Internet of Things 33 (2025) 101676

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot

Research article

Optimization strategies for neural network deployment on FPGA: An energy-efficient real-time face detection use case

Mhd Rashed Al Koutayni *, Gerd Reis, Didier Stricker

German Research Center for Artificial Intelligence, DFKI, 67663 Kaiserslautern, Germany

ARTICLE INFO

Keywords:
IoT
Edge computing
Artificial intelligence
Deep learning
Optimization
Hardware acceleration
Quantization
High-level synthesis
FPGA
System-on-a-chip (SoC)
Face detection

ABSTRACT

Field programmable gate arrays (FPGAs) are considered promising platforms for accelerating deep neural networks (DNNs) due to their parallel processing capabilities and energy efficiency. However, Deploying DNNs on FPGA platforms for computer vision tasks presents unique challenges, such as limited computational resources, constrained power budgets, and the need for real-time performance. This work presents a set of optimization methodologies to enhance the efficiency of real-time DNN inference on FPGA system-on-a-chip (SoC) platforms. These optimizations include architectural modifications, fixed-point quantization, computation reordering, and parallelization. Additionally, hardware/software partitioning is employed to optimize task allocation between the processing system (PS) and programmable logic (PL), along with system integration and interface configuration. To validate these strategies, we apply them to a baseline face detection DNN (FaceBoxes) as a use case. The proposed techniques not only improve the efficiency of FaceBoxes on FPGA but also provide a roadmap for optimizing other DNN-based applications for resource-constrained platforms. Experimental results on the AMD Xilinx ZCU102 board with VGA resolution ($480 \times 640 \times 3$) input demonstrate a significant increase in efficiency, achieving real-time performance while substantially reducing dynamic energy consumption.

1. Introduction

Field-programmable gate arrays (FPGAs) have emerged as promising platforms for accelerating deep neural networks (DNNs), particularly for resource-constrained applications requiring real-time performance and energy efficiency. Unlike traditional CPUs and GPUs, FPGAs offer parallel processing capabilities and customizable hardware configurations, enabling tailored optimizations for specific workloads, making them attractive for computer vision tasks where DNN inference demands both high throughput and low latency. A key advantage of FPGAs over GPUs is their superior timing predictability, as they can operate on a single image at a time with deterministic latency, making them more suitable for critical applications. In contrast, GPUs rely on batch processing for efficiency. Additionally, FPGA designs are custom solutions tailored to specific workloads, allowing fine-grained control over resource allocation and execution timing. However, deploying DNNs on FPGA platforms presents significant challenges, including limited computational resources, constrained power budgets, and complex system integration requirements.

A key challenge in FPGA-based DNN deployment is balancing computational efficiency with hardware constraints. While modern FPGAs provide considerable parallel processing potential, their on-chip memory and logic resources are limited compared to GPUs. Consequently, optimizing DNN models for FPGA execution involves various strategies to reduce computational overhead

* Corresponding author.
E-mail address: rashed.al_koutayni@dfki.de (M.R. Al Koutayni).

https://doi.org/10.1016/j.iot.2025.101676
Received 23 September 2024; Received in revised form 15 April 2025; Accepted 8 June 2025
Available online 10 July 2025

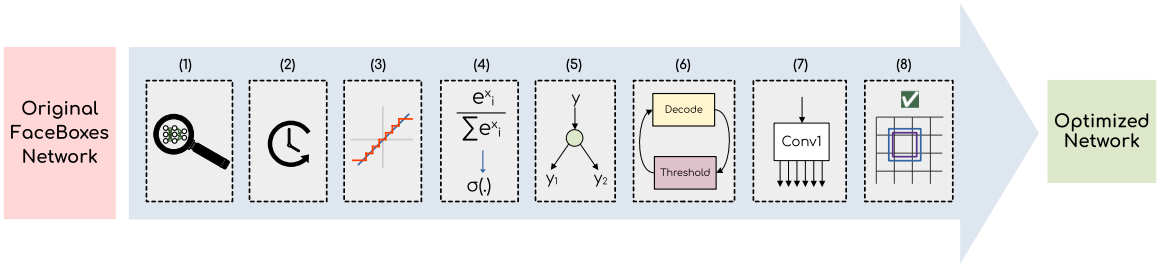

Fig. 1. Overview of the network optimizations, illustrating the sequential improvements applied to the baseline network (FaceBoxes). (1) Architectural optimizations, where the layers with minimal contribution are identified and removed. (2) Optimizing input writing time by moving the subtraction of dataset RGB average to the PL. (3) Quantization and datatype optimization. (4) Sigmoid-based softmax. (5) Split network outputs. (6) Swap post-processing steps: exponential decoding and thresholding. (7) Parallelize the first convolutional layer to increase the overall inference speed. (8) Offline generation of SSD priors to avoid repetition of calculations.

and enhance data flow efficiency. These optimizations include architectural modifications, fixed-point quantization, computation reordering, and parallelization techniques that exploit the FPGA's inherent parallelism. Additionally, efficient data movement and memory management play a crucial role in minimizing performance bottlenecks and energy consumption. Another critical aspect of optimizing DNN inference on FPGAs is hardware/software co-design. By strategically partitioning workloads between the processing system (PS) and programmable logic (PL), designers can optimize task allocation to maximize performance and minimize latency. The PS typically handles high-level control and preprocessing tasks, while the PL accelerates the core DNN computations. Effective hardware/software partitioning requires careful consideration of computation scheduling, memory access patterns, and interface configurations to ensure seamless integration between software and hardware components.

To validate these optimization methodologies, we apply them to a baseline face detection DNN, FaceBoxes [1], as a representative use case. Face detection is a fundamental computer vision task with applications in security, surveillance, and human–computer interaction. The FaceBoxes model, designed for real-time face detection, serves as an ideal benchmark to evaluate FPGA-based optimizations due to its balance of accuracy and computational efficiency. By systematically applying our proposed techniques, we demonstrate how FPGA-based face detection can achieve real-time performance while maintaining energy efficiency.

In fact, the single use case of face detection serves as a representative case study to demonstrate the proposed techniques for optimizing and deploying DNNs on FPGA SoCs. The proposed optimization strategies are broadly applicable to other DNN architectures. Specifically, any network with binary classification can use the sigmoid trick, latency bottlenecks can be parallelized, operations can be reordered, quantization reduces the computation, and input writing time can be minimized. While additional models could further validate the approach, our focus is on the optimization pipeline rather than benchmarking multiple networks. Future work could extend this methodology to other networks, but the current study effectively illustrates how the proposed techniques can be applied in practice.

The deployment of the DNN on the FPGA SoC is performed using our integrated framework DeepEdgeSoC [2]. The framework provides DeepGUI, a user-friendly interface for architecture visualization and editing without coding. Existing neural networks can be imported to DeepEdgeSoC using a parsing module called Tracer. Behind the scenes, a module called Fixifier extends PyTorch's neural network module, enabling quantization of parameters and activations to arbitrary bitwidths. Furthermore, The Exporter module extracts layer parameters and format them for hardware deployment. Lastly, DNN2HLS offers an AXI4 stream-based library for efficient hardware implementations of deep learning layers and operations.

Our experimental evaluation is conducted on the AMD Xilinx ZCU102 FPGA board, processing VGA-resolution ($480 \times 640 \times 3$) input images. The results highlight substantial improvements in inference speed and energy consumption compared to an unoptimized FPGA implementation. Specifically, our optimizations enable real-time face detection with significantly reduced dynamic energy usage, showcasing the effectiveness of our approach. Furthermore, the methodologies presented in this work provide a generalizable framework for optimizing other DNN-based applications on resource-constrained FPGA platforms.

Overall, this study contributes to the growing body of research on FPGA-based DNN acceleration by presenting a systematic approach to optimizing neural network inference for real-time applications. By demonstrating these techniques in the context of face detection, we offer insights that can be extended to a wide range of vision-based tasks and embedded AI applications. Future work will explore further refinements in hardware-aware DNN design, adaptive reconfiguration techniques, and broader applications of our optimization strategies to diverse FPGA architectures.

The main contributions of this work can be summarized in the following points:

- Efficient hardware design and optimization techniques to achieve high performance and low power consumption.
- An optimized implementation of the FaceBoxes model for face detection. Compared to the original model with 1.01 M parameters, the number of parameters is reduced to 98.15 K.

The paper is structured as follows: in Section 2, we compare our work to the related work in the literature. Section 3 illustrates the proposed approach of this work. Section 4 demonstrates the experimental setup. Section 5 provides quantitative as well as qualitative results. Section 6 discusses the strengths and limitations of this work. Lastly, the work is summarized and the improvement opportunities are highlighted in Section 7.

2. Related works

In this section, we examine the previous works after grouping them into two distinct categories: Conventional Face Detection on FPGA and Deep Learning-Based Face Detection on FPGA.

2.1. Conventional face detection on FPGA

Several studies have made significant contributions to the field of face detection. Rao et al. [3] introduced a dual-stage system architecture that combines skin tone detection with the Viola–Jones framework, achieving high accuracy in face tracking under time constraints. Nguyen et al. [4] demonstrated a system capable of accurately detecting faces of up to five individuals within a 1.5-meter range, processing 320 × 240 pixel images within several hundred milliseconds. Zivkovic et al. [5] implemented face detection using a high-speed VHDL on a ZYBO ZYNQ 7010 SoC, while Byun et al. [6] focused on designing and implementing real-time face detection using local patterns on an FPGA, analyzing hardware resource usage. Wang et al. [7] utilized the SDAccel tool from AMD Xilinx to create a heterogeneous computing platform for face detection, with the FPGA serving as a coprocessor. Chou et al. [8] reported a face detection accuracy rate of 96.14%, demonstrating the effectiveness of their real-time multi-face detection system. Spagnolo et al. [9] achieved a frame rate of up to 125 fps for QVGA resolution using a novel parallel processing scheme and adaptable run-time strategy. Fang et al. [10] developed a face mask detection algorithm using the Haar cascade classifier to detect facial features and mask-wearing status. Mohanty et al. [11] proposed acceleration techniques for real-time face detection on a CPU-FPGA platform using a state-of-the-art algorithm with numerous simple classifiers. Fekih et al. [12] designed a hardware architecture based on the Viola–Jones object detection system, achieving real-time face detection with a 6.46× speed-up over an equivalent OpenCV software solution. Finally, He et al. [13] introduced a novel SoC architecture for ultra-fast face detection using a robust algorithm with a cascade of ANN classifiers trained on AdaBoost Haar features.

2.2. Deep learning-based face detection on FPGA

Several advancements have been made in FPGA-based face detection systems. LPYOLO [14] redesigned and deployed the TinyYolov3 architecture for face detection, utilizing a CNN-based object detection method optimized for embedded systems with a high degree of parallelism applied to the FPGA's logical resources. Xu et al. [15] presented an FPGA-based real-time multi-face detection system aimed at crowded area surveillance, employing a CNN and proposing a hardware-friendly fully quantization strategy, with results tested on the WIDER FACE dataset. Fu et al. [16] designed an FPGA-based system to accelerate MTCNN, one of the most accurate face detection neural networks, making it a promising candidate for edge computing in mobile robot applications.

3. Proposed approach

The objective of this work is to demonstrate effective hardware optimization strategies by using FaceBoxes as a case study DNN and tailoring it for efficient deployment on the AMD Xilinx UltraScale+ MPSoC platform. While other networks exist (e.g., TinyYolo), FaceBoxes was chosen as a use case because, although it is a lightweight face detection model, it is originally designed to run on a CPU or GPGPU and is not inherently optimized for FPGA deployment. Although we focus on FaceBoxes as a single use case, the proposed optimization strategies are broadly applicable to other DNN architectures. Our emphasis remains on demonstrating an effective optimization pipeline rather than benchmarking multiple networks. This section begins with an overview of the baseline network, FaceBoxes. Subsequently, the comprehensive design process is outlined. Afterwards, the optimizations that have enabled the implementation on FPGA SoC are explained (Fig. 1). Finally, the end-to-end inference flow of the implemented network is highlighted.

3.1. Baseline network: FaceBoxes

FaceBoxes [1], illustrated in Fig. 2, is a single fully convolutional neural network (FCN) for face detection, initially built upon the architecture of single-shot detector (SSD). The network comprises two parts: rapidly digested convolutional layers (RDCL) and multiple scale convolutional layers (MSCL). RDCL is mainly responsible for shrinking the input by a factor of 32 using convolutional and pooling layers. The kernel size for these layers is chosen carefully so that a trade-off between efficiency and effectiveness is reached. Furthermore, C.ReLU activation function is used to double the number of output channels by concatenating the outputs with their negated version before applying ReLU. MSCL consists of 3 Inception modules followed by 4 convolutional layers. As the feature maps pass through these layers, the size of the corresponding receptive fields increases. Such multiscale receptive fields enhance the network's ability to handle faces of different sizes. Furthermore, the Inception modules enrich the receptive fields further by processing features via multiple convolution branches. In fact, this contributes to the network's ability to detect faces of different scales.

Similarly to SSD, each grid cell can be assigned with multiple anchors i.e., prior boxes. Anchors are predefined bounding boxes used to predict the location and size of objects in an image. By using anchors, the network only needs to predict the offset of the object from the anchor, rather than the entire bounding box. This reduces the search space as well as network training time. Anchors can also help to improve the accuracy of object detection by providing a good starting point for the network. In practice,

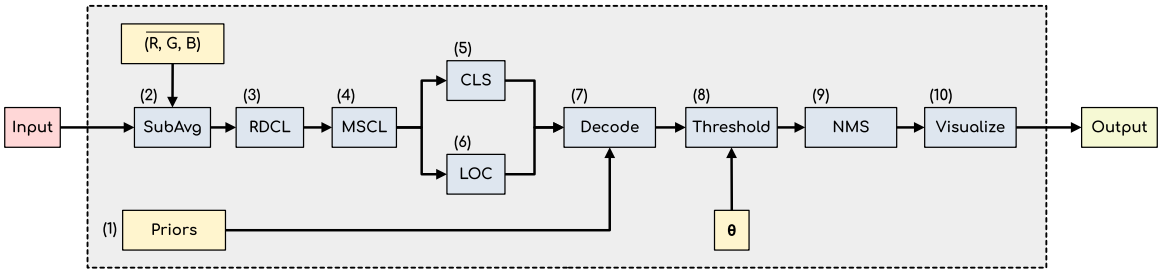

Fig. 2. FaceBoxes algorithm. (1) Generate priors. (2) Subtract the average RGB value of the entire dataset from the input to help center the input around zero and avoid dataset bias. (3) Generate downscale features using rapidly digested convolutional layers (RDCL). (4) Generate multiscale features in which each pixel corresponds to a receptive field in the input image using multiple scale convolutional layers (MSCL). (5) Classify each pixel (i.e. input receptive field) using a classification head (CLS). (6) Regress bounding boxes for each pixel (i.e. for each input receptive field) using a localization head (LOC). These 2 outputs are considered (raw outputs). (7) Decode outputs then match with predefined priors and ground truth. (8) Eliminate undesired results by applying a threshold θ on the confidence scores. (9) Remove duplicated detections using non-maximum suppression (NMS). (10) Visualize results using OpenCV.

each anchor box is specified by an aspect ratio and a zoom level. The network can then fine-tune the anchors to better match the ground truth boxes. Throughout training, the ground truth bounding boxes $(x_{min}, y_{min}, x_{max}, y_{max})$ are encoded to facilitate network training. Conversely, during the inference phase, the predicted bounding box values need to be decoded to their original locations to reverse the encoding applied during training. The network recall rate is further improved by applying anchor densification technique, which eliminates the anchor tiling density imbalance between small and large anchors. This involves adding extra small anchors around the center of a receptive field and consequently increase the network capability to detect small faces.

The network accepts an RGB input of size ($480 \times 640 \times 3$), and generates 2 raw outputs, namely localization regression and classification. These 2 outputs are flattened and concatenated to form a single output vector of size 384,000. The output size is independent of the number of faces in the images, as it only depends on the number of predefined anchors. Specifically, the localization regression output has the size [num_anchors, 4] while the classification output has the size [num_anchors, 2]. This is because each bounding box can be specified by 4 values, namely $(x_{min}, y_{min}, x_{max}, y_{max})$, while there are only 2 classes (face, background). Since the total number of anchors in our implementation is fixed at 64,000 by design, the final flattened output size is: $\text{num\_anchors} \times 4 + \text{num\_anchors} \times 2 = 64{,}000 \times 4 + 64{,}000 \times 2 = 384{,}000$.

3.2. Design process overview

The implementation is carried out on the ZCU102 Evaluation Board, leveraging the XCZU9EG-2FFVB1156E MPSoC. Within this architecture, the PS is a multiprocessor SoC that combines an ARM flagship Cortex-A53 64-bit quad-core processor with a Cortex-R5 dual-core real-time processor. On the other hand, the PL features 912 block RAMs, 2,520 DSP48 units, 274k look-up tables, and 548k flip-flops.

Since a SoC architecture is targeted, hardware/software partitioning is necessary to determine which part of the network algorithm executes on the PL so that the rest of the work is off-loaded to the PS. The partitioning decision is determined by the static or dynamic nature of the operations. Pre-processing includes resizing arbitrary-size inputs to a fixed size. This operation is inherently dynamic and therefore is executed on the PS. Similarly, post-processing tasks like thresholding, decoding, and the recursive non-maximum suppression (NMS), which handle dynamic output sizes and adaptive operations, are also suited for the PS. In contrast, DNN inference is well-suited for acceleration on the PL due to the fixed and parallelizable computation patterns. Therefore, we aim at implementing the computationally intensive part of the network on the PL, while the pre- and post-processing steps are handled by the PS.

The end-to-end design flow is performed using our integrated framework DeepEdgeSoC [2], which covers the design process from designing and training the network to FPGA implementation. In the beginning, the baseline FaceBoxes network is imported to DeepEdgeSoC. Afterwards, the evaluation method mentioned in the original paper is applied to assess the model performance. It is necessary to profile the network size and performance at this stage to obtain a baseline for later comparison. This is where the number of parameters, the number of multiply-accumulate (MAC) operations, inference run-time (i.e. number of frames per second) and network accuracy are observed and recorded. Generally, reducing the size of a DNN while maintaining good accuracy is considered a challenging task, yet it is successfully performed in this work. This includes multiple techniques such as identifying redundant layers and removing them, tweaking the number of channels and reordering operations while having negligible impact on accuracy. Although architectural modifications typically require retraining the network from scratch, the filters of the surviving layers can be initialized using the values of the original filters. This approach leverages the fact that these filters still retain useful features (e.g., quasi-orthogonality), allowing for fine-tuning rather than full retraining, which can significantly improve training times. Furthermore, a quantitative evaluation of the resulting network's performance is necessary. Optionally, a visual examination of the results (i.e. by running a dataset demo and/or a live demo) can give an additional impression of the network quality. It is worth noting that the training hyper-parameters and loss functions are adopted from the original paper.

The next step involves quantizing the network since it currently relies on 32-bit floating-point data types for both parameters and activations, resulting in excessive chip utilization. We aim at fitting the whole network including its parameters on-chip.

Quantization is the process of obtaining a fixed-point version of the network where arbitrary bit-widths can be used to represent each layer's parameters and activations. Represented as (ap_fixed<N,I>), this annotation signifies a fixed-point data type with N total bits and I integer bits, providing a flexible approach for numerical precision in hardware design. A fast and optimistic quantization methodology is post-training quantization (PTQ) in which the parameters and activations are quantized only after training is done. However, when the network accuracy drops significantly, quantization-aware training (QAT) should be followed instead. It is important to note that DeepEdgeSoC allows exploring different quantization bit-width combinations using PyTorch.

After performing the quantization, we transition from software to hardware design. In Vitis HLS, the network is simulated first so that its behavior can be cross-checked against its software version. Consequently, high-level synthesis is performed to convert the network to a synthesizable IP core. Afterwards, the system integration is conducted using Vivado where the interface between the PS, the PL and the DDR4 RAM is configured and the operation frequency is determined. The resulting bitstream can be deployed on the targeted FPGA to configure the PL. It is necessary to run a driver on the PS which is responsible for retrieving the input image, pre-processing it, firing the PL and post-processing the results. DeepEdgeSoC provides off-the-shelf layers that can be customized and aggregated via AXI streams to form an underlying streaming architecture for the desired network.

3.3. Hardware architecture

At the hardware level, the DNN is transformed into a streaming architecture by assigning each layer to a distinct hardware block, rather than using a single configurable engine. These blocks are connected through AXI4 stream channels, creating a dataflow pipeline. Unlike traditional computation, where each layer must wait for all inputs, a streaming layer can start processing as soon as it receives the necessary inputs from the previous layer. The entire model, along with its parameters, is stored on-chip, with the parameters saved in Block RAM (BRAM). The AXI interface between the PL and PS is configured such that the PL operates as the master, while the PS functions as the slave for both input and output transactions.

Fig. 3 illustrates the hardware architecture of convolutional and max pooling layers (a), batch normalization layers (b), and ReLU activation layers (c). The convolutional layer is a fundamental building block for feature extraction, applying learned filters to the input tensor through element-wise multiplications followed by accumulations. To efficiently implement this operation in hardware, the design utilizes two key memory structures: the Line Buffer and the Window Buffer. The Line Buffer temporarily stores input data until enough rows are available for processing, while the Window Buffer holds small patches of input data, ensuring they are readily accessible for parallel multiplications. Once the required input patch is loaded, a Multiplier Array performs simultaneous multiplications, and the results are passed through an Adder Tree to accumulate partial sums and add the learned bias before being streamed to the Output Buffer. A similar architectural approach is applied to the pooling layer, which reduces spatial dimensions and computational complexity. Instead of a Multiplier Array, pooling operations use comparators for max pooling. By leveraging the same line and window buffering techniques, pooling layers maintain efficiency while performing their respective operations. This structured hardware design ensures adaptability, making it suitable for accelerating deep learning models on FPGA platforms. The batch normalization layer normalizes each input channel using the running mean and variance, then scales and shifts the result with learned parameters. To optimize hardware efficiency, the normalization and scaling steps are combined into a single equation, replacing division with precomputed constants W and B, reducing on-chip resource usage. This allows batch normalization to be efficiently implemented using a simple multiplication and addition per channel, with parameters stored as ring buffers for sequential access during inference. The hardware implementation of the ReLU activation function is highly efficient, utilizing a simple multiplexer controlled by the sign bit of the input. If the input is positive, the sign bit remains zero, allowing the multiplexer to pass the value directly to the output. Conversely, for negative inputs, the sign bit triggers the multiplexer to output zero. This design ensures minimal computational overhead, making ReLU an ideal activation function for FPGA and hardware-accelerated deep learning applications.

3.4. Optimization details

In this subsection, the detailed implementation steps are explained along with the optimization techniques applied to enable the deployment on FPGA SoC. Due to the relatively large number of parameters, numerous layers and multiple skip connections, certain modifications are required. Thus, it became necessary to obtain a lightweight version of the network while striving to retain the highest possible accuracy.

3.4.1. Architectural optimization

Fig. 4 is a visual comparison between the original and optimized networks. The optimization process begins by identifying and removing layers with minimal contribution to the overall performance of the network. This is accomplished based on the ablation study mentioned in the original FaceBoxes paper as well as through trial and error investigation. As mentioned before, the filters of the surviving layers can be initialized with the values of the original filters, thereby notably decreasing training time. As a result of this exploration, the Inception module as well as a couple of convolutional layers from MSCL were removed. Furthermore, C.ReLU modules were replaced by conventional ReLU activations. This figure shows also changes in the number of channels. Similarly to the original FaceBoxes, the optimized network produces 2 raw outputs (localization, classification) from a single RGB input. Table 1 shows the final model details.

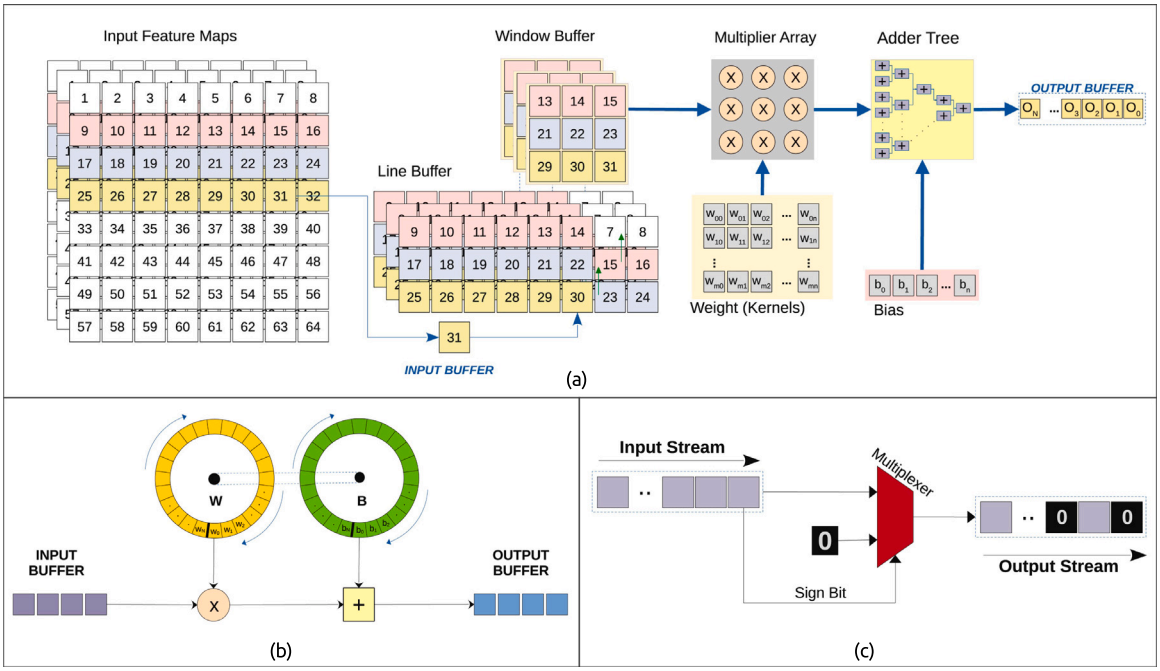

Fig. 3. Hardware blocks from DeepEdgeSoC: (a) Convolutional and max pooling layers. (b) Batch normalization layer. (c) ReLU activation layer.

Table 1
Face detection optimized architecture. CLS and LOC stand for Classification and Localization heads respectively. The input and output shapes are given as: H × W ×C.

Layer	Input shape	Kernel	Stride	Padding	Output shape	Activation
RDCL-Conv0	480 × 640 × 3	7 × 7	4 × 4	3 × 3	120 × 160 × 32	BN/ReLU
RDCL-MaxPool0	120 × 160 × 32	3 × 3	2 × 2	1 × 1	60 × 80 × 32	–
RDCL-Conv1	60 × 80 × 32	5 × 5	2 × 2	2 × 2	30 × 40 × 32	BN/ReLU
RDCL-MaxPool1	30 × 40 × 32	3 × 3	2 × 2	1 × 1	15 × 20 × 32	–
MSCL-Conv0	15 × 20 × 32	3 × 3	1 × 1	1 × 1	15 × 20 × 32	BN/ReLU
MSCL-Conv1	15 × 20 × 32	3 × 3	2 × 2	1 × 1	8 × 10 × 32	BN/ReLU
MSCL-Conv2	8 × 10 × 32	3 × 3	2 × 2	1 × 1	4 × 5 × 32	BN/ReLU
LOC-Conv0	15 × 20 × 32	3 × 3	1 × 1	1 × 1	15 × 20 × 84	–
LOC-Conv1	8 × 10 × 32	3 × 3	1 × 1	1 × 1	8 × 10 × 4	–
LOC-Conv2	4 × 5 × 32	3 × 3	1 × 1	1 × 1	4 × 5 × 4	–
CLS-Conv0	15 × 20 × 32	3 × 3	1 × 1	1 × 1	15 × 20 × 42	–
CLS-Conv1	8 × 10 × 32	3 × 3	1 × 1	1 × 1	8 × 10 × 2	–
CLS-Conv2	4 × 5 × 32	3 × 3	1 × 1	1 × 1	4 × 5 × 2	–

3.4.2. Optimizing input writing time

During the training, the average RGB value of the entire dataset is subtracted from the input channel by channel. This helps to keep the input distribution centered around the same mean value across all images in the dataset, ensuring training stability and achieving better generalization. The same subtraction operation needs to be performed during the inference phase (i.e., on the FPGA). Initially, 8 bits per channel are sufficient to represent the RGB input. However, after subtracting the average RGB value, this number increases to 9 bits. Naively, the subtraction operation is seen as a pre-processing step that must be implemented on the PS. Nonetheless, this would require transferring the data from the PS to the PL in 16-bit words, with 7 bits being redundant and unused in each transfer. This is because the AXI interface in SoC architectures requires data transfers between the PS and the PL to be aligned to byte boundaries or occur in units of bytes. To overcome this inefficiency, the subtraction operation is delegated to the PL, allowing the network input to remain as 8-bit words. Additionally, the efficient *memcpy()* function can be called on the PS to speed up the data transfer process from the off-chip memory to the PL.

3.4.3. Quantization and datatype optimization

Originally, the floating-point datatype is used for network parameters, activations, input and output. Quantization plays an important role in facilitating the implementation of the network on the FPGA. A 7-bit PTQ quantization scheme (ap_fixed<7,1>) for the network parameters is applied. Furthermore, intermediate activations are represented as fixed-point (ap_fixed<32,16>). As

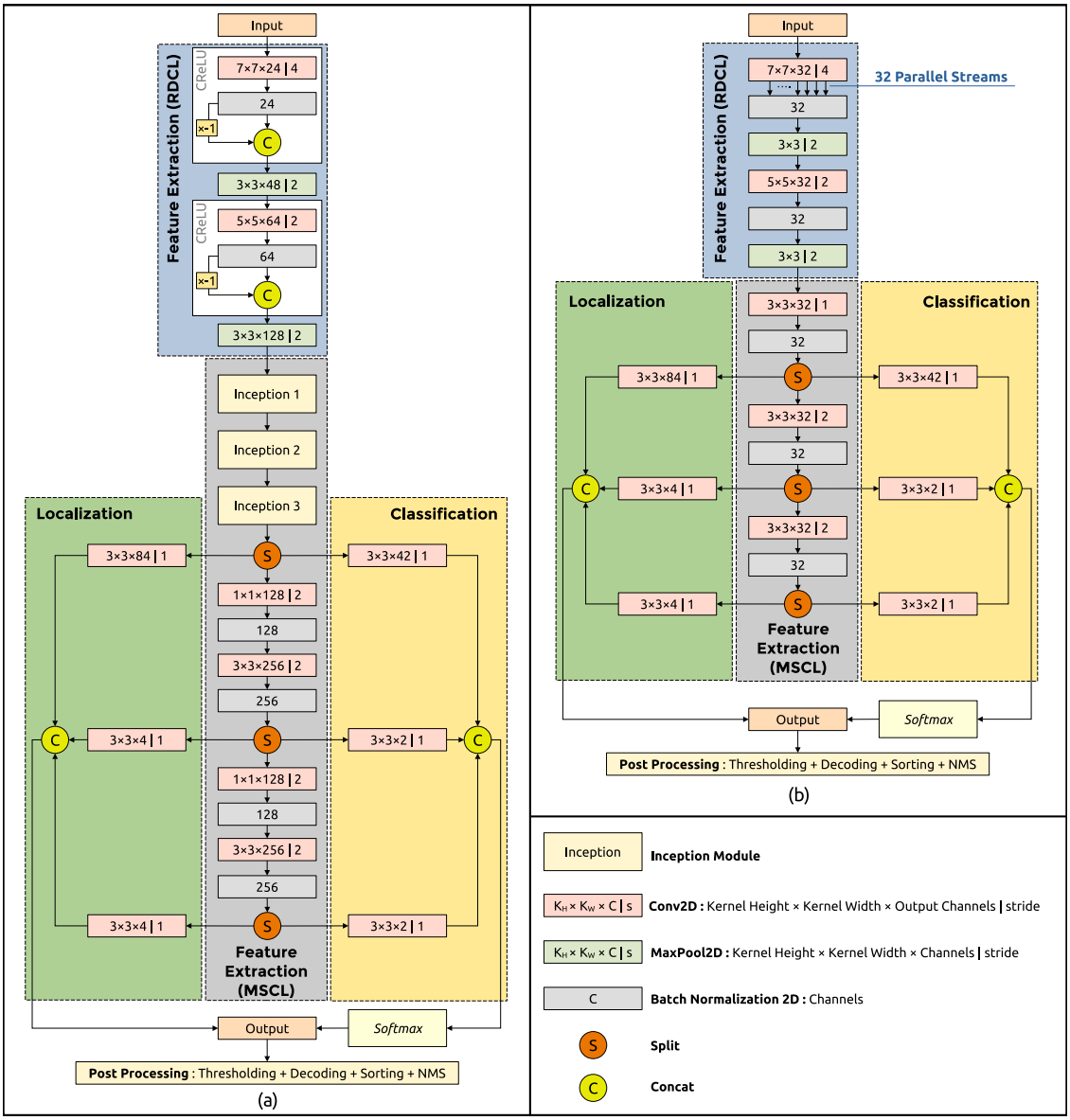

Fig. 4. FaceBoxes core network: (a) The original network. (b) The optimized network. The optimization process involves removing layers with minimal impact on performance, based on an ablation study and trial-and-error. This led to the removal of the Inception module and some convolutional layers from MSCL, as well as replacing C.ReLU with ReLU activations. The first layer was also parallelized to improve efficiency. Additionally, the number of channels was adjusted. The optimized network, like the original FaceBoxes, generates two raw outputs (localization and classification) from a single RGB input.

for the input, shifting the RGB average subtraction from the PS to the PL reduced the required datatype from (int16_t) to (uint8_t), as detailed in Sub Section 3.4.2. Finally, the raw output is converted from (ap_fixed<32,16>) to (int32_t) before being sent back to the PS for post-processing. Table 2 shows the network quantization summary.

3.4.4. Sigmoid-based softmax

Like any object detection task, face detection is a union of a localization task and a classification task. The latter is a binary decision task that indicates whether each prior box contains a face or background. Softmax (Eq. (1)) is used to convert the classifier's output into probabilities within the range $[0, 1]$ so that a fixed threshold (e.g., 0.6) can be used to filter out insignificant detections.

$$\text{Softmax}(x_1|(x_1, x_2)) = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} \tag{1}$$

Table 2
Network quantization summary.

Quantity	Quantization scheme
Parameters	ap_fixed<7,1>
Activations	ap_fixed<32,16>
Input	uint8_t
Output	ap_fixed<32,16>

where e^x is the exponential function and $\text{Softmax}(x_1|(x_1, x_2))$ is the Softmax function for the first class x_1 given a total of two classes (x_1, x_2). As in DeepEdgeSoC, Softmax is implemented as a modified Sigmoid function by dividing both the numerator and the denominator by the non-zero value e^{x_1}, as shown in Eq. (2).

$$\text{Softmax}(x_1|(x_1, x_2)) = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} = \frac{1}{1 + e^{x_2 - x_1}} = \frac{1}{1 + e^{-(x_1 - x_2)}} = \sigma(x_1 - x_2) \tag{2}$$

where $\sigma(x)$ is the sigmoid function. This approach reduces the computational complexity to a single lookup for subtraction. Furthermore, it is important to note that only half of the classification output is needed. This means that the confidence score only needs to represent one of the two classes (face or no face), rather than both simultaneously. As a result, the amount of classification data that needs to be transferred to the PS is reduced by half.

3.4.5. Split outputs

Before raising the "DONE" flag, the PL needs to flatten and concatenate both network outputs, as in the original network. However, waiting for the whole outputs to be ready before proceeding results in additional run-time delay. Since each raw output is computed by a separate network branch, this issue can be solved by allocating a separate Master AXI interface for each output. As a result, the outputs can be transferred concurrently and independently from the PL to the PS. In other words, while one output is still being computed, the other can already begin its transfer. This optimization helps to minimize the overall run-time delay of the face detection network and therefore improve its efficiency.

3.4.6. Swap thresholding and decoding

As mentioned in FaceBoxes description in Section 3.1, training involves encoding bounding boxes, which are then decoded in the inference phase. Encoding involves the use of the logarithm function ($ln(x)$), while Decoding depends on the exponential function (e^x). Since the inference is specifically being implemented on the FPGA, the focus is solely on optimizing the decoding step as part of the post-processing stage. As can be seen in the upper part of Fig. 5, these three post-processing steps are: datatype casting, decoding, and thresholding. Moreover, the lower part of Fig. 5 illustrates how these operations are reordered to increase the implementation efficiency. Originally, the PS receives the raw output from the PL in the form of (int32_t), which is converted into floating-point format. This datatype casting is required for the subsequent decoding step. Once the results are decoded, undesired detections with a confidence score below a predefined threshold (e.g. 0.6) are ignored to keep the meaningful ones. This raises the following question: Could a reordering of these steps increase the efficiency of the implementation while maintaining the correctness of the results? It is observed that decoding is a costly operation since it comprises exponentiation and multiplication. In fact, among the total number of decoded results (64,000), only a small portion of useful detections are required, while the decoding performed for the remaining results is redundant. Since thresholding is much cheaper than decoding, we optimize the workflow and therefore the detection run-time by reordering the three operations mentioned above. Instead of converting the output to the floating-point domain, it is kept in the original (int32_t) format, whereas the threshold is shifted to the (int32_t) domain by multiplying it by 2^{16} (left shift 16 times). In this case, the unnecessary results are discarded by applying the threshold in the (int32_t) domain. Afterwards, the filtered output is converted to floating-point. Finally, the computationally expensive decoding can be applied on the few surviving detections.

3.4.7. Parallelize the first convolutional layer

After applying the previous optimizations, the frame rate achieved is 17 fps. In order to improve it further, a comprehensive run-time analysis for each layer in the network is conducted based on the HLS report. As a result, the first convolutional layer is identified as the bottleneck. This layer employs a (7×7) kernel size on the VGA input $(480 \times 640 \times 3)$ to generate 32 feature maps. Typically, feature pixels flow sequentially between consecutive layers through single stream pipes. To accelerate the network, the FPGA parallel capabilities are leveraged by modifying the first convolutional layer so that all 32 output pixels, each belonging to a separate channel, are calculated simultaneously. For this purpose, 32 parallel streams are set up between this layer and the subsequent batch normalization layer. This optimization elevated the throughput from 17 to 55.44 fps. Fig. 4-b shows the parallel streams between the first convolutional and batch normalization layers.

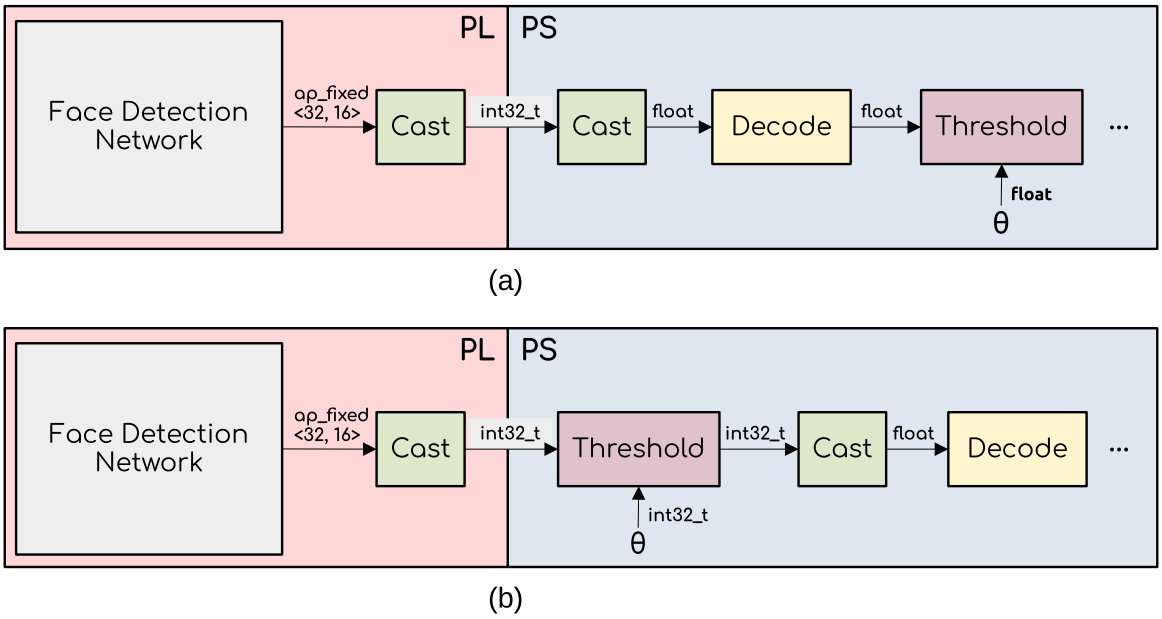

Fig. 5. Reordering thresholding and decoding operations on the PS. (a) The original workflow, where the raw output from the PL (int32_t) is first converted to floating-point format, followed by decoding and thresholding to discard detections below a confidence score (e.g., 0.6). (b) The optimized approach, where thresholding is applied first in the int32_t domain by scaling the threshold (e.g., left shift by 16 bits). This eliminates unnecessary detections before converting the remaining results to floating-point, allowing decoding to process only the filtered outputs, improving overall efficiency.

3.4.8. Offline generation of priors

As mentioned in Section 3.1, priors (or anchors) are predefined default bounding boxes chosen with specific sizes, aspect ratios and location in the image. It should be emphasized that the model predicts offsets for each predefined prior box rather than the entire bounding box. While in the original FaceBoxes implementation the same prior boxes are generated anew for every input image, the inference run-time can be reduced by generating these priors once at compile time. This can be achieved by dumping them as an array in a C++ header file which can be then included in the PS driver. This approach allows for the reuse of the same prior values to decode the encoded bounding box predictions for each inference iteration.

3.5. Inference flow

Fig. 6 illustrates the inference flow of the optimized system i.e., how an input image travels through the network and how detection results are obtained. The complete inference journey involves three phases: (1) input pre-processing, (2) network execution and (3) post-processing. In the pre-processing phase, input RGB images are read from local file (in case of offline demo) or USB camera (in case of online demo) and resized to ($480 \times 640 \times 3$). Afterwards, the input pixels are transferred as (uint8_t) to the memory space where the PL expects the input to be available. The PS driver triggers the PL by setting the *Start* flag to first subtract the dataset average RGB value then to run the network. Once the PL raises the *Done* flag, the raw output is ready to be transferred back to the PS for post-processing.

As soon as the raw output is completely received by the PS, a thresholding function is invoked to eliminate low score detections. Afterwards, the detections are decoded to reverse the offset regression encoding done during training. Lastly, non-maximum suppression (NMS) is invoked to remove overlapped duplicated detections of the same face. At this stage, the end-to-end face detection task is complete and the final results can be visualized by overlaying the resulting bounding boxes on the original input image. It is worth mentioning that for the AXI4 interface betweeen the PS and the PL, the PL plays the master's role (AXI_M) and pushes the raw results to the memory where the PS can read them via its slave (S_AXILite) interface for post-processing.

4. Experimental setup

In this section, the datasets used for network training and evaluation are mentioned. Afterwards, the training strategy is presented. Subsequently, the performance metrics used to evaluate face detection performance are explained.

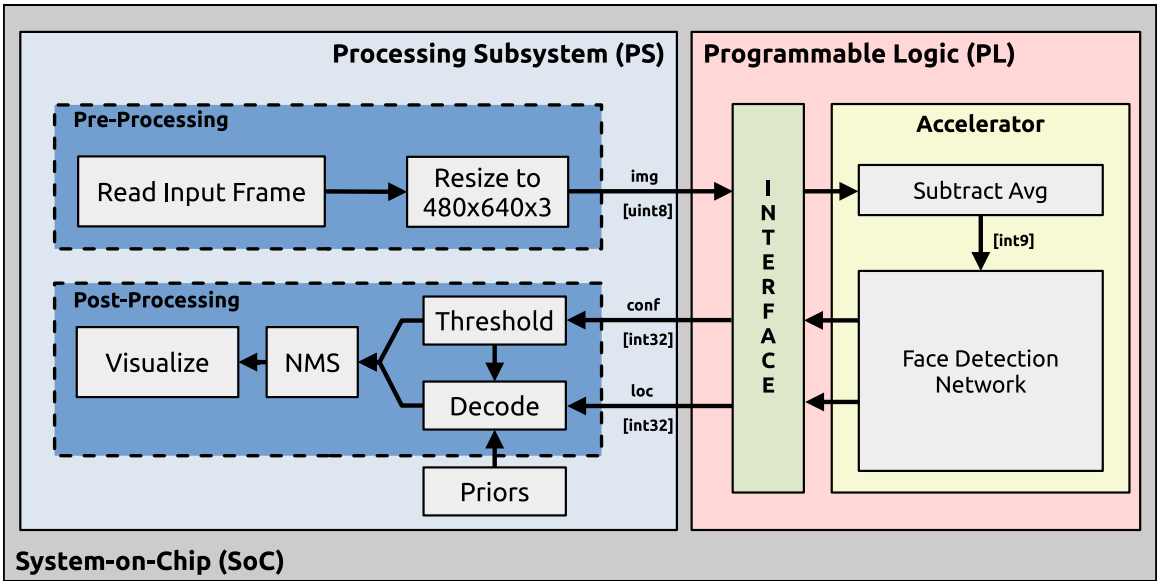

Fig. 6. SoC-based face detection inference flow. The inference process consists of three phases: (1) pre-processing, (2) network execution, and (3) post-processing. In pre-processing, input images are resized and transferred to the PL. The PS triggers the PL to subtract the average RGB and run the network. Once the PL completes, the raw output is sent to the PS for post-processing, which includes thresholding low-score detections, decoding, and applying non-maximum suppression (NMS) to remove duplicates. The final results are visualized by overlaying bounding boxes. The AXI4 interface allows the PL to push results to memory for the PS to process.

4.1. Datasets

Throughout the development of the real-time face detection network on the FPGA, the following datasets are required:

- **WIDER FACE** [17] dataset consists of 32,203 images with a total amount of 393,703 labeled faced of different scales and poses. This dataset is split into three subsets. The training, validation and testing subsets represent 40%, 10% and 50% of the whole dataset (i.e. 12,280, 3226 and 16,097 images) respectively.
- **Annotated Faces in the Wild** [18] (AFW) dataset involves 205 images with 468 faces. This dataset is used for testing only, since it involves a small quantity of images.

4.2. Training strategy

Following the same training strategy mentioned in FaceBoxes paper, the optimized face detection network was trained using the WIDER FACE dataset, which provides a diverse set of images with varying face scales, occlusions, and poses. To enhance the model's generalization ability, data augmentation techniques such as random cropping, horizontal flipping, and color distortion were applied. The loss function used during the training process is a combination of Cross Entropy Loss for binary classification and Smooth L1 Loss for bounding box regression. The model is trained on an NVidia GTX 1070 GPU for 300 epochs using the SGD optimizer with a batch size of 32, incorporating weight decay of 5×10^{-4}. The initial learning rate is 10^{-3}, which was reduced to 10^{-4} and 10^{-5} at the beginning of the 201^{st} and 251^{st} epochs respectively. Training settings are summarized in Table 3.

4.3. Performance metrics

Before presenting the quantitative results, it is important to first highlight the metrics used to evaluate DNN implementations. Mean Average Precision (mAP), frame rate, and dynamic energy consumption are among the most significant performance indicators.

4.3.1. Mean Average Precision (mAP)

In object detection, precision and recall are essential metrics for assessing the accuracy of the detector. Precision is calculated as the ratio of true positive detections (correctly predicted faces) to the total number of detections made (both true positives and false positives). Mathematically, precision is expressed as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{3}$$

Table 3
Training settings.

Hyperparameter	Value
Optimizer	SGD
Momentum	0.9
Weight decay	5×10^{-4}
Batch size	32
Number of epochs	300
Learning Rate	10^{-3}
Loss Functions	Cross Entropy Loss for binary classification and Smooth L1 Loss for regression.
Data augmentations	Color distortion Random cropping Horizontal flipping Hard negative mining
GPU	NVidia GTX 1070
Framework	PyTorch

On the other hand, recall measures the ratio of true positive detections to the total number of ground truth faces (the actual faces present in the image). It reflects the detector's ability to find all relevant faces. Recall is calculated as:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{4}$$

To calculate mean average precision (mAP), the system computes precision and recall at different thresholds of confidence for each detection, then calculates average precision (AP) for each class by averaging precision values at multiple recall levels. The mAP is the mean of these AP scores across all face detection categories, providing a comprehensive measure of detection quality. When calculating precision and recall, a detection is considered a true positive if the Intersection over Union (IoU) between the predicted bounding box and the ground truth bounding box exceeds a threshold (typically 0.5), a false positive if it does not match any ground truth, and a false negative if a ground truth face is not detected. In this work, we use the mAP calculation script[1] provided by Cartucho et al. [19].

4.3.2. Frame rate

The frame rate (fps) indicates the number of frames that the system can process per second, serving as a measure of its real-time performance. It can be calculated as the inverse of the inference run-time T_{image}, which denotes the time required to process a single frame. In practice, a relatively large number of images (e.g., 1000) is fed into the system, and the average frame inference run-time is calculated. Notably, to ensure fair comparison, both pre- and post-processing are included in the run-time measurement.

4.3.3. Dynamic energy

Measuring energy consumption is crucial in evaluating the efficiency of modern computational systems, particularly in resource-constrained environments such as embedded systems, mobile devices, and edge computing platforms. Energy efficiency directly impacts battery life, operational costs, and the environmental footprint of a system. The main objective is to evaluate the dynamic energy consumption required to process each frame effectively. Therefore, the idle power consumption P_{idle} is first measured. This is the power that the hardware platform withdraws without running the network. Afterwards, the total power consumption P_{total} is measured during inference. Given the inference run-time T_{image}, the dynamic energy required by the hardware to process a single frame is given as: $E_{image} = (P_{total} - P_{idle}) \times T_{image}$.

It is worth noting that both P_{total} and P_{idle} correspond to the power consumption of the entire SoC, not just the PL. This means that the measured power includes contributions from both pre-and post-processing operations. Additionally, the power required for I/O traffic is also accounted for, ensuring a comprehensive evaluation of the system's overall power consumption.

Since related works report power/energy consumption for varying input sizes, directly comparing energy consumption values may not provide a fair evaluation. Instead, computing E_{pixel}, which represents the energy consumption per processed pixel, offers a more standardized and meaningful basis for comparison. This metric accounts for differences in input resolution, enabling a more accurate assessment of energy efficiency across different implementations.

4.3.4. Energy-Accuracy Trade-off Score (EAT Score)

The energy consumption per processed pixel E_{pixel} expresses implicitly three main performance aspects, namely run-time, power consumption, and input size. However, it cannot be used as a holistic comparison score, because it does not take the accuracy into account. To further enhance the fairness of comparison with related works, we introduce the energy-accuracy tradeoff (EAT) score, which quantifies the relationship between accuracy and energy consumption per pixel. The EAT score is defined as:

$$\text{EAT} = \frac{mAP}{E_{pixel}} \tag{5}$$

[1] https://github.com/Cartucho/mAP

The rationale behind this formula is to establish a comprehensive metric that is directly proportional to mAP while being inversely proportional E_{pixel}, ensuring a balanced evaluation of both performance and energy efficiency. A higher EAT score indicates a more efficient model, achieving better accuracy with lower energy consumption. This metric allows a holistic and fair comparison across different architectures, ensuring that energy-efficient designs do not disproportionately sacrifice accuracy. By incorporating both factors into a single metric, the EAT Score enables a more meaningful assessment of various implementations, ensuring that energy-efficient designs do not compromise detection quality.

5. Results

In this section, the results of our FPGA-based real-time face detection are presented, with a comparison to related works in terms of accuracy, speed, and power consumption.

5.1. Cross-platform performance

The proposed FPGA SoC implementation is compared to two existing embedded platform implementations, namely NVidia Jetson AGX Xavier (JX) and RaspberryPi 3B+. The NVidia Jetson AGX Xavier features a 512-core Volta GPU with Tensor Cores, an 8-core ARM v8.2 64-bit CPU, and 16 GB of 256-bit RAM. It was selected as it is a widely used embedded GPU. This provides a strong baseline for evaluating energy-efficient DNN acceleration. In comparison, the Raspberry Pi 3B+ is equipped with a Broadcom BCM2837B0 SoC, which includes a quad-core Cortex-A53 (ARMv8) 64-bit CPU running at up to 1.4 GHz, along with 1 GB of SDRAM. We had to rerun the network on NVidia GTX 1070 in order to measure the power and energy, since the GPU used in the original FaceBoxes paper is not reported.

It can be seen from Table 4 that the Raspberry Pi 4 demonstrates very limited performance, achieving only 0.33 fps with a high energy consumption of 6621 mJ per image, making it impractical for real-time tasks. The NVIDIA Jetson Xavier CPU running the original model reaches 9.11 fps at 3230 mJ. After applying optimization techniques without quantization, the performance improves to 20.17 fps and 1510 mJ, while further quantization to FxP7 maintains similar speed at 19.61 fps but also raises energy to 2210 mJ.

On the NVIDIA GTX 1070, the original model achieves 53 fps with an energy cost of 1050 mJ per image. Optimizing the model without quantization pushes the frame rate up to 64.3 fps and lowers energy consumption to 875 mJ. Using the natively supported FP16 quantization achieves a high throughput of 62.7 fps with an energy cost of 896 mJ and 91.59% mAP. The slight decrease in the frame rate and the slight increase in the energy consumption are most likely because the data is fetched from the memory as FP32 then converted into FP16. However, the difference is larger when using 7-bit fixed-point quantization as it leads to a reduced frame rate of 38.2 fps and increased energy consumption of 1413 mJ, with a slight improvement in accuracy to 91.73%. The differences in speed and energy consumption are coming from the lack of native support for 7-bit quantization on the GPU, which results in additional computational overhead during inference.

Looking at the NVIDIA Jetson Xavier GPU, the original model achieves 43.09 fps with 306 mJ energy per image, already indicating a strong balance between speed and efficiency. With optimization but no quantization, performance rises to 56.79 fps, though energy also increases to 217 mJ. Further quantization with FP16 gives 53.16 fps at 229 mJ. Meanwhile, similarly to GTX 1070, 7-bit fixed-point quantization reduces the frame rate to 29.87 fps and increases energy consumption to 412 mJ, again reflecting the inefficiencies of using non-native precision.

Compared to GPU-based implementations, our ZCU102 design achieves a good trade-off between accuracy, speed, and energy. While maintaining an accuracy of 91.73%, our implementation reaches a frame rate of 55.44 fps. However, where the ZCU102 clearly stands out is in energy efficiency: it consumes only 46 mJ per image, which is significantly lower than all GPU implementations. For instance, the closest configuration on the JX GPU (opt, no quant) requires 217 mJ, nearly 5× more, while the best GTX 1070 variant (opt, no quant) consumes 875 mJ, making our design nearly 19× more efficient.

5.2. Comparison with related FPGA implementations

Table 5 compares various methods for real-time image processing across diverse platforms, emphasizing metrics like frame rate (fps), accuracy (mAP), power consumption (P, mW), energy per frame (E_{image}, mJ), energy per pixel (E_{pixel}, nJ), and energy-accuracy trade-off score (EAT Score). Among these methods, our design achieves a notable balance between performance and energy efficiency. With an input size of 480 × 640, our implementation delivers a frame rate of 55.44 fps and an mAP of 91.73%, consuming 46.25 mJ per frame and 150.55 nJ per pixel.

In comparison, other methods either sacrifice energy efficiency or performance. For example, the approach by Spagnolo et al. achieves a higher frame rate of 125 fps with a slightly better mAP of 97.0%, but its smaller input size of 320 × 240 results in limited applicability for larger images. Meanwhile, Mohanty et al. consume significantly more energy per frame (4934.69 mJ) and per pixel (64,253.75 nJ), despite operating at a lower frame rate of 6.89 fps. Similarly, LPYOLO consumes 133.33 mJ per frame and 770.46 nJ per pixel for a smaller frame rate of 18 fps and mAP of 75.7%. Chou's implementation balances good frame rates and high accuracy. Fang achieves high frame rates for larger inputs. However, both Chou and Fang do not report power consumption. Based on the EAT score, our implementation achieves the best trade-off, demonstrating superior energy efficiency without compromising accuracy, outperforming all other works. Note that Rao's work reports only the PL power and energy consumption. Table 6 presents a comparison of hardware utilization metrics for various AMD Xilinx FPGA-based object detection implementations. The metrics include the utilization of BRAMs, DSPs, flip-flops, and look-up tables on their respective platforms.

Table 4
Profiling original and optimized FaceBoxes network on different hardware platforms. Params indicates the number of parameters. FP32, FP16 and FxP7 stand for 32-bit Floating-Point, 16-bit Floating-Point and 7-bit Fixed-Point respectively. Par and Act denote the datatype used for parameters and activations, respectively. MAC stand for multiply-accumulate operations. The frame rate measurement takes the pre- and post-processing into account. The accuracy is measured as mean average precision (mAP). E_{image} is the dynamic energy per frame measured for the whole platform. "Orig" and "opt" refer to the original and optimized networks, respectively. "quant" refers to applying PTQ on the network, while "no quant" indicates that no quantization is applied.

Platform	Params (K)	Datatype (Par/Act)	MACs (G)	Frame Rate (fps)	Accuracy (mAP %)	E_{image} (mJ)
Raspberry Pi4	1.01 M	FP32/FP32	0.28	0.33	98.65	6621
NVidia JX CPU (orig)	1.01 M	FP32/FP32	0.28	9.11	98.65	3230
NVidia JX CPU (opt, no quant)	98.15 K	FP32/FP32	0.14	20.17	91.59	1510
NVidia JX CPU (opt, quant, FxP7)	98.15 K	FxP7/FxP32	0.14	14.16	91.73	2210
NVidia GTX 1070 (orig)	1.01 M	FP32/FP32	0.28	53	98.65	1050
NVidia GTX 1070 (opt, no quant)	98.15 K	FP32/FP32	0.14	64.3	91.59	875
NVidia GTX 1070 (opt, quant, FP16)	98.15 K	FP16/FP16	0.14	62.7	91.59	896
NVidia GTX 1070 (opt, quant, FxP7)	98.15 K	FxP7/FxP32	0.14	38.2	91.73	1413
NVidia JX GPU (orig)	1.01 M	FP32/FP32	0.28	43.09	98.65	306
NVidia JX GPU (opt, no quant)	98.15 K	FP32/FP32	0.14	56.79	91.59	217
NVidia JX GPU (opt, quant, FP16)	98.15 K	FP16/FP16	0.14	53.16	91.59	229
NVidia JX GPU (opt, quant, FxP7)	98.15 K	FxP7/FxP32	0.14	29.87	91.73	412
ZCU102 (Ours)	98.15 K	FxP7/FxP32	0.14	55.44	91.73	46

Table 5
Comparison with related FPGA implementations of face detection, detailing metrics such as input size (pixels), frame rate (fps), mean average precision (mAP), power consumption (P), energy per image (E_{image}), energy per pixel (E_{pixel}), and energy-accuracy trade-Off score (EAT Score). The table highlights the trade-offs between accuracy, performance, and energy efficiency across diverse hardware configurations and optimization strategies. Fields marked with '-' are not reported. For the work done by Mohanty et al. we calculated the accuracy value based on the precision–recall graphs provided in the referenced paper.

Method	Input Size	Frame Rate (fps)	Accuracy (mAP %)	P (mW)	E_{image} (mJ)	E_{pixel} (nJ)	EAT Score
LPYOLO [14]	416 × 416	18	75.7	2400[a]	133.33[a]	770.46[a]	0.98
Xu et al. [15]	512 × 288	37	84.6	–	–	–	–
Fu et al. [16]	320 × 240	11.9	–	–	750.0	9765.63	–
Fekih et al. [12]	480 × 640	16.53	92.0	–	–	–	–
Byun et al. [6]	480 × 640	60	–	–	–	–	–
Chou et al. [8]	480 × 640	60	96.14	–	–	–	–
Fang et al. [10]	1280 × 720	45.79	96.5	–	–	–	–
He et al. [13]	80 × 60	625	–	5950[a]	9.52[a]	1983.33[a]	–
Nguyen et al. [4]	480 × 640	6.55	92.0	–	–	–	–
Mohanty et al. [11]	320 × 240	6.89	87.5	34,000	4934.69	64,253.75	–
Rao et al. [3]	480 × 640	16.53	94.5	90.2[b]	5.46[b]	17.76[b]	53.21[b]
Spagnolo et al. [9]	320 × 240	125	97.0	1700	13.60	177.08	5.48
Zivkovic et al. [5]	320 × 240	0.7	79.0	134	191.43	2492.56	0.32
Wang et al. [7]	320 × 240	66.6	–	–	–	–	–
Ours	480 × 640	55.44	91.73	2564.1	46.25	150.55	6.16

[a] Represent total SoC power and energy consumption, including P_{idle}, as no breakdown of power or energy usage is provided in the referenced papers.
[b] Represent only PL power and energy consumption, as PS power or energy is not provided in the referenced papers.

5.3. Qualitative results

Fig. 7 illustrates a visual example of face detection performed by the developed system using the AFW data set. The blue bounding boxes are the groundtruth boxes, while the green bounding boxes are the post-processed detections. It can be observed that, while the detections are not perfect, they closely approximate the ground truth for various face sizes. This level of performance is notable given the significant reduction in the number of parameters and the corresponding improvement in energy efficiency.

6. Discussion

This section will address the limitations of our research and explore specific aspects of our findings that may influence their generalizability.

Complexity-performance trade-off. Considering the available space in the FPGA fabric, further optimizations to the network architecture are possible. However, such changes might influence the model's accuracy, as they often involve trade-offs between computational complexity and performance. Likewise, although additional hardware-level optimizations might improve computational efficiency, they might also lead to higher energy consumption due to more intensive utilization of the FPGA fabric. Therefore,

Table 6
Comparison of FPGA resource utilization across various methods and platforms using AMD Xilinx FPGAs, showing number of units of BRAMs, DSPs, flip-flops, and look-up tables used. Fields which are marked with '-' are not reported.

Method	Platform	Block RAM	DSP	Flip-Flops	Look-Up Tables
LPYOLO [14]	PYNQ-Z2	91	202	54,264	39,368
Xu et al. [15]	ZCU104	–	–	–	–
Fu et al. [16]	ZC706	196	880	222,456	133,783
Fekih et al. [12]	Z706	84	33	9908	11,047
Byun et al. [6]	Virtex-7	–	–	49,096	41,149
Fang et al. [10]	PYNQ-Z2	–	–	–	–
He et al. [13]	Virtex-5	276	161	37,828	67,704
Rao et al. [3]	Virtex 4VSX35	–	–	–	–
Spagnolo et al. [9]	XC7Z020	37	39	6481	7259
Zivkovic et al. [5]	ZYBO ZYNQ 7010	59	8	2271	3462
Wang et al. [7]	KCU1500	86	55	12,996	25,992
Ours	ZCU102	195	314	31,257	111,548

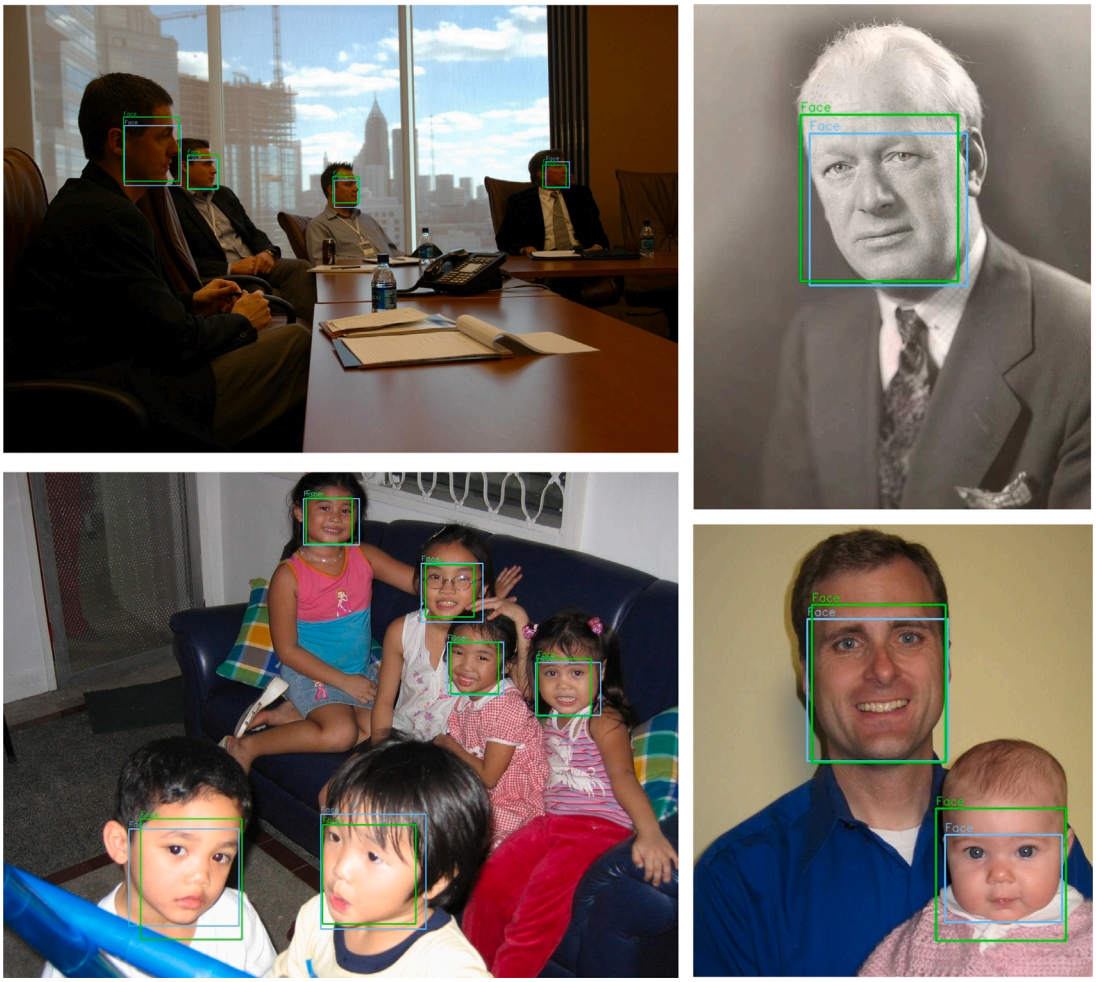

Fig. 7. Visual representation of face detection results produced by the proposed system. The figure compares detected bounding boxes (in green) against ground truth annotations (in blue) for various face sizes. It can be seen that even small far faces can be detected accurately despite the reduced number of parameters. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the current design strikes a balance that meets the stated objectives by achieving satisfactory performance, accuracy, and energy efficiency within the given constraints. Our focus in this work is not on exploring different design points through NAS but rather on demonstrating effective methodologies for optimizing and deploying deep neural networks on FPGA SoCs. The selected trade-off between accuracy and throughput aligns with our objective of showcasing deployment strategies rather than searching for the

optimal network architecture. While further reductions in model complexity could yield higher throughput, such explorations fall outside the scope of this work.

Acceptable accuracy drop. The acceptable accuracy drop largely depends on the specific application and its tolerance for errors, such as false positives or false negatives. For instance, in real-time embedded systems, such a small reduction in accuracy might be acceptable if it results in significant gains in speed and energy efficiency. However, for safety-critical applications, even a minor drop in accuracy may not be acceptable, and further optimizations should aim to balance both accuracy and performance.

Sigmoid-based softmax. While it is true that softmax for more than two classes cannot be directly replaced by a sigmoid function, our approach is specifically tailored for binary classification, which is widely used in many real-world applications, including face detection, object presence verification, and medical diagnosis. Using a sigmoid function in this case is a well-established practice that significantly reduces computational complexity while maintaining accuracy.

NVidia Jetson AGX Xavier energy efficiency. When comparing our FPGA SoC implementation to the NVidia Jetson AGX Xavier GPU implementations (Table 4), the energy efficiency gain ranges from 6.6× and 8.9×. This is less than the 10× energy efficiency gain that is often expected with FPGA implementations of optimized DNNs. Although these results reflect significant energy savings and performance improvements, they do not meet the 10× energy efficiency gain that is often anticipated with FPGA implementations of optimized DNNs. The limitation in energy savings arises due to the post-processing operations, such as NMS and decoding. These operations involve dynamic computations, including handling variable-sized outputs and overlapping detections, which consume additional energy. Moreover, the integration of both the PL for the network and the PS for post-processing means that energy savings are distributed across the entire chip, diluting the efficiency gains achieved in the PL alone.

DSP throughput. Looking at the number of DSPs utilized in our implementation (314), the theoretical peak performance is expected to be around 30 GMACS at 100 MHz. This, however, assumes ideal conditions where all DSPs operate at full capacity without any stalls or memory bottlenecks. On the other hand, the achieved frame rate of 55.44 fps is not purely based on computation time but also includes the time taken to read input data from off-chip memory and write processed outputs back to memory. These I/O operations introduce additional latency, making the measured performance less than the raw computational peak. Furthermore, the proposed design follows a dataflow streaming architecture, where layers are mapped to dedicated hardware blocks interconnected via AXI stream channels. This architecture balances computation and memory access efficiency but does not guarantee full DSP utilization at all times. In addition to that, pipeline inefficiencies, control overhead, and scheduling dependencies introduce practical limitations on achieving peak MAC utilization. Lastly, DNN workloads are inherently irregular, and different layers have varying computational intensities, leading to variations in hardware utilization. Despite these constraints, the proposed design achieves a frame rate of 55.44 fps while maintaining a strong balance between efficiency, accuracy, and resource usage. Furthermore, compared to existing FPGA-based DNN accelerators, our design demonstrates competitive performance while optimizing for real-time processing on an embedded FPGA-SoC platform.

7. Conclusion and future work

This work presents a comprehensive approach for optimizing and deploying DNNs on FPGA SoC platforms, with a focus on achieving energy-efficient, real-time performance for resource-constrained applications. We have demonstrated a set of optimization strategies, including architectural modifications, fixed-point quantization, computation reordering, and parallelization, to enhance the performance of DNN inference on FPGA. Through a careful hardware/software co-design, we effectively partitioned workloads between the PS and PL to maximize throughput and minimize latency. The case study of FaceBoxes, a lightweight face detection model, served as an effective use case to validate the proposed optimizations. Experimental results on the ZCU102 board demonstrated the effectiveness of the introduced optimization techniques, achieving an impressive 55.44 fps frame rate with only 46.25 mJ energy consumption per frame, marking a 21× improvement in energy efficiency over the original network.

While this study has successfully demonstrated how the proposed optimization techniques can be applied to a specific use case, there are several open doors for future research. One key direction involves benchmarking the FPGA implementation against a wider range of embedded GPU platforms, including newer, low-power devices that have become increasingly common in edge AI applications. Additionally, while FaceBoxes served as a representative model in this work, applying the same optimization pipeline to other DNN architecture would help demonstrate the generality and scalability of the proposed techniques. Another important aspect to explore is the trade-off between throughput and accuracy; a more granular investigation into how different architectural modifications, quantization levels, and parallelization strategies affect this balance could inform design choices for specific application needs. Finally, extending the evaluation to include different input image sizes (e.g., 360 × 480, or full HD) would offer valuable insights into how resolution impacts system performance, energy consumption, and model accuracy.

CRediT authorship contribution statement

Mhd Rashed Al Koutayni: Visualization, Validation, Methodology, Investigation, Conceptualization. **Gerd Reis:** Supervision, Funding acquisition, Conceptualization. **Didier Stricker:** Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially funded by the Federal Ministry of Education and Research of the Federal Republic of Germany as part of the research project FAIRe (Grant number 01IS23074). Furthermore, this work has been partially funded by the European Union as part of the research project dAIEDGE (Grant number 101120726).

Data availability

The datasets used in this work are taken from other papers or benchmarks available online.

References

[1] S. Zhang, X. Zhu, Z. Lei, H. Shi, X. Wang, S.Z. Li, Faceboxes: A CPU real-time face detector with high accuracy, in: 2017 IEEE International Joint Conference on Biometrics, IJCB, IEEE, 2017, pp. 1–9.
[2] M.R. Al Koutayni, G. Reis, D. Stricker, DeepEdgeSoC: End-to-end deep learning framework for edge IoT devices, Internet Things 21 (2023) 100665, http://dx.doi.org/10.1016/j.iot.2022.100665, URL https://www.sciencedirect.com/science/article/pii/S2542660522001469.
[3] M. Rao, P.R. Kumar, T. Balaji, A high performance dual stage face detection algorithm implementation using FPGA chip and DSP processor, J. Inf. Syst. Telecommun. (JIST) 4 (40) (2022) 241.
[4] H.L. Nguyen, M.S. Nguyen, T.N. Do, Real-time face detection and human tracking system on FPGA Cyclone-V, REV J. Electron. Commun. 12 (3–4) (2022).
[5] M. Živković, M. Herceg, N. Pjevalica, M. Subotić, Face detection in images using an FPGA, in: 2021 Zooming Innovation in Consumer Technologies Conference, ZINC, IEEE, 2021, pp. 96–101.
[6] J.Y. Byun, J.W. Jeon, Face detection using local patterns in FPGA, in: 2021 15th International Conference on Ubiquitous Information Management and Communication, IMCOM, IEEE, 2021, pp. 1–2.
[7] J. Wang, W. Leng, Accelerating face detection algorithm on the FPGA using SDAccel, in: Quality, Reliability, Security and Robustness in Heterogeneous Systems: 15th EAI International Conference, QShine 2019, Shenzhen, China, November 22–23, 2019, Proceedings, Springer, 2020, pp. 154–165.
[8] K.-Y. Chou, Y.-P. Chen, Real-time and low-memory multi-faces detection system design with naive Bayes classifier implemented on FPGA, IEEE Trans. Circuits Syst. Video Technol. 30 (11) (2019) 4380–4389.
[9] F. Spagnolo, S. Perri, P. Corsonello, Design of a real-time face detection architecture for heterogeneous systems-on-chips, Integration 74 (2020) 1–10.
[10] T. Fang, X. Huang, J. Saniie, Design flow for real-time face mask detection using PYNQ system-on-chip platform, in: 2021 IEEE International Conference on Electro Information Technology, EIT, IEEE, 2021, pp. 1–5.
[11] A. Mohanty, N. Suda, M. Kim, S. Vrudhula, J.-s. Seo, Y. Cao, High-performance face detection with CPU-FPGA acceleration, in: 2016 IEEE International Symposium on Circuits and Systems, ISCAS, IEEE, 2016, pp. 117–120.
[12] H.B. Fekih, A. Elhossini, B. Juurlink, An efficient and flexible FPGA implementation of a face detection system, in: International Symposium on Applied Reconfigurable Computing, Springer, 2015, pp. 243–254.
[13] C. He, A. Papakonstantinou, D. Chen, A novel SoC architecture on FPGA for ultra fast face detection, in: 2009 IEEE International Conference on Computer Design, IEEE, 2009, pp. 412–418.
[14] B. Günay, S.B. Okcu, H.Ş. Bilge, LPYOLO: low precision YOLO for face detection on FPGA, 2022, arXiv preprint arXiv:2207.10482.
[15] H. Xu, Z. Wu, J. Ding, B. Li, L. Lin, J. Zhu, Z. Hao, FPGA based real-time multi-face detection system with convolution neural network, in: 2019 8th International Symposium on Next Generation Electronics, ISNE, IEEE, 2019, pp. 1–3.
[16] C. Fu, Y. Yu, FPGA-based power efficient face detection for mobile robots, in: 2019 IEEE International Conference on Robotics and Biomimetics, ROBIO, IEEE, 2019, pp. 467–473.
[17] S. Yang, P. Luo, C.-C. Loy, X. Tang, Wider face: A face detection benchmark, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 5525–5533.
[18] M. Koestinger, P. Wohlhart, P.M. Roth, H. Bischof, Annotated facial landmarks in the wild: A large-scale, real-world database for facial landmark localization, in: IEEE International Conference on Computer Vision Workshops, IEEE, 2011, pp. 2144–2151.
[19] J. Cartucho, R. Ventura, M. Veloso, Robust object recognition through symbiotic deep learning in mobile robots, in: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, IEEE, 2018, pp. 2336–2341.