Hardware Acceleration of CNN-Based Image Segmentation for Fire Detection

Miguel Reis
Instituto Superior Técnico,
Universidade de Lisboa
miguel.angelo.reis@tecnico.ulisboa.pt

***Abstract*—Image segmentation is an important task for many applications, like surveillance, security, and medical, among others. Many of these applications need to be executed on edge devices to take real-time decisions. Hence, low-cost and fast solutions are needed for image segmentation. One such application is automatic fire detection systems to facilitate the intervention and reduce the cost of firefighters travel in case of false occurrences.**

This work optimizes an image segmentation algorithm based on a deep learning model (Deeplabv3+ with a Modified Aligned Xception) backbone and implements it in a System-on-Chip Field Programmable Gate Array (Xilinx Zynq UltraScale+ ZU3EG SoC).

The model was optimized with low bitwidth quantization to improve the hardware acceleration without sacrificing accuracy. The optimized model was applied to an image segmentation task for fire detection. The results show that an accuracy higher than that achieved with floating-point representation is with low bitwidth fixed point quantization. When the accelerator is integrated into the final hardware design system, the network has a throughput of 1 frame per second and a low consumption of 6 Watts.

***Index Terms*—Convolutional Neural Network, Image Segmentation, Fire Detection, Quantization, Hardware Accelerator, High-Level Synthesis**

I. Introduction

The goal of this work is to develop a fire detection deep neural network embedded system using a low-cost SoC FPGA to be used on IoT Devices. This chapter describes the motivation, the main objectives for this work and the report outline.

A. Motivation

Due to the increase of forest fires in the last decade, the purpose chosen for this work is the early detection of fires in order to decrease the damage of these disasters. Image processing in deep neural networks has been continuously evolving throughout the years. Due to the computation-intensive nature of deep neural networks, they are predominantly executed with the assistance of a CPU or a GPU in real-world applications, which are costly in power consumption and money wise. Running them on edge devices is an improvement regarding cost and power consumption but it is a challenge due to the limited resources attached to them. FPGAs can be used as an edge device due to their low cost and possible low-power consumption but several tradeoffs must be done in order to run CNNs on them. Speed, accuracy and throughput are some of the compromises that must be made in order for the system to run in a low-cost FPGA. For detecting fires a minimal throughput of 0.1 FPS with a low power consumption should be achieved.

II. State of the Art

This section introduces the concept of Convolutional Neural Networks (CNN) and their use in image classification and image segmentation along with the most relevant models that have been proposed. Implementation methods on FPGAs are also introduced along with two architecture types.

A. Convolutional Neural Networks

CNNs are a category of DNN mostly suited for image-focused tasks such as image classification or image recognition. CNNs work by processing input images in the form of matrices with pre-defined dimensions, for example, an RGB image would have three channels and a black and white image would only have one, and getting an output that can be later turned into a prediction. In order for a CNN to be able to output a desirable prediction it needs to be trained over several iterations with the help of already classified datasets. These datasets consist of an agglomerate of images, with several classes, each with its corresponding truth label. A well trained CNN will output correct predictions of the labels associated with the input images that are processed.

CNNs are made up of several layers, where each one gets its input FM from the previous layer, and outputs its resulting FM to the next one. Feature maps are the input and output of each layer, composed of two-dimensional arrays of pre-determined sizes. The number of filters used in a layer will result in the same number of output FMs. There are four types of layers which together make up a model of a CNN: the convolutional layers, the pooling layers, the activation function layers, and the fully connected layers.

Some CNNs also include a Batch Normalization layer to normalize the average and the standard deviation of the layers output FM to zero and one, respectively [1]. This improves overall accuracy and speeds up the training process.

The convolutional layer is the most important component of a DNN, it is responsible for extracting features out of its input FM. This is done by convoluting the input FM with a predefined kernel. A kernel is a small two-dimensional array composed of weights that are trained in order to achieve the best predictions possible. A 2D convolution is done by

processing a single kernel with its input FM with also only one two-dimensional array. The kernel is overlaid with the input FM array and the dot-product is performed: the multiplications of each overlapped value are added together and then stored on a position of the output array. A bias value is also added to the result which is also a trained parameter. The process is repeated one position to the right until the output array is filled. There are two parameters which alter the way the convolution is done: the padding and the stride. The stride dictates how many positions the kernel shifts after each dot-product and the padding determines how many 0-valued positions are added to the edges of the input array. The size of the output FM is calculated with Equation 1 [2] where i_h and i_w are the input FM array height and width, the k_h and k_w are the kernel height and width, the s is the stride and the p is the padding.

$$OFM_h * OFM_w = (\frac{i_h - k_h + 2p}{s} + 1) * (\frac{i_w - k_w + 2p}{s} + 1) \quad (1)$$

The number of total multiplications in a 2D convolution can be calculated using equation 2 where OFM is the output FM and k is the kernel.

$$\#ofMultiplications = OFM_h * OFM_w * k_h * k_w \quad (2)$$

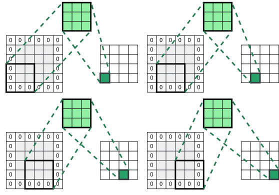

Fig. 1. Example of a 2D Convolution

Figure 1 shows an example of a 2D convolution, showing only 4 dot-products for simplicity, with an input FM array size of 4×4, a kernel with a size of 3×3 and a stride and padding of 1. The kernel is represented by the light-green box with several weights represented by its internal squares. The input and output FMs are also represented on the left and right, respectively, of each dot-product. In each iteration, the dot-product result is represented by the dark-green squares on the output FM. After each dot-product, except the last one, a one-position shift to the right is shown. The output FM array has a size of 4×4 that is confirmed by equation 1.

For a 3D convolution, the output FM is comprised of several two-dimensional arrays, or channels, the same number as the number of the filter used. A filter is composed of several channels, comparable to a 3D kernel. As in the 2D convolution, several dot-products are performed with the same channels (e.g the first filter channel is only computed with the first input FM channel, the second filter channel with the second input FM channel, and so forth) and the results of these dot-products are added together along with the bias and stored in a position of a channel of the OFM. Each filter used will result in a different channel on the OFM.

Figure 2 shows an example of a 3D convolution of a 3-channel input FM with 2 filters.

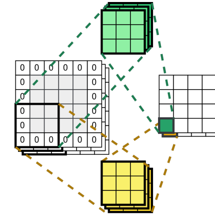

Fig. 2. Example of a single iteration of a 3D Convolution

An input FM with 3 channels can be seen with two filters, shown with two different colours, with also 3 channels and an output FM of 2 channels, the same number of channels as the number of filters. Only one iteration of the convolution is shown, for simplicity, where each filter is related to a single channel of the output FM.

The number of total multiplications in a 3D convolution can be calculated using equation 3 where OFM is the output FM, $N_{filters}$ is the number of filters and F is the Filter.

$$\#ofMults = OFM_h * OFM_w * N_{filters} * F_{channels} * F_h * F_w \quad (3)$$

Depthwise separable convolutions are a less computationally expensive way of doing 3D convolutions with a large number of filters [3]. The depthwise separable convolution works by using channel-independent dot-products, much like several 2D convolutions, resulting in an intermediate FM with the same number of channels as the filter used, and then doing a 3D convolution with several filters with 1×1 sized channels with the intermediate FM of the previous convolution. Like the 3D convolution, the output FM has the same number of channels as the number of filters used in the depthwise convolution.

The number of total multiplications in a depthwise separable convolution can be calculated using equation separableconv-mults where OFM is the output FM, IFM is the output FM.

$$\begin{aligned}\#ofMults =& OFM_h * OFM_w * F_h * F_w * IFM_{channels} \\ &+ N_{filters} * IFM_{channels} * OFM_h * OFM_w\end{aligned} \quad (4)$$

While a 3D convolution of a 3-channel 12×12 IFM with 256 filters sized 5×5 that results in a 256 channel 8×8 output FM requires a total of 1,228,800 multiplications, a depthwise separable convolution needs only 4,800 multiplications for the first convolution and then 49,152 multiplications for the depthwise convolution for a total of 53,952 multiplications.

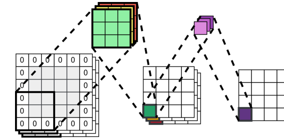

Fig. 3. Example of a Depthwise Separable Convolution

Figure 3 shows an example of a depthwise separable convolution of a 3 channel 4×4 image with one filter with 3

channels for the channel independent convolutions and one 1×1 3 channel filter for the depthwise convolutions. Figure 3 shows one iteration of a depthwise separable convolution, such that the intermediate FM, in the middle, has 3 channels where each coloured value is computed with a 2D convolution of each channel of the input FM with its respective filter channel. Afterwards, the output FM is calculated as a 3D convolution with a single 3 channel filter sized 1×1.

Atrous convolutions allow the control of the resolution of the convolutions by adding a new parameter, the rate. The rate defines the number of spaces between the values of the kernel. This allows for the convolutions to have a wider field of view with the same kernel size.

The size of this convolution output is calculated with equation 5 where i_h and i_w are the input height and width, the k_h and k_w are the kernel height and width, the s is stride, the p is the padding and r is the rate.

$$\begin{aligned} OutputSize = &(\frac{i_h - (k_h * r - (r-1)) + 2p}{s} + 1)* \\ &(\frac{i_w - (k_h * r - (r-1)) + 2p}{s} + 1) = O_h * O_w \end{aligned} \tag{5}$$

The number of multiplications of an Atrous Convolution is also given by equation 3.

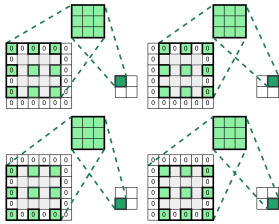

Fig. 4. Example of a Atrous Convolution

Figure 4 shows an example of a full atrous convolution of a single channel 4×4 input FM, a kernel of 3x3 with a stride and padding of 1 and rate of 2 where four dot-products are performed where the values of the input FM are separated by one value, defined by the rate parameter of 2. The shift after each dot-product, except the last one, is defined by the stride parameter of 1.

B. Image Classification

Image classification works by attributing a percentage of certainty of prediction to several classes, the more probable one being the one with a higher value. The metrics to evaluate these models are the top-1 error and top-5 error, which indicate how many times the network has predicted the correct label with the highest probability and how many times the correct label appears in the network's top five predicted classes, respectively.

C. Xception

The Xception model [4] uses mostly separable depthwise convolutions, instead of the 3D convolutions used by the previous networks, followed by batch normalizations, ReLU Functions and/or max pooling layers. The use of depthwise separable convolutions was inspired by the use of Inception modules on the Inception V3 network. The typical Inception modules can be simplified as large 1×1 convolutions followed by convolutions that operate on non-overlapping segments of the output FM. It was found that the depthwise separable convolutions performed similarly with a similar number of weights.

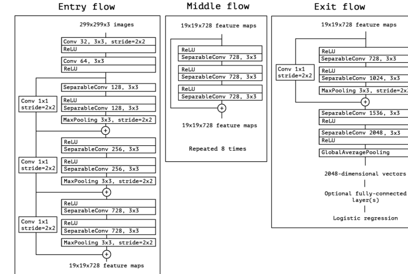

Fig. 5. Xception Network from [4]

III. Semantic Segmentation

Semantic segmentation is the process of assigning a label to each pixel of an input image. This is useful for object detection where several objects can be identified. A common architecture for semantic segmentation is the Encoder-Decoder structure [5], [6]. The Encoder-Decoder structure is a two-stage network where the encoder compresses its input through several convolutions, which captures the semantic information, while the decoder predicts the output from the encoder's output FM. The encoder is usually a network such as ResNet or Xception that captures the semantic information, through its several convolutions, while the decoder assigns each output pixels label using the encoder's semantic information.

Figure 6 shows an example of an output of a Semantic Segmentation CNN where it is shown that the motorcycle and the driver are separately identified with different colours, as they are different classes.

Fig. 6. Example of an output of an Image Segmentation CNN [7]

A. DeepLabV3+

DeepLabV3+ [8] is an architecture that employs the Encoder-Decoder structure. A simple decoder is added to its predecessor DeepLabV3 which makes use of the ASPP. ASPP applies several atrous convolutions with different rates in parallel in order to compute features in multiple scales. ASPP can help with detecting objects that might be too far away or too close to the camera due to the use of atrous convolutions with different rates.

IV. CNNs in Field Programmable Gate Arrays

The computations involved in CNNs are highly intensive due to the high number of MAC operations. As CNN models get deeper the number of operations and the storage requirements increase. In order to efficiently use FPGAs, the CNNs can be compressed in order to reduce the use of memory, communications and operations while taking into account the minimization of accuracy loss. Compression of a CNN can be done with quantization and pruning techniques.

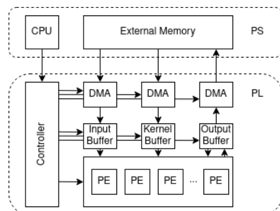

Fig. 7. Typical FPGA-based CNN accelerator based on [9]

In typical FPGA-based CNN accelerators [9], as shown in Figure 7, there are several levels of memory hierarchy. These systems are composed of input buffers that fetch the input FMs and the kernel values from the external memory through DMA.

The workload is sent to the controller from the CPU in the PS which then send the control signals to the other modules. In order to decrease the critical path of the circuit and increase the throughput the computations are usually pipelined.

Data can be quantized to any number of bits reducing the complexity of the hardware necessary for the computations and the storage capacity. It also can increase the throughput of the system but it may also decrease the overall accuracy of the model, being necessary to choose the best trade-off between accuracy and throughput and/or resource consumption. A fixed point number representation is usually chosen due to the less computationally expensive operations associated with it, in comparison to floating point representation, and is characterized by its Bitwidth (BW), the number of bits used to store the number, and the Fractional Offset (FO) which is the number of fractional bits.

Quantization can be performed after the model is trained by simply converting the floating point values into the chosen data size, or while training where the weights and activations are converted (during the forward pass), the latter giving better results due to the implicit fine-tuning.

V. Processing Flow Approaches

Two types of processing flow approaches can be used in order to run CNNs on FPGAs, one in which one or several CLP computes each layer iteratively, and another where the whole model computation is mapped on the FPGA and each layer is computed in sequence in a pipelined fashion.

Figures 8 show a representation of the CLP accelerators. For the CLP accelerator, there is only one customizable CLP, where the convolution parameters, such as sizes, padding and stride, are sent from the CPU, and in the pipelined accelerator

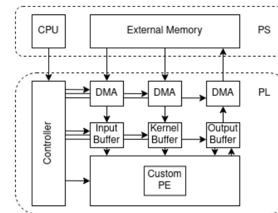

Fig. 8. Configurable Layer Processor Accelerator

there is a chain of hardwired layer processor with fixed convolution parameters tied to each layer processor.

While pipelined accelerators might be faster due to the low memory transfers, they are constrained to a specific CNN model and they can be more hardware expensive than all-purpose CLPs because all the individual layers must be simultaneously mapped in the hardware.

VI. Design and Optimization of the Network Model

A. Design Flow from Model to FPGA

In order to deploy this neural network into an SoC FPGA several tasks must be performed, according the design flow shown in Figure 9.

The network model has to be chosen along with the dataset to train the model with known tools such as Pytorch and Brevitas. After training, the trained model is evaluated to check the accuracy and to obtain checkpoints useful to verify the correctness of the outputs produced by the developed system.

A tool was developed, named Brevitas Converter, that converts the Pytorch network into a quantized Brevitas network along with the trained weights. Afterwards, a study on the quantization of data of this network, using Brevitas, has to be done taking into account the tradeoffs between area and performance from reducing the data bitwidth. This study must be diverse regarding the size in bitwidth of the data used throughout the network in order for the best bitwidth configuration to be chosen.

The accuracy and efficiency of inference of the network are then improved by merging the convolutional layers with the batch normalization layers with a tool developed in Python named Batch Normalization Merger, thus reducing the number of overall layers.

Fine-tuning is done to improve the accuracy further. The network will then be modelled by developing IP capable of running configurable layers where each of the network layers will be processed. The drivers for these IPs will also be developed in order to configure the IPs for each layer correctly. The weights will then be exported into the FPGA to be used with the developed system using another developed tool in Python named Weights Extractor. These tools are described in the following sections.

B. Tools for Training and Quantization

In order to deploy this neural network into an SoC FPGA several tasks must be performed, according to the design flow

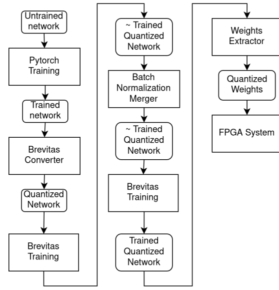

Fig. 9. Design Flow

shown in Figure 9. For this work the chosen Network is DeepLabV3+ with a Modified Aligned Xception Backbone, starting from the work done by Houda [10]. The network was trained with Pytorch[1] while the quantization training was performed with Brevitas[2].

PyTorch is an open-source machine learning framework implemented in Python optimized for both CPUs and GPUs. The training of networks with this framework is done by feeding the training script, where several training parameters are defined such as learning rates, metrics and loss functions, and a description of the neural network to be used.

Brevitas is a PyTorch library used for quantization-aware training that implements a set of building blocks at different levels of abstraction to model a reduced precision hardware data path at training time. This library provides several quantized versions of the standard PyTorch layers that can be replaced with the original PyTorch model.

C. Helper Tools

To better integrate Pytorch, Brevitas and FPGA deployment three python-based tools were developed. These tools are:

- **Brevitas Converter:** a python script that takes the description file of a Pytorch network and generates a description file for the Brevitas framework and converts the already trained weights to the generated Brevitas Network.
- **Batch Normalization Merger:** a tool that merges Batch Normalization layers with the previous convolution layer.
- **Weights Extractor:** a tool that extracts the weights and bias into binary files in order to be loaded into the SoC DDR memory.

D. Network Description

This work uses the DeepLabV3+ network with a Modified Aligned Xception as a backbone. This network consists of 142 Convolutions and ReLu as activation functions. This network contains FM modifying features such as FM sums, average pooling and interpolations.

Figures 10 and 11 show the Modified Aligned Xception and the DeepLabV3+ Networks, respectively, with all the convolutional layers stated in order of execution with their

[1] https://pytorch.org/

[2] https://github.com/Xilinx/brevitas/

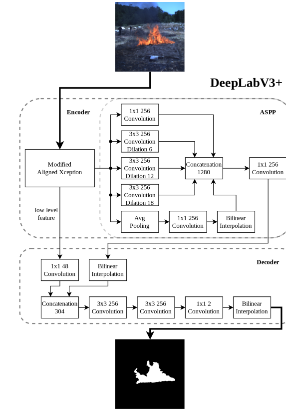

Fig. 10. DeepLabV3+

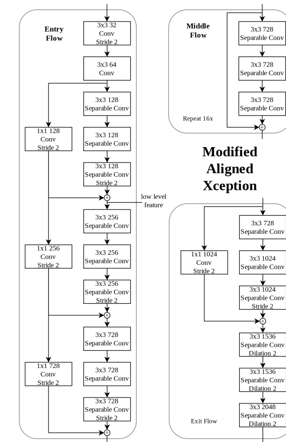

Fig. 11. Modified Aligned Xception

most important parameters as well. This network features Separable Convolutions, Atrous Convolutions, FM Sums, Average Pooling, Bilinear Interpolations and Concatenations. Each Separable Convolution block is composed of a depthwise convolution, a batch normalization layer, a pointwise convolution, another batch normalization layer and a ReLU layer. A convolution block is composed of a Convolution, a batch normalization layer and ReLU Layer, except for the last convolution where there is no ReLU layer. The Network is divided in several parts the Encoder which is composed of a backbone network, in this case, the Modified Aligned Xception which could be replaced with another one, and the ASPP block which gathers features in multiple scales with the use of atrous convolutions. Next to the encoder is the decoder block which is responsible for decoding the features gathered by the encoder. As for the Modified Aligned Xception, it is divided into three sections: the Entry flow which consists of several Separable Convolutions, some with a stride of 2 in order to reduce the size of the FMs. The Middle Flow is repeated 16 times in sequence and the Exit flow contains some convolutions with dilation characteristics.

The Corsican dataset[3] is composed of a total of 1135 images of fires acquired in the visual range (i.e RGB values) and in the near-infrared (i.e single channel image) under various conditions of positioning, weather, vegetation, distance to the fire and brightness.

E. Work flow

The dataflow regarding the network will be the following: the Network will be trained using only non-quantized Pytorch Functions where the maximum accuracy will be recorded. Afterwards, the network description file along with the trained weights will be converted to a quantized version using the developed Tool Brevitas Converter, followed by a small fine-tuning. Then the batch normalization layers will be merged with the previous convolutions in order to reduce the total number of layers during inference. Then the quantized network will be submitted to a final stage of fine-tuning where the optimal accuracy will be achieved.

VII. HARDWARE ACCELERATOR FOR DEEPLABV3+

The hardware accelerator was designed in order for the network to run from start to end without software intervention as it would prove to be a bottleneck for the whole system. All features of the network such as the convolutions, sum of maps, average pooling and interpolation were kept in mind when developing the accelerator.

The hardware accelerator was developed using Xilinx Vitis 2022.1 High Level Synthesis. The accelerator development targeted implementation in the PL of Xilinx Zynq UltraScale+ SoC ZU3EG FPGA, more specifically part number xczu3eg-sbva484-1-i, present in the Ultra96-v2 development board.

Vitis HLS is a high-level Synthesis tool that allows C and C++ functions to be compiled into RTL components that can be implemented in the PL fabric. The design flow of Vitis HLS consists of compiling and simulating the algorithm, analysing and optimising the design, synthesising it into an RTL design, verifying the RTL implementation and then exporting the RTL IP to Vivado to be implemented on the device.

The Hardware architecture for this work is composed of Processing Blocks which compute each and every layer iteratively. The PS will send via DMA the input FMs and weights of each layer and then receive, also through DMA, the output of those layers.

A. Hardware Design

For the development of the hardware, a few choices had to be made beforehand. The bitwidth of the weights, bias and activations were chosen run time, to be statically configurable (at compiling time), and the data structure was chosen to be stored z-wise i.e. in the same pixel of a map the values of channels are stored consecutively.

The architecture is split into 2 different convolution types: Regular Convolution and Depthwise Convolution. The Regular Convolution block also computes the pointwise convolutions. For the FM modifying features present in the network such as FM Sums, Interpolation and Averaging, separate blocks were developed depending on whether they are done after or before the Convolutions. The data will be transferred to and from the respective processing Blocks using the DMAs. Since the pre and post-processing elements are only relevant to the regular convolution block, the components developed for these elements will only be connected to the Convolution IP, as shown in figure 12.

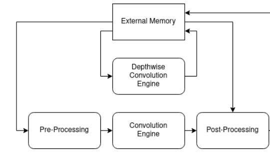

Fig. 12. IP Topology

All developed components use the AXI4-Lite protocol for parameter configuration, while the data inputs and outputs are transferred to and from the external memory with the use of the AXI4-Stream protocol. The AXI4-Lite protocol is a point-to-point bidirectional interface between an IP core and the processor while the AXI4-Stream protocol is designed for transporting arbitrary unidirectional data, one value per clock cycle.

The bit-width chosen for the AXI4-Stream data was 64-bit, which for an example of 4-bit weights and 8-bit bias, results in 16 values of weights streamed per clock cycle and 8 values of bias per clock cycle.

The Convolution IP features enough PEs to fill a 64-bit output word, for an example of 4-bit activations there would be 16 PEs and each PE would also perform 16 MAC operations. In order to keep local memory usage as low as possible while reducing the number of data transfers, there are only enough BRAMs to store a pre-determined number of biases and weights and 4 lines of the biggest input map of the network, since the biggest kernel size is also 3 and the extra line is used to store the next line of the feature map.

In the beginning, the IP receives, through the AXI4-Lite port, the parameters of the convolution and then begins to store into the BRAMs the bias and weights values from the AXI-Stream input port and then receives 3 lines of the map. Then, for each output pixel, the PEs accumulators are reset to the values of the corresponding filter bias. Afterwards, for each output pixel, the dot products of the weights with the corresponding map values are calculated. When the PEs computations are done, each value of the accumulators is scaled and concatenated into a 64-bit value which is then streamed out through the output AXI-Stream port. While the PEs are working the next pixel of the input FM is being read into the extra line of BRAMs. This process repeats until the convolution is done. The entire IP is pipelined in order to maximize the throughput.

Figure 13 shows the block diagram of the convolution IP, where the behaviour described above is illustrated.

The Depthwise Convolution IP works similarly to the Convolution IP, the difference being that each PE does 9 MAC

[3]http://cfdb.univ-corse.fr/

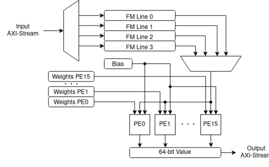

Fig. 13. Convolution IP

operations, corresponding to a 3×3 kernel. For an example of 4-bit activations, the Depthwise Convolution IP has 16 PEs with 9 MACS each and therefore is able to compute a total of 144 MACs/Cycle. As in the Convolution IP, at the beginning of the execution, the bias and weights are read from the AXI-Stream input followed by 3 lines of the input map. Since the IP does a full 3×3 kernel in each PE per cycle, 9 input map values, as well as 9 weights values, are fed into each PE. For the Input Map, there are 12 groups of BRAMS or 4 groups of lines, 3 for the convolution and 1 for storing the next line in parallel with the computation. Each PE computes its output value and then all the values are concatenated and sent out to the output AXI-Stream Port.

Figure 14 shows the block diagram of the Depthwise Convolution IP, where the behaviour described above is illustrated.

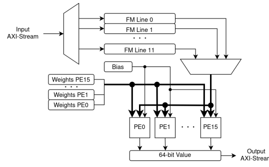

Fig. 14. Depthwise IP

The Pre-Processing IP is able to average a variable-sized map into a 1x1 map by reading the input stream values and adding them to the previously read values. Once all the values are read, the division by the total number of pixels of the resulting value of the sums is done. The addition, as well as the division of the values, is done in parallel 16 times, for an example of 4-bit activations, in order for there to be an iteration interval of 1. The division is replaced by a multiplication of a value calculated at the beginning of the IP's execution ($\frac{2^{25}}{N}$) and a bit shift represented by a division by 2^{25} in order to reduce the hardware and execution time, as shown in equation 6.

$$Avg = \frac{\sum_{i=0}^{N-1} val[i]}{N} = \frac{\sum_{i=0}^{N-1} val[i] * \frac{2^{25}}{N}}{2^{25}} \tag{6}$$

If there is no need for averaging of maps then the streamed values from the input port are then streamed out.

The Post-Processing IP implements the addition and interpolations of maps. In order to achieve an iteration interval of 1 while adding maps, it is necessary to have 2 input AXI4-Stream ports, one connected directly to the output of Convolution and the other to the external memory. When a value from each port is received the 64-bit values are partitioned into 4-bit values, if using 4-bit activations, and added in parallel. The additions are set to be signed or unsigned by setting the signed parameter previously via AXI4-Lite. Afterwards, the values are attached again into a 64-bit value and then streamed out to the external memory.

Bilinear interpolation works by applying linear interpolation in two directions. The linear interpolation formula can be seen in formula 7, where x1 and x2 are the known coordinates values in a 1-dimensional plane, Q1 and Q2 are the values associated with those coordinates and P is the value to be calculated for the new x.

$$P = Q_1 \frac{x_2 - x}{x_2 - x_1} + Q_2 \frac{x - x_1}{x_2 - x_1} \tag{7}$$

The Bilinear Interpolation formula, seen in formula 8, can be deduced from the linear interpolation formula when moving on to a 2 dimensional plane.

$$\begin{aligned} P =& Q_{11} \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)} + Q_{21} \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)} + \\ & Q_{12} \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)} + Q_{22} \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)} \end{aligned} \tag{8}$$

Implementing this formula with fixed point numbers requires scaling the position of the known points (smaller map) in order for there not be any fractional bits. The scaling factor is related to the size of the bigger map e.g if the bigger map has a size of 10 pixels then the scaling factor would be 9.

The final division of the interpolation can be replaced by a bit shift by first multiplying a factor that is the result of the division between a power of two and the actual denominator. This factor is calculated at the beginning of the interpolation process since the value is always the same. The formula can then be rewritten:

$$\begin{aligned} F =& \frac{2^{25}}{(x_2 - x_1)(y_2 - y_1) * f^2} \\ P =& \frac{(Q_{11} first + Q_{21} second + Q_{12} third + Q_{22} fourth) * F}{2^{25}} \end{aligned} \tag{9}$$

Before starting the Bilinear Interpolation the IP stores into BRAMs 2 lines of the map. Afterwards, at every clock cycle, 4 64bits values, containing 16 values each, are loaded and the interpolation is performed while loading the next line of values. The interpolation is performed for each value to be streamed out of the IP achieving a parallelism of 16.

If there is no need for interpolation or the addition of maps, the streamed values from the input ports are directly streamed out.

B. FPGA Image Build

The IPs were exported from Vitis HLS to Vivado where they were interconnected according to the topology designed in Figure 12. Two FIFO BRAM blocks were added at the end of the Depthwise Convolution and at the end of the Post-Processing IP in order for the values to be instantly written out of the IPs and not halt their processing. The sizes of these FIFO blocks were set to 2048 64-bit values due to the available BRAMs on device. Three DMAs were also added in order for the inputs to be written to the IPs and for the outputs to be read into the external memory. 2 DMAs have a write and read

channel for writing the weights and inputs FMs and reading the output FMs, one (dma2) has the input and output ports connected to the Depthwise IP and the other (dma0) has the input port connected to the Pre-Processing Ip and the output port connected to the Post-Processing IP. The remaining DMA (dma1) only has a write port for writing input FMs to be added in post-processing and is connected to the second input port of the Post-Processing IP.

C. Baremetal Deployment

The network will be executed layer by layer by sending the corresponding data to the correct IPs and receiving the output FMs through the DMAs with the use of the developed drivers. A platform was created in Vitis IDE in order for the drivers to be written in C. It was necessary to write drivers for different kinds of convolution flows: normal flow, partitioned flow, and dilated flow. The normal flow consists of setting the convolutions parameters through AXI-lite which can be done by writing to the necessary addresses, starting the IP execution by setting the IP's start signal on the control register by AXI-Lite, sending in order the bias and weights through AXI-Stream/DMA from the external memory to the IPs in a contiguous fashion, setting up the transfer from the IPs output to the external memory through AXI-Stream/DMA and then sending the FM through AXI-Stream/DMA. Setting up the output transfer before the input FM transfer makes sure that minimal time is lost between transfers because once the output is ready it can be transferred immediately. After every transfer of inputs and before starting the next one it is verified if the transfer is already finished in order for no errors to occur. The convolution is considered done once the output transfer is finished and the signal done of the control register from the IP AXI-Lite is read. The partitioned flow has the same steps in the beginning as the normal flow, the difference being when receiving the output FM, since it is partitioned the memory positions won't be contiguous for different pixels, and the output DMA Transfer is set up as many time as there are pixels, each one set to the correct memory address. FM concatenations can be done with the use of the partitioned flow.

Since dilated convolution can't be produced in a straightforward way with the developed IPs, they have to be partitioned as in [11].

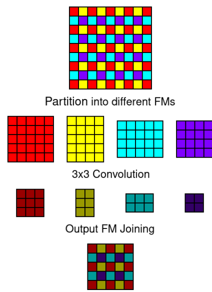

Fig. 15. Atrous Convolution with dilation rate of 2 with partitioned FMs

Figure 15 demonstrates how atrous convolutions can be processed with partitioned FMs, using the example of a 3×3 Figure 15 demonstrates how atrous convolutions can be processed with partitioned FMs, using the example of a 3×3 Convolution with a dilation rate of 2. First, the original FM is partitioned into $rate^2$ sections followed by separate convolutions for each section. Then, the outputs of these separate convolutions are joined with their pixels being interleaved in the same manner as the first partitioning.

The dilated flow consists of the same steps as the beginning of the partitioned flow, this time the difference being that, when sending the input FM, several DMA transfers are set up to correctly send the required pixel positions, since they aren't in a contiguous memory address. The output FMs are also saved with several DMA Transfers since the output values are also not in contiguous positions for different pixels.

Since the partitioned flow and the dilated flow have many more DMA transfers compared to the normal flow, they are less time efficient than the normal flow.

VIII. Results

This chapter describes and analyzes the results obtained throughout this work, from the training and testing of the Pytorch network, through the Quantization attempts and the final quantized network results. The hardware resources and timing values of the Built FPGA System will also be displayed along with the developed system timing values.

A. Pytorch Training

Using PyTorch, a DeepLabV3+ with an Xception Backbone was trained with the Corsican Fire dataset using only RGB images sized 300x300 due to the highest accuracy results achieved here [10]. The dataset was divided randomly into three parts: the first being the training partition where 682 images are used for training the network, the second being the validation partition where 226 images are tested after each epoch of training to keep track of the network train process and the third being the test partition where 226 images tested to evaluate the network final mIoU. The Training was done with a learning rate of 0.02 and with a Cross-Entropy loss function. The training machine used a 12GB Nvidia RTX2080Ti for training and it took around 1 hour to train on 100 epochs. The mIoU reached for validation and testing are 92.45% and 91.5% respectively. The reason the values of [10] were not reached might be due to a different dataset partition organization and the small adjustments made to the Xception backbone.

B. Quantization

After training the network, several quantization attempts with varying weights and activation bitwidths were done to determine the best mIoU to Hardware Resources compromise achievable. These quantization attempts consisted of using the developed tool Brevitas Converter and fine-tuning for 30 epochs. Since Brevitas training requires more computations than Pytorch-only training the combined time of processing

TABLE I
TEST MIOU(%) OF QUANTIZED NETWORK WITH VARYING BITWIDTHS

		Activations					
		8	7	6	5	4	3
Weights	8	92.02	92.05	91.97	91.67	90.16	77.24
	7	92.02	92.02	91.95	91.23	88.33	77.32
	6	92.02	92.03	91.92	91.66	87.87	76.40
	5	92.09	92.06	92.02	91.54	87.71	77.72
	4	92.00	92.01	91.82	91.47	**87.31**	76.22
	3	91.84	92.05	91.81	91.42	85.62	70.82

these attempts was of around 72 hours. Table I shows the results obtained.

The bitwidths were tested from 8 to 3 for both activation and weights as lowering, even more, the size of the bitwidths would wield mIoU results below 80%. These results are probably not the best achievable with these bitwidths, since we stopped training after 30 epochs as running for more epochs would take a considerable amount of time. However, at 30 epochs the loss variation is small, meaning that only a small improvement would be achieved with more training epochs. Decreasing both the weights and activations bitwidths lower the accuracy of the network, and with an activation bitwidth of 3, the accuracy decreases significantly independently of the weights bitwidth. Since using weights with a bitwidth size that is not a power of 2 wastes memory resources, and as the achieved mIoU of 87.31% for a bitwidth of 4 for both activations and weights is only less than 5% lower than the mIoU achieved with the original Pytorch network, the rest of this work will use this bitwidth configuration.

Afterwards, the Batch normalization layers were merged with their respective convolutional layers, using the developed tool Batch Normalization Merger, which decreased the mIoU to 81.21%. After further 50 Epochs of fine-tuning training, the mIoU reached 92.76% (an even higher value than the original Pytorch mIoU of 91.53%). The reason the quantized network performs better than the non-quantized Pytorch network might be due to the noise associated with the quantization.

C. Vitis HLS Results

TABLE II
VITIS HLS RESOURCE ESTIMATES

	DSP	FF	LUT
Available	360	141120	70560
Convolution	14	5349	24259
Depthwise	9	9742	23080
PreProcessing	20	1477	4157
PostProcessing	105	5104	12738
Total	148	21672	64234

Table II shows the LUT, FF and DSP available resources and resource estimates for the IPs developed. All of the IPs hardware resources combined do not exceed the device hardware limitations.

The local memory of all the IPs must be sized according to the network model to be executed.

TABLE III
TOTAL NUMBER OF BRAMS NEEDED FOR EACH IP

	FM BRAMs	Bias BRAMs	Weights BRAMs	Total BRAMs
Convolution	19.0	2.0	128.0	149.0
Depthwise	24.0	2.0	9.0	35.0
Pre-Processing	6.5	X	X	6.5
Post-Processing	8.0	X	X	8.0
			Total	198.5

As seen in table III, the total number of BRAMs needed is thus reduced to 198.5 BRAMs, less than the total 216 available.

D. FPGA Build

Several Synthesis and Implementation runs were done in order to determine the highest frequency possible of these IPs and the maximum frequency reached is 175MHz. Table IV shows the Hardware Utilization of the whole system.

As shown, no hardware limits are hit, the highest resource usage being the BRAMs. The power report also states a power consumption of 2.82 W, where 1.64 W are for the Processing System alone. The FPGA bitstream was then generated and exported along with the necessary files to Vitis IDE in order to be able to run the network.

E. System Results

The network was then run in the Ultra96-v2 Board, the 226 test images were processed and their respective mIoU was evaluated. The input images, weights and truth images were stored, before execution, into the board DDR memory. The input images were saved already normalized, so no pre-processing on them is needed. Every output image was compared with the truth images, calculating their mIoU and confirming that the implementation is correct. Then each of the layer's execution times were recorded and averaged over the 226 runs in order to figure out the system's throughput. The timing is recorded by calculating the difference between the number of PS clock cycles before and after executing each layer, and so these values include both the PL execution time and the data transfer times.

Regular convolutions with no Pre-Processing or Post-Processing take around 5% more than the theory time while the Depthwise convolutions with the same conditions take 15% more or higher. This is due to the number of MACs and size of memory transfers associated with each convolution as shown in the table. Partitioned and dilated convolutions also have a significant timing increase compared to the theoretical times, this is due to the elevated number of DMA transactions generated in between processing. Interpolated and Averaged convolutions also have a Time Factor value higher than other convolutions, this is due to the increased number of cycles associated with the pre and post-processing and the theoretical times do not include these extra cycles.

Table V details the theoretical time, the real-time and the extra time for each of the convolution flows used. The

TABLE IV
VIVADO IMPLEMENTATION RESOURCES

	LUTs (70560)		FFs (141120)		BRAMs (216)		DSPs (360)	
Block Name	**Qty**	**%**	**Qty**	**%**	**Qty**	**%**	**Qty**	**%**
Convolution IP	15075	21.4	4221	3.0	140.5	65.1	18	5.0
Depthwise IP	14503	20.6	6641	4.7	26.0	12.0	9	2.5
Pre-Processing IP	1487	2.1	1321	0.9	6.5	3.0	21	5.8
Post-Processing IP	4007	5.7	3024	2.1	8.0	3.7	106	29.4
FIFOs	146	0.2	122	0.1	9.0	4.2	0	0.0
DMAs	4062	5.8	5833	4.1	22.5	10.4	0	0.0
Others	6007	8.5	9098	6.5	0.0	0.0	0	0.0
Total	45287	64.2	30260	21.4	212.5	98.4	154	42.8

TABLE V
TIMING SUMMARY FOR VARIOUS TYPES OF CONVOLUTIONS FLOWS

		Theoretical		Real		Extra		
	#Convs	**Time**	**%**	**Time**	**%**	**Time**	**%**	**%/Conv**
Dilated Flow	6	264	31.2	366	36.6	101	67.0	11.2
Partitioned Flow	7	67	8.0	90	9.0	22	14.6	2.1
Normal Flow	129	514	60.8	543	54.4	28	18.5	0.1
Total	142	847	100.0	999	100.0	152	100.0	0.7

whole network takes 999ms to run 1 image, which means it has a throughput of 1 frame per second. The theoretical time, without pre-processing and post-processing, is 847ms which would mean a theoretical throughput of 1.18 frames per second. It is relevant to point out that the Atrous convolutions and partitioned convolutions take 81.6% of the extra time involved in the network, while only making up for only 13 of the 142 total convolutions. Future work could improve this by making use of DMAs with Scatter and Gather which are able to store DMA transaction requests in sequence, thus eliminating the delay between waiting for a transaction to end and starting a new one. If the dilated and partitioned flows layers took as much extra time as the normal flow layers (average time factor is 1.11%) then the estimated throughput would be 1.07 FPS or 931 ms per image.

IX. CONCLUSION

This work succeeded in training a fire-detecting DeepLabV3+ with a Modified Aligned Xception neural network with a final mIoU accuracy of 91.53%. The network was quantized and fine-tuned to a Fixed Point representation of 4 bits for both weights and activations. The quantized network achieved a final mIoU accuracy of 92.76%. A network hardware accelerator was developed and implemented on a Xilinx Zynq UltraScale+ SoC ZU3EG. A Configurable Layer Processor processing flow was implemented where the same accuracy was obtained. A final throughput of 1 FPS was achieved with an estimated power consumption of 5.82W thus beating the initial requirement of 0.1 FPS and low power consumption. The hardware resources used for the network hardware accelerator were 45K LUTs, 30K Flip Flops, 212.5 BRAMs and 154 DSPs, which equates to roughly 55% of the available resources of the ZU3EG device. This work can fit in low to mid-range devices.

X. ACKNOWLEDGMENTS

This work was partially supported by a research grant through INESC-ID and Fundação para a Ciência e a Tecnologia (FCT) with reference BI| 2022/246 and by project IPL/2021/smartSPACE ISEL through Instituto Politécnico de Lisboa.

REFERENCES

[1] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” 2015.

[2] K. Smeda. (2019, Nov) Understand the architecture of CNN. Accessed 31-Dec-2021. [Online]. Available: https://towardsdatascience.com/understand-the-architecture-of-cnn-90a25e244c7

[3] C.-F. Wang. (2018, Aug) A basic introduction to separable convolutions. Accessed 31-Dec-2021. [Online]. Available: https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728

[4] F. Chollet, “Xception: Deep learning with depthwise separable convolutions,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1800–1807.

[5] S. Minaee, Y. Y. Boykov, F. Porikli, A. J. Plaza, N. Kehtarnavaz, and D. Terzopoulos, “Image segmentation using deep learning: A survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2021.

[6] Y. Xing, L. Zhong, and X. Zhong, “An encoder-decoder network based fcn architecture for semantic segmentation,” *Wireless Communications and Mobile Computing*, vol. 2020, pp. 1–9, 07 2020.

[7] “Visual object classes challenge 2012 (voc2012).” [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/#devkit

[8] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, “Encoder-decoder with atrous separable convolution for semantic image segmentation,” 2018.

[9] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, “Accelerating cnn inference on fpgas: A survey,” 2018.

[10] H. Harkat, J. M. Nascimento, and A. Bernardino, “Fire detection using residual deeplabv3+ model,” in *2021 Telecoms Conference (ConfTELE)*, 2021, pp. 1–6.

[11] K.-W. Chang and T.-S. Chang, “Efficient accelerator for dilated and transposed convolution with decomposition,” in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5.