply a sliding window to avoid duplication of loading activations. One of the strategies is operating $2 \times 2$ max-pooling channel by channel, as shown in Fig. 11. However, this would require the re-organization of Ifms. Besides, the large amount of re-organized Ifms increases the memory resource, which involves additional off-chip data transfer and slowing down the speed.

Another strategy we could try is similar to the sliding cache and sliding buffer when computing $3 \times 3$ convolutional computation, i.e. storing a receptive field and two rows of Ifms, with each row containing $fn$ channels, as shown in Fig. 12. This strategy would avoid the re-organization of the Ifms. However, this strategy requires storing $2 \times \text{row_in} \times fn$ number of activations, where $\text{row_in}$ represents the Ifm size.

In comparison, the third strategy we designed includes a small cache with the size of $2 \times fn$ and store the outputs in a buffer size of $\text{row_out} \times fn$, where $\text{row_out}$ indicates the Ofm size. Since $\text{row_out} = \text{row_in}/2$, the on-chip storage requirement can be reduced by 4 times approximately. The pooling procedure is operated row by row, with each row containing $fc$ channels of activations. As shown in Fig. 13, we introduce a small cache to operate max_pooling with a window size of $1 \times 2 \times fn$ across a row of the Ifms, and the produced intermediate results are temporarily stored in an output buffer the size of $\text{row_out} \times fn$. When the small cache slides to the second row of Ifms, it would first operate the $1 \times 2$ max-pooling and then compare it with the previous intermediate result stored in the output buffer to produce an output activation. Compared with the first two strategies we designed, our third max_pooling data-flow strategy benefits from directly streaming the Ifms into the max_pooling operation without re-organization and offering the most efficient resource-constrained...
max_pooling strategy. Therefore, we finally choose our third max_pooling dataflow strategy in this work.

5 Experimental results and evaluation

5.1 Experimental setup

In an experiment, our implementation is based on the ZCU104 platform containing a Xilinx Zynq UltraScale+ MPSoC ZU7EV FPGA that includes 11 Mb BRAMs. To focus on low-resource and low-power consumption, we only consider the BRAMs for storage in this experiment. There exist a quad-core ARM Cortex-A53 processor and 2 GB DDR4 memory. Vivado HLS 2019 is utilized to generate the RTL circuit, and the hardware implementation is developed using Vivado 2019.

The experiment adopts the original YOLOv2 [30] that is trained by Joseph Redmon for object detection tasks using datasets of PASCAL VOC 2007 and VOC 2012 [31]. The YOLOv2’s configurations are downloaded from Joseph Redmon’s official Github [32], and the parameters are downloaded from his official website [33]. Our experiment does not involve re-configuration or tuning of the YOLOv2.
In the experiment, the YOLOv2 is quantized using the post quantization method introduced in [26]. The mAP is measured after quantization. Both weights and images are represented in 8-bit format and saved in .bin files. The processing system (PS) could handle the allocation of DDR4 memory to store the weights, images, intermediate results, and feature maps. The PS could schedule the implemented overlay, and the YOLO header in the PS could predict the bounding boxes and recognize the classes. The platform of Vivado HLS is adopted to implement YOLOv2 with our proposed cross-layer on-chip computing dataflow, and multi-layer data reuse and buffering strategies, which involve HLS programming to reduce the FPGA resource utilization and improve power efficiency. Our custom accelerator is written in HLS, and then the RTL file is generated automatically in the Vivado HLS. Finally, our custom overlay is developed by importing the generated RTL file and designing the block diagram in Vivado. To meet the requirement of the sequential operation of IoT applications [3], the YOLOv2 is operated with batch size = 1. To validate the performance, we executed our developed custom overlay on the FPGA and measured the resource utilization, power consumption, and speed, and compared the performance of our implementation with the previous FPGA-based YOLO implementations. Since the reports from Vivado are considered in the majority of existing publications [21, 34, 35], we consider the resource utilization, in terms of on-chip memory, DSPs, LUTs, and FFs reported in Vivado, and the power consumption, i.e. the total on-chip power, reported from the power analyser of Xilinx in Vivado. On one hand, by validating the results in two ways: (i) whether the result of on-chip memory resource utilization is under 11 M bits, and (ii) whether the result of power consumption is lower than 5 Watts, we can judge the performance of the resource-constraint and power-efficiency of our YOLOv2 implementation. On the other hand, by comparing our implementation with previous works in terms of resource utilization (including BRAMs, DSPs, and LUTs), size (including the size of the model, weights, and that of input frame), power consumption, and speed, we can verify the advantages and disadvantages of our implementation.

5.2 Results

Table 2 shows the total resource utilization, and that of different types of layers on the FPGA. The overall utilization includes 461 BRAMs (18 Kb each), 387 DSPs, 50,716 FFs, and 91,167 LUTs. From Table 2, we can see that the main resource utilization is the BRAMs, which covers $461 \times 18 \text{ kb} = 8.23 \text{ M bits} < 11 \text{ M bits}$. The BRAMs are mainly utilized for on-chip memory and M_AXI interfaces. The on-chip memory consumes 189 BRAMs, accounting for 30.3%, to store weights and bias. Moreover, the developed multi-level buffers of the $3 \times 3$ convolutional layer consume 15% BRAMs to load Ifms. In addition, max-pooling consumes 7% BRAMs. Compared with the $3 \times 3$ convolutional layer, the $1 \times 1$ convolutional layer and scale layer are implemented without using BRAMs. Besides, the M_AXI interfaces consume 128 BRAMs to generate high bandwidth interfaces for on-chip and off-chip data transfer. The major utilization of DSPs is contributed by the $3 \times 3$ convolutional layer with 15% DSPs used for the MAC computing. Besides, 5% DSPs are...
used for the scale layer for operating the multiplication of the Leaky ReLU. From the above analysis, we can see that our implementation mainly uses resources for operating the $3 \times 3$ convolutional layer, and it does not consume significant resources for implementing the $1 \times 1$ convolutional layer, max-pooling layer, and scale layer.

In this experiment, YOLOv2 contains a large model size, involving 50.6 MB weights and 34.90 GOP approximately. Table 3 reports the performance in terms of power consumption and throughput. Our implementation achieves the power consumption of 4.8 W for inference of YOLOv2, which meets the goal of less than 5 Watts. The throughput and power efficiency are measured by GOP/s and GOP/s/W, respectively. The frequency is set to 200 MHz in the experiment.

### 5.3 Comparison, discussion, and future work

Table 4 shows the comparison of our implementation with other related works. We have chosen the same features for a fair comparison, including the input image frame of $416 \times 416$, the PASCAL VOC dataset, and the FPGA frequency of 200 MHz. From Table 4, it can be seen that our implementation reduces the BRAMs by 2.7 to 6.9 times compared with previous works [20, 21, 24, 25]. Our implementation consumes 6.96 times less BRAM than [20], and it requires 2.75 times less BRAM than [24]. Moreover, compared with [21], our implementation reduces the FFs utilization by 10.3 times. Compared with [25], our implementation reduces the number of DSPs and LUTs by 8.9 times and 7.0 times, respectively. This shows the significant reduction of on-chip resource utilization achieved by our implementation. The low-resource utilization allows the deep CNN of YOLOv2 to be executed on a low-resource IoT edge device.

Besides, our implementation archives the lowest power consumption compared with the previous works [20, 21, 24, 25], reducing power consumption by 2.3 times to 5.7 times. Compared with [25], our implementation reduces power consumption by 4.4 times while handling a larger weight size of the CNN. The [20] and [24] could achieve high power efficiency due to their aim of achieving very high throughput. However, such high throughput implementations require higher overall power consumption and significant on-chip resources, which could not be suitable for a low-power and low-resource edge device. Compared with [20] and [24], we focus on low-resource and low-power implementation, and our implementation reduces the overall power consumption by 5.4 times and 2.3 times. Compared with [21], the same level of power efficiency has been achieved by our implementation using significantly lower resources of BRAMs, LUTs, and FFs. In conclusion, our implementation outperforms the previous work in terms of using low-resource utilization for a battery-powered low-resource IoT platform.

Our implementation has resolved the key challenges of the low-resource and low-power execution of YOLOv2 with deep CNN involved. In our future work, we would like to focus on improving the speed for low-resource CNN inference. In this experiment, our low-resource implementation has achieved a throughput of 100.33 GOP/s. The YOLOv2 inference speed measured in frame rate is around 2.16 FPS. This is caused by the conflict between the huge model size of YOLOv2 and the low-resource requirement. The speed could be further improved by combining the implementation with CNN approximation approaches, e.g., channel pruning, i.e., reducing the number of filters in each convolutional layer. In the future, we would like to reduce the model size and improve the implementation for high-speed CNN inference. Besides, in the next step, we would like to adopt more efficient FPGA-based boards, e.g., PYNQ-Z2 and Ultra96-V2 boards, which are suitable for low-resource and low-power IoT platforms.

Furthermore, beyond the topic of this work, the emerging advanced computing technologies such as quantum computing is highly expected to bridge the gap

| Model size | Weights | Frequency | Power consumption | Throughput | Power efficiency |
|------------|---------|-----------|-------------------|------------|------------------|
| 34.90 GOP  | 50.6 MB | 200 MHz   | 4.8 W             | 100.33 GOP/s | 20.90 GOP/s/W    |
between the limited computing cloud resources and the rapid growth of complexity of the computationally expensive AI applications. As the next generation of computing methods, noisy intermediate-scale quantum technology has emerged [37] in addition to quantum mapping, i.e. mapping a quantum AI algorithm to a specific quantum hardware platform.

Besides, the research of deep neural network (DNN) optimization and quantization is the necessary front step to fit DNN into a low-resource FPGA, and it has attracted leading companies’ attention. For instance, Xilinx has released a development stack for AI inference, named Vitis AI [38], which includes the AI optimizer and quantizer, expected to further improve the efficiency of the DNN inference. In the future, we would like to take advantage of the Vitis AI, as well as focus on developing efficient DNN optimization and quantization algorithms.

### 6 Conclusion

This paper presented an FPGA-based YOLOv2 implementation for object detection tasks on low-resource computer vision-based IoT edge devices including drones, smart cameras, smart glasses, and remote control vehicles with cameras. Our FPGA implementation addresses the issue of large data reloading and frequent off-chip memory access for resource-limited platforms by developing efficient dataflow and multi-level buffers to maximize data transfer on-chip and minimize off-chip data access. The proposed dataflow strategy offers filter-level data reuse in the PE array for parallel computing and allows output activations transferred across different types of layers. Moreover, the proposed multi-level on-chip buffers involve a sliding buffer and a sliding cache to reuse the fetched Ifms on-chip. Our implementation achieves low memory resource utilization, i.e., 8.3 M bits of on-chip memory, with 4.8 W of low-power consumption for YOLOv2 inference. The limitations of the current work include: (i) the conflict between the large size of YOLOv2 and the low-resource requirement of an edge device makes it hard to optimise the speed performance, and (ii) supporting only 8-bit precision operation. Future extensions of this work include improving the speed and flexibility of supporting various precisions, as well as developing hardware-oriented CNN optimization algorithms for low-resource CNN inference on low-resource and low-power IoT edge devices.
Funding  Open Access funding enabled and organized by CAUL and its Member Institutions.

Declarations

Conflict of interest  The authors declare that they have no conflict of interest.

Open Access  This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Shi Y, Yang K, Jiang T, Zhang J, Letaief KB (2020) Communication-efficient edge AI: algorithms and systems. IEEE Commun Surv Tutor 22(4):2167–2191
2. Xu C, Jiang S, Luo G, Sun G, An N, Huang G, Liu X (2020) The case for FPGA-based edge computing. IEEE Trans Mob Comput. https://doi.org/10.1109/TMC.2020.3041781
3. Han S, Liu X, Mao H, Pu J, Pedram A, Horowitz MA, Dally WJ (2016) EIE: Efficient inference engine on compressed deep neural network. ACM SIGARCH Comput Archit News 44(3):243–254
4. Liu Z, Zheng T, Xu G, Yang Z, Liu H, Cai D (2020) Training-time-friendly network for real-time object detection. In: proceedings of the AAAI conference on artificial intelligence, vol 07, pp 11685–11692
5. Zou Z, Shi Z, Guo Y, Ye J (2019) Object detection in 20 years: a survey. arXiv preprint arXiv:190505055
6. Zhang Z, Kouzani AZ (2020) Implementation of DNNs on IoT devices. Neural Comput Appl 32(5):1327–1356
7. Arshad MA, Shahriz S, Sagahyroon A (2020) On the Use of FPGAs to Implement CNNs: a Brief Review. In: 2020 International conference on computing, electronics & communications engineering (iCCECE), IEEE, pp 230–236
8. Murshed M, Murphy C, Hou D, Khan N, Ananthanarayanan G, Hussain F (2019) Machine learning at the network edge: A survey. arXiv preprint arXiv:190800080
9. Garg D, Sharma K, Singla A (2018) Designing a green data processing device using different input/output standards on FPGA. In: 2018 fifth international conference on parallel, distributed and grid computing (PDGC), IEEE, pp 75–79
10. Zhao Z-Q, Zheng P, Xu S-t, Wu X (2019) Object detection with deep learning: a review. IEEE Trans Neural Netw Learn Syst 30(11):3212–3232
11. Dhillon A, Verma GK (2020) Convolutional neural network: a review of models, methodologies and applications to object detection. Prog Artif Intell 9(2):85–112
12. Redmon J, Divvala S, Girshick R, Farhadi A (2016) You only look once: unified, real-time object detection. In: proceedings of the IEEE conference on computer vision and pattern recognition, pp 779–788
13. Sanchez J, Sawant A, Neff C, Tahkhi H (2020) AWARE-CNN: automated workflow for application-aware real-time edge acceleration of CNNs. IEEE Internet Things J 7(10):9318–9329
14. Ahmad A, Pasha MA, Raza GJ (2020) Accelerating Tiny YOLOv3 Using FPGA-Based Hardware/Software Co-Design. In: 2020 IEEE international symposium on circuits and systems (ISCAS), IEEE, pp 1–5
15. Yu Z, Bouganis C-S (2020) A parameterisable FPGA-tailored architecture for YOLOv3-tiny. In: international symposium on applied reconfigurable computing, Springer, pp 330–344
16. Nguyen DT, Kim H, Lee H-J (2020) Layer-specific optimization for mixed data flow with mixed precision in FPGA design for CNN-based object detectors. IEEE Trans Circuits Syst Video Technol 31(6):2450–2464
17. Bozorgzadeh B, Covey DP, Heidenreich BA, Garris PA, Mohseni P (2014) Real-time processing of fast-scan cyclic voltammetry (FSCV) data using a field-programmable gate array (FPGA). In: 2014 36th annual international conference of the IEEE engineering in medicine and biology society, IEEE, pp 2036–2039
18. Xu J, Nie Y, Wang P, López AM (2019) Training a binary weight object detector by knowledge transfer for autonomous driving. In: 2019 international conference on robotics and automation (ICRA), IEEE, pp 2379–2384
19. Dinelli G, Meoni G, Rapuano E, Fanucci L (2020) Advantages and limitations of fully on-chip CNN FPGA-based hardware accelerator. In: 2020 IEEE international symposium on circuits and systems (ISCAS), IEEE, pp 1–5
20. Wang Z, Xu K, Wu S, Liu L, Liu L, Wang D (2020) Sparse-YOLO: hardware/software co-design of an FPGA accelerator for YOLOv2. IEEE Access 8(2020):116560–116585
21. Li S, Luo Y, Sun K, Yadav N, Choi KK (2020) A novel FPGA accelerator for real-time and ultra-low power deep convolutional neural networks compared with titan X GPU. IEEE Access 8(2020):105455–105471
22. Gschwend D (2020) Zynqnet: an fpga-accelerated embedded convolutional neural network. arXiv preprint arXiv:200506892
23. Xu K, Wang X, Liu X, Cao C, Li H, Peng H, Wang D (2021) A dedicated hardware accelerator for real-time acceleration of YOLOv2. J Real-Time Image Proc 18(3):481–492
24. Nguyen DT, Nguyen TN, Kim H, Lee H-J (2019) A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection. IEEE Trans Very Large Scale Integr (VLSI) Syst 27(8):1861–1873
25. Ding C, Wang S, Liu N, Xu K, Wang Y, Liang Y (2019) REQ-YOLO: a resource-aware, efficient quantization framework for object detection on FPGAs. In: proceedings of the 2019 ACM/SIGDA international symposium on field-programmable gate arrays, pp 33–42
26. Jacob B, Kligys S, Chen B, Zhu M, Tang M, Howard A, Adam H, Kalenichenko D (2018) Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: proceedings of the IEEE conference on computer vision and pattern recognition, pp 2704–2713
27. Yang Y, Deng L, Wu S, Yan T, Xie Y, Li G (2020) Training high-performance and large-scale deep neural networks with full 8-bit integers. Neural Netw 125(2020):70–82
28. Abdiyeva K, Tibevey T, Lukac M (2020) Capacity limits of fully binary CNN. In: 2020 IEEE 50th international symposium on multiple-valued logic (ISMVL), IEEE, pp 206–211
29. Guan Y, Liang H, Xu N, Wang W, Shi S, Chen X, Sun G, Zhang W, Cong J (2017) FP-DNN: an automated framework for mapping deep neural networks onto FPGAs with RTL- HLS hybrid templates. In: 2017 IEEE 25th annual international symposium on field-programmable custom computing machines (FCCM), IEEE, pp 152–159
30. Redmon J, Farhadi A (2017) YOLO9000: better, faster, stronger. In: proceedings of the IEEE conference on computer vision and pattern recognition, pp 7263–7271
31. Everingham M, Van Gool L, Williams CK, Winn J, Zisserman A (2010) The pascal visual object classes (voc) challenge. Int J Comput Vision 88(2):303–338
32. Redmon J (2018) yolov2-voc.cfg. https://github.com/pjreddie/darknet/blob/master/cfg/yolov2-voc.cfg
33. Joseph R (2016) YOLO: real-time object detection. https://pjreddie.com/darknet/yolov2/
34. Stanisz J, Lis K, Gorgon M (2021) Implementation of the pointpillars network for 3D object detection in reprogrammable heterogeneous devices using FINN. J Signal Process Syst, 1–16
35. Zhang N, Wei X, Chen H, Liu W (2021) FPGA implementation for CNN-based optical remote sensing object detection. Electronics 10(3):282
36. Wang J, Gu S (2021) FPGA implementation of object detection accelerator based on Vitis-AI. In: 2021 11th international conference on information science and technology (ICIST), IEEE, pp 571–577
37. Kusyk J, Saeed SM, Uyar MU (2021) Survey on quantum circuit compilation for noisy intermediate-scale quantum computers: artificial intelligence to heuristics. IEEE Trans Quant Eng 2(2021):1–16
38. Adaptable & real-time AI inference acceleration. (2022). https://github.com/Xilinx/Vitis-AI

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.