This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

A Full Featured Configurable Accelerator for Object Detection with YOLO

DANIEL PESTANA[1], PEDRO R. MIRANDA[1], JOÃO D. LOPES[1], RUI P.DUARTE[1],(Member, IEEE), MÁRIO VÉSTIAS[2],(Member, IEEE), HORÁCIO C. NETO[1], AND JOSÉ T. DE SOUSA[1],(Member, IEEE)

[1]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, 1000-029 Lisboa, Portuga
[2]INESC-ID, Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa, 1500-310 Lisboa, Portugal

Corresponding author: Mário Véstias (e-mail: mario.vestias@isel.pt).

"This work was supported in part by the Fundação para a Ciência e Tecnologia (FCT) under Grant UIDB/50021/2020, and in part by the projects PTDC/EEI-HAC/30848/2017, through INESC-ID and IPL/IDI&CA/2020/TRAINEE/ISEL through Instituto Politécnico de Lisboa."

ABSTRACT Object detection and classification is an essential task of computer vision. A very efficient algorithm for detection and classification is YOLO (You Look Only Once). We consider hardware architectures to run YOLO in real-time on embedded platforms. Designing a new dedicated accelerator for each new version of YOLO is not feasible given the fast delivery of new versions. This work's primary goal is to design a configurable and scalable core for creating specific object detection and classification systems based on YOLO, targeting embedded platforms. The core accelerates the execution of all the algorithm steps, including pre-processing, model inference and post-processing. It considers a fixed-point format, linearised activation functions, batch-normalisation, folding, and a hardware structure that exploits most of the available parallelism in CNN processing. The proposed core is configured for real-time execution of YOLOv3-Tiny and YOLOv4-Tiny, integrated into a RISC-V-based system-on-chip architecture and prototyped in an UltraScale XCKU040 FPGA (Field Programmable Gate Array). The solution achieves a performance of 32 and 31 frames per second for YOLOv3-Tiny and YOLOv4-Tiny, respectively, with a 16-bit fixed-point format. Compared to previous proposals, it improves the frame rate at a higher performance efficiency. The performance, area efficiency and configurability of the proposed core enable the fast development of real-time YOLO-based object detectors on embedded systems.

INDEX TERMS Object detection, Convolutional Neural Network, FPGA, Lightweight YOLO.

I. INTRODUCTION

Object detectors have a wide range of application fields such as security, transportation, military and medical. Their task consists of classifying and locating multiple objects in an image from predefined categories. Object detection has been under extensive research in both academia [1], [2] and real-world applications [3], [4]. Traditional approaches use handcrafted low-level features and shallow trainable architectures [5].

Recent technological breakthroughs led to the fast evolution of object detectors. The main contributions include the development of Deep Neural Networks (DNNs), mostly Convolutional Neural Networks (CNNs), and the increase of hardware computing power. State-of-art object detectors use deeper CNN models to learn more complex features without designing them manually. There are two object detectors categories: two-stage detectors based on region proposal and one-stage detectors based on regression/classification.

Two-stage detectors follow the traditional object detection pipeline by scanning the whole scenario and then focusing on regions of interest. The first stage generates region proposals, known as candidate bounding boxes, and the second stage extracts features from each candidate box to perform the classification and bounding box regression tasks. The most popular two-stage detectors are R-CNN [6], Fast R-CNN [7], R-FCN [8], Faster R-CNN [9], and Mask R-CNN [10].

One-stage detectors treat object detection as a regression/classification problem by adopting a unified framework to obtain the labels and locations directly. These detectors map straightly from image pixels to bounding box coordinates and class probabilities. They do this by proposing predicted boxes directly from input images without the re-

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

gion proposal step. The most well-known one-stage detectors are You Only Look Once (YOLO) [11], its successors YOLOv2 [12], YOLOv3 [13] and YOLOv4 [14], and RetinaNet [15]. See [1], [16], [17] for a full review of object detectors.

The selection between one-stage and two-stage detectors resides on a choice between speed and accuracy. Two-stage detectors provide higher localisation and object recognition accuracy, while one-stage detectors achieve higher inference speed.

Among the several object detectors, YOLO is the one that presents the best trade-off between accuracy and execution time. However, despite its good results and lower complexity compared to other object detectors, YOLO still has high computational requirements because of its core neural network model complexity.

Graphics Processing Units (GPUs) have been the most common programmable accelerators for executing DNNs due to their high parallelism and high-speed floating-point computing power. However, their high energy consumption and die area limits the usability of GPUs in embedded platforms.

The lightweight YOLO versions Tinier-YOLO, Tiny-YOLOv3 and Tiny-YOLOv4 have fewer parameters and computations than the full versions and show some accuracy reduction. However, for deploying them on embedded computing platforms and achieving real-time detection, architectures with high performance and low energy consumption are required.

Recent studies [18], [19] have been using Field Programmable Gate Arrays (FPGAs) as a more energy-efficient alternative to GPUs for executing DNNs. FPGAs provide advantages such as high dedicated hardware design flexibility, fixed-point calculation, parallel computing and low power consumption. Besides, unlike application-specific integrated circuits, FPGAs can follow the fast evolution of object detectors.

The problem is that designing a new dedicated accelerator for a new version of YOLO is a time-consuming process. Instead, we consider that a parameterisable and algorithm oriented hardware core is the solution for designing these real-time object detectors.

This paper proposes a configurable IP core to efficiently execute complete object detector systems based on YOLO's lightweight versions. Contrary to YOLO's previously proposed dedicated accelerators that only consider a particular model and only accelerate the IP core CNN, the currently proposed IP core is easily configurable to support different versions of YOLO. Moreover, it considers different frame rates and accelerates the pre- and post-processing steps. Our all-Verilog IP core is portable to FPGA devices from all vendors and Application-Specific Integrated Circuits (ASICs).

As a test case, the IP core was integrated into a complete hardware/software System-on-Chips (SoCs) solution using a minimal RISC-V processor to control the algorithm and the hardware accelerator. The accelerator has been configured to run Tiny-YOLOv3 and Tiny-YOLOv4 in real-time, with performance over 30 frames per second.

We prototyped the complete system using an UltraScale XCKU040 FPGA. While running Tiny-YOLOv3 for 768×576 images, the solution achieves a performance of 32 frames per second, with the hardware accelerator executing with a 143 MHz clock frequency. For Tiny-YOLOv4, the system analyses 31 frames per second. Compared to previous dedicated solutions, the proposed IP core is significantly more competitive and easy to design.

This document is organised as follows. Section 2 introduces the background of CNNs, object detection with YOLO and FPGA-based solutions. Section 3 describes the architecture of the IP core developed to accelerate YOLO. Section 4 presents the final solution's performance results in terms of resource consumption, the detectors' execution time, and the comparison with other FPGA-based works. Finally, Section 5 concludes the work and highlights the significant achievements and suggestions for future work.

II. BACKGROUND AND RELATED WORK

A. CONVOLUTIONAL NEURAL NETWORKS

CNNs consist of a sequence of interconnected layers implementing two main stages: feature extraction and classification, as shown in Figure 1. The stages realise convolutional, pooling and fully connected layers. The network comprises repeated blocks for feature extraction, each composed of a convolutional layer, an optional batch-normalisation layer, a non-linear layer with an activation function, and an optional pooling layer. For classification purposes, fully connected layers, optionally followed by a regression function, are typically applied after the last block of the feature extraction stage. Modern CNN models add other types of layers such as shortcut, route and upsample layers.

Convolutional layers perform 3D convolutions, which is equivalent to a set of 2D convolutions. In 2D convolutions, a 2D kernel is overlapped and shifted as a sliding window throughout the entire 2D input feature image, generating a 2D Output Feature Map (OFM). Each overlap performs a Multiply-ACcumulate (MAC) operation. Padding, which consists of adding new elements around the Input Feature Map (IFM) edges, allows the output to keep the same size. Usually, zero-padding is applied.

The input of convolutional layers is a set of 2D feature maps, designated channels, and another set of 3D kernels, with each 3D kernel having the same number of 2D channels. Applying a 3D kernel corresponds to performing a 2D convolution between each channel of the IFM and each channel of the given 3D kernel, accumulating the convolution results across all the channels. Adding the former result with a shared bias associated with each 3D kernel creates the OFM. Therefore, each 3D kernel creates one OFM.

Batch-normalisation layers are used for speeding up the training by normalising the input data, that is, zero mean and unit standard deviation [20]. Furthermore, the normalised value is scaled and shifted. Eq. 1 expresses the computation

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

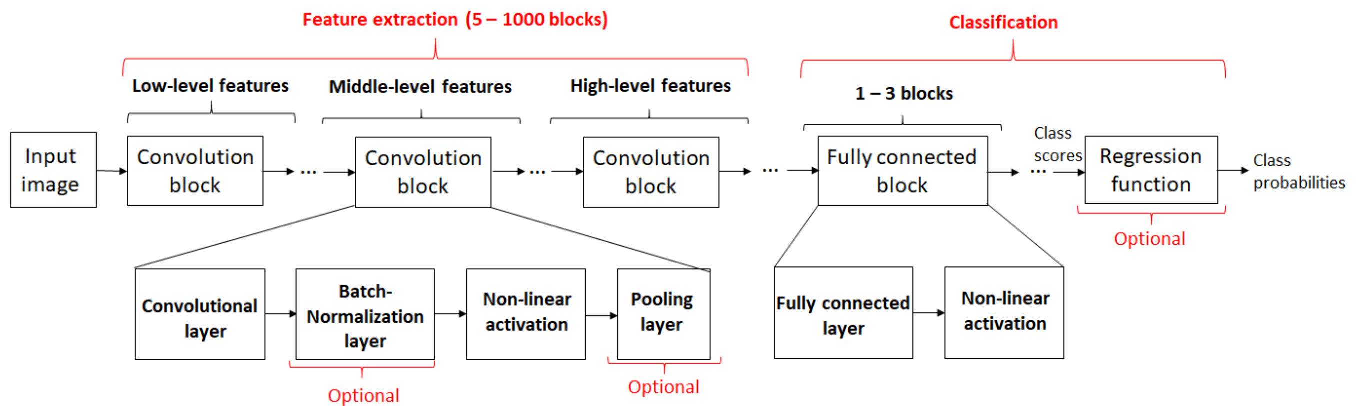


FIGURE 1. Architecture of a typical CNN

performed by this layer for each input element, x, where the mean, μ, and the variance, σ^2, are statistics collected from training. The scale factor, γ, and the shift factor, β, are parameters learned during training. ϵ is a small constant that avoids dividing by zero. The shift factor can include bias computation instead of computing it in the convolutional layer.

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \quad (1)$$

In inference, the values of μ, σ^2, γ and β are known. Thus, Eq. 1 can be reformulated as one multiplication and one addition, as shown in Eq. 2, where γ_i and β_i are the new scale and shift factors.

$$y = x \times \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} + \left(-\frac{\mu \times \gamma}{\sqrt{\sigma^2 + \epsilon}} + \beta\right) = x \times \gamma_i + \beta_i \quad (2)$$

The pooling layer downsamples the feature maps, leading to a reduction of the number of inputs in the subsequent layers. Each pooling block is replaced by the maximum (maxpooling) or the mean (average pooling) value of its activations. Some CNNs, instead of using pooling layers, apply a stride of 2 in the convolutional layers. However, pooling layers are more robust as they turn the network invariant to small shifts and distortions when downsampling [18].

The shortcut layer skips one or more layers by adding a former layer's output to the current layer's input. CNN's focused on object detection tasks use routing and upsampling layers [13]. The routing layer concatenates the output from a former layer with the current layer's input by stacking them into separate channels. This procedure allows the detection of fine-grained features, improving the localisation of small objects. The upsampling layer upsamples a feature map, typically by a factor of two, which allows to detect objects at different scales and obtain more meaningful semantic information from the features.

AlexNet [18] was one of the first CNN-based models for image classification, followed by VGG-16 [18]. Both base themselves on the architecture presented in Figure 1. They mainly differ in the number of layers and the number and size of the kernels. A more distinct model is GoogLeNet [18]. Different sized filters are convoluted in parallel for the same input feature map, and the results are further concatenated. Therefore, the input is processed at multiple scales.

ResNet [18] was the first CNN that exceeded the human-level accuracy for image classification by deploying a deeper network than the models mentioned above. Those models suffered from the vanishing gradient problem, restraining them from getting deeper. When training, after several multiplications, the gradient becomes infinitely small during backpropagation, affecting the update of the weights in early layers for profound networks. Shortcut layers were added to the network to avoid that problem.

Darknet-53 [13] is a more recent CNN model that employs the shortcut layers first introduced by ResNet to allow a deeper network. Darknet-53 is the backbone model of the YOLOv3 object detector. CSPDarknet-53 [14] extends DarkNet-53 with a split and merge strategy that improves the gradient flow through the network model. This model is used as the backbone of YOLOv4. Both of these networks have a lightweight version to reduce the number of weights and computations. We refer to them as Darknet-53-Tiny, used in YOLOv3-Tiny, and CSPDarknet-53-Tiny, used in YOLOv4-Tiny. Table 1 summarises the number of parameters and the number of multiply-accumulate operations of the CNNs mentioned above.

TABLE 1. Complexity of known convolutional neural networks.

Model	# Million Parameters	# GMAC
AlexNet	60	0.65
VGG16	138	7.80
ResNet-101	40	3.80
ResNet-152	55	5.65
Darknet-53	62.2	32.90
CSPDarknet-53	27.6	26
Darknet-53-Tiny	8.8	2.78
CSPDarknet-53-Tiny	6.5	3.75

The tiny versions of Darknet and CSPDarknet reduce the

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

complexity of the original models. The number of parameters of the tiny versions is considerably reduced compared to ResNet. However, the number of operations is only around 25% lower because the parameters are more reused during inference.

B. OBJECT DETECTION WITH YOLO

A general-purpose object detector's task is to locate and classify existing objects in an image from predefined categories. The most common way to label and output the located object's coordinates is to draw a bounding box around it. State-of-art object detectors are DNN-based, and their backbone network for feature extraction consists of networks for image classification excluding the last fully connected layers [1].

The best performance for both PASCAL VOC 2007/2012 and Microsoft COCO [16] datasets is achieved by two-stage detectors, R-FCN and Mask R-CNN [1]. Higher frame rates are achievable with one-stage detectors such as YOLO and its successors [21]. Several versions are available for the detectors, which mainly differ in the input feature maps' sizes of the first layer. However, the topology of the network is the same for any version. For instance, YOLOv3 has three versions with input feature maps of 320×320, 416×416 or 608×608. Bigger input feature maps tend to lead to higher accuracy but lower speed. YOLOv3 and recently YOLOv4 present the best trade-off between accuracy and execution time.

The YOLO detector process flow starts by resizing the input images. Then, the YOLO network extracts features using the CNN backbone and returns candidate bounding boxes from those features for three different scales: 52×52, 26×26 and 13×13. These specific scales apply to the 416x416 input case. Candidate bounding boxes are then filtered based on their objectness score and the score of each class. Finally, non-maximum suppression is used to remove multiple detections of the same object, and the final detections, bounding boxes and class labels, are drawn over the original input image.

The method scales the input image to have the exact size of the YOLO network's first layer by maintaining its aspect ratio and adding padding until meeting the desired dimensions. The input image's maximum width or height is scaled to the resized image's desired dimension, while the other dimension is scaled proportionally to keep the aspect ratio. The resize method is based on a bilinear interpolation. For each unknown pixel of the resized image, this method considers the closest 2×2 neighbourhood of available pixels from the input image surrounding the unknown pixel. It interpolates its final value by calculating the weighted average of the four available pixels.

The resizing procedure can be divided into three steps. First, the pre-processing stage determines the indexes and factors that are related to each possible pixel of the resized image. In other words, the positions and weights of the closest 2×2 pixels are determined for each unknown pixel based on the dimensions of both input and resized images. The input image's width is then resized by calculating the weighted sum of two of the closest pixels based on each unknown pixel's horizontal indexes and factors. This second step generates an intermediate image with the resized image's desired width while keeping the input image's height. For instance, a 768×576 input image is converted into a 416×576 intermediate image. The final step resizes the image's height by determining the other two closest pixels' weighted sum based on each unknown pixel's vertical indexes and factors. Finally, this converts a 416×576 intermediate image to the desired 416×312 resized image. Note that the resizing of both width and height is done for all the input image channels.

The CNN-based YOLO network is composed of convolutional, shortcut, yolo, upsample and route layers. The yolo layer applies the logistic activation in the predictions of each bounding box, excluding the width and height parameters. The route layer concatenates the output from a former layer with the input of the current layer by stacking them into different channels. There are no fully connected layers, and convolutional layers with stride two are used instead of maxpool layers. All convolutional layers include batch-normalisation and the Leaky ReLU with $a = 0.1$ as the activation function, except the layer immediately before each yolo layer, which uses a linear activation function.

Objects of various sizes are detected with different feature map scales through a structure similar to the FPN [16]. YOLO uses three scales: 52×52 to detect small objects, 26×26 to detect medium objects and 13×13 to detect big objects.

After executing the YOLO network block, there are several candidate bounding boxes for each scale. However, only a few correspond to actual detections depending on the number of objects in the image. The actual detections correspond to the bounding boxes whose product between the objectness score and each class's conditional probability is above a given threshold with a default value of 0.5. The bounding boxes can be multi labelled, i.e., can have more than one object, and the objects can belong to different classes.

Due to detecting objects with different scales and with three bounding boxes per grid cell, multiple bounding boxes of the same object might be found. The Non-Maximum Suppression (NMS) algorithm is used to remove these multiple detections [22]. After applying the NMS algorithm, the detector's post-processing phase ends by drawing the bounding boxes and respective class labels over the original input image.

In this paper, YOLOv3-Tiny and YOLOv4-Tiny are used as case studies. Compared to YOLOv3, the tiny network detects objects using only two scales: 26×26 and 13×13, downsamples the feature maps using max-pooling instead of convolutions with stride two and does not use shortcut layers. The last maxpool layer has a stride of 1 and uses half padding such that the feature map keeps its resolution.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

C. FIXED-POINT QUANTISATION

The CNN execution can be accelerated by approximating the computation at the cost of minimal accuracy drop. One of the most common strategies is reducing the precision of operations. During training, the data is typically in single-precision floating-point format. For inference in FPGAs, the feature maps and kernels can be converted to fixed-point format with less precision, typically 8 or 16 bits, reducing the storage requirements, hardware utilisation and power consumption [23].

In general, the feature maps of deeper layers tend to present a more extensive numerical range than the ones from initial layers, and the weights also tend to be much smaller than the pixels of feature maps [24]. The precision must be increased for intermediate results to prevent overflow. Thus, different precision values are typically used for weights, intermediate results, and output feature maps from different layers.

The proposed IP core considers both weights and activations represented in fixed-point format to improve the object detection throughput. The proposed architecture can be easily modified to consider integer or floating-point formats.

Weights can be directly converted to fixed-point, given a trained floating-point model. Then, the quantised network can be fine-tuned with a post-training step to improve the model's accuracy after quantisation.

The quantised models executed by the IP core can be fine-tuned or not. Different layers require different fixed-point scales. Hence, dynamic quantisation is considered where different layers are allowed to have different fixed-point scales.

To analyse the accuracy drop caused by the conversion to fixed-point, YOLOv3-Tiny and YOLOv4-Tiny detectors were run with the MS COCO 2017 test dataset, with a post-training quantised model without fine-tuning, and the mAP_{50} metric was evaluated using the CodaLab platform [25]. The results show that a 16-bit fixed-point model presents a mAP_{50} drop below 2.1 compared to the original floating-point model. Fine-tuning would most probably improve the precision of the network and would allow lower fixed-point formats. The bit width of the fixed-point representation establishes a trade-off between the area and performance of the hardware accelerator and the accuracy of the network model.

D. BATCH-NORMALIZATION

This work adopted the batch-normalization folding method proposed in [26] to implement batch-normalization. It consists of a linear transformation to fold the parameters of the batch-normalisation layer into the preceding convolutional layer, consequently reducing the number of parameters and operations of the network. As a result, the pre-trained floating-point weights w and biases b are updated to their new values w' and b' according to Eq. 3.

$$w' = \frac{\gamma \times w}{\sqrt{\sigma^2 + \epsilon}} \qquad b' = b - \frac{\mu \times \gamma}{\sqrt{\sigma^2 + \epsilon}} \tag{3}$$

Quantisation is applied to w' and b'.

E. CNN ACCELERATION WITH FPGAS

One of the main advantages of accelerating CNNs with FPGAs, rather than with a Central Processing Unit (CPU) or GPU, is the flexibility to design customised hardware to exploit various parallelism sources and use dedicated distributed on-chip buffers to support data reuse. As networks get deep, the number of operations and storage requirements increase. Thus, external memory is required, whose access results in high latency and significant energy consumption. FPGAs present a density of hard-wired Digital Signal Processing (DSP) blocks and a collection of On-Chip Memories (OCMs) that can be used to perform the MAC operations and reduce the number of external memory accesses.

The most common approaches for accelerating CNN inference in FPGAs in previous works [19] are mainly focused on exploiting the parallelism of the MAC operations of the convolutions and on approximating the model with fixed-point computation.

Typical FPGA-based CNN accelerators [27]–[31] introduce several levels of memory hierarchy. The system in [31] is composed of two on-chip input buffers, one for fetching the feature maps and the other for fetching the parameters from the external memory using a Direct Memory Access (DMA). The data is streamed into configurable cores responsible for computing the MAC operations. Each core has its on-chip registers. The on-chip output buffer stores the intermediate results and outputs the feature maps. These are transferred back, if needed, to the external memory. The CPU issues the controller's workload, which in turn generates control signals to the other modules. The multipliers and adder trees present in each core are usually pipelined to reduce the circuit's critical path and increase the throughput.

The computation of each convolutional layer can be seen as the application of four nested loops. Each loop is associated with a source of parallelism:

- intra-convolution: multiplications in 2D convolutions are implemented concurrently
- inter-convolution: multiple 2D convolutions are computed concurrently
- intra-FM: multiple pixels of a single output Feature Map (FM) are processed concurrently
- inter-FM: multiple output FMs are processed concurrently.

The sources of parallelism to be exploited depend on the convolutional layers' characteristics, like the number of channels and input feature map size, and on the FPGA resources. The architectural configuration of the cores, number of MACs and registers, and the temporal data scheduling are defined by applying loop optimisation techniques, such as loop unrolling and loop tiling [20]. According to a given unroll factor, loop unrolling parallelises the loops' execution at the expense of resource utilisation. Loop tiling divides the data into multiple blocks to increase the data locality.

The CNN execution can be further accelerated by approximating the computation at the cost of a minimal accuracy

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

drop. Two of the most common strategies are reducing the precision and number of operations. During training, data are represented in single-precision floating-point format. During inference, data can be converted to fixed-point, typically 8 or 16 bits, to reduce storage requirements, hardware utilisation, and power consumption [20].

The most common approaches to reducing the number of operations are weight pruning and low-rank approximation [32]. A fine-tuning phase often follows these methods to counterbalance the accuracy drop. Operation reduction methods are not considered in this work.

Zhang et al. [28] were one of the first to employ loop optimisation strategies to accelerate the execution of AlexNet in an FPGA. The accelerator achieves a performance of 62 Giga Operations Per Second (GOPS), using 32-bit floating-point arithmetic. Suda et al. [29] reach a performance of 187 GOPS for the same AlexNet using loop optimisation techniques alongside 16-bit fixed-point arithmetic. Since then, many other proposals have considered data quantisation and data pruning to design optimised hardware accelerators for CNN inference [19].

Considering specifically the acceleration of YOLO, Wei et al. [33] proposed an FPGA-based architecture for the acceleration of YOLOv2-tiny. The Leaky ReLU activation function is replaced with ReLU to reduce resource consumption. The authors report an inference throughput of 19 Frames Per Second (FPS) in a Zynq 7035 FPGA. Liu et al. [34] also proposed an FPGA-based accelerator of YOLOv2-tiny with a 16-bit fixed-point number representation for both inputs and outputs. The system achieves 69 FPS using an Arria 10 GX1150 FPGA.

Some works proposed various simplifications of YOLOv2 [35], [36]. The former considers a mixed algorithmic solution with convolutional layers and a Support Vector Machine (SVM). The solution achieves 40 FPS in a Zynq XCZU9EG FPGA device with a 1.5pp accuracy drop. The second adopts a pipelined CNN architecture where each layer is mapped to a dedicated hardware module. The solution was implemented in a Virtex XC7VX485T FPGA, with a sizeable on-chip memory, achieving 109 FPS at 18.3 W of power.

While YOLOv2 targets only 24 classes of objects [37], YOLOv3 targets 80 classes, thus requiring a higher number of operations per inference. Oh et al. [38] implemented the YOLOv3-tiny model on a Zynq XCZU9EG FPGA. The network is trained with a pedestrian signalling dataset, and the weights and activations are quantised with an 8-bit fixed-point. The backbone network is implemented in the programming logic, whilst the programming system handles the detection layers and the remaining functionality. The authors claim a throughput of 104 FPS without detailing the hardware architecture or resource consumption. The utilisation of an 8-bit fixed-point data representation improves the performance but reduces the precision by about 15%.

Ahmad et al. [39] apply batch-normalisation folding and post-training quantisation of 18 bits. Only the convolutions are handled in hardware, and all the other layers and activation functions are implemented in software. The hardware architecture exploits the inter-FM, inter-convolution and intra-convolution parallelisms with a total parallelism factor of 2304. The actual throughput is not reported.

Yu et al. [40] accelerate all YOLOv3-Tiny layers in hardware. They exploit intra-convolution instead of the intra-FM parallelism, and the data is quantised with 16 bits with a reduction in mAP of 2.5 pp. The solution achieves a latency of 532 ms per inference, less than 2 FPS. The high latency reduces its applicability to real-time scenarios.

This paper proposes a configurable core for the efficient execution of full object detectors based on Tiny YOLO. Contrary to previous solutions that only accelerate the CNN of a particular YOLO version, the YOLO core proposed in this work accelerates all algorithm steps, including pre-CNN acceleration (width and height resize), CNN acceleration and post-CNN acceleration (detection phase).

III. HARDWARE ACCELERATOR IP CORE

This section describes the hardware architecture designed to accelerate lightweight versions of YOLO, including the layers of the background CNN network, pre-CNN and post-CNN operations.

A. HIGH-LEVEL ARCHITECTURE

The architecture of the accelerator is composed of three main modules, *xWeightRead*, *xComp* and *AXI-DMA*, as represented in Figure 2.

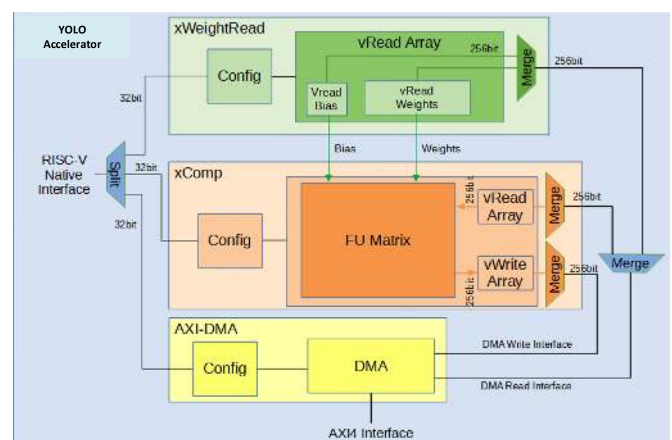


FIGURE 2. High-level architecture of the accelerator.

The **xWeightRead** stage reads weights and biases from the external memory and stores them in the on-chip memory. This stage is constituted by an array of configurable *vRead* units. A *vRead* unit is a dual-port memory. One port is used for writing the weights and biases read from the external memory via the DMA, and the other port is used for reading those values and feeding them to the *xComp* stage. The write and read addresses are generated by two independent Address Generator Units (AGUs) so that the write and read accesses can be performed simultaneously.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

The **xComp** stage computes convolutions, maxpool and activation functions and stores the results back to the on-chip memory. This stage is composed of a matrix of configurable custom computing Functional Units (FUs), where each row shares a *vRead* unit and a *vWrite* unit. The *vRead* unit is used for reading tiles of the input feature map from the external memory. The *vWrite* unit includes a dual-port memory, an internal AGU to store the computation results, and an external AGU to write the stored results to the external memory via DMA. The various *vRead* and *vWrite* units share a merge block each, which is a priority encoder for allowing DMA access to only one *vRead* and one *vWrite* at a given time.

The function of the **AXI-DMA** block is to read/write data from/to the external memory. It handles 256-bit-wide data and allows configurable bursts for both reads and writes. It has two data native interfaces, allowing the **xComp** and **xWeightRead** modules to read and write from memory and an AXI-4 interface to access the external memory. It also has a native configuration interface driven by the CPU. Like the FUs, the DMA can be configured while executing so that configurations, data transfers, and FU computing can co-occur.

Each stage has a configuration module with specific configurations shared between the same FU types within the stage. Apart from the internal configurations of each stage, there are global control and status registers that are common to all stages:

- **Run**: starts the execution of the configurations stored in the shadow registers of each stage;
- **Clear**: resets the configurations stored in the register files of each stage;
- **Done**: indicates the end of execution of all configurations of all the stages.

B. DETAILED ARCHITECTURE

The detailed architecture of the IP core is shown in Figure 3. The custom FUs are reconfigurable, allowing them to form different hardware datapaths for different computations.

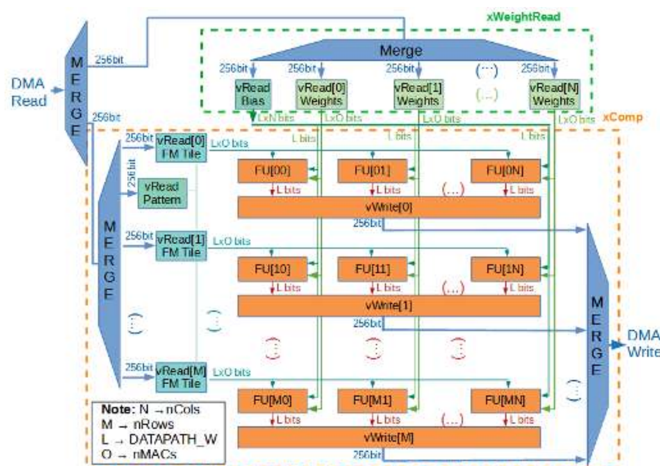


FIGURE 3. Detailed architecture of the accelerator.

Each FU in the same row receives the same feature map tile but a different 3D kernel, which corresponds to computing multiple OFM (Output Feature Map) in parallel (inter-FM parallelism), corresponding to an unrolling factor defined by `nCols`. In turn, each FU in the same column receives the same 3D kernel but a different FM tile, corresponding to the computation of multiple pixels of a single output FM in parallel (intra-FM parallelism), where the unroll factor is defined by `nRows`. Therefore, the total number of FU is `nCols x nRows`. Multiple 2D convolutions are computed inside each FU in parallel (inter-convolution parallelism), with an unroll factor defined by `nMACs`.

The data flow is the following. The *vRead* units read data from the external memory using the DMA and store them internally. Simultaneously, the data in the *vRead* units, obtained from the external memory in the previous run, is broadcast to columns or rows of FUs, depending on the type of *vRead* unit. Each custom FU computes a different 3D convolution. The computation results of the custom FUs are concatenated and stored in the *vWrite* unit. Simultaneously, the results of the previous run are written back to the external memory via the DMA.

1) xWeightRead stage

Figure 4 shows the detailed architecture of the *xWeightRead* stage, omitting the *merge* module. This module is composed of a *Bias vRead* unit and an array of *Weight vRead* units. These units read data from the external memory using their external AGUs, write these data to their internal Bias and Weights memories and, at the same time, read previous data from their internal memories to feed the compute FUs.

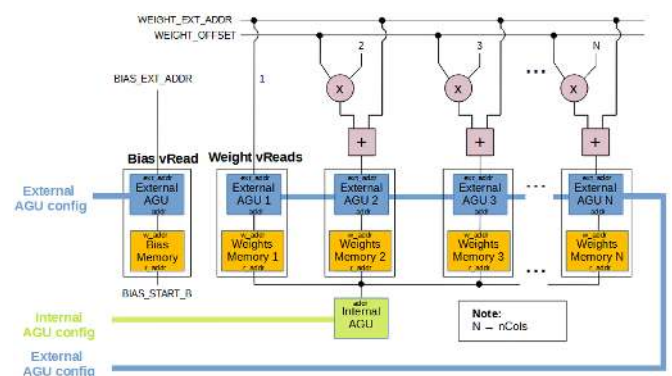


FIGURE 4. xWeightRead stage detailed architecture.

The *Weight vRead* units share the Internal AGU to read the data from the internal memory. The external AGUs are individual, as each uses a different base address value. However, their configuration is shared because only the first external AGU's base address is configurable (`WEIGHT_EXT_ADDR` runtime parameter). The base addresses of the other external AGUs are calculated in hardware by adding the product of the external AGU position and the address offset (`WEIGHT_OFFSET` runtime parameter) to the base (`WEIGHT_EXT_ADDR`). This scheme results from the same

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

convolutional layer's kernels being stored sequentially in the external memory and having the same size. Despite requiring the use of `nCols-1` multipliers in the design, performing this calculation in hardware allows keeping the configuration size independent of the number of `vReads`.

The weight memories are asymmetric dual-port memories, with an external bus of 256 bits and an internal bus of `nMACs × DATAPATH_W` bits, allowing `nMACs` weights to be read simultaneously from the same 3D kernel to perform inter-convolution parallelism. As all the custom FUs share the same bias in the same matrix column, the *Bias vRead* unit uses only a single read address defined by the `BIAS_START_B` runtime parameter, which does not require an internal AGU.

The runtime parameters of the *xWeightRead* stage are used by the internal and external AGUs that control the access pattern of the weights and bias memories. For the *vReads*, the external AGU controls the write address of the memories whilst the internal AGU controls the read address.

The communication with the external memory is done through the native external memory interface, where the address is calculated by adding a base value with an offset value. The communication with the internal memory is done via the native internal memory interface. The address is determined by adding a base value with an offset calculated using a sequential counter inside the external AGU.

2) xComp stage

The *xComp* stage is composed of an array of FM Tile *vRead* units, a matrix of custom FUs and an array of *vWrite* units. The *vRead* units operate similarly to the *vRead* units from the *xWeightRead* stage. Each custom FU receives a bias, weights and pixels from the FM tile to compute 3D convolutions. The *vWrite* units write the results to their internal memories and send the previous results back to the external memory.

The *vReads* for the input FM tiles are similar to the *vReads* for the weights. Namely, the base address of each external AGU is calculated with `nRows-1` multipliers; the configurations are shared between the external AGUs; and the asymmetric dual-port memories that store the FM tiles use a single internal AGU and operate as ping-pong buffers. Figure 5 shows the detailed architecture of the *xComp vReads*, for simplification omitting the calculation of the external AGU's base addresses, the *merge* module and the dataflow.

The read address of the memories can be configured to come from the internal AGU or values stored in other memory.

The detailed architecture of the custom FU is represented in Figure 6. The reconfigurable interconnections inside the FU allow forming different datapaths to accelerate different CNN layers. The FUs can be configured to form convolutional, with or without bias, or maxpool datapaths. A set of activation functions (e.g., Leaky ReLU, sigmoid) can also be configured.

The default operation is the 3D convolution performed by the MACs in parallel. Each MAC performs one tile

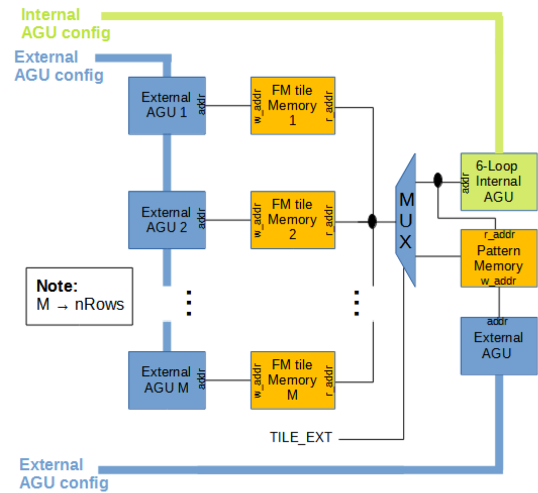


FIGURE 5. xComp vReads detailed architecture.

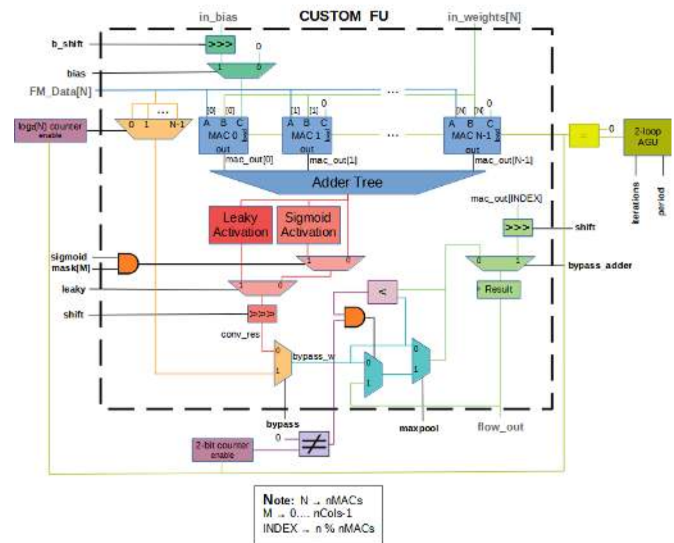


FIGURE 6. Custom FU detailed architecture.

convolutions of input channels, and an adder tree sums the convolution results across the channels. The internal AGU controls the number of accumulations to perform by resetting the accumulator of the MACs when the output address is zero. The bias can be included in the convolutions' computation by enabling the `bias` runtime parameter. The `b_shift` runtime parameter indicates the number of bits to shift the bias before the accumulation, according to its quantisation format.

The IP core implements the activation functions after the convolution. The `leaky` runtime parameter enables the leaky activation, which is implemented with two adders, one multiplexer and shifters. The `sigmoid` runtime parameter enables the sigmoid activation, which is implemented with simple comparators, multiplexers, adder/subtracters and a priority encoder. In case the sigmoid activation is not applied to all output channels, the `mask` runtime parameter provides

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

a second enable for the sigmoid computation, which is individual for each custom FU in the matrix row. After the optional activation blocks, the result is shifted considering the value of the `shift` runtime parameter and the quantisation format of the results.

The IP core is designed to compute the convolutional and maxpool layers' in the same run. The computation of the Maxpool layer, which is enabled by the `maxpool` runtime parameter, is performed with a 2-bit counter to handle 2×2 blocks of pixels, a comparator to find the maximum value in the 2×2 block and a multiplexer to select that value. The Maxpool can also be performed standalone without performing convolutions in the same run by bypassing the pixels from the FM tile to the Maxpool computation input. The `bypass_adder` runtime parameter provides the option of routing one of the MACs' result to the output of the custom FU. It is used when only needing to compute individual accumulations with a single MAC.

The runtime parameters for the *xComp vWrites* are used by the internal and external AGUs that control the access pattern of the memories that store the computation results. For the *vWrites*, the internal AGU controls the write-address of the memories whilst the external AGUs control the read address of the memories. Therefore, the `direction` parameter of the external AGUs is hard-wired to one, which indicates that the data is read from the internal memories and written into the external memory. Similarly to the *vReads*, the *vWrites* require `nRows-1` multipliers for the calculation of the base address of each external AGU, use a single internal AGU shared by all memories and share the runtime parameters between the external AGUs.

3) AXI-DMA

The AXI-DMA module converts the requests of the *vReads* and *vWrites* of the native interface to the AXI4 read and write transactions of the AXI4 interface.

For the read transactions (*vReads*), the runtime parameter defines the total number of transfers in a single run. The AXI4 protocol supports a maximum of 256 transfers per burst. Therefore, the DMA contains an internal counter, initialised at the beginning of the configuration run, that decrements each time a transfer is done. For instance, if the total number of transfers is 500, the first burst will have 256 transfers while the second burst will have 244 (`500-256`).

C. IMPLEMENTATION OF PRE- AND POST-CNN YOLO FUNCTIONS

The proposed IP core can also accelerate the pre- and post-CNN operations of the YOLO algorithm. As stated in section II-B, the pre-CNN image resizing method is divided in width and height resize and accelerated in the IP core by performing two accumulations per MAC.

For resizing the width, each row of the input image is stored in a different FM tile *vRead* and padding rows are added if the height of the input image is not multiple the number of rows of the matrix of FUs. The pixels in the *vRead* memories are addressed by the pattern memory that stores the irregular patterns of the horizontal indices instead of using the internal AGU directly. The horizontal factors are stored in the weight *vReads* unit. For each matrix row, only one custom FU is used per input channel instead of all the row's custom FUs. Unlike the kernels in the convolutional layers, there is no parallelism between horizontal factors. Also, only one MAC is used per custom FU by setting the `bypass_adder` parameter to one as the channels are not summed as in the convolutions.

The rows from the intermediate image generated by the width resize step to be stored in the FM tile *vReads* depend on the vertical indexes' values, considering the height resizing. As these values do not follow any regular pattern, only the first row of custom FUs is used, and the `ext_addr` parameter of its external AGU is determined by the base address configured in the software according to the rows selected by the vertical indexes. The vertical factors are stored in the weight *vReads*, and only one matrix row with four MACs is used. The total parallelism factor for the height resizing computation is only four.

The RISC-V soft-processor software entirely handles the post-CNN functions to identify the detections with a score above the threshold and applies non-maximum suppression. The IP accelerates the function that draws the labels and boxes when any detections are found. When drawing labels, each label pixel has to be multiplied by the RGB value computed during the setup, as each class is associated with a different RGB colour. Hence, the FM tile *vRead* stores the label and the weight *vRead* stores the RGB values. The post-CNN is bounded by communication due to irregular writes to the external memory. Therefore, only four MACs are used for the computation of the coloured labels. When drawing the boxes, only the RGB values are stored in the FM tile *vRead*, and the weight *vRead* is not used.

D. CONFIGURATION OF THE YOLO IP CORE

The synthesis parameters that determine the internal architecture of the proposed YOLO IP core are described in Table 2.

TABLE 2. Synthesis parameters of the YOLO IP core.

Parameter	Description
nCols	Number of FUs per row
nRows	Number of FUs per column
nMACs	Number of MAC per FU
DDR_ADDR_W	Address width of external memory
DATAPATH_W	Datapath width
VREAD_TILE_EXT_ADDR_W	External address width of FM tile
VREAD_BIAS_ADDR_W	Internal address width of bias memory
VREAD_WEIGHT_ADDR_W	Internal address width of weight memory
VREAD_TILE_ADDR_W	Internal address width of FM tile
VREAD_PATTERN_ADDR_W	Internal address width of vRead memory
VWRITE_ADDR_W	Internal address width of vWrite memory

The execution time for the computation of the CNN-only in the YOLO IP core can be roughly estimated according to Eq. 5, assuming a balanced overlap between communication and computation.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

$$ExecT \approx \frac{NumOperations\,of\,model}{2 \times Parallelism\,factor \times Freq} \times Eff \quad (4)$$

where the $NumOperations\,of\,model$ is the total number of operations necessary to execute an inference of the model, the $Parallelism\,factor$ is the number of MAC units performing computations in parallel in the design, and $Freq$ is the operating frequency of the circuit.

The total number of MAC units is given by:

$$NumberofMAC = nMAC \times nCols \times nRows \quad (5)$$

The efficiency factor (Eff) depends on the ratio between the number of kernels and the number of FUs per row and the ratio between the feature maps' size and the number of feature map tiles. The maximum efficiency is achieved if the size of kernels or tiles is a multiple of `nCols` and `nRows`, respectively. As expected, the execution time decreases when increasing the parallelism factor. A tool was developed to automatically determine this efficiency based on the IP core and the description of the CNN model.

The pre- and post-CNN processing depend on the size of the images, (i_x, i_y, i_z) as explained in section III-C. The pre-CNN processing includes the weight and high resize steps. The post-CNN processing takes the detections, filters the boxes, and draws the boxes and their labels. The first two are implemented in software, and the drawing phase is implemented in the accelerator.

The number of cycles to resize the image in weight is the maximum of the number of computational, wR_{comp}, and communication, wR_{comm}, cycles, according to equations 6 and 7.

$$wR_{comp} = \frac{(Ni_x \times i_y \times i_z)}{nMACs \times nRows} \quad (6)$$

$$wR_{comm} = \frac{((i_x + Ni_x) \times i_y \times i_z) \times CH \times 2}{MemoryBW} \quad (7)$$

where Ni_x is the weight of the resized image, CH is the number of channels, and $MemoryBW$ is the memory transfer rate.

The number of cycles to resize the image in height is the maximum of the number of computational, hR_{comp}, and communication, hR_{comm}, cycles, according to equations 8 and 9.

$$hR_{comp} = \frac{(Ni_x \times Ni_y \times i_z)}{nMACs} \quad (8)$$

$$hR_{comm} = \frac{((i_x) \times (i_y + Ni_y) \times i_z) \times CH \times 2}{MemoryBW} \quad (9)$$

where Ni_y is the height of the resized image.

The execution time of the post-CNN processing is dependent on the number of labels and boxes. The process is communication bounded with irregular write accesses to external memory.

The address widths of the on-chip memories are found after deciding the size of each type of memory. The size of weight memories is chosen as twice the largest filter size to allow ping-pong buffering. The size of the tile memories is chosen to permit storing a tile of the input feature map necessary to calculate at least a line of the output feature map. The tiles cover the entire width of the input FM for the layers where the FM tile memories are not used in a ping-pong fashion, as they are transferred only once or twice depending on the scale. Thus, unlike the other memories in the design, FM tile memories do not need to have twice their data size due to the ping-pong operation.

The address width of the output memories depends on the number of results computed per tile in each run, which depends on the tiling factor chosen for the width of the tiles. The memory is doubled to work in ping-pong. The bias memory size must have enough space to store all bias associated with the filters present in the weight memories.

IV. RESULTS

This section reports the results obtained with the proposed IP core, configured to run the YOLOv3-Tiny and the YOLOv4-Tiny detectors. The development board used to evaluate this work is the Kintex UltraScale KU040 [41]. The synthesis parameters used to determine the proposed IP core's internal architecture are shown in Table 3.

TABLE 3. Synthesis parameters chosen for the IP core for the two versions of Tiny-YOLO.

Parameter	Tiny-YOLOv3	Tiny-YOLOv4
nCols	16	24
nRows	13	13
nMACs	4	4
DDR_ADDR_W	32	32
DATAPATH_W	16	16
VREAD_TILE_EXT_ADDR_W	15	15
VREAD_BIAS_ADDR_W	3	3
VREAD_WEIGHT_ADDR_W	14	14
VREAD_TILE_ADDR_W	15	15
VREAD_PATTERN_ADDR_W	10	10
VWRITE_ADDR_W	8	8

The number of `nCols`, `nRows` and `nMACs` were chosen to achieve a target frame rate around 30, according to the equations of section III-D. Also, the number of `nRows` was chosen to best map the size (multiples of 13) of the feature maps of YOLOv3-Tiny and YOLOv4-Tiny. The pre- and post-CNN processing have an estimated execution time of 7 ns.

The dimension of the tile memory was chosen to accommodate the biggest FM tile with 32256 pixels. Therefore, `VREAD_TILE_ADDR_W` is 15. As each pixel is 16 bits, each FM tile memory requires 64kB.

The size of the output memories was chosen to store the maximum number of results per tile (104). As these

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

memories always work in ping-pong fashion, their size must be doubled; hence, `VWRITE_ADDR_W` is 8.

The weight memory was chosen to store the largest 3D kernel with a total of 4608 weights. Since this memory also works in a ping-pong fashion, twice this size must be considered; therefore, `VREAD_WEIGHT_ADDR_W` is 14. The weights are also represented in 16 bits, so each weight memory requires 32Kb.

The YOLO core accelerator was integrated into a RISC-V based SoC, and the whole system was called SoC-YOLO. The system uses a low-performance RISC-V soft-processor to control the memory sub-system and peripherals. The peripherals' set includes a boot controller, internal memory to store the firmware, external memory to store the image and weights, timer to measure the application's time performance, UART for the bootloader and debugging and Ethernet to load images to the board. Both the soft-processor and the IP core main hardware components operate with a clock frequency of 143MHz.

Table 4 presents the resource consumption of the SoC-YOLO system-on-chip, in terms of the FPGA primitives. As highlighted, most of the resources are occupied by the hardware accelerator.

TABLE 4. Resource consumption of the SoC-YOLO accelerator for different configurations.

Component	BRAM	FF	LUT	DSP
AXI Interconnect	0	9,887	3,442	0
DDR4 Controller	25.5	11,918	9,697	3
RISC-V CPU	0	902	2,569	4
Internal memory	17	41	60	0
AXI Cache	1	592	629	0
YOLO IP core v3	339	86,319	103,655	832
YOLO IP core v4	403	124,761	146,820	1248
Ethernet	1	382	193	0
UART	0	89	86	0
Timer	0	130	2	0
Others	0	728	480	0
Total v3	383.5	110,988	138,946	839
Total v4	447.5	149430	182,111	1255

Each DSP Block implements one multiply-accumulate operation. Therefore, the peak performance of the system totals $\texttt{nCols}\times\texttt{nRows}\times\texttt{nMACs}\times 2\times freq = 238$ GOPS for Tiny-YOLOv3 and 357 GOPS for Tiny-YOLOv4. The expected efficiency running the CNN was determined as 96 % for the first model and 82% for the second.

A. PERFORMANCE ANALYSIS RUNNING YOLOV3-TINY

The software baseline of the YOLO application was divided into four sections: peripherals initialisation and preparation of the data in the external memory, pre-CNN, CNN and post-CNN. The software-only version's execution time running on the RISC-V soft-processor at 143MHz is 969 seconds (above 16 minutes).

The execution time of the YOLOv3-Tiny detector implemented in the SoC-YOLO platform is detailed in Table 5. The total execution time is 30.9 ms, achieving a frame rate of 32 FPS.

TABLE 5. Performance of the YOLOv3-Tiny detector in the SoC-YOLO platform.

Section	Execution time (ms)
Width resize	1.09
Height resize	1.94
CNN (DarkNet-53-Tiny)	24.4
Get detections	1.94
Filter boxes	0.14
Draw detections	1.37
Total time (ms)	30.9

Compared to the software baseline, the pre-CNN procedure was accelerated from 1 s to only 3 ms. The post-CNN procedure's detection drawing method was improved from approximately 12 ms to nearly 1.4 ms. Both accelerations are mainly due to reducing the communication time between the FPGA and the external memory using the DMA engine inside the IP core. The highest speed-up of $40k\times$ was achieved for CNN's acceleration, from 968 seconds to only 24.4 ms, which is very close to the expected value. The expected value is the CNN number of operations over the IP peak performance times its efficiency, which results in $5.56\,GOPS/(238\,GOPS\times 0.96) = 24.3\,ms$.

Table 6 compares the execution time of the fixed-point model of the YOLOv3-Tiny detector implemented in the SoC-YOLO platform with the original floating-point model from Darknet-53-Tiny executed in both CPU and GPU. SoC-YOLO is nearly 27$\times$ faster than the CPU version and only 2$\times$ slower than the GPU version without batch, being, however, the only platform suitable for embedded systems.

TABLE 6. Performance comparison of the YOLOv3-Tiny detector in different platforms.

Platform	Time (ms)	FPS
CPU (Intel i7-8700 @ 3.2 GHz)	828.3	1.2
GPU (RTX 2080 Ti)	15.4	64.9
FPGA (SoC-YOLO)	30.9	32.4

The execution time of CNN is always constant as long as the parallelism factor is kept. On the other hand, the pre-CNN execution time depends on the input image's dimensions, whilst the post-CNN execution time depends on the number of candidates and final detections. For both software baseline and final firmware, the timing results reported were for a 768$\times$576 input image from which the detector finds five candidate boxes and four final detections.

Figure 7 shows the variation of the pre-CNN execution time according to the input image's resolution. The width-resize method is slower for wide input images, and the height-resize method increases its execution time for square input images.

In turn, Figure 8 shows how the post-CNN execution time rises when detecting more candidate boxes and final detections, which could result from the reduction of the threshold score. The increase in the detector's execution time when requiring a higher input image resolution or a lower

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

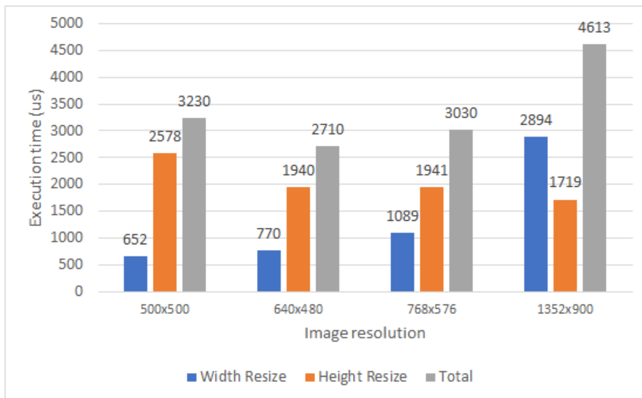


FIGURE 7. Execution time of the pre-CNN depending on the input image resolution.

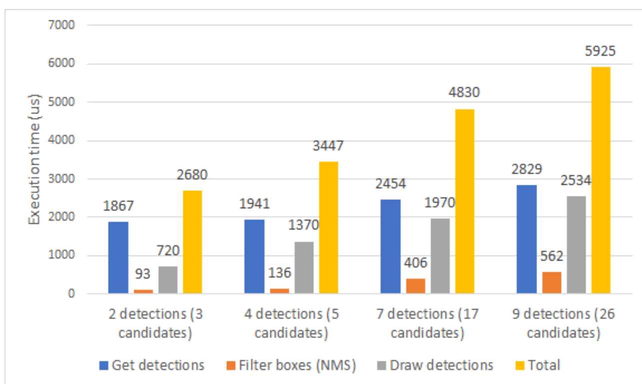


FIGURE 8. Execution time of the post-CNN depending on the number of detections.

score threshold can be compensated by reducing the CNN's execution time by applying a higher parallelism factor.

The pre- and post-CNN processing stages are independent of the number of `nCols`. Therefore, the execution times represented in figures 7 and 8 are the same for both YOLO-Tiny models.

B. PERFORMANCE ANALYSIS RUNNING YOLOV4-TINY

The execution time of the YOLOv4-Tiny detector on the SoC-YOLO platform with the proposed IP core is 32.1 ms, around 31 FPS.

SoC-YOLO runs the background CNN in 26.9 ms, close to the expected value of 25.6 ms. In this case the expected value is 7.51 $GOPS/(357\, GOPS \times 0.81) = 25.6\, ms$.

Table 7 compares the execution time of the fixed-point model of the YOLOv4-Tiny detector implemented over the SoC-YOLO platform with the original floating-point model executed in both CPU and GPU. SoC-YOLO is nearly 33× faster than the CPU version and only 1.6× slower than the GPU version and, as referred, is more suitable for embedded systems.

TABLE 7. Performance comparison of the YOLOv4-Tiny detector in different platforms.

Platform	Time (ms)	FPS
CPU (Intel i7-8700 @ 3.2 GHz)	1054.1	0.9
GPU (RTX 2080 Ti)	19.7	50.7
FPGA (SoC-YOLO)	32.1	31.2

C. COMPARISON WITH FPGA-BASED IMPLEMENTATIONS

The SoC-YOLO implementation with the proposed IP core configured for YOLOv3-Tiny was compared to previous detectors in FPGA referred in subsection II-E (see Table 8). All implementations consist of hardware/software co-design. As far as we know, there are no reported works on implementations of YOLOv4-Tiny on FPGA, so this implementation is not compared to previous works.

Oh et al. [38] claims a throughput of 104 FPS without detailing the hardware architecture or resource consumption. The hardware architecture in Ahmad et al. [39] achieved a total parallelism factor of 2304, which is over 2.5× higher than that of SoC-YOLO and would justify a higher throughput. However, the only performance metric reported is the number of MAC operations per second, calculated from the product between the number of DSPs and the frequency. The actual throughput is not reported, and therefore no fair performance comparison can be made between the two works. Yu et al. [40] execute all YOLOv3-Tiny layers in the programmable logic of a Zynq 7020 device, which has between 4× and 8× fewer hardware resources than the UltraScale XCKU040 used herein. They achieved a throughput of 2 FPS, which is about 17× lower than SoC-YOLO. Both works perform 16-bit MAC operations, and based on the MAC operations per second, SoC-YOLO has better performance and better area efficiency in LUT and DSP consumption. Despite the resources necessary for dynamically configuringSoC-YOLO, its area efficiency (MOPS/s/kLUT and MOPS/s/DSP) is about 3.2× higher compared to [40].

The SoC-YOLO platform uses a soft processor with lower performance and executes at a lower frequency than the other works' hard processor. Still, the system can achieve a real-time performance as the software development is simplified at the expense of the IP core accelerator's features. For instance, the DMA integration frees the CPU from handling data transfers to only execute the configurations of the FUs. The number of configurations is invariant to the number of FUs in the design, consequently reducing the configuration time.

All the other works only focus on the acceleration of the CNN part of the YOLOv3-Tiny detector. This work executes the detector's complete process flow by adding the image resize before the CNN and drawing the detections after the CNN.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

TABLE 8. Comparison of FPGA-based implementations of the YOLOv3-Tiny detector.

	[38]	[39]	[40]	SoC-YOLO
FPGA	UltraScale+ XCZU9EG	Virtex-7 XC7VX485T	Zynq 7020	UltraScale XCKU040
Freq.(MHz)	-	200	100	143
LUT (K)	-	49	26	139
BRAM	-	70	93	384
DSP	-	2304	160	839
FPS	104.2	-	1.9	32.4
FP (bits)	8	18	16	16
GOPS	-	-	10.5	180
MOPS/s/kLUT	-	-	403.8	1295.0
MOPS/s/DSP	-	-	66.3	215.5
Power (W)	-	4.81	3.36	3.87

V. CONCLUSIONS

This work proposes a new configurable hardware accelerator for the execution of the Tiny versions of YOLO. The IP core is parameterisable, taking into account the characteristics of the underlining CNN of the object detector and the available resources in the target device. Hardware optimisations such as the approximation of the activation functions, batch-normalisation, folding and post-training dynamic quantisation were considered to improve performance efficiency.

The IP core consists of a matrix of vector functional units that share configurations among the same type, an integrated DMA for fast data transfers, heterogeneous stages, automatic ping-pong memories and address generation units handling 6-level nested loops without software intervention. The custom MAC-based FUs are organised in a matrix structure to exploit Inter-FM, Intra-FM and Inter-Convolution parallelism and enhance pixel and weight sharing.

The yolo, upsample and most of the maxpool layers are executed alongside the precedent convolutional layer. The IP core also accelerates the pre-CNN procedure and the drawing of the detections in the post-CNN procedure.

The accelerator was integrated into a RISC-V-based SoC platform and then configured for real-time execution of YOLOv3-Tiny and YOLOv4-Tiny object detectors. The fixed-point models without fine-tuning induce a mAP_{50} drop of less than 2.1 compared to the original floating-point models. The system achieves a frame throughput of 32 and 31 FPS for the complete YOLOv3-Tiny and YOLOv4-Tiny detectors, showing significantly higher performance and resource efficiency than previous works.

As future work, the hardware accelerator can be improved with additional types of layers. For example, to accelerate the shortcut layers present in the YOLOv3 network, an optional adder can be placed between each two FM tile *vReads* to add pixels from different input FMs before the convolution operation.

REFERENCES

[1] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, and R. Qu, "A Survey of Deep Learning-Based Object Detection," IEEE Access, vol. 7, pp. 128 837–128 868, 2019.
[2] L. Liu, W. Ouyang, X. Wang, P. W. Fieguth, J. Chen, X. Liu, and M. Pietikäinen, "Deep Learning for Generic Object Detection: A Survey," International Journal of Computer Vision, pp. 1 – 58, 2018.
[3] E. Unlu, E. Zenou, N. Riviere, and P.-E. Dupouy, "Deep learning-based strategies for the detection and tracking of drones using several cameras," IPSJ Transactions on Computer Vision and Applications, vol. 11, pp. 1–13, 2019.
[4] R. Simhambhatla, K. Okiah, S. Kuchkula, and R. Slater, "Self-Driving Cars: Evaluation of Deep Learning Techniques for Object Detection in Different Driving Conditions," SMU Data Science Review, vol. 2, no. 1, 2019.
[5] N. O. Mahony, S. Campbell, A. Carvalho, S. Harapanahalli, G. A. Velasco-Hernández, L. Krpalkova, D. Riordan, and J. Walsh, "Deep learning vs. traditional computer vision," CoRR, vol. abs/1910.13796, 2019. [Online]. Available: http://arxiv.org/abs/1910.13796
[6] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 580–587.
[7] R. Girshick, "Fast r-cnn," in 2015 IEEE International Conference on Computer Vision (ICCV), 2015, pp. 1440–1448.
[8] J. Dai, Y. Li, K. He, and J. Sun, "R-fcn: Object detection via region-based fully convolutional networks," in Proceedings of the 30th International Conference on Neural Information Processing Systems, ser. NIPS'16. Red Hook, NY, USA: Curran Associates Inc., 2016, p. 379–387.
[9] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," IEEE Trans. Pattern Anal. Mach. Intell., vol. 39, no. 6, p. 1137–1149, Jun. 2017. [Online]. Available: https://doi.org/10.1109/TPAMI.2016.2577031
[10] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in 2017 IEEE International Conference on Computer Vision (ICCV), 2017, pp. 2980–2988.
[11] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 779–788.
[12] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," in 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 6517–6525.
[13] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," 2018.
[14] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," 2020.
[15] T. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 42, no. 2, pp. 318–327, 2020.
[16] Z. Zhao, P. Zheng, S. Xu, and X. Wu, "Object Detection With Deep Learning: A Review," IEEE Transactions on Neural Networks and Learning Systems, vol. 30, no. 11, pp. 3212–3232, Nov 2019.
[17] X. Feng, Y. Jiang, X. Yang, M. Du, and X. Li, "Computer vision algorithms and hardware implementations: A survey," Integration, vol. 69, pp. 309–320, 2019, doi:10.1016/j.vlsi.2019.07.005.
[18] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," Proceedings of the IEEE, vol. 105, no. 12, pp. 2295–2329, Dec 2017.
[19] M. P. Véstias, "A survey of convolutional neural networks on edge with reconfigurable computing," Algorithms, vol. 12, no. 8, 2019. [Online]. Available: https://www.mdpi.com/1999-4893/12/8/154
[20] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating CNN inference on FPGAs: A Survey," 2018.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

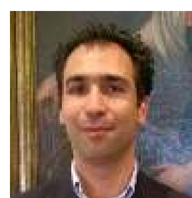
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

[21] X. Feng, Y. Jiang, X. Yang, M. Du, and X. Li, "Computer vision algorithms and hardware implementations: A survey," Integration, vol. 69, pp. 309–320, 2019.
[22] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2015.
[23] A. Gonçalves, T. Peres, and M. Véstias, "Exploring data size to run convolutional neural networks in low density fpgas," in Applied Reconfigurable Computing, C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz, Eds. Cham: Springer International Publishing, 2019, pp. 387–401.
[24] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A Survey of FPGA-Based Neural Network Accelerator," 2017.
[25] CodaLab, "COCO Detection Challenge (Bounding Box)," https://competitions.codalab.org/competitions/20794#participate, 2020.
[26] P. Kluska and M. Zieba, "Post-training Quantization Methods for Deep Learning Models," in Intelligent Information and Database Systems, N. T. Nguyen, K. Jearanaitanakij, A. Selamat, B. Trawiński, and S. Chittayasothorn, Eds. Cham: Springer International Publishing, 2020, pp. 467–479.
[27] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, and et al., "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2016, p. 26–35.
[28] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks," in Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2015, p. 161–170.
[29] 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 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2016, p. 16–25.
[30] 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 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '17. ACM, 2017, pp. 45–54.
[31] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto, "A fast and scalable architecture to run convolutional neural networks in low density fpgas," Microprocessors and Microsystems, vol. 77, p. 103136, 2020.
[32] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto, "Fast convolutional neural networks in low density fpgas using zero-skipping and weight pruning," Electronics, vol. 8, no. 11, 2019. [Online]. Available: https://www.mdpi.com/2079-9292/8/11/1321
[33] G. Wei, Y. Hou, Q. Cui, G. Deng, X. Tao, and Y. Yao, "Yolo acceleration using fpga architecture," in 2018 IEEE/CIC International Conference on Communications in China (ICCC), 2018, pp. 734–735.
[34] X. Xu and B. Liu, "Fclnn: A flexible framework for fast cnn prototyping on fpga with opencl and caffe," in 2018 International Conference on Field-Programmable Technology (FPT), 2018, pp. 238–241.
[35] 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 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 31–40.
[36] D. T. Nguyen, T. N. Nguyen, H. Kim, and H. Lee, "A high-throughput and power-efficient fpga implementation of yolo cnn for object detection," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 27, no. 8, pp. 1861–1873, 2019.
[37] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results," http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.
[38] S. Oh, J. H. You, and Y. K. Kim, "Implementation of Compressed YOLOv3-tiny on FPGA-SoC," in 2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia), 2020, pp. 1–4.
[39] A. Ahmad, M. A. Pasha, and G. J. Raza, "Accelerating Tiny YOLOv3 using FPGA-Based Hardware/Software Co-Design," in 2020 IEEE International Symposium on Circuits and Systems (ISCAS), 2020, pp. 1–5.
[40] Z. Yu and C.-S. Bouganis, "A Parameterisable FPGA-Tailored Architecture for YOLOv3-Tiny," in Applied Reconfigurable Computing. Architectures, Tools, and Applications, F. Rincón, J. Barba, H. K. H. So, P. Diniz, and J. Caba, Eds. Cham: Springer International Publishing, 2020, pp. 330–344.
[41] Avnet, "Kintex UltraScale KU040 Development Board: Hardware User Guide," December 2015.

DANIEL PESTANA received the MSc in Electrical and Computer Engineering from Instituto Superior Técnico. Currently working as hardware and embedded systems engineer at Deimos Engenharia for space mission projects. His main fields of interest include digital system design, hardware acceleration, FPGA design, embedded systems and microcontroller programming.

PEDRO R. MIRANDA received the MSc in Electrical and Computer Engineering from Instituto Superior Técnico. He is currently developing reconfigurable computing architectures for Real-Time SAR imagery. His main interest is digital system design, with an emphasis on reconfigurable computing.

JOÃO D. LOPES is a PhD student of electrical and computer engineering at the Department of Electrical and Computer Engineering of the Instituto Superior Técnico (IST), University of Lisbon and a junior researcher at the INESC-ID research institute in Lisbon, Portugal, working on reconfigurable computing with seven published technical papers.



RUI P. DUARTE received his PhD from Imperial College London, UK, in 2014. He is now a researcher at the Electronic Systems Design and Automation (ESDA) research group at INESC-ID in Lisbon, Portugal. His research interests include reconfigurable computing, fault-tolerant and low-power architectures.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3081818, IEEE Access

MÁRIO P. VÉSTIAS received the PhD in electrical and computer engineering in 2002 from the Technical University of Lisbon, Portugal. He is a Coordinate Professor at the Polytechnic Institute of Lisbon, School of Engineering (ISEL), Department of Electronics, Telecommunications and Computer Engineering (DEETC), where he is responsible for undergraduate and graduate courses on computer architecture and digital systems design. He is also a senior researcher at the ESDA (Electronic Systems Design and Automation) group at INESC-ID in Lisbon. His research interests include Computer Architectures and Digital Systems for Embedded Reconfigurable Computing.

HORÁCIO C. NETO is an Associate Professor at the University of Lisbon, School of Engineering (IST), Department of Electrical and Computer Engineering (DEEC). He is responsible for the Electronic Systems Design and Automation (ESDA) research group at INESC-ID, a research institute associated with IST. His main research interests are Digital Systems Design and Computer Architecture, with an emphasis on Reconfigurable Computing. He has a PhD in Electrical and Computer Engineering from the Technical University of Lisbon.

JOSÉ T. DE SOUSA is a Lecturer at the Department of Electrical and Computer Engineering at the University of Lisbon, School of Engineering (IST), and a senior researcher at INESC-ID, a research institute associated with IST (1999-present). He is also a tech entrepreneur in the area of Semiconductor Intellectual Property, having founded and managed three companies: Coreworks (2001-2013, co-founder and CEO), IPbloq (2017-2019, co-founder and CEO), IObundle (2018-present, owner and founder). His main interests are Digital Systems Design and Computer Architecture, with an emphasis on Reconfigurable Computing. He holds four international patents, is co-author of one book and has published more than 70 technical papers in international journals and conferences. He was General Chair of the 2013 Field Programmable Logic and Applications Conference, co-editor of its proceedings and a related special issue on the IEEE Transactions on Computers journal.

• • •

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/