FPGA Implementation of a CNN for Oriented Object Detection in Aerial Images

Francisco Carrilho
Instituto Superior Técnico,
Universidade de Lisboa
Email: francisco.carrilho@tecnico.ulisboa.pt

***Abstract*—The objective of this work is to design and implement a hardware/software system for oriented object detection of aerial images. The system is based on a convolutional neural network (CNN) detector and is aimed at processing one aerial image per second using a system-on-chip field- programmable gate array (SoC FPGA).**

Object detection in aerial images and videos is an important and challenging computer vision problem with important real-world applications such as emergency rescue, disaster relief, and surveillance. Oriented object detection considers both the position of the object and its rotation angle or orientation which makes detections significantly more accurate but more computationally intensive. Most target applications must be locally computed on unmanned aerial vehicles (UAVs), which requires implementing efficient solutions on edge devices, namely SoC FPGAs.

The hardware/software system implements an optimized version of an oriented object detection model based on the YOLO object detection algorithm. The original YOLO model was optimized and quantized, with both weights and activations represented with a specific 8-bit fixed-point format, to provide an efficient hardware-friendly solution. The system is composed by a dedicated hardware accelerator, which accelerates the inference of the main layers of the CNN model by executing 256 multiply-accumulate operations in parallel, and by a software processor that executes the less computing intensive functions. The final hardware/software system, implemented in a Zynq SoC FPGA, executes the inference of the R-YOLOv4 with a frame rate close to 1 FPS and a power consumption of only 7.3 W.

***Index Terms*—Object Detection, Quantization, Convolutional Neural Network, FPGA, Hardware/Software Codesign, Hardware Acceleration.**

I. Introduction

Object detection images or videos is an important and challenging computer vision problem with a wide range of applications, like surveillance, medical healthcare, and autonomous driving. In aerial images, object detection has become more important with the rise of the Unmanned Aerial Vehicle (UAV) and has many real-world applications such as emergency rescue, disaster relief, urban planning, surveillance, weather prediction, etc. [14].

Advances in deep learning technology have led to the creation of Convolutional Neural Network (CNN) based models for object detection whose accuracy surpasses the classical computer vision solutions [9] and are being used for research in aerial sensing using UAVs [10].

State-of-the-art object detectors are usually run on Central processing unit (CPU) or Graphics processing unit (GPU) that, for edge computing, namely in UAVs, are too expensive and energy inefficient. Therefore, for this kind of application, the use of field-programmable gate arrays (FPGAs) can be an efficient alternative even if, due to the limited resources, a compromise between performance and accuracy would need to be found in order for one of these systems to be deployed in a low-end FPGA.

The objective of this work is the implementation of a state-of-the-art CNN-based object detector in an System-on-chip (SoC) FPGA able to perform real-time object detection in aerial images.

State-of-the-art object detection models will be studied, in particular models oriented for object detection in aerial images. These models are based on CNNs and therefore it is important to study their structures and how they can be implemented in a SoC FPGA. Afterward, the selection, optimization, training, and quantization of the selected model is done. The training is performed on a machine with a high-performance GPU to mitigate the time spent on this time-consuming task. Furthermore, the hardware accelerator is designed in hardware and validated. The hardware accelerator is then integrated in a SoC architecture and mapped in a SoC FPGA. The embedded software necessary to for the network deployment will be developed and then, the HW/SW system is validated and tested.

This paper is organized as follows. Section 2 introduces deep learning concepts, in particular CNNs, reviews the state-of-the-art object detection solutions based on deep-learning, strategies, and techniques that help to implement CNN on an FPGA and CNN accelerators. Section 3 describes the design and optimizations done of the object detection model selected for this project, the dataset used for training and testing, details about the software tools used and the development workflow. Section 4 presents the design of the hardware accelerator that will be integrated in the FPGA. Section 5 presents the integration of the hardware accelerator into the embedded system, creating the HW/SW system that will run the model in section 2. The systems performance results are also presented. Finally, section 6 concludes the work and proposes future development to further improve the project.

II. Background and State of the Art

A. Convolutional Neural Network

A convolutional neural network (CNN) is one of the many types of Deep Neural Network (DNN). Similarly to other types

of neural networks, a CNN consists of multiple layers of interconnected nodes. However, the CNN is specifically designed to process grid-like structured data making it especially good when used for image analysis and pixel data processing.

A CNN is composed of three key layers: convolutional, pooling, and fully connected. There are some other layers that can be added to help improve the model and the training in various ways such as batch normalization and upsampling layers. All of these can and are usually followed by an activation function.

The convolutional layer applies the convolution operation on the input map. In the case of a 3D convolution the input is no longer a simple 2D matrix but instead, a set of 2D maps. Each of these maps is then passed by a group of filters which are basically 3D tensors composed of one kernel for each of the channels of the image.

A pooling layer reduces the spatial dimensions of the input data, reducing the number of parameters by applying a pooling operation to it. A pooling operation sweeps the input with a kernel and a given stride and performs a statistical summary of all neighboring pixels within the receptive field. The max-pooling is used in this work.

In a fully connected layer every node of the input layer is connected to every output node with an associated weight. Each output node is the result of the sum of the multiplication of every input and its respective weight and a bias term.

Activation functions are nonlinear functions that are applied to the input feature map. The most commonly used activation functions are the sigmoid, the hyperbolic tangent (tanh), the rectified linear unit (ReLU), the leaky ReLU, the exponential linear unit (ELU), and the softmax.

The shortcut layer is used to mitigate the vanishing gradient problem, which happens in deeper DNN models, where the gradient gets smaller and smaller making it difficult for the network to learn. Shortcut connections skip one or more layers and then add their output to the output of a layer further ahead.

Batch normalization is an optional layer that aims to normalize the inputs of each layer inputs to reduce internal covariate shift. This is achieved by initially dividing the data into mini-batches and then normalizing the values based on its average (μ) and standard deviation (σ).

Finally, the upsampling layer increases the size of the FM to be fed to the next layer. A possible upsampling strategy is replicating each pixel into a 2 by 2 square, doubling the height and width of the input FM.

B. CNN models for Object Detection

Object detection is a computer vision problem that aims to determine the location and identity of a given object in an image or video. This problem can be divided into three stages: region selection, feature extraction, and object classification [17]. There are a variety of deep learning-based solutions for this problem, including one-stage and two-stage detectors accompanied by various neural network architectures.

These models tend to detect objects by placing bounding boxes around them and classifying them within the correct category. These bounding boxes can be simple horizontal bounding boxes (HBBs) or oriented bounding boxes (OBBs). Both are valid representations, that are dependent on the applications. However, for aerial image object detection, OBB is more appropriate, since it allows the observer to more accurately distinguish objects in instances that are closely packed. A figurative example can be observed in Figure 1.

(a) (b)

Fig. 1. OBB(a) and HBB(b) representation comparison presented in [2].

As can be seen, the left image uses oriented boxes and so objects are easily identified In the image on the right, horizontal bounding boxes are used. In this case, boxes overlap and it is difficult to identify objects.

Object detection models are divided into two major groups: the more accurate two-stage detectors such as the R-CNN family of detectors [5, 4, 16]; and the faster but less precise one-stage detectors such as the YOLO [13, 11, 12, 1], the SSD [8] and the RetinaNet [7]. These models' use cases depend on the application. Typically, two-stage object detectors are used when accuracy is the most important aspect. One-stage object detectors are used in embedded devices where some accuracy must be traded-off for performance.

Two-stage detectors divide the object detection problem into two steps. In the first step, a model for feature extraction and region proposals is used, while in the second step, each region previously proposed is fed to a classifier architecture to determine if and what objects are present in said region. One-stage object detectors are designed to solve the same problem but in a single forward pass through a single network that is used for all stages of the object classification problem.

One of the most used one-stage object detector is YOLO [13, 11, 12, 1]. The YOLO model has had many versions and one of its adaptations is the YOLOv4, which introduces some new key features and improvements over the previous versions. This version not only introduces a new backbone but also uses a variety of techniques that either improve performance without introducing further inference cost or accuracy, but have an impact on inference time.

The R-YOLOv4 is a YOLOv4 modification proposed in [6] that allows the model to make accurate predictions with OBB instead of HBB. This is achieved by adding the rotation angle, θ, as an additional attribute to the bounding boxes and adding more anchor boxes at different rotating angles.

When compared to other models, R-YOLOv4 shows superior accuracy. Several models were tested with aerial images from the UCAS-AOD [18] dataset which is used for Object detection in aerial images (ODAI) (see Table I).

As can be seen from the table, R-YOLOv4 is the most accurate model for oriented object detection among the compared models.

TABLE I
mAP RESULTS ON UCAS-AOD DATASET ADAPTED FROM [3] AND [6]

Model	$mAP(\%)$
Faster-RCNN	84.26
CNN-SOSF	86.52
YOLOv2	44.63
YOLOv3	91.09
CNN-AOOF	92.42
R-YOLOv4	95.05

C. CNN implementations on FPGA

Due to their computational complexity, CNNs are usually run on GPUs or implemented in reconfigurable devices such as CGRAs or FPGAs.

Reduction of precision of the representation of the operands can be used not only to reduce data transfer overhead but also to maximize the number of operations possible for the same resources.

There are two main possible architectural approaches. The first involves mapping the entire CNN to the FPGA and all layers processed in a full dataflow. This approach is certainly the faster of the two since it minimizes memory transfers. However, it requires either small network configurations, an extremely large FPGA or even multiple FPGAs connected since mapping all operations performed in CNN requires a large number of resources. The second approach consists of processing each layer in sequence with a single engine. The engine is configured according to the characteristics of each layer (kernel size, sizes of the input and output maps, stride, etc.). This approach, known as Configurable Layer Processor (CLP) accelerator, requires more external memory accesses but supports models with different layer configurations, and bigger CNN architectures in a less expensive FPGA device, since the resource consumption of an individual layer is much smaller.

III. DESIGN AND OPTIMIZATION OF THE ORIENTED OBJECT DETECTION MODEL

The model chosen for rotated object detection in aerial images was a modified version of the R-YOLOv4 [6]. Based on the collected information, this is the most appropriate model for the task proposed. Not only is this model based on the YOLOv4, an object detection model with good accuracy for real-time applications, but its modifications also allow it to be more accurate when detecting oriented objects, which perfectly fits the objective of the work. Some modifications are made to the model at a cost of a minor precision loss to make it better suited for hardware acceleration.

The original network contains 111 convolutional layers, each followed by one of three types of activation functions: Leaky ReLU, mish or linear (which is only applied to the output layers). Moreover, the network contains max pooling, upsampling, and shortcuts with concatenation.

To be more hardware-friendly, a few modifications were made to the original model at the cost of a small decrease in accuracy. These modifications resulted in a network that contains 66 convolutional layers all followed by the Leaky ReLU except for the output layers that have linear activation functions. Additionally, all concatenations were turned into sums, and the max pooling layers were removed. The substitution of concatenation layers with sums helped reduce the depth of various layers reducing the number of parameters and thus making the model consume less memory and have to perform less operations.

The accuracy of the original and modified versions of the RYOLOv4 are represented in table II.

.

From the results, it can be observed that the total accuracy reduces 2.5%, but the complexity (MACS - Multiply-Accumulate) reduces about $2\times$ and the number of parameters also reduces about $3\times$.

A. Tools for Training and Quantization

After the selection of the model, it has to be trained, quantized and its parameters extracted to be used by the hardware accelerator. The adopted training framework was PyTorch and the quantization framework was Brevitas. A few custom tools were developed to help export the model to hardware.

Pytorch is a python open-source machine-learning framework that allows for the development, training, and testing of machine-learning models. Even though it supports a wide variety of models, this framework is mostly used for deep learning and neural networks. These models can be defined in Pytorch as several functions, each representing a layer and where the arguments are the layer's parameters such as kernel size, stride, number of channels for the output and input, and padding.

Brevitas is a Python library that extends Pytorch, allowing for quantization-aware training to be applied to neural network models. There are quantized versions of the PyTorch layers that can be replaced in the original PyTorch model to help recreate hardware's low precision datapath during training, reducing the quantization error during inference in hardware. These layers are similar to the normal Pytorch layers, but require a quantization engine where all the quantization configuration parameters are declared.

To help the development of the accelerator, some extra tools were needed, not only to convert and integrate the original Pytorch model with a Brevitas quantized one, but also for parameter extraction and test generation. Most of these tools were originally developed in [15] and were modified to fit the requirements of this project. These tools are the Brevitas converter, the batch normalization merger, the weights extractor, and the bin file generator. All these tools were developed using the Python programming language.

The Brevitas converter is a program that takes the original Pytorch model description and automatically converts it to a Brevitas quantized model description. This tool allows for a fast quantization of the original model.

The tool has two additional scripts that allow for the conversion of the weights trained from the original model into quantized, Brevitas-compatible weights, which allows for training with pre-trained weights, instead of training the

model from scratch which speeds up the very time-consuming training process.

The batch normalization merger is a program that iterates through every layer and merges the batch normalization layer with its respective convolutional layer. This is possible because since a batch normalization layer is effectively a 1x1 convolution and there are no non-linear layers between the convolutional and batch normalization layers.

This batch normalization merging is useful because it reduces memory consumption and computational needs, further speeding up the training process. It also makes the extraction of the weights easier.

The hardware accelerator runs the inference of the model using the weights determined after training. Therefore, a program is used to extract the weights produced during training to be used by the hardware accelerator. The weights extractor tool is a program that extracts the weights from the Brevitas quantized model and stores them into binary files in a fixed point format.

The Bin File Generator developed in this work generates random arrays of any shape, determined by a configuration specification, and then stores them in a bin file in z-wise, 128-bit words. The purpose of this script was to create tests to validate the functionality of the accelerator without having to run the whole network and wait for the training to be complete. With this tool, multiple tests for different convolution configurations can be generated and then used to validate the application.

B. Design Flow

Using Pytorch, Brevitas and the custom tools, the model quantization and export tasks to generate the quantized weights for the SoC FPGA are achieved following the design flow represented in figure 2.

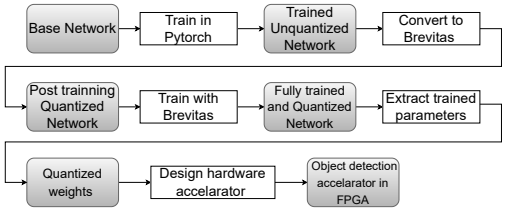

Fig. 2. Proposed project design flow.

The base model is first trained using the PyTorch framework, producing a trained model with weights and activations represented in single-precision floating-point. Then, using the Brevitas Converter tool, the model is quantized alongside its previously trained floating point weights and activations which generates a quantized Brevitas network and quantized weights and activations. Afterwards, Brevitas is used so that quantization-aware training can be applied to the network to reduce quantization error in inference. At this step, multiple quantization configurations are explored to find a good trade-off between computational complexity and accuracy. Typically, bitwidths of 2, 4, and 8 are considered for both weights and activations.

After quantization-aware training, the model is ready to have its quantized weights extracted, using the weights extractor tool, and used in the design, testing, and validation of the hardware accelerator. The hardware accelerator will be designed by developing an Intellectual Property (IP) core capable of running configurable layers. The design and development of the hardware accelerator will be discussed in Chapter 4.

C. Quantization Analysis of R-YOLOv4

To train and test the model, the UCAS-AOD dataset was chosen, which contains 1510 RGB images from Google Earth, having 2 total classes, cars and plains, with 14596 instances. Furthermore, this dataset supports OBB representation, hence matching all the dataset requirements. In the figure 3 are a few examples of images of the UCAS-AOD dataset.

Fig. 3. Example images from UCAS-AOD dataset.

Different quantizations were tried with the reduced R-YOLOv4 model with varying weights and activation bitwidths. The mAP_{50} was annotated for each quantization.

The quantization process followed the design flow described in section 3.4. The models were trained for 100 epochs with the Adam optimizer. Table III reports the results for two different quantizations: 8×8 and 8×4.

.

The accuracy of the modified model with quantization 8×8 reduced 2.2 percentual points compared to the same non-quantized model. Considering the more aggressive quantization, 8×4, the model accuracy dropped about 11 percentual points. Therefore, it was decided to consider the 8×8 quantization. Compared to the original model, the model implemented in hardware has an accuracy drop of 5.7 percentual points.

TABLE II
ORIGINAL AND MODIFIED RYOLOV4 ACCURACY.

Model	MACS	Parameters	Car $map_{50}\%$	Airplane $map_{50}\%$	$map_{50}\%$
Original RYOLOv4	55.8 G	56.7 M	78.7	96.5	87.6
Modified RYOLOv4	26.7 G	18.1 M	75.5	94.7	85.1

TABLE III
RYOLOV4 ACCURACY WITH AND WITHOUT QUANTIZATION. TWO DIFFERENT QUANTIZATIONS WERE CONSIDERED: 8×8 AND 8×4

Model	MACS	Parameters	Car $map_{50}\%$	Airplane $map_{50}\%$	All $map_{50}\%$
Original RYOLOv4	55.8 G	56.7 M	78.7	96.5	87.6
Modified RYOLOv4	26.7 G	18.1 M	75.5	94.7	85.1
Quant. Modified RYOLOv4 8×8	26.7 G	18.1 M	73.5	92.3	82.9
Quant. Modified RYOLOv4 8×4	26.7 G	18.1 M	62.9	85.4	74.2

IV. HARDWARE ACCELERATOR DESIGN

The accelerator was developed using Xilinx Vitis High level synthesis (HLS) version 2022.1, and the target FPGA was an AMD/Xilinx Zynq Ultrascale+ MultiProcessor System-On-Chip (MPSoc) XCZU7EV from Xilinx integrated in a ZCU104 board. Besides the FPGA, the board also includes 2GB of DDR4 memory and interfaces to peripherals.

Vitis HLS is a high-level synthesis tool where hardware descriptions can be done using the C/C++ programming languages as well as in built pragmas for optimization. The development process consists of producing the logic for the operations meant to be performed by the accelerator and then testing and validating their functionality through simulation. Afterwards, the design is synthesized and into an Register-transferlevel (RTL) design and its implementation is validated. Finally the RTL IP is exported and added to the Vivado's block design in order to finalize the implementation of the accelerator in the device.

A. IP Core Design

The core was designed to execute the layers in sequence, that is, a single engine is considered that can be configured according to the characteristics of each layer of the network model. The IP core is therefore configurable to support the execution of any convolutional layer present in the RYOLOv4 model. More specifically, the IP supports convolutions with kernels of size 3×3 and 1×1, and strides of 1 or 2. It also only supports the leaky ReLU activation function.

The IP explores two types of parallelism: inter-layer parallelism (where multiple output maps are generated in parallel) and intra-kernel parallelism (where multiple MAC units are used to run a single convolution in parallel). The developed architecture has 16 processing elements (PEs) to explore inter-layer parallelism each with 16 parallel MAC units. A larger number of PEs and MACs can be considered at the cost of more hardware resources. Considering that the activations are quantized with 8 bits, 16 output activations are produced in parallel and packed in a 128-bit word output to be sent to the output interface. Since each PE runs 16 multiply–accumulates (MACs) operations, it receives 16 activations and weights in parallel packed also in 128 bits. The architecture of the IP is represented in figure 4.

Before running a layer, all weights are stored in distributed on-chip memory. The higher number of weights among all layers is considered to size the local memories of the IP allocated to store the weights. These are stored in the Ultra random access memory (URAM) of the FPGA. To guarantee a single transfer of weights for each execution of a convolutional layer, the size of the local weight memories is set according

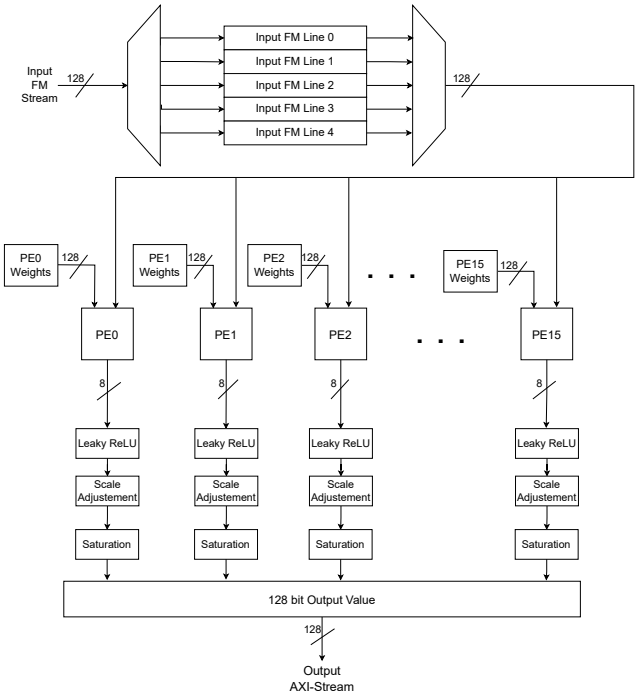

Fig. 4. Convolution IP Diagram.

to the layer with the larger number of weights. Input and output maps are only stored partially in on-chip memory since these are processed in stream. Only 5 lines from the input feature map (FM) are stored locally in Block random access memory (BRAM) and the output activations are packed and sent immediately to external memory to be read by the next layer. This on-chip storage allows the IP to process any convolution configuration while reducing the number of data transfers with the external memory which would affect performance.

Furthermore, the memory for the filters is partitioned into 16 sections, one for each PE for local PE access, allowing for simultaneous access to different sections of the filters memory. In order to ensure a PE workload balance, the filters are evenly distributed through the partitions, meaning that for a convolution with 32 filters, each PE stores locally two filters.

The IP is configured to receive both the maps and the weights in a z-wise configuration, meaning that they are sorted through their Z-axis. The easiest way to understand this data configuration is to imagine a map with $5 \times 5 \times 24$, which means that this map has 24 input channels. This map is organized so that, in the first place, the IP receives all 24 channels at position (0,0), then all the 24 channels of position (1,0), and

so on until all the 25 positions are sent. This is done by first iterating through X and then Y, meaning, all the 24 channels of a FM line are sent before going onto the next. The values are packed into 128-words and each of the map values is 8 bits meaning each bus has 16 values. For the previous example since there are 24 channels, two 128 bit words are needed to represent one pixel of the map, with the first 126-bit word being the first 16 channels and the second being the remainder 8. Since 8 activations are not enough to fill the 128-bit bus, the remainder free positions are filled with zeros, so that these useless positions will not affect the computations. In figure 5 there is a visual representation of this data organization. This representation is true for both maps and filters.

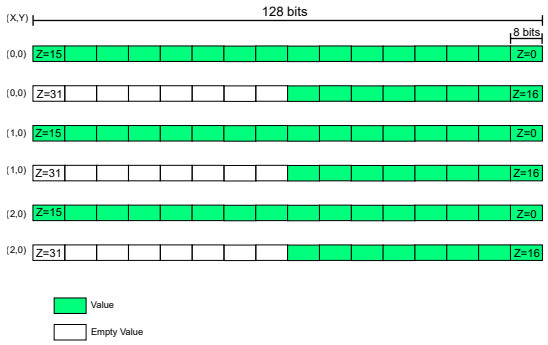

Fig. 5. Z-wise data organization.

The figure illustrates an example of packing activations of a map with 24 channels. Since the number of channels is not multiple of 16, the data is padded to 16 positions.

The order in which the input activations are read depends on the kernel size. The neural network model considered in this work includes kernels of size 3×3 and 1×1. The z-lines of pixels must be read to follow the window of the kernel. With a kernel of size 3×3, the PEs receive three vectors of three sequential lines. For a kernel of size 1×1 only a single vector is read. The sequence of reading addresses of the feature map lines is generated by an address generator unit that shares the output address with all PEs (not represented in the figure). The filters in each PE are read in sequence since the order in which they are stored in memory matches the sequence with which the activations are read.

To execute one layer, the IP starts by receiving the configuration parameters of said layer: map size, kernel size, padding, number of input channels, number of filters, stride, activations, via an AXI4-Lite port. Then, it reads through the AXI-Stream port all the weight values and stores them in the URAMs, and also reads the initial three or four liens from the input FM (depending on the kernel size) to enable the execution of the first convolutions with any value of stride. The remaining lines of the input map are read in parallel with the calculation of the convolutions over the lines present in the input map memory.

The processing flow for the convolution operation begins by resetting all PE accumulators. Next a 128-bit word, which contains the first 16 channels of the pixel of the FM, is sent to the PE along with another 128-bit word containing the 16 corresponding weights for the convolution operation. Inside the PE, the dot product between the activations and the weights is calculated by performing 16 MAC operations. In the subsequent iteration, the next 16 channels of the first input activation are sent, repeating the previous step until all channels of the activation are processed. Once all channels of the first input activation are processed, the convolution moves to the next activation, performing all z-wise operations and continuing this process until all activations required for the output are iterated over.

A visual representation of a PE is represented in 6.

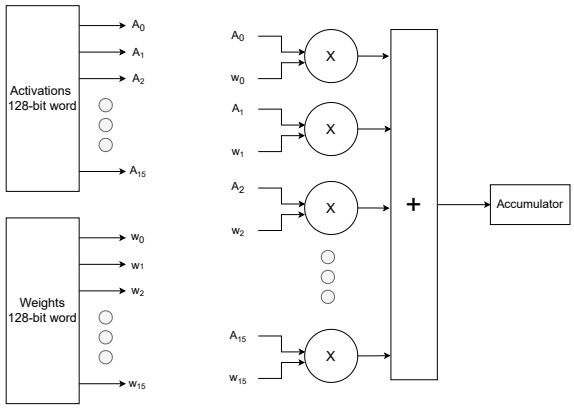

Fig. 6. Processing Element Diagram.

The sixteen parallel multipliers are followed by an adder tree and an accumulator. Once the accumulator of each of the PEs has an output activation ready, these values are passed through the Leaky ReLU function, scaled according to the scale of the fixed point representation, and then saturated within the 8-bit signed representation range (-127 to 127) to avoid over and underflow. Finally, the accumulator values are concatenated into a 128-bit word to be streamed out through the output AXI-Stream port.

Since each PE accesses different filters, the output pixels produced will correspond to different channels of the output, which will preserve the z-wise organization in the output feature map.

Simultaneously, at every iteration of the processing elements, a 128-bit word composed of 8-bit activations is read through the AXI-Stream port and stored in local memory. This allows reading the next input FM lines to be processed while processing the previous ones, masking the data transfer overhead. This process needs to be controlled cautiously to prevent the newly read lines to overwrite the yet to be processed lines read before. The IP is entirely pipelined in order to maximize throughput.

A modified version of the Leaky ReLU function was considered to simplify its hardware implementation. The original Leaky ReLU function is defined as $max(0.1 \times x, x)$. This requires a multiplication by 0.1. The modified Leaky ReLU function is defined as $max(0.09375 \times x, x)$ to simplify this operation. Knowing that $0.09375 = 2^{-4} + 2^{-5}$, a multiplication by this constant is implemented as an addition of two shifted values. The modified Leaky ReLU was considered during the training of the model.

B. Vits HLS Results

The Vitis HLS tool generates reports detailing the pipeline characteristics and utilization estimates of the IP. These pipeline characteristics are obtained after synthesis, allowing for the observation of the iteration interval and iteration latency for each loop. The iteration interval represents the number of clock cycles required to complete one full iteration of a loop from start to finish, while iteration interval, also known as initiation interval is the number of clock cycles needed to start consecutive iterations of a loop. While the impact of iteration latency on performance becomes less significant as the number of iterations in a loop increases, the iteration interval is crucial for the overall performance of the loop. A lower iteration interval allows iterations to start more frequently, resulting in more effective pipeline utilization.

The pipeline characteristics of the IP are represented in the table IV. It is important to mention that the clock period considered is $10\,\mathrm{ns}$.

TABLE IV
PIPELINE CHARACTERISTICS OF THE CONVOLUTION IP CORE.

	Iteration Interval (clk cycles)	Iteration Latency (clk cycles)
Read Weights Loop	1	1
Read Initial FM Loop	1	2
Convolution Loop	1	43

As it can be observed in IV the iteration interval for all of the loops is 1 which very important for the performance of the IP. An iteration interval greater than 1 would have a multiplicative effect on the time needed to run the loop, as it reduces the frequency at which new iterations start, thereby decreasing throughput and increasing total execution time.

The utilization estimates from HLS can also be obtained post-synthesis, however, these are very conservative estimations. A much more accurate resource utilization estimate is obtained from the post-implementation report. The resources estimate from the post-implementation report of Vitis HLS is in table V.

TABLE V
RESOURCE UTILIZATION ESTIMATES FROM VITIS HLS.

Resource	Used	Available	Utilization (%)	Guideline(%)
LUT	18293	230400	7.9	70
FF	9461	460800	2.1	50
DSP	281	1728	16.3	80
URAM	64	96	66.7	80
BRAM (36K)	21.5	312	6.9	80

As intended the resource consumption of the accelerator does not exceed the device hardware limitations.Most of the DSPs used are bound to the PEs to execute the MAC operations. However, there are a few more that are used for execute calculations and other control operations.

The local memory of the IP was divided between BRAMs and URAMs. The BRAMss are used to store the feature map lines and were dimensioned to be able to support the biggest possible line. This happens in the second layer of the model when the input map dimensions are $608 \times 608 \times 32$. The map values are organized z-wise and the IP only stores 5 lines of the map, that is, $97\,\mathrm{KB}$ of memory are needed to store the lines. Since each BRAM has $36\,\mathrm{Kb}$, the amount of BRAms needed to store the FM lines can be calculated, as it is shown in (1).

$$\#\text{BRAMs} = \frac{95KB}{36Kb} = 21.1 \approx 21.5 \text{ BRAMs} \tag{1}$$

The URAMs is used to store the weights and were dimensioned to support the largest layer of the model. The largest layer features $512 \times 384 \times 3 \times 3$ weights which is equivalent to $1,69$MB of weights. Since the memory is partitioned into 16 sections, each partition needs to be able to store $110.6\,\mathrm{KB}$. Knowing that each URAM has a capacity of $288\,\mathrm{Kb}$, the ammount of URAMs needed for each partition can be calculated, as it is shown in (2).

$$\#\text{URAMs} = \frac{110,6KB}{36Kb} = 3.1 \approx 4 \text{ URAMs} \tag{2}$$

Since there are 16 partitions and each uses 4 URAMs, the total number of URAMs used is 64.

V. HW/SW SYSTEM IMPLEMENTATION AND RESULTS

A. Hardware/Software Architecture

The hardware/software architecture of the system is comprised of the accelerator and the processor. Both the Processing System (PS) and the Programmable Logic (PL) of the target FPGA (Zynq UltraScale+ XCZU7EV) are used to implement the system. The PL block implements the IP explained in the previous section which was exported as an IP core and integrated into the FPGA. The PS block equipped with a quad-core Arm Cortex-A53 Application Processing Unit (APU) runs the software of the system.

For the interface between the PS and the PL blocks, the Advanced eXtensible interface (AXI) communication standard is used, which provides high bandwidth, low latency connections, resulting in efficient data transfers.

The system requires the transfer of the FM and filters from the external memory to the hardware accelerator. The access to the external memory is done through one high-performance port of the PS-PL interface and with the use of Direct memory access (DMA) blocks, which are set up and configured by the PS to send the FM and filters as well as receive the outputs produced by the accelerator.

The hardware/software architecture was elaborated in Vivado, whose block design is represented in figure 7.

As it can be seen in 7, other then the PL and PS, a DMA is included to transfer the weights and input FM values from external memory into the IP and transfer the output FM values produced by the IP to the external memory via the AXI-Stream protocol.

The DMA connects to the external memory controller through HP0 (High-Performance) port of the ZYNQ Ultrascale+ PS. This connection is done by the AXI SmartConnect block.

There is also an AXI Interconnect block, that connects the AXI-Lite interface of the PS to both the DMA and the PL block. This allows the PS to send the configuration parameters

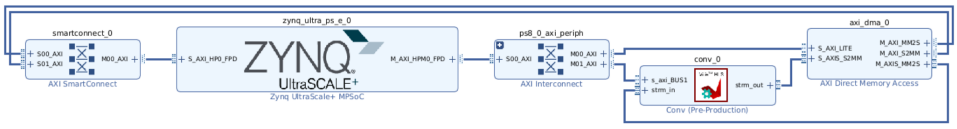

Fig. 7. Vivado Block Design.

TABLE VI
VIVADO'S RESOURCE UTILIZATION.

	LUT (230 400)		FF (460 800)		DSP (1728)		BRAM (312)		URAM (96)	
	Used	%	Used	%	Used	%	Used	%	Used	%
Convolution IP	25 577	11.1	10 720	2.3	163	9.4	21.5	6.9	64	66.7
DMA	2077	0.9	3217	0.7	0	0	5	1.6	0	0
ps8_axi_periph	1321	0.6	1446	0.2	0	0	0	0	0	0
Smartconnect	2151	0.9	3910	0.8	0	0	0	0	0	0
Total	31 135	13.5	19 326	4.2	163	9.4	26.5	6.9	64	66.7

of the convolutional layer via the AXI-Lite protocol, as well as to control the scheduling of data being transferred between the IP and the external memory through the DMA.

The model is executed by running the convolutional layers one at a time until all layers are executed.

The software platform created in the Vitis IDE runs in the ARM processor of the ZYNQ FPGA. It iteratively configures the convolutional engine and instructs the accelerator to start executing the convolution. The configuration parameters of the engine are sent through the AXI-Lite interface. Then, a start signal through the same interface instructs the core to start operating. The accelerator first reads all weights and stores them in the local memories of the PEs. These are read in the predefined order from the external memory. This transfer is executed by the DMA, which are configured to transfer the right number of weights, using the AXI-Stream interface. The output transfer is done by the same DMA, which is configured before starting the stream of input FM values so that the outputs can be streamed out of the IP and into the external memory as soon as they are ready.

The processor then waits for the end of the transfer of output activations to external memory, meaning that the core has finished the execution. Once the output stream finishes and the signal done of the control register of the AXI-Lite is read, the convolution is completed. The process described must be repeated and chained together by feeding the output of a previous layer as the output of the next. The upsampling and FM sum layers are not supported, so these must be executed in software.

B. Resource Utilization

The resource utilization of the hardware architecture illustrated in Figure 7 after place and route is found in table VI.

As shown in Table VI, the majority of the FPGA resources are utilized by the developed convolution IP and the DMA. Both the BRAM and URAM usage estimates from the HLS (Table V) are accurate. The remaining components have a minimal impact on FPGA resource usage.

The most used resource is memory, in particular, URAM to store the weights of the model. The architecture can be designed with less memory. However, this would mean that the weights of the larger layers would not fit into internal memory at the same time. Therefore, the input map would have to be read more than once and the output maps would be produced in chunks which would then have to be concatenated. Besides, reducing data transfers with external memory also reduces the energy consumption.

Another important aspect is that only around 13% of LUTs and 10% of DSPs were used. Since the proposed accelerator is scalable in terms of parallelism, the system's performance could be improved by increasing parallelism at the cost of more resources.

C. Performance Results

Each convolutional layer from the model was run individually on the ZCU104. The output of each layer was validated as correct and its timings were registered. Furthermore, a software-only convolution was also developed using simple C, with which the outputs were validated and the execution times were also collected to be compared with those obtained with the accelerator. It is relevant to mention that the C code for the software code is not optimized and thus the software execution results could be potentially better.

The processing time of each layer was calculated, including the data transfers in and out of the FPGA, with the accelerator running with an operating frequency of 100 MHz. The theoretical time the accelerator would need to complete each layer was also calculated by multiplying the number of cycles each layer takes to complete by the clock period of 10 ns. The number of cycles the accelerator takes to complete a layer is calculated by dividing the number of MAC operation of a layer by the number of MACS the accelerator performs per cycle, which in this case is 256. The timing results for each layer can be found in table VII.

As shown in table VII, each layer the accelerator timings have a slight discrepancy with the theoretical ones. This is

TABLE VII
PERFORMANCE RESULTS PER LAYER.

Conv #	Input Size	Input Channels	Kernel Size	Stride	Padding	Output Channels	Output Size	#MACs (M)	#Cycles	Theoretical time (ms)	Real Time (ms)	SW Time (ms)
1	608	16	3	1	1	32	608	1703.41	6 653 952	66.540	69.29	320 160
2	608	32	3	2	1	64	304	1703.41	6 653 952	66.540	74.115	319 028
3	304	64	1	1	0	32	304	189.27	739 328	7.393	11.437	38 087
4	304	64	1	1	0	32	304	189.27	739 328	7.393	10.896	38 081
5	304	32	1	1	0	32	304	94.63	369 664	3.697	5.087	19 139
6	304	32	3	1	1	32	304	851.71	3 326 976	33.270	34.562	159 924
7	304	32	1	1	0	32	304	94.63	369 664	3.697	5.087	19 141
8	304	32	1	1	0	64	304	189.27	739 328	7.393	8.828	38 233
9	304	64	3	2	1	128	152	1703.41	6 653 952	66.540	70.209	318 745
10	152	128	1	1	0	64	152	189.27	739 328	7.393	9.17	38 007
11	152	128	1	1	0	64	152	189.27	739 328	7.393	9.172	38 005
12	152	64	1	1	0	64	152	94.63	369 664	3.697	4.432	19 071
13	152	64	3	1	1	64	152	851.71	3 326 976	33.270	34.013	1 569 683
14	152	64	1	1	0	64	152	94.63	369 664	3.697	4.418	19 063
15	152	64	1	1	0	128	152	189.27	739 328	7.393	8.13	38 080
16	152	128	3	2	1	256	76	1703.41	6 653 952	66.540	68.827	318 764
17	76	256	1	1	0	128	76	189.27	739 328	7.393	8.35	37 963
18	76	256	1	1	0	128	76	189.27	739 328	7.393	8.22	37 965
19	76	128	1	1	0	128	76	94.63	369 664	3.697	4.098	19 023
20	76	128	3	1	1	128	76	851.71	3 326 976	33.270	33.829	159 314
21	76	128	1	1	0	128	76	94.63	369 664	3.697	4.097	19 025
22	76	128	1	1	0	256	76	189.27	739 328	7.393	7.814	37 999
23	76	256	3	2	1	512	38	1703.41	6 653 952	66.540	69.055	317 584
24	38	512	1	1	0	256	38	189.27	739 328	7.393	8.054	37 934
25	38	512	1	1	0	256	38	189.27	739 328	7.393	8.055	37 934
26	38	256	1	1	0	256	38	94.63	369 664	3.697	3.991	19 004
27	38	256	3	1	1	256	38	851.71	3 326 976	33.270	34.247	158 665
28	38	256	1	1	0	256	38	94.63	369 664	3.697	3.991	19 002
29	38	256	1	1	0	512	38	189.27	739 328	7.393	7.775	37 955
30	38	512	3	2	1	384	19	638.78	2 495 232	24.952	27.711	118 455
31	19	384	1	1	0	192	19	26.62	103 968	1.040	1.226	5 379
32	19	384	1	1	0	192	19	26.62	103 968	1.040	1.226	5 364
33	19	192	1	1	0	192	19	13.31	51 984	0.520	0.601	2 723
34	19	192	3	1	1	192	19	119.77	467 856	4.679	5.143	22 180
35	19	192	1	1	0	192	19	13.31	51 984	0.520	0.6	2 724
36	19	192	1	1	0	512	19	35.49	138 624	1.386	1.549	716 233
37	19	512	1	1	0	256	19	47.32	184 832	1.848	2.147	9 525
38	19	256	3	1	1	512	19	425.85	1 663 488	16.635	18.233	78 720
39	19	512	1	1	0	256	19	47.32	184 832	1.848	2.147	9 523
40	19	256	1	1	0	256	19	23.66	92 416	0.924	1.061	479 397
41	19	256	3	1	1	512	19	425.85	1 663 488	16.635	18.233	78 716
42	19	512	1	1	0	512	19	94.63	369 664	3.697	4.168	18 994
43	19	512	1	1	0	256	19	47.32	184 832	1.848	2.147	9 524
44	38	512	1	1	0	256	38	189.27	739 328	7.393	8.055	37 929
45	38	256	1	1	0	128	38	47.32	184 832	1.848	2.103	9 534
46	38	128	3	1	1	256	38	425.85	1 663 488	16.635	17.089	79 360
47	38	256	1	1	0	256	38	94.63	369 664	3.697	3.991	19 006
48	38	256	1	1	0	128	38	47.32	184 832	1.848	2.102	9 537
49	76	256	1	1	0	128	76	189.27	739 328	7.393	8.349	37 965
50	76	128	1	1	0	64	76	47.32	184 832	1.848	2.239	9 543
51	76	64	3	1	1	128	76	425.85	1 663 488	16.635	16.838	79 695
52	76	128	1	1	0	128	76	94.63	369 664	3.697	4.097	19 025
53	76	128	3	1	1	256	76	1703.41	6 653 952	66.540	67.295	318 571
54	76	256	1	1	0	561	76	829.53	3 240 336	32.403	33.449	166 194
55	76	128	3	2	1	256	38	425.85	1 663 488	16.635	17.463	79 394
56	38	256	1	1	0	128	38	47.32	184 832	1.848	2.103	9 528
57	38	128	3	1	1	256	38	425.85	1 663 488	16.635	17.089	79 359
58	38	256	1	1	0	384	38	141.95	554 496	5.545	5.883	28 475
59	38	384	3	1	1	512	38	2555.12	9 980 928	99.809	102.48	475 871
60	38	512	1	1	0	561	38	414.76	1 620 168	16.202	17.04	83 058
61	38	384	3	2	1	512	19	638.78	2 495 232	24.952	27.645	118 505
62	19	512	1	1	0	256	19	47.32	184 832	1.848	2.147	9 521
63	19	256	3	1	1	512	19	425.85	1 663 488	16.635	18.233	79 558
64	19	512	1	1	0	384	19	70.98	277 248	2.772	3.158	14 261
65	19	384	3	1	1	512	19	638.78	2 495 232	24.952	27.361	118 046
66	19	512	1	1	0	561	19	103.69	405 042	4.050	4.547	20 805

due to the data transfer times not being accounted for in the theoretical times calculations.

The total execution time of the model, based on the run times of each layer, can be found on the table VIII. It is noteworthy that this execution time is only an estimate of the real network total runtime, since the upsampling and FM layers were not implemented and, if implemented solely on software, these layers would have a significant impact on the models' performance.

TABLE VIII
TOTAL EXECUTION TIMES.

Theoretical HW Time (s)	Real HW Time (s)	SW Only Time (s)
1.05	1.13	7710.77

To further compare the system results, the network model was ran and tested in other platforms. In addition to the software-only solution run in the Arm Cortex-A53 present on the ZCU04 development kit, the model was also tested in with a GPU-only execution, using a NVIDIA GeForce RTX 3090, and a CPU-only execution using an Intel(R) Core(TM) i7-11700F. These tests were done using the Pytorch floating point model and measuring its overall execution time. The execution of the model in the GPU and CPU was done by running the model in inference mode for 453 images from the UCAS-AOD dataset. The performance comparison alongside power consumption of these different systems can be found in table IX. The power consumption values for the ZCU104 were obtaned via the Vivado power report, with an additional estimated 3 W accounted for external memory and component access. The power consumption figures for the CPU and GPU were based on the manufacturer's maximum specified power usage

TABLE IX
PERFORMANCE AND EFFICIENCY OF DIFFERENT PLATFORMS

Device	Time (s)	FPS	Power (W)
ZCU104	1.13	0.88	7.27
Arm Cortex-A53	7710,77	1.2×10^{-3}	5.63
Intel(R) Core(TM) i7-11700F	0.15	6.67	65
NVIDIA GeForce RTX 3090	0.0082	122.46	350

The highest throughput is achieved with the GPU which achieves 122 FPS. Both the GPU and the Intel processor are faster but require high power. The proposed solution in hardware achieves a frame rate close to 1 FPS with only 7.3 W. As stated above, the parallelism of the architecture using the same FPGA can be increased by a factor of 6 or 7, permitting increasing the throughput of the accelerator to close to 6 FPS.

Almost 50% of the power consumed by the FPGA is by the ARM core. With the complete model implemented in hardware, the processor is only used to configure the accelerator and the DMA. Using a small soft processor for these tasks implemented in the programmable logic would permit using an FPGA with only programmable logic and, thus, reduce the power.

This makes the hardware/software solution a far superior choice for real-time, on-site object detection in aerial images. Although it may be less performant, it boasts significantly lower power consumption compared to GPU and CPU platforms.

VI. CONCLUSION AND FUTURE WORK

This work presents the development and implementation of a hardware/software solution for real-time object detection in aerial images using a Convolutional Neural Network (CNN).

The R-YOLOv4 model was selected as the most suitable for this project due to its balance between accuracy and performance, as well as its support for oriented object detection. The model was modified to improve its suitability for hardware deployment. These modifications decreased the depth of the network and reduced the number of parameters, thereby lowering the computational and memory requirements at the cost of $2.5\,\%$ of accuracy. The model was quantized with 8 bits for both activations and weights, with a final mAP50 of 82.9%, 5.7% lower than the original model.

A custom hardware accelerator was designed and implemented in a Zynq Ultrascale+ MPSoc FPGA XCZU7E, with a total of 256 MACs that run in parallel. Experimental results demonstrated that the proposed solution achieves a frame rate of nearly 1 FPS with a power consumption of only 7.3 W. Although this performance is lower compared to high-end GPU and CPU platforms, the significant reduction in power consumption makes it an attractive solution for edge computing applications. It is worth noting that the upsampling layers and feature map sum layers were not implemented in hardware.

As future work, the model deployment can be completed by implementing the upsampling and FM sum layers. Initially, these can be implemented in a software-only solution; however, for better performance, they should also be implemented in the FPGA using the remaining available space.

The FM sum layers can be implemented by adding another DMA, which would use the AXI-Stream interface to send the map to be summed to the output into the IP and adds them to the output of the PEs before streaming out.

The upsampling operation, which simply duplicates each pixel every pixel of the map, could be achieved by creating an IP that receives the results from the convolution IP and, after receiving a full pixel, streams the pixel twice into the external memory. Additionally, various PE configurations could be tested to determine the extent to which increasing the processing parallelism of the IP improves performance.

References

[1] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. arXiv:2004.10934 [cs, eess]. Apr. 2020. URL: http://arxiv.org/abs/2004.10934 (visited on 01/03/2023).

[2] Gong Cheng et al. *Towards Large-Scale Small Object Detection: Survey and Benchmarks*. en. arXiv:2207.14096 [cs]. Dec. 2022. URL: http://arxiv.org/abs/2207.14096 (visited on 01/03/2023).

[3] Zhipeng Dong et al. "Multi-Oriented Object Detection in High-Resolution Remote Sensing Imagery Based on Convolutional Neural Networks with Adaptive Object Orientation Features". en. In: *Remote Sensing* 14.4 (Jan. 2022). Number: 4 Publisher: Multidisciplinary Digital Publishing Institute, p. 950. ISSN: 2072-4292. DOI: 10.3390/rs14040950. URL: https://www.mdpi.com/2072-4292/14/4/950 (visited on 01/05/2023).

[4] Ross Girshick. *Fast R-CNN*. arXiv:1504.08083 [cs]. Sept. 2015. DOI: 10.48550/arXiv.1504.08083. URL: http://arxiv.org/abs/1504.08083 (visited on 01/03/2023).

[5] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". en. In: (Nov. 2013). DOI: 10.48550/arXiv.1311.2524. URL: https://arxiv.org/abs/1311.2524v5 (visited on 01/03/2023).

[6] *kunnnnethan/R-YOLOv4*. en. URL: https://github.com/kunnnnethan/R-YOLOv4 (visited on 01/05/2023).

[7] Tsung-Yi Lin et al. "Focal Loss for Dense Object Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.2 (Feb. 2020). Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 318–327. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2018.2858826.

[8] Wei Liu et al. "SSD: Single Shot MultiBox Detector". en. In: vol. 9905. arXiv:1512.02325 [cs]. 2016, pp. 21–37. DOI: 10.1007/978-3-319-46448-0_2. URL: http://arxiv.org/abs/1512.02325 (visited on 01/03/2023).

[9] Niall O' Mahony et al. *Deep Learning vs. Traditional Computer Vision*. Vol. 943. arXiv:1910.13796 [cs]. 2020. DOI: 10.1007/978-3-030-17795-9. URL: http://arxiv.org/abs/1910.13796 (visited on 01/06/2023).

[10] Upesh Nepal and Hossein Eslamiat. "Comparing YOLOv3, YOLOv4 and YOLOv5 for Autonomous Landing Spot Detection in Faulty UAVs". en. In: *Sensors* 22.2 (Jan. 2022), p. 464. ISSN: 1424-8220. DOI: 10.3390/s22020464. URL: https://www.mdpi.com/1424-8220/22/2/464 (visited on 01/03/2023).

[11] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger". en. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, July 2017, pp. 6517–6525. ISBN: 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.690. URL: http://ieeexplore.ieee.org/document/8100173/ (visited on 01/03/2023).

[12] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement*. en. arXiv:1804.02767 [cs]. Apr. 2018. URL: http://arxiv.org/abs/1804.02767 (visited on 01/03/2023).

[13] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. en. arXiv:1506.02640 [cs]. May 2016. URL: http://arxiv.org/abs/1506.02640 (visited on 01/03/2023).

[14] Vladimir Reilly, Haroon Idrees, and Mubarak Shah. "Detection and Tracking of Large Number of Targets in Wide Area Surveillance". en. In: *Computer Vision – ECCV 2010*. Ed. by Kostas Daniilidis, Petros Maragos, and Nikos Paragios. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 186–199. ISBN: 978-3-642-15558-1. DOI: 10.1007/978-3-642-15558-1_14.

[15] Miguel Reis. *Hardware Acceleration of CNN-Based Image Segmentation for Fire Detection*. Nov. 2022.

[16] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. en. arXiv:1506.01497 [cs]. Jan. 2016. URL: http://arxiv.org/abs/1506.01497 (visited on 01/03/2023).

[17] Zhong-Qiu Zhao et al. *Object Detection with Deep Learning: A Review*. en. arXiv:1807.05511 [cs]. Apr. 2019. URL: http://arxiv.org/abs/1807.05511 (visited on 01/03/2023).

[18] Haigang Zhu et al. "Orientation robust object detection in aerial images using deep convolutional neural network". In: *2015 IEEE International Conference on Image Processing (ICIP)*. Sept. 2015, pp. 3735–3739. DOI: 10.1109/ICIP.2015.7351502.