Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

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

Research paper

A hardware accelerator to support deep learning processor units in real-time image processing

Edoardo Cittadini [*], Mauro Marinoni, Giorgio Buttazzo

Scuola Superiore Sant'Anna, Pisa, Italy

ARTICLE INFO

Keywords:
Hardware acceleration
Object detection
Deep neural networks
Real-time processing
Image pre-processing acceleration
Deep processing unit

ABSTRACT

Deep neural networks are becoming crucial in many cyber–physical systems involving complex perceptual tasks. For those embedded systems requiring real-time interactions with dynamic environments, as autonomous robots and drones, it is of paramount importance that such algorithms are efficiently executed onboard on properly designed hardware accelerators to meet the required performance specifications. In particular, some neural network architectures for object detection and tracking, as You Only Look Once (YOLO), include heavy computational stages that need to be executed before and after the model inference. Such stages are typically not incorporated in traditional accelerators and are executed on general-purpose processors, thus introducing a bottleneck in the overall processing pipeline. To overcome such a problem, this paper presents a general-purpose accelerator on a field-programmable gate array (FPGA) able to run pre-processing and post-processing operations typically required by vision tasks. The proposed solution has been tested in combination with a YOLO object detector accelerated on an Advanced Micro Devices (AMD) Xilinx Kria KR260 board mounting an UltraScale+ multiprocessor system-on-chip, achieving a significant improvement in terms of both timing performance and power consumption, and enabling onboard visual processing into drones. The proposed solution is able to boost the traditional object detection process by a factor of 4.4, allowing the execution of the full processing pipeline at 60 frames per second (fps), versus 13.6 fps reachable without the proposed accelerator. As a result, this work enables the use of high-speed cameras for developing more reactive systems that can respond to incoming events with lower latency.

1. Introduction

The tremendous evolution of artificial intelligence (AI), in particular machine learning and deep neural networks, is playing a crucial role in modern Cyber-Physical Systems (CPS), like drones (Ahmed et al., 2022; Xing et al., 2023), rovers (Jones et al., 2023; Cantero et al., 2022), and edge nodes for the Internet of Things (IoT) (Rosero-Montalvo et al., 2024; Wulfert et al., 2024). Such AI technologies, particularly those developed for vision tasks, such as object detection and tracking, are becoming fundamental for the development of intelligent systems, ranging from autonomous vehicles to smart factories. Since these systems are becoming ubiquitous, optimizing their implementation to satisfy the increasing performance requirements of CPS and IoT applications is crucial for their fruitful adoption.

The need for efficient AI processing has stimulated the development of dedicated hardware solutions that meet the stringent requirements of low latency, high throughput, time predictability, and energy efficiency. Among these solutions, Field-Programmable Gate Arrays (FPGAs) provide parallelism, adaptability, and low power consumption, making them an ideal candidate for accelerating AI workloads. Also, they underwent substantial improvements in software support for a seamless design and integration. For example, AMD Xilinx proposed Vitis-AI (AMD Xilinx, 2023d), a comprehensive development stack to design and deploy AI-based applications in their heterogeneous platforms, also exploiting the Deep Learning Processor Unit (DPU) (AMD Xilinx, 2023b), which is a programmable engine dedicated to Convolutional Neural Networks (CNN).

Despite the growing adoption of the DPUs in FPGA-based platforms, they are primarily used to accelerate the CNN model itself, leaving the pre-processing and post-processing phases to the CPUs, thus creating a critical bottleneck in the processing pipeline. This partitioning of the processing pipeline between the FPGA and the CPUs produces a suboptimal solution, which overloads the processors with additional computational activities that increase the interference on computational resources, such as the shared cache, the memory controller,

[*] Correspondence to: Via Cettigne 30, Cagliari, 09129, Italy
E-mail addresses: edoardo.cittadini@santannapisa.it (E. Cittadini), mauro.marinoni@santannapisa.it (M. Marinoni), giorgio.buttazzo@santannapisa.it (G. Buttazzo).

https://doi.org/10.1016/j.engappai.2025.110159
Received 16 April 2024; Received in revised form 10 October 2024; Accepted 23 January 2025
Available online 4 February 2025

and the bus. As a consequence, the execution times of the processing pipeline become longer and highly variable, reducing the performance of the overall cyber–physical system, which become less predictable and less controllable.

1.1. Paper contributions

To address the aforementioned issues, this paper presents the Image Processing Unit (IPU), which is a flexible hardware accelerator implemented on FPGA to speedup typical pre-processing and post-processing operations required in vision tasks. The neural network inference is supposed to be performed by a dedicated neural co-processor, as the Xilinx DPU. The proposed IPU can be integrated with a wide range of accelerated neural networks, like those provided by AMD Xilinx in its Model Zoo (AMD Xilinx, 2023c), but can also be combined with custom solutions. Using the IPU requires the deployment of a single IP on the FPGA that has to be configured at runtime.

The main motivation behind the development of the IPU was the need to process the images taken from a camera at the same rate at which they are captured. In most cases, this rate is around 30 frames per second (fps), corresponding to a period of 33.3 ms.

While the inference of most deep neural networks accelerated on a DPU can normally be performed within this period (5.5 ms for the YOLOX-Nano, 8.9 ms for the YOLOv3), there are other pre-processing operations (e.g., image resizing and normalization) and post-processing computations (e.g., bounding box coordinate decoding) that are typically not implemented in the accelerated models available in the AMD Xilinx Model Zoo (AMD Xilinx, 2023c).

If such computations are executed by software on the processing system of the platform, they introduce a bottleneck in the overall pipeline that leads to exceed the 33.3 ms bound by a significant extent. For instance, extensive experiments carried out in this work on an AMD Xilinx Kria KR260 UltraScale+ platform, shown that accelerating a YOLOX network by a DPU and executing all the pre-and post-processing operations on the Cortex-A53 quad-core processor requires an average execution time of 73 ms. Vice versa, when accelerating the same operations on the proposed IPU, the average time for executing the overall detection pipeline reduces to 13.6 ms, which allows increasing the frame rate of the camera up to 60 fps, thus enabling the development of more reactive systems that can respond to incoming events with a lower latency.

1.2. Paper structure

The rest of the paper is organized as follows: Section 2 discusses the related work; Section 3 introduces object detection strategies; Section 4 presents the architecture of the proposed IPU accelerator; Section 5 reports some experimental results aimed at showing the advantages of the proposed approach; and Section 6 states the conclusions and future work.

2. State of the art

Hardware acceleration on embedded systems went through remarkable advancements driven by the growing demand for machine learning and deep learning capabilities in resource-constrained environments. It plays a crucial role in enhancing the performance of embedded platforms and nowadays there are several mature solutions, as Graphics Processing Units (GPUs), Tensor Processing Units (TPUs), and Field Programmable Gate Arrays (FPGAs), which can be used for this purpose. Historically, GPUs were the first effective devices able to support the execution of machine learning by exploiting their parallel nature. As an example, AlexNet (Krizhevsky, 2014) was one of the first deep neural networks accelerated by GPUs during both training and inference.

However, GPUs present a number of issues when used in embedded systems. One of the major problems is their relatively high energy consumption, which is particularly relevant in battery-powered devices, like flying drones or small autonomous robots. This is caused by the high number of processing engines and because these devices were not originally developed for DNN inference tasks, so their structure is not optimized for that specific computation. Recent research has highlighted the growing importance of power-efficient systems for embedded devices using hardware accelerators, particularly in resource-constrained environments such as IoT and edge devices. Studies comparing FPGA-based systems with other accelerators, such as GPUs and TPUs, have shown that while GPUs offer significant computational power, their energy consumption remains a major concern in embedded and battery-powered systems. Conversely, TPUs and FPGAs have demonstrated more favorable energy efficiency, especially for tasks requiring high parallelism and real-time performance (Smith and Lee, 2024; Rodriguez and Martinez, 2024; Jones and Nguyen, 2024). Another trend to improve the energy consumption is to design machine learning models to optimize efficiency, as reported in the study of Zhou et al. (2020a). For instance, lightweight models, like the MobileNet family (Howard et al., 2017; Sandler et al., 2018; Howard et al., 2019; Zhou et al., 2020b), SqueezeNet (Iandola et al., 2016), EfficentNet (Tan and Le, 2020), and ShuffleNet (Zhang et al., 2018) rely on special layers aimed at reducing the number of operations and memory requirements to make them suitable for executing on resource constraint devices.

Another problem of GPUs is their high response time variability, caused by the concurrent accesses to shared devices existing on such architectures, as buses, memory controllers, and high-level caches, which create significant interference on the computations, introducing long and variable delays in application tasks (Buttazzo, 2022). For instance, Cavicchioli et al. (2017) observed significant and variable delays when using GPU acceleration on heterogeneous embedded platforms due to the contention occurring on shared memory, especially for memory-intensive GPU tasks.

Finally, GPUs are closed systems that schedule tasks in a non-preemptive fashion. This means that, if the system includes multiple neural networks with different complexity and periodicity requirements, those with shorter periods will be more likely to experience longer delays and higher response time variability. To solve this problem, Capodieci et al. (2018), in collaboration with NVIDIA, proposed to modify the GPU internal scheduler with a preemptive scheduler based on Earliest Deadline First (EDF) (Liu and Layland, 1973), also providing bandwidth isolation by means of a Constant Bandwidth Server (CBS) (Abeni and Buttazzo, 2004). Unfortunately, however, this solution is not yet available on commercial NVIDIA GPU platforms.

Another approach to accelerate neural models is to use tensor processing units (TPUs) (Google, 2023), which are application specific integrated circuits (ASICs) specifically designed to accelerate operations, like convolutions and vector multiplications, which are common to deep neural networks. A detailed analysis of TPU performance for different neural network models was presented by Seshadri et al. (2022) and Asyraaf Jainuddin et al. (2020).

A third way to accelerate machine learning models is by exploiting programmable logic circuits in FPGA devices. These devices offer an interesting trade-off between energy efficiency and flexibility with respect to GPUs (Qasaimeh et al., 2019), and lower construction and development costs with respect to ASICs. Additionally, the improvements made to software development tools, like high-level synthesis (HLS) compilers (Nane et al., 2016; Ye et al., 2022) and frameworks like hls4ml (Fahim et al., 2021), Catapult (Siemens, 2023), Vitis HLS (AMD Xilinx, 2023e) from AMD, and Quartus prime HLS (Altera, 2023) from Intel, allowed a wide range of developers to deploy their solutions into FPGA-based platforms. Another high-level software interface for FPGA programming is the OpenCL standard (Stone et al., 2010).

To implement a neural network on FPGA, two approaches can be adopted: (i) encoding the entire network (including the model parameters) into the programmable logic and executing it on the FPGA; or (ii) executing tensor operations on a general-purpose co-processor implemented as a softcore on the FPGA. Both approaches have pros and cons.

Using hardware description languages or high-level synthesis tools, it is possible to encode entire neural networks in programmable logic, achieving very performant task-oriented accelerators (Gunay et al., 2022; Pestana et al., 2021; Ma et al., 2018). This approach, however, requires a much larger amount of FPGA resources, which become the major bottleneck when implementing large deep neural models. This solution is also less flexible, since a slight change in the model parameters requires reprogramming the accelerator. Furthermore, this type of approach requires input and output data to have specific encoding, which often requires the execution of operations that can significantly impact run-time performance.

The solution relying on a softcore co-processor is less performant compared with network-specific accelerators, but far more flexible and more resource efficient, since it can perform the inference of multiple neural networks, whose parameters can be stored in RAM. To use this approach, neural networks need to be previously adapted using quantization methods (Gholami et al., 2022) and parameter pruning (Cheng et al., 2023) to make them compatible with the FPGA hardware.

There are several implementations of co-processors for accelerating generic CNNs, like the ones reported by Gholami et al. (2022). The most significant example of a general-purpose co-processor on FPGA is the AMD Xilinx Deep-learning Processor Unit (DPU) (AMD Xilinx, 2023b), but there are other custom implementations described in the literature (Liu et al., 2017; Chakradhar et al., 2010).

A significant limitation of the solutions based on general-purpose co-processors is that only layers common to standard CNNs are accelerated, leaving other operations to the CPU. For instance, all the pre-processing activities, as input rescaling, color depth quantization, and normalization are left to the application, as well as other post-processing tasks, such as the softmax layer and the bounding box construction needed in object detection networks. Some of these processing activities are computationally expensive and can become a bottleneck when implementing a full perception task in an embedded system. Additionally, several hardware accelerators are not open-source technologies, and consequently they cannot be modified to accommodate application-specific requirements.

To fill this gap, this work advances the state of the art by proposing a special-purpose accelerator able to perform image pre-processing and post-processing tasks that are typical in object detection, including the softmax activation function used for classification. The proposed unit can be paired with a CNN co-processor to accelerate a complete pipeline of a vision task or can be paired with a network-specific accelerator to offload some operations and save space on the FPGA. The proposed device was developed to provide an extension of the set of operations that can be accelerated, without focusing on a specific neural network or application, but providing an interface suitable for detection and classification networks.

3. Background

3.1. Convolutional neural networks in object detection

Convolutional neural networks (CNNs) drastically changed the field of computer vision, particularly in tasks such as object detection and image classification. CNNs are specialized neural networks designed to recognize visual patterns directly from an image. They consist of interconnected sequential layers, including convolutional layers for feature extraction, pooling layers for downsampling and dimensional reduction, and fully connected layers for classification and regression tasks. The success of CNNs in object detection lies in their ability to automatically learn relevant patterns and shapes from images and integrate them into higher level features. This feature extraction capability is crucial for object detection, where the network needs to identify objects by their distinguishing features regardless of their position, scale, or orientation. Object detection requires solving two main subtasks: object localization, which involves identifying the precise position of objects through bounding box coordinates, and classification, where the detected objects are categorized into predefined classes or categories. CNNs significantly contributed to advance object detection techniques through various architectures specifically tailored for this task. Traditional methods relied on handcrafted features and sliding window techniques, which were computationally expensive and lacked robustness. Modern architectures, such as Faster R-CNN (Ren et al., 2017), SSD (Ma et al., 2018), and YOLO (Redmon and Farhadi, 2018; Ge et al., 2021), have significantly improved detection accuracy, efficiency, and speed. Such CNN-based object detectors often leverage pre-trained networks, as VGG (Liu and Deng, 2015), ResNet (He et al., 2015), MobileNet (Howard et al., 2019), or Darknet (Redmon and Farhadi, 2018), as feature extractors, allowing the detection models to learn rich representations from large-scale image datasets. More recently, the development of anchor-based and anchor-free detection architectures has further improved the performance of object detection in image processing.

3.2. Anchor-based and anchor-free object detection

Object detection involves two main approaches: anchor-based and anchor-free methodologies. Anchor-based detectors, such as Faster R-CNN (Ren et al., 2017), rely on predefined box shapes (anchors) to predict object bounding boxes. Such anchors act as reference points across the image, aiding object localization based on predefined scales and aspect ratios. Vice versa, anchor-free detectors, like YOLOX (Ge et al., 2021) and FCOS (Tian et al., 2019), remove the use of anchors directly predicting bounding box coordinates and objectness scores from key points. This method is simpler, more flexible and often achieves competitive performance with reduced computational complexity. For these reasons, most modern real-time embedded applications adopt anchor-free detection models to contain computational complexity and meet real-time constraints, while also reducing the overall energy consumption.

3.3. YOLOX

This section describes the YOLOX neural network used in the object detection pipeline. YOLOX is a new efficient and accurate version of the YOLO neural network family. In particular, the YOLOX nano model provided by AMD Xilinx was used for all the experiments. Its architecture is illustrated in Fig. 1.

YOLOX is an evolution of the YOLOv3 architecture (Redmon and Farhadi, 2018), where the anchor-based detection mechanism is replaced with an anchor-free mechanism, which is more general and can easily be extended to key-points and 3D object detection, while the anchor-based approach is limited to bounding box prediction. The first part of the YOLOX architecture is a DarnkNet-53 convolutional network (backbone), while the second part is based on a Feature Pyramid Network (FPN) (Lin et al., 2017) (pyramid neck), which extracts features at different resolutions. Unlike the previous YOLO models, from version 1 to version 5, the prediction of classes (Class), objectness (OBJ), and bounding boxes (BBox) is performed by decoupled heads. This helps improve the detection accuracy by reducing conflicts between classification and regression tasks. The YOLOX model used in this paper was trained on the COCO dataset (Lin et al., 2014), containing over 330,000 annotated images divided into 80 object categories. The neural network requires input images of size $416 \times 416 \times 3$ (where 3 is the number of color channels) and produces three output volumes of shapes $52{\times}52$, $26{\times}26$, and 13×13, and depth 85. Overall, the network produces an output of 301,665 elements to be post-processed. Fig. 2 shows the organization of each output volume.

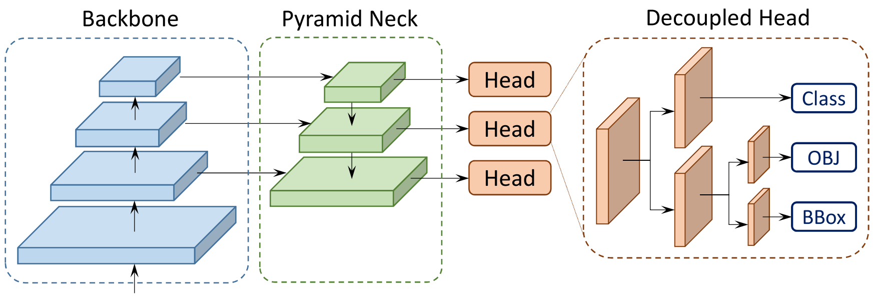


Fig. 1. Architecture of the YOLOX neural network.

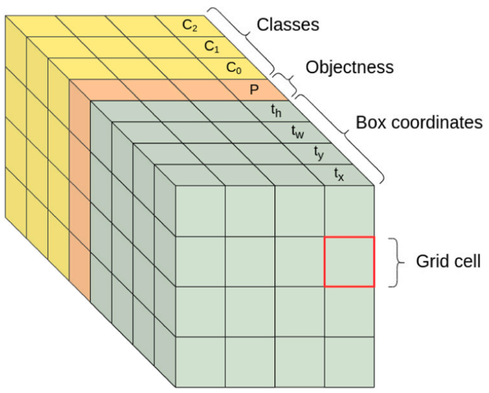


Fig. 2. Organization of an output volume in the YOLOX network.

Each element of an output volume, highlighted by a red square in Fig. 2, is called a grid cell and is responsible to predict one bounding box. The bounding box coordinates, represented in green, are not directly provided by the decoupled heads, but they are expressed using four values: t_x, t_y, t_w, t_h. The box center coordinates (C_x, C_y), its width B_w, and its height B_h can then be derived by post-processing through Eqs. (1)–(4), where S is the stride and D_x, D_y denotes the displacement of the grid cell in the image:

$$C_x = S * (D_x + t_x) \quad (1)$$

$$C_y = S * (D_y + t_y) \quad (2)$$

$$B_w = S * e^{t_w} \quad (3)$$

$$B_h = S * e^{t_h} \quad (4)$$

The objectness score, represented in orange, is a score in the range (0,1) that indicates the validity of the bounding box prediction for the given grid cell. It is obtained by applying the sigmoid function to the value produced by the corresponding output neuron. The validity of the box prediction is tested using a threshold value, which is a hyperparameter of the model. A common threshold value is 0.5. The class probability scores, represented in yellow, are obtained, like the objectness, by applying the sigmoid function to every element. Then, the class of the object inside the box is the one with the highest score.

To accelerate the YOLOX, the network inference is processed by the AMD Xilinx DPU, while all the post-processing computations required to decode the outputs produced by the network are accelerated via the proposed IPU, because the required operations are not included in the DPU instruction set. The proposed configuration allows for a complete acceleration of the object detection pipeline, improving both inference time and throughput.

4. System architecture

The processing pipeline for the inference of a generic single-stage neural network for object detection can be divided into three main phases, as shown in Fig. 3.

In the pre-processing phase, the input frame is elaborated to match the input specifications of the convolutional network used for feature extraction. Often, this phase consists of resizing the image and normalizing the pixel values. The second phase is the core of the algorithm performing the inference, where all the network layers are evaluated to produce the output. The computations involved in this phase are usually accelerated, due to the large amount of parallel multiply and accumulate operations that can efficiently be executed on a dedicated hardware.

In the last post-processing phase, the output of the neural network is further elaborated, typically using CPUs, to produce the final result. In particular, in single-stage detection networks, the input image is divided into smaller regions (grids), where each region can detect an objects whose coordinates are relative to the grid. As a result, the confidence scores produced by the network, for each grid and for each object category, are encoded into an output volume, which must be elaborated by several complex math operations to obtain meaningful data. Furthermore, each object may be detected in different grids and with different confidence score. In this case, a Non-Maximum Suppression algorithm (Hosang et al., 2017) is used to prune the redundant boxes and select the one with the highest confidence score.

Unfortunately, such pre-processing and post-processing phases are computationally intensive operations that, if executed on CPUs, may vanish the advantages of accelerating the model inference. This issue is even more crucial when the target computing platform is an embedded board or an IoT device with a power-oriented processing system.

Cittadini et al. (2023) reported a set of experiments on an AMD Xilinx ZCU104 UltraScale+ MPSoC development board that clearly highlight the negative impact of the pre-processing and post-processing phases in object detection tasks. The maximum execution time reported for the two non-accelerated phases resulted to be almost four times larger than the time required for accelerating the CNN inference.

The solution presented in this paper allows removing such a performance gap by accelerating the whole processing pipeline illustrated in Fig. 3 on the FPGA of a heterogeneous platform. To contain the number of read and write operations, the proposed solution uses a chain of pointers on shared memory locations that significantly reduces communication delays and latency. Fig. 4 shows the memory mapping of data in DDR memory and the interactions between different components.

4.1. Image processing unit

The solution proposed in this paper is implemented as an Image Processing Unit (IPU) acting as a specific hardware accelerator used to speedup both the pre-processing operations (such as image resizing

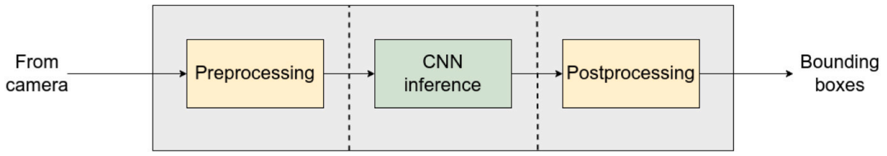


Fig. 3. Processing pipeline of a neural network for object detection.

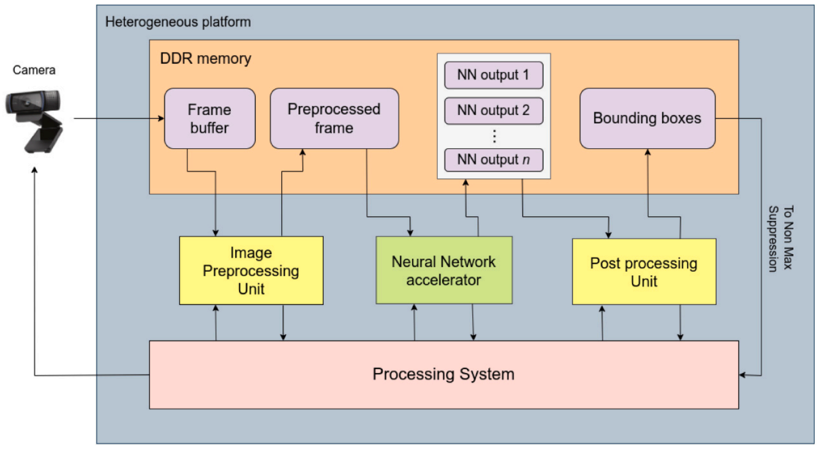


Fig. 4. Proposed acceleration flow.

and normalization) and post-processing algorithms (such as bounding box decoding and Softmax), which are typically not carried out by the accelerated networks available in the AMD Xilinx Model Zoo (AMD Xilinx, 2023c). The purpose of such a design choice is to reduce the workload on the processing system by offloading the pre- and post-processing operations to the IPU, thus achieving a more balanced utilization of resources, reaching higher frame rates as well as more predictable response times suitable for real-time vision applications.

The proposed IPU consists of three units:

- A pre-processing unit to accelerate the pre-processing activities typically required in image processing tasks (implemented in HLS);
- A post-processing unit to accelerate the activities typically required in object detection tasks (implemented in HLS);
- A control unit to coordinate and configure the previous units by interacting with the Programmable System through an AXI slave port (implemented in HDL).

A block diagram of the IPU is illustrated Fig. 5. Note that, to improve efficiency, all IPU units operate on shared memory buffers to avoid expensive memory copies.

Each accelerated function is triggered by the software application by writing a specific register in the control unit. The completion of the accelerated task can be notified either by polling or by interrupt.

4.1.1. Pre-processing unit

The pre-processing unit implements a large subset of the OpenCV[1] library. In particular, the core is capable of taking an RGB or grayscale image and resizing it to an arbitrary dimension. Although the OpenCV resize function allows choosing among different scaling algorithms, the bilinear interpolator was implemented, being the best compromise between speed and quality. The image shape presented to the neural model must be the same as the one used during training. To provide higher flexibility, the interpolator was designed to provide two scaling methods: one that preserves the aspect ratio and one that stretches the original image. The user can also specify a padding value to match the input specifications through a dedicated register of the control unit. When specified, pad filling and image resizing are simultaneously executed in hardware.

To achieve a flexible pixel quantization, a dedicated input scale factor can be specified to reduce the range of the pixel values. Such a scale factor value is model-specific and must be provided by the user in a dedicated register. Its default value is 1, meaning no input scaling.

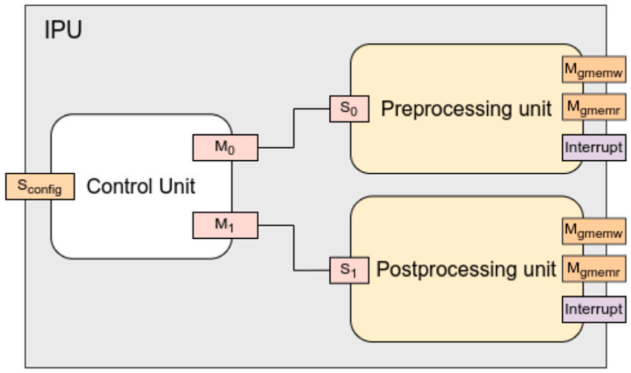


Fig. 5. Architecture of the image processing unit.

4.1.2. Post-processing unit

The post-processing unit accelerates the final processing phase of single-stage detectors. In particular, the unit accelerates the computation of the four bounding box coordinates, the objectness, and the scores for each class. Such operations require the implementation of the sigmoid and exponential functions inside the post-processing unit. This unit can be configured by the application using a set of registers in the

[1] OpenCV library: https://opencv.org/.

control unit. Such a device converts integers to floating point values using a dedicated register in the control unit that stores the output scale.

This unit also implements the softmax layer to normalize the classification scores. It is worth noting that, for applications using multiple neural networks, accelerating the softmax function on the post-processing unity allows for a higher degree of parallelism that could not otherwise be exploited using the softmax implemented in the DPU. In fact, the DPU cannot accelerate convolutional layers in parallel with the softmax execution. Thus, moving the latter on a different hardware allows parallelizing the two operations and increasing the throughput of the system.

For object detection tasks, the computation is more complex and requires specifying the following parameters in the corresponding registers:

1. Image size, specifying the side of a square image;
2. Stride for the output volume, used, paired with the image size, to obtain the dimension of a grid cell;
3. Pointer to the input memory buffer;
4. Pointer to the memory location where the result will be written;
5. Number of classes to be detected by the model.

It is worth noting that for classification tasks the output of the CNN is a single array, whereas for object detection tasks, the accelerator must be invoked multiple times and each output volume must have a reserved memory region to write the computation result matching the dimension of the volume specified in the registers. Allocating such a memory, however, is not in charge of the IPU, but is handled by the device driver, which is described in Section 4.2.

The very last processing stage for a detection task, the Non Maximum Suppression, is still executed on the processors because it involves array sorting, which would achieve a negligible benefit from a hardware acceleration.

4.1.3. Control unit

The control unit provides a set of general registers, which can be set by the application to configure the other two units, and two control registers for setting the operational mode (polling or interrupt) for each device. Overall, it provides the following registers:

1. Mode register 1: it specifies the operational mode (polling or interrupt) for the pre-processing unit;
2. Mode register 2: it specifies the operational mode (polling or interrupt) for the post-processing unit;
3. Input width register: it specifies the width of the input image;
4. Input height register: it specifies the height of the input image;
5. Output width register: it specifies the width of the output image;
6. Output height register: it specifies the height of the output image;
7. Padding register: it specifies the padding value;
8. Input scale register: it specifies the input scale factor;
9. Input address register: it is a pair of 32-bit registers used to obtain a 64-bit value representing the pointer to the input buffer in memory;
10. Output address register: it is a pair of 32-bit registers used to obtain a 64-bit value representing the pointer to the location where the result will be written in memory.

4.2. Software support

This section provides an overview of the possible software solutions used to support the IPU in different embedded platforms. From a software perspective, embedded systems can be divided into three groups: (i) platforms directly running the bare metal application, (ii) platforms that use a real-time operating system (RTOS), and (iii) platforms that use a complex operating system, like Linux. Among high-end FPGA-based heterogeneous platforms, accelerators are integrated as devices in Linux, and their functions are accessed as services via drivers and system calls.

A common problem of accelerators used in systems employing artificial intelligence is that they typically work with large amounts of data that must be contiguous in RAM to allow the use of read and write bursts, which improve the overall system performance. The pre-processing and post-processing units of the proposed device access the data through two 64-bit registers (represented using a pair of 32-bit registers). The content of these registers represents an address that, paired with an offset value, identifies a memory location to fetch or store the data. Note that, if only a single 32-bit register were used, then the mapping of accelerators and peripherals would be limited to the lowest 4 GB of RAM.

One problem when interfacing with memory-mapped devices is the use of the cache. In fact, cache coherency must be avoided, since it can lead to inconsistent data reads or writes producing undefined results. Memory-mapped peripherals are not aware of memory hierarchy, hence, if some data are cached in the address range of the device and the device reads them before a cache write back, it will read inconsistent data. However, invalidating the data cache for every acceleration request can have a significant impact on the overall system performance.

To support the use of the IPU across various embedded settings (i.e., bare-metal applications, RTOS-supported systems, or higher-level Linux-based applications) the IPU driver has been designed with multiple levels of hardware abstraction. Specifically, for RTOS and bare-metal environments, the driver can directly access memory using physical addresses, effectively avoiding caching issues. This is achieved by leveraging dedicated CPU instructions to bypass the cache for specific ranges of memory addresses, particularly for the memory segments where the accelerators are mapped. When integrating the IPU in Linux, the driver structure becomes more complex. Fig. 6 illustrates a block diagram describing the interactions between the operating system components, the user application, and the physical hardware device.

Unlike the previous cases, Linux does not allow complete access to all the platform resources from user space, hence reserving the memory that the accelerators need for the I/O buffers requires the implementation of a kernel module. Because of virtual memory and paging, the RAM, from the application perspective, is no longer a flat and homogeneous address space and it is handled by the operating system through the memory management unit (MMU). Also, Linux uses a privileged system that isolates the kernel space from the user space, which requires additional operations to provide an interface to access the kernel memory space and a direct interface for the user space applications using the acceleration device. The Contiguous Memory Allocator (CMA), reported in Fig. 6, is a Linux kernel component that handles the allocation and mapping of contiguous memory buffers required by device drivers and memory mapped peripherals. The memory area on which it operates must be known at boot time, so it requires rebuilding the kernel, specifying the amount of memory that must be reserved. When the user application calls the configure service, the device driver first sets the registers to specify the size of the input and output image, then it allocates the contiguous memory buffer using the CMA, and finally maps both buffers and registers in the user space. Mapping buffers directly to the user space is crucial to significantly improve performance. In fact, the solution of passing data from user to kernel space and vice versa requires multiple buffers copies, which is highly inefficient because it requires twice the memory and scales poorly, as the data transfer times increase due to the copy operation. It also increases the memory traffic, which causes a higher interference on the system.

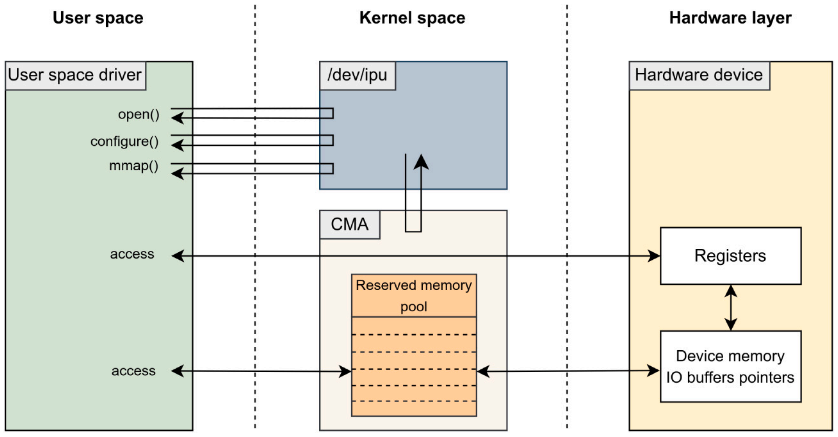


Fig. 6. Architecture of the Linux kernel module.

5. Experiments

This section presents a set of experiments aimed at showing the advantages of the proposed Image Processing Unit (IPU) in reducing the overall computation times of the complete detection pipeline, using an AMD Xilinx Kria KR260 UltraScale+ MPSoC as a reference platform. First, we report a set of measurements carried out to show the performance improvement obtained by the accelerated pre-processing and post-processing units with respect to the corresponding software routines. Then, we present the results obtained on a fully accelerated pipeline performing an object-detection task with the YOLOX (Ge et al., 2021) single-stage anchor-free detector to highlight the advantages of the proposed approach in a real application scenario. Note that no evaluation is reported on the accuracy of the neural model, since the proposed solution uses a pretrained and accelerated network available in the AMD Xilinx Model Zoo (AMD Xilinx, 2023c) without any modification.

It is worth noting that the IPU is not influenced by the complexity of the neural network models themselves. Instead, it focuses on accelerating the pre- and post-processing tasks; hence, the only factors that affect its computation times are (i) the dimensions of the raw image generated by the camera that becomes the input of the pre-processing unit, (ii) the size of the rescaled image produced by the pre-processing unit, (iii) the number of classes involved in the post-processing computation, and (iv) the number of output levels to be processed after the CNN accelerator to obtain the coordinates of the bounding boxes. In all the experiments, the input frames were acquired using a Logitech C920 USB camera that can be configured to grab images with a variable resolution from 320 × 240 to 1280 × 720.

5.1. Hardware setup

The experimental setup used to carry out the performance tests on an AMD Xilinx UltraScale+ MPSoC is shown in Fig. 7. The board is an AMD Xilinx Kria KR260 and the CNN inference was accelerated by an AMD Xilinx DPU (AMD Xilinx, 2023b) (DPUCZDX8G version 4.1.), through the Vitis-AI (AMD Xilinx, 2023d) software stack version 2.5. The DPU was instantiated as a single-core version. The image pre-processing and post-processing tasks were accelerated by the IPU, instantiated with the most flexible configuration. The optional softmax core was excluded from the DPU, since it has been implemented in the IPU post-processing unit to allow for a higher degree of parallelism in the presence of multiple neural networks. In fact, when a DPU core is processing the softmax, it cannot accelerate other CNN operations. On the contrary, moving the softmax computation from the DPU to the IPU allows parallelizing the softmax of a CNN and the inference of another model on the two acceleration devices.

Both the DPU and the IPU have an AXI slave connection with the Programmable System (PS), used for device configuration. Such connections use a low-performance port, referred to as S_{config} in Fig. 7, because configuration operations do not require high-bandwidth channels. The same also applies for the master instructions port (M_{instr}) of the DPU. Vice versa, the two master data ports of the DPU (M_{data}) and the four master ports of the IPU (M_{rpre}, M_{rpost}, M_{wpre}, and M_{wpost}) are high-performance ports to ensure the maximum possible throughput.

This setup has been used to test each hardware acceleration unit (pre-processing, inference, and post-processing) and the fully-accelerated processing pipeline for object detection tasks. Table 1 reports, for each accelerated unit, the utilization of the different FPGA resources, expressed both as the total number of resources and relative percentage. It is worth observing that, since the resource utilization changes depending on the number of iterations, the results reported in Table 1 are obtained by setting the maximum number of iterations required to process the data. This is done by setting the `loop tripcount` parameter in the HLS tool to the maximum value required by the application, thus obtaining the maximum resource utilization of the devices.

Managing the DPU requires the Linux operating system because the driver is only distributed inside the Petalinux software tool and is not released as open-source code. Hence, the Vitis-AI (AMD Xilinx, 2023d) framework is compiled for Linux. The setup adopted for the IPU allows processing input images up to 1280 × 720 pixels, producing output images up to 640 × 480 pixels.

The DPU DSP clock has been set to 630 MHz, close to the maximum frequency specified in the product guide (AMD Xilinx, 2023a), whereas the implemented IPU instance can support a clock frequency up to 245 MHz, derived by the Vitis HLS analyzer.

For the reported experiments, the interrupt-based approach was used by the IPU units to notify the processor of a job termination. In particular, the ARM v8 is notified by an interrupt generated by each accelerator (the IPU units or the DPU) every time a hardware-accelerated portion of the inference pipeline is completed.

Note that, since time measurements in Linux can be altered by task preemption, context switches, and kernel activities, all computation times of the accelerated tasks were measured through a performance counter IP , also implemented on the FPGA. Such a device is connected to 1 MHz clock to measure time with 1 ms resolution, starting with the corresponding start bit in the control register of the control unit, and stopping with the interrupt arrival at job termination. The aim of this device, which is only present in the experimental setup, is to provide

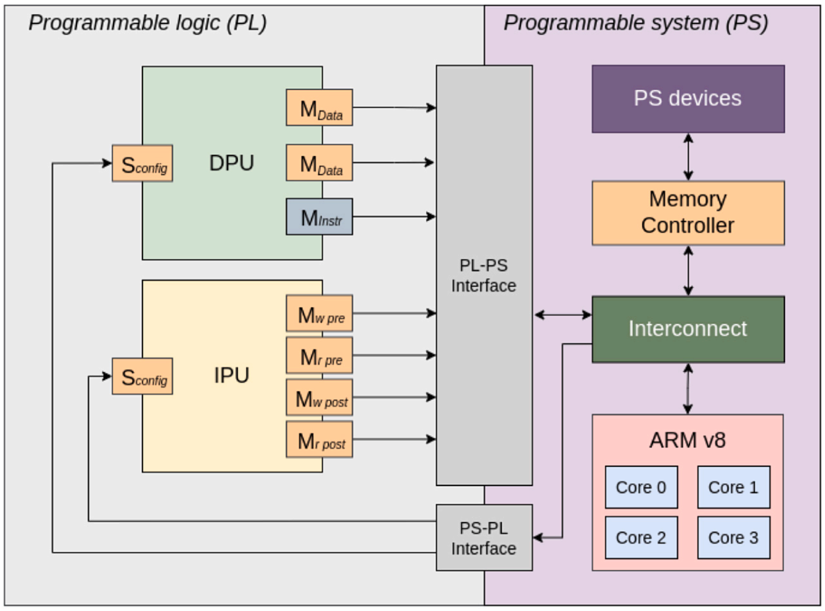


Fig. 7. Overview of the experimental platform.

Table 1
Total number of FPGA resources (and relative percentage) used to instantiate each device on a Kria KR260 UltraScale+ board to implement the full hardware accelerated object detection pipeline.

Device	LUT	FF	BRAM	DSP
Pre-processing unit	15 428 (13.17%)	13 165 (5.62%)	26.5 (18.4%)	55 (4.41%)
DPU	51 639 (44.09%)	99 714 (42.57%)	105 (72.9%)	710 (56.89%)
Post-processing unit	11 951 (10.20%)	13 147 (5.61%)	1 (0.7%)	56 (4.49%)

accurate measures for acceleration times regardless of the platform in which the IPU is operating. Computation times of the corresponding software application were measured using the high precision timer available in the ARM v8 through the Linux API. In order to reduce interference on the monitored task the program was executed under the SCHED_FIFO scheduling policy with a priority level set to 99, which corresponds to the highest value in Linux.

5.2. Pre-processing unit

This subsection reports the results of an experiment aimed at showing the execution times of the IPU pre-processing unit with respect to its software counterpart. The image pre-processing task performs the sequence of operations described in Section 4.1 on the images captured by the camera to match the DPU input requirements.

Table 2 reports the minimum, average, and maximum execution times of the pre-processing task executed both in software and accelerated by the IPU. These measurements have been obtained over 10,000 frames acquired from the camera, for each considered input–output resolution. In particular, with respect to the software execution, the IPU achieved a speedup ranging from 4× to 32× on the average execution time, depending on the input–output resolution. Note that, while the execution times of the hardware task are strictly related to the input and output resolutions, the times resulting from the software task are not monotonically related to the image sizes, probably due to the optimizations performed by the compiler as a function of the array sizes.

Fig. 8 illustrates the two execution times distributions obtained with an input image resolution of 640 × 480 pixels and an output size of 416 × 416. The distributions achieved with the other resolutions are reported in Appendix. In order to better appreciate the distribution of the tasks in Fig. 8, a few outliers have been left out from the plots.

As clear from the graphs, the proposed solution not only reduces the execution time of the task, but also reduces its variability over multiple executions, making the application more predictable. Table 3 reports the standard deviations on the execution times of the pre-processing task executed in software and accelerated by the IPU for different input/output image resolutions. Notice that the standard deviation reduces from a maximum of 120 ms (in software) to a maximum of 11 ms (with the IPU).

5.3. Post-processing unit

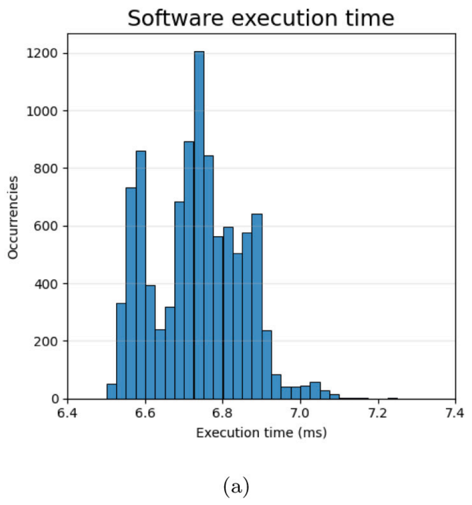
As done for the pre-processing task, this subsection compares the performance of the post-processing task when accelerated by the IPU, with respect to the corresponding software implementation. Since the shapes of the YOLOX output volumes are defined in the model and do not depend on the input image size, the timing performance of this task has been tested by varying the number of output classes. In this experiment, the number of classes has been varied from 10 to 80, which is the full number of classes in the COCO dataset (Lin et al., 2014), with a step of 10.

The obtained results are reported in Table 4, from which it can be seen that the IPU allows achieving a speedup factor on the average execution times that ranges from 4.5× to 7.2×.

Fig. 9 illustrates the two execution times distributions obtained with 80 classes. The distributions achieved with the other number of classes are reported in Appendix. To better appreciate the distribution of the tasks in Fig. 9, a few outliers have been left out from the plots.

Table 2
Execution times obtained on the pre-processing task executed in software and accelerated by the IPU for different input/output image resolutions.

Input resolution	Output resolution	Software task			Hardware accelerated task		
		Min (ms)	Avg (ms)	Max (ms)	Min (μs)	Avg (μs)	Max (μs)
1280 × 720	416 × 416	9.637	9.733	11.167	1049.701	1049.975	1199.630
1280 × 720	250 × 250	3.596	3.724	7.846	1023.290	1023.539	1166.149
640 × 480	416 × 416	6.502	6.733	10.068	377.497	378.902	528.216
640 × 480	250 × 250	4.211	4.256	5.100	363.017	364.370	508.686
320 × 240	416 × 416	5.166	5.266	6.942	180.398	180.736	521.945
320 × 240	250 × 250	3.309	3.362	4.144	105.699	105.864	179.278



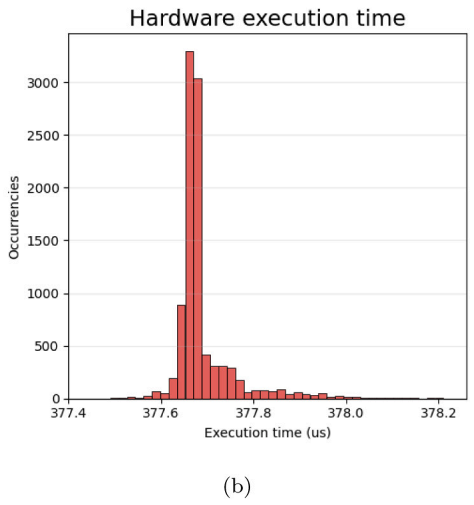
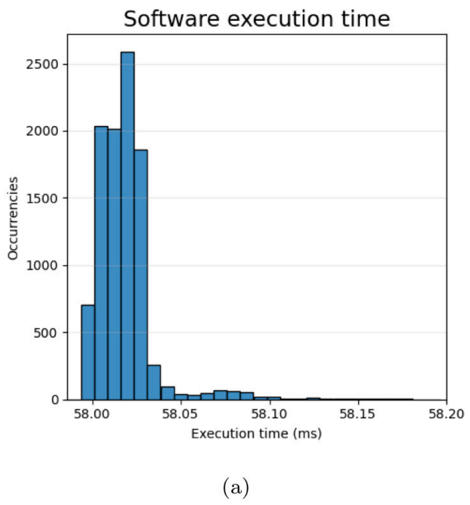


Fig. 8. Image pre-processing execution times distributions for the software (a) and hardware solution (b), with input images of 640 × 480 pixels and output images of 416 × 416.



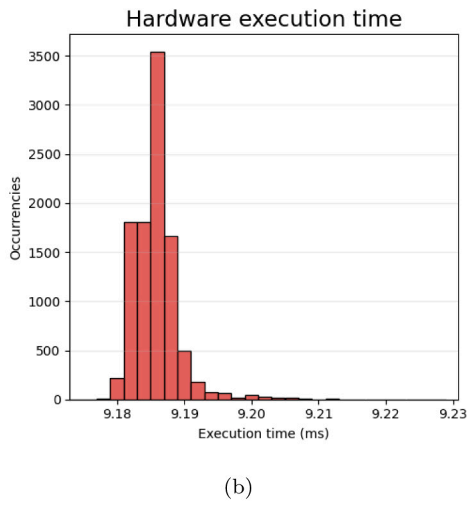


Fig. 9. Post-processing execution times distributions for the software (a) and hardware solution (b), with the number of classes defined in COCO (i.e., 80).

Table 3
Standard deviations on the execution times of the pre-processing task executed in software and accelerated by the IPU for different input/output image resolutions.

Input resolution	Output resolution	Software task Standard deviation (μs)	Hardware task Standard deviation (μs)
1280 × 720	416 × 416	69.004	2.104
1280 × 720	250 × 250	103.137	1.553
640 × 480	416 × 416	121.006	11.032
640 × 480	250 × 250	30.383	10.355
320 × 240	416 × 416	77.578	4.695
320 × 240	250 × 250	42.279	1.336

As clear from the graphs, the IPU not only reduces the execution time of the task, but also reduces its variability over multiple executions, making the application more predictable.

Table 5 reports the standard deviations on the execution times of the post-processing task executed in software and accelerated by the IPU for different number of classes. Notice that the standard deviation reduces from a maximum of 81.8 ms (in software) to a maximum of 15.6 ms (with the IPU), depending on the number of classes.

Table 4
Comparison of the execution times for the post-processing phase between the classical software solution and the proposed hardware acceleration for different values of detected classes.

Number	Software task			Hardware accelerated task		
of classes	Min (ms)	Avg (ms)	Max (ms)	Min (ms)	Avg (ms)	Max (ms)
10	7.428	7.452	8.902	1.638	1.641	1.748
20	14.628	14.642	16.668	2.724	2.728	2.856
30	21.871	21.894	23.627	3.805	3.809	6.854
40	29.069	29.089	32.957	4.872	4.878	6.359
50	36.308	36.331	38.058	5.948	5.954	6.211
60	43.532	43.558	45.268	6.033	6.038	6.386
70	50.762	50.796	54.806	8.114	8.121	9.012
80	57.992	58.022	62.045	9.179	9.185	9.555

Table 5
Standard deviations on the execution times of the post-processing task executed in software and accelerated by the IPU for different number of classes.

Number of classes	Software task Standard deviation (μs)	Hardware task Standard deviation (μs)
10	25.819	2.387
20	41.894	2.522
30	35.021	3.747
40	49.848	15.636
50	51.177	5.813
60	51.788	7.711
70	70.872	12.768
80	81.801	7.679

Table 6
Execution times of the complete YOLOX detection pipeline for the two considered implementations: SW-based, where only the YOLOX is accelerated by the DPU, and IPU-based, where also the pre-processing and post-processing tasks are accelerated by the IPU.

	Min (ms)	Avg (ms)	Max (ms)
SW-based	73.285	73.371	82.264
IPU-based	16.575	16.683	19.755

5.4. End-to-end pipeline

To evaluate the performance gain introduced by the IPU on the overall vision application, this experiment compares the fully accelerated pipeline (where the YOLOX model is accelerated by the DPU and the pre-processing and post-processing tasks are accelerated by the IPU) against the typical case in which only the network model is accelerated by the DPU and the other tasks are executed in software. The last system implementation comes from the fact that the YOLOX model is optimized by the Vitis-AI compiler for the AMD Xilinx DPU, producing a binary output which does not allow us to replicate the same computation flow via software.

In this setting, the system takes images from a camera, and since the YOLOX model takes input images with a resolution of 416 × 416 pixels, the camera resolution was set to 640 × 480 pixels, which is the closest one among those available from the camera. The output volumes are sized to detect all the 80 classes available in the COCO dataset.

The obtained results are summarized in Table 6, which reports the minimum, maximum, and average execution times of both solutions, achieved over 10,000 repetitions. It is worth noting that the proposed IPU allows processing the input stream with an average rate of 59.94 fps, against an average rate of 13.63 fps without the IPU.

The execution time distributions of the two implementations are illustrated in Fig. 10. As clear from the graphs, the IPU is beneficial not only for reducing the execution time of the vision task, but also its variability. In particular, the standard deviations resulted to be of 172.537 ms for the software-based solution and only 41.847 ms for the IPU-based solution.

5.5. Power consumption

A last set of experiments was carried out to evaluate the power consumption of the pre-processing and post-processing tasks executed by the IPU with respect to the corresponding software implementations. In particular, Table 7 reports the average power consumption of the platform in a fixed interval of time equal to 800 s, in which the pre-processing and post-processing tasks were executed via software or accelerated by the IPU.

In a first experiment, in order to have the same number of measurements in the same time interval, both tasks were activated with the same period ($T_{SW} = T_{IPU}$) equal to 80 ms, equivalent to the minimum period that can be reached by the software implementation. In this setting, both tasks perform the same number of cycles ($N_{SW} = N_{IPU} =$ 10,000) in 800 s. The obtained results show that the process accelerated by the IPU is slightly more power-demanding with respect to the one executed in software. In fact, the average power consumption measured for the software task (P_{SW}) resulted in 12.116 W, while the one related to the IPU (P_{IPU}) resulted in 12.292 W, with a little increment in power consumption ($\Delta P = 0.176$ W), corresponding to 1.452 percent increment with respect to the software case.

In a second experiment, each task was reactivated to process a new image as soon as the previous computation was completed and a newly acquired frame was available from the camera. In this case, the software task was able to run with a period $T_{SW} = 73.9$ ms, slightly shorter than before, executing for N_{SW} = 10,814 cycles in the overall test interval, whereas the hardware task was able to run with a period $T_{IPU} = 16.75$ ms for N_{IPU} = 47,761 cycles in the same interval, corresponding to a speedup of 4.42×. In this setting, the software task consumed an average power $P_{SW} = 12.326$ W, while the one using the IPU consumed $P_{IPU} = 12.523$ W, with a little increment in power consumption equal to $\Delta P = 0.197$ W, corresponding to 1.598 percent increment with respect to the software case.

6. Conclusions

This paper presented an Image Processing Unit (IPU) for the hardware acceleration of typical pre-processing and post-processing functions that must be performed when using deep neural networks in vision tasks, as image classification, scene segmentation, object detection, and tracking. The proposed accelerator has been implemented on FPGA as a general programmable special-purpose accelerator that can work in conjunction with other accelerators normally used to speedup the inference of convolution neural networks.

The problem addressed in this work specifically focuses on the bottlenecks created by pre-processing and post-processing functions, which are often neglected in common FPGA accelerated neural networks. When these functions are implemented in software and are executed by CPUs, the execution time of the overall vision pipeline increases significantly, vanishing the advantage of having an accelerated inference and limiting the achievable frame rates substantially. The proposed IPU is able to remove such bottlenecks by fully offloading these tasks to dedicated special-purpose accelerators. To show the advantages of the proposed solution, a set of experiments have been carried out testing the IPU in combination with the AMD Xilinx Deep Learning Processor Unit (DPU) (AMD Xilinx, 2023b) for accelerating the full processing pipeline of a YOLOX object detector. The results of the experiments clearly showed that the proposed accelerator is crucial to achieve a real-time performance in embedded systems and IoT devices that integrate deep learning components and have to perform perception tasks within stringent timing constraints.

In particular, extensive tests performed on an AMD Xilinx UltraScale+ MPSoC executing the latest version of the YOLOX single-stage anchor-free detector showed that the use of the IPU allowed increasing the frame rate of the object detection pipeline from about

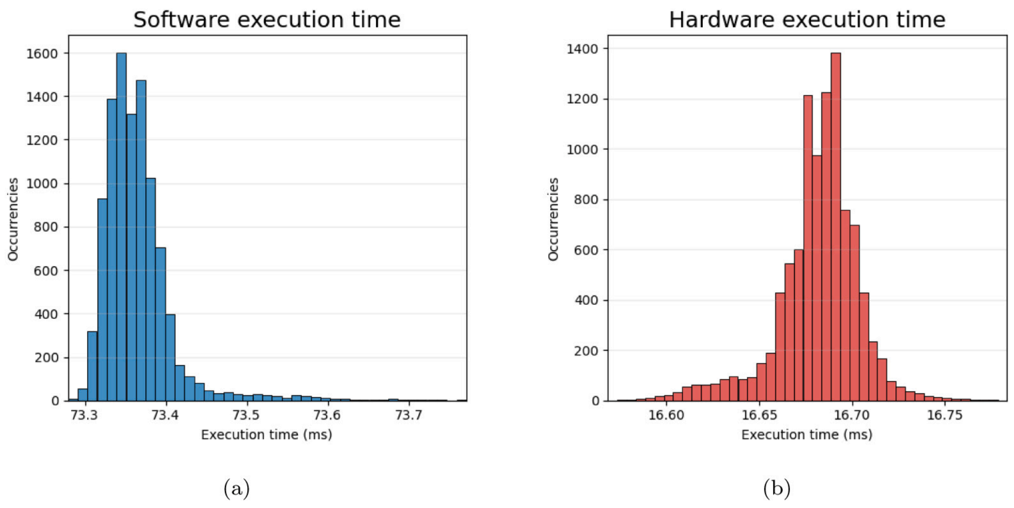


Fig. 10. YOLOX end-to-end execution times distributions for the software (a) and hardware solution (b).

Table 7
Average power consumption of the platform in the end-to-end pipeline executed via software and accelerated by the IPU. In both cases, the YOLOX is accelerated by the DPU.

T_{SW} (ms)	T_{IPU} (ms)	N_{SW}	N_{IPU}	$Speedup$	P_{SW} (W)	P_{IPU} (W)	ΔP (W)	$100 * \Delta P/P_{SW}$
80	80	10,000	10,000	1	12.116	12.292	0.176	1.452%
73.9	16.75	10,814	47,761	4.42	12.326	12.523	0.197	1.598%

13 fps (achieved with the standard solution provided by AMD Xilinx in which only the YOLOX is accelerated by the DPU) up to 60 fps (achieved by exploiting the IPU to also accelerate the pre-processing and post-processing tasks). More specifically, compared to the software-based implementation, the IPU pre-processing unit achieved a speedup factor ranging from 9.3× (with the highest camera resolution and the largest pre-processed image size) to 32× (with the lowest camera resolution and the smallest pre-processed image size). Concerning the post-processing unit, it reached a speedup between 4.5× (achieved with 80 output classes) and 7.2× (achieved with 10 output classes). These improvements also led to a reduced variability of the execution times, which is essential for achieving more predictable response times in real-time applications. For instance, the proposed solution has been successfully tested on a quadcopter (Cittadini et al., 2023) to perform a real-time tracking of multiple moving targets onboard.

Another advantage of the IPU is to offload the CPUs from such heavy auxiliary computations, leaving them available for other application activities that can be performed in parallel with the vision tasks.

Concerning power consumption, the experiments reported in Section 5.5 showed that the IPU is able to provide the aforementioned performance boost consuming only 1.5% more power with respect to the corresponding software implementation, which is crucial for executing deep learning models onboard in small cyber–physical systems or IoT devices.

As a future work, we plan to extend the IPU to support the post-processing functions of other deep learning models, such as those used for image segmentation, which requires the assignment of the class labels at a pixel level, and natural language processing, requiring the efficient computation of similarity scores, intensively used also in tasks like object tracking. Finally, we would also like to optimize the IPU architecture to further reduce power consumption without compromising performance, further extending the operational lifetime of battery-powered cyber–physical systems.

CRediT authorship contribution statement

Edoardo Cittadini: Writing – original draft, Validation, Software, Investigation, Conceptualization. **Mauro Marinoni:** Writing – review & editing, Supervision. **Giorgio Buttazzo:** Writing – review & editing, Supervision.

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 was partially supported by the Department of Excellence of Robotics and AI of the Sant’Anna School of Advanced Studies and by the Italian Ministry of University and Research (MUR) under the VIRMA project funded by the EU PNRR plan with DM-737/2021.

Appendix. Extra results

This section presents some extra results to better highlight the performance of the proposed hardware acceleration under different conditions. In particular, the graphs reported from Figs. A.11 to A.16 extend the results presented in Fig. 8 by showing the execution times distributions of the software and hardware tasks for the different combinations of input and output image resolutions listed in Table 2.

Similarly, the graphs reported from Figs. A.17 to A.24 extend the results presented in Fig. 9 by showing the post-processing execution times distributions for the software and hardware tasks over 10,000 runs, for the different number of output classes reported in Table 4.

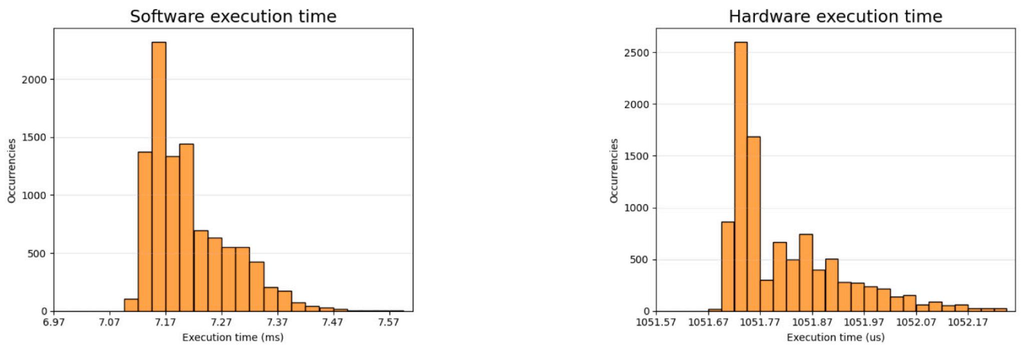


Fig. A.11. Pre-processing execution times distributions, measured over 10,000 runs, of the software task and hardware task, for a 1280 × 720 input image and a 640 × 480 output image.

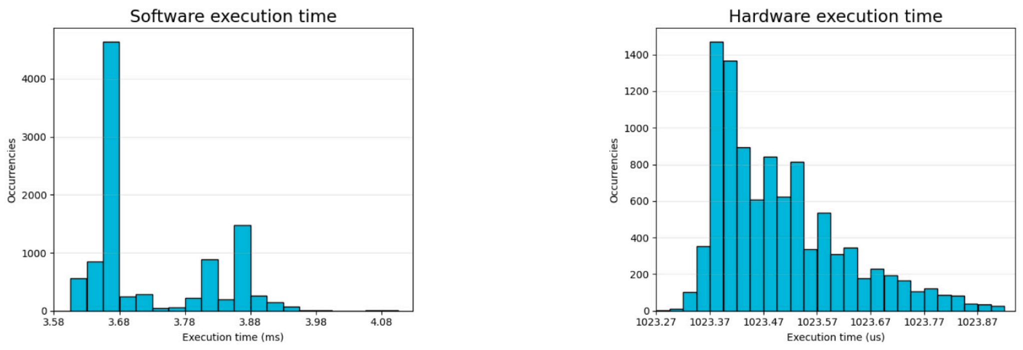


Fig. A.12. Pre-processing execution times distributions, measured over 10,000 runs, of the software task and hardware task, for a 1280 × 720 input image and a 250 × 250 output image.

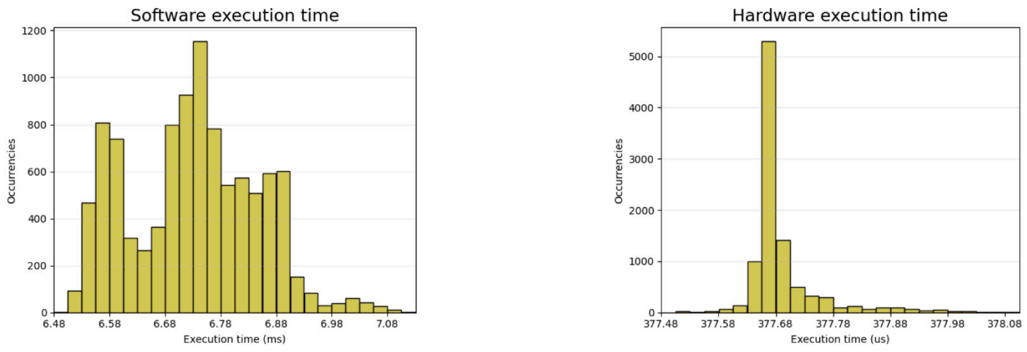


Fig. A.13. Pre-processing execution times distributions, measured over 10,000 runs, of the software task and hardware task, for a 640 × 480 input image and a 416 × 416 output image.

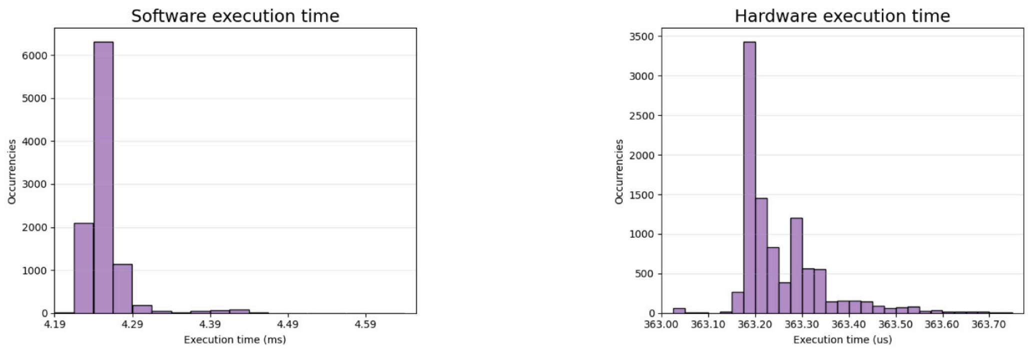


Fig. A.14. Pre-processing execution times distributions, measured over 10,000 runs, of the software task and hardware task, for a 640 × 480 input image and a 250 × 250 output image.

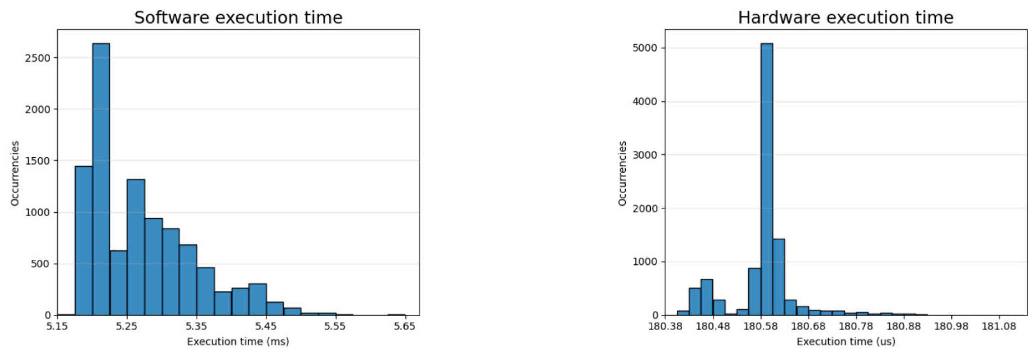


Fig. A.15. Pre-processing execution times distributions, measured over 10,000 runs, of the software task and hardware task, for a 320 × 240 input image and a 416 × 416 output image.

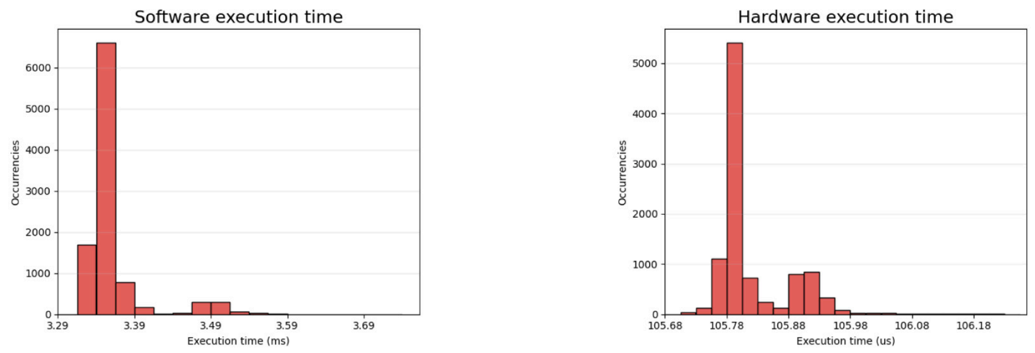


Fig. A.16. Pre-processing execution times distributions, measured over 10,000 runs, of the software task and hardware task, for a 320 × 240 input image and a 250 × 250 output image.

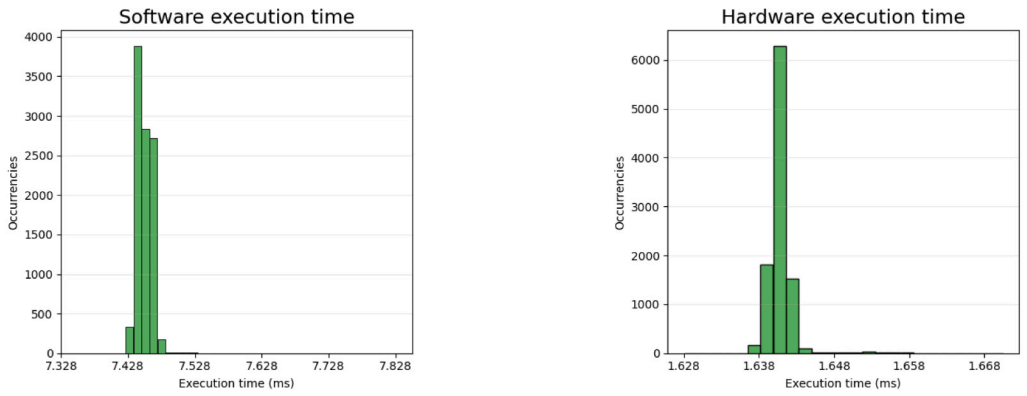


Fig. A.17. Post-processing execution times distributions, measured over 10,000 runs, of the software task (left column) and hardware task (right column), for 10 output classes.

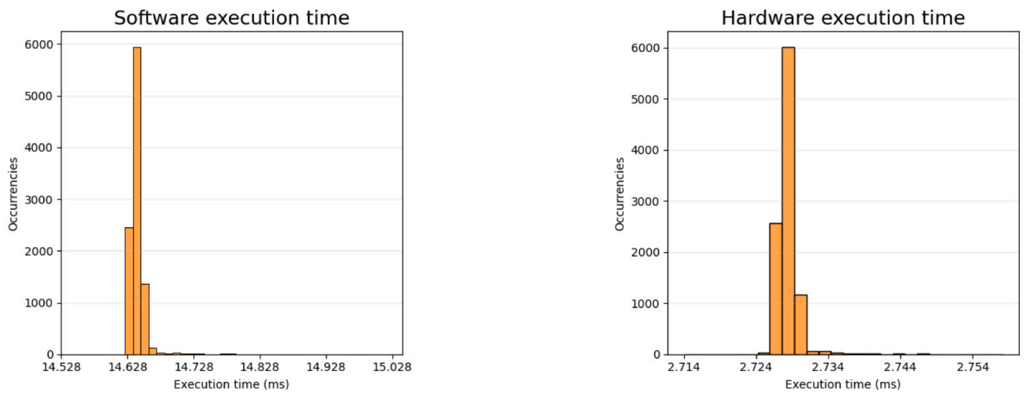


Fig. A.18. Post-processing execution times distributions, measured over 10,000 runs, of the software task (left column) and hardware task (right column), for 20 output classes.

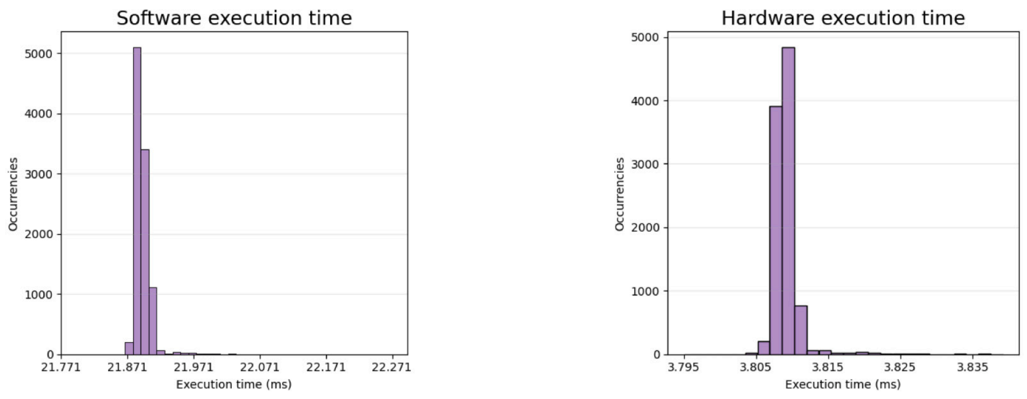


Fig. A.19. Post-processing execution times distributions, measured over 10,000 runs, of the software task (left column) and hardware task (right column), for 30 output classes.

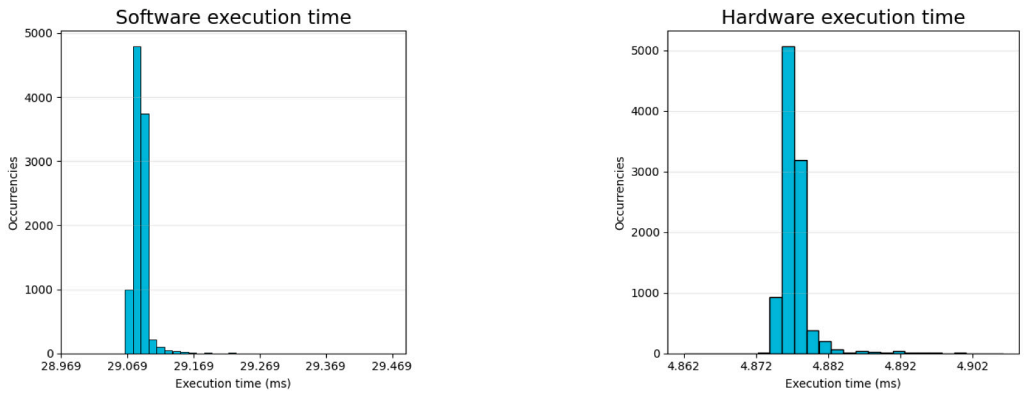


Fig. A.20. Post-processing execution times distributions, measured over 10,000 runs, of the software task (left column) and hardware task (right column), for 40 output classes.

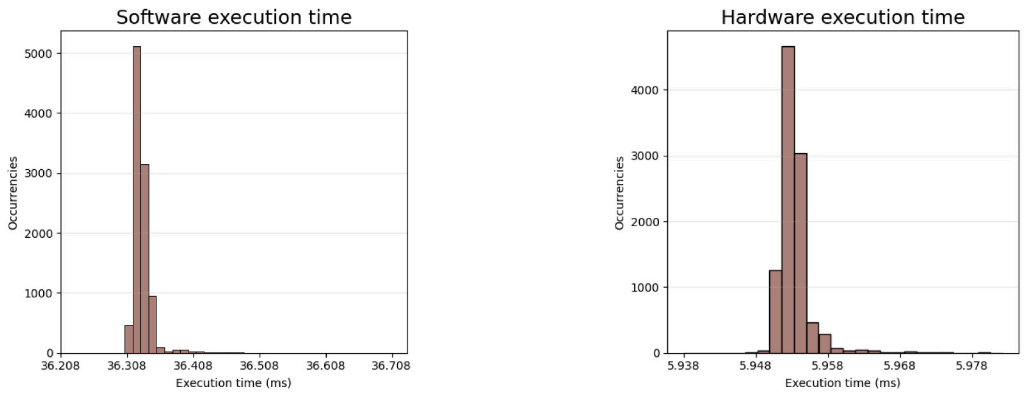


Fig. A.21. Post-processing execution times distributions, measured over 10,000 runs, of the software task (left column) and hardware task (right column), for 50 output classes.

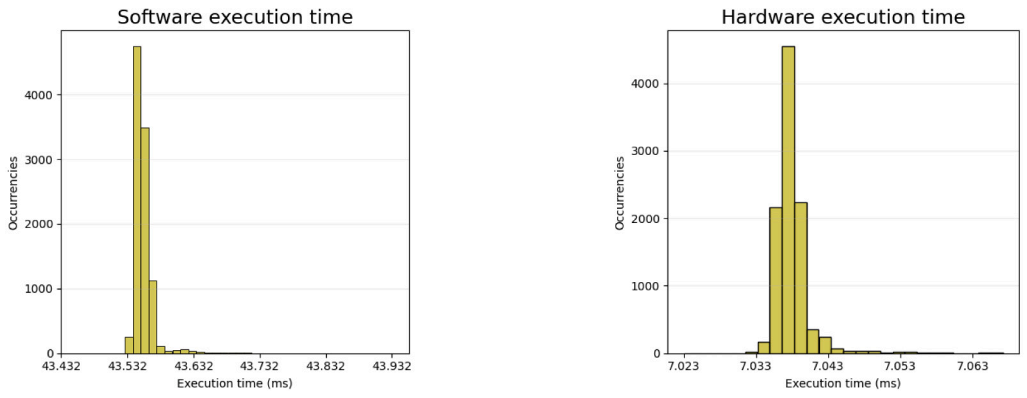


Fig. A.22. Post-processing execution times distributions, measured over 10,000 runs, of the software task (left column) and hardware task (right column), for 60 output classes.

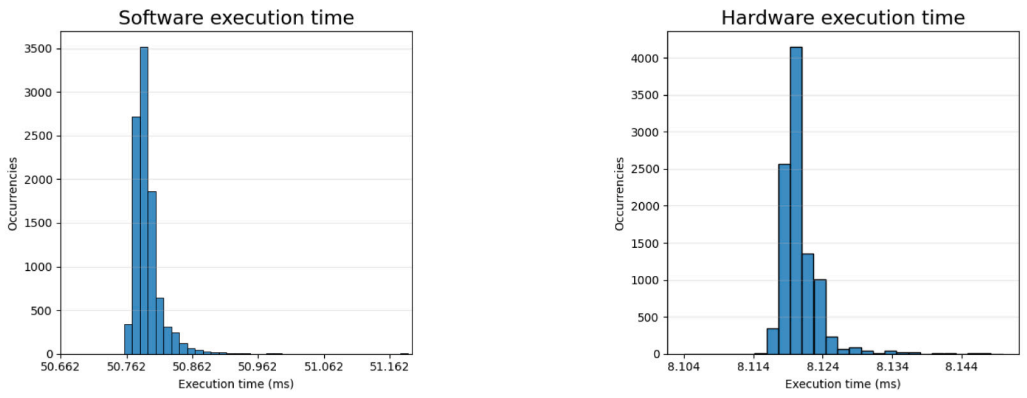


Fig. A.23. Post-processing execution times distributions, measured over 10,000 runs, of the software task (left column) and hardware task (right column), for 70 output classes.

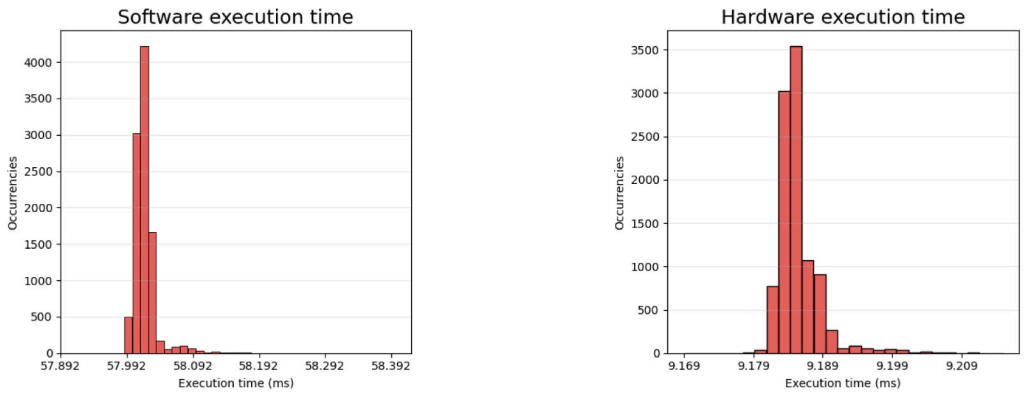


Fig. A.24. Post-processing execution times distributions, measured over 10,000 runs, of the software task (left column) and hardware task (right column), for 80 output classes.

Data availability

The data that has been used is confidential.

References

Abeni, Luca, Buttazzo, Giorgio, 2004. Resource reservation in dynamic real-time systems. Real-Time Syst. 27 (2), 123–167.

Ahmed, M., Mohanta, J., Sanyal, A., 2022. Inspection and identification of transmission line insulator breakdown based on deep learning using aerial images. Electr. Power Syst. Res. URL https://www.sciencedirect.com/science/article/pii/S0378779622004084.

Altera, Intel, 2023. HLS compiler for quartus prime design software. (Accessed 20 September 2023). URL https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

AMD Xilinx, 2023a. DPUCZDX8G for zynq UltraScale+ MPSoCs product guide (PG338). (Accessed 20 September 2023). URL https://docs.xilinx.com/r/en-US/pg338-dpu/Configuring-Clock-Wizard.

AMD Xilinx, 2023b. DPU - deep learning processing unit for convolutional neural network. (Accessed 20 September 2023). URL https://www.xilinx.com/products/intellectual-property/dpu.html.

AMD Xilinx, 2023c. Model zoo. (Accessed 20 September 2023). URL https://github.com/Xilinx/Vitis-AI/tree/master/model_zoo/model-list.

AMD Xilinx, 2023d. Vitis-AI. (Accessed 20 September 2023). URL https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html.

AMD Xilinx, 2023e. Vitis HLS. (Accessed 20 September 2023). URL https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html.

Asyraaf Jainuddin, Ahmad Ammar, Hou, Yew Cheong, Baharuddin, Mohd Zafri, Yussof, Salman, 2020. Performance analysis of deep neural networks for object classification with edge TPU. In: 2020 8th International Conference on Information Technology and Multimedia. ICIMU, pp. 323–328. http://dx.doi.org/10.1109/ICIMU49871.2020.9243367.

Buttazzo, Giorgio C., 2022. Can we trust AI-powered real-time embedded systems? In: Bertogna, M., Terraneo, F., Reghenzani, F. (Eds.), Third Workshop on Next Generation Real-Time Embedded Systems, June 22, 2022, Budapest, Hungary. In: OASIcs, vol. 98, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 1:1–1:14.

Cantero, D., Esnaola-Gonzalez, I., Miguel-Alonso, J., Jauregi, E., 2022. Benchmarking object detection deep learning models in embedded devices. Sensors 22 (11), 4205. http://dx.doi.org/10.3390/s22114205.

Capodieci, N., Cavicchioli, R., Bertogna, M., Paramakuru, A., 2018. Deadline-based scheduling for GPU with preemption support. In: Proc. of the 39th IEEE Real-Time Systems Symposium. RTSS 2018, Nashville, Tennessee, USA.

Cavicchioli, R., Capodieci, N., Bertogna, M., 2017. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In: Proc. of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation. ETFA 2017, Limassol, Cyprus.

Chakradhar, Srimat, Sankaradas, Murugan, Jakkula, Venkata, Cadambi, Srihari, 2010. A dynamically configurable coprocessor for convolutional neural networks. In: Proceedings of the 37th Annual International Symposium on Computer Architecture. ISCA '10, pp. 247–257. http://dx.doi.org/10.1145/1815961.1815993.

Cheng, Hongrong, Zhang, Miao, Shi, Javen Qinfeng, 2023. A survey on deep neural network pruning-taxonomy, comparison, analysis, and recommendations. http://dx.doi.org/10.48550/arXiv.2308.06767, arXiv:2308.06767.

Cittadini, E., Marinoni, M., Biondi, A., Cicero, G., Buttazzo, G., 2023. Supporting AI-powered real-time cyber-physical systems on heterogeneous platforms via hypervisor technology. Real-Time Syst. http://dx.doi.org/10.1007/s11241-023-09402-4.

Fahim, Farah, Hawks, Benjamin, Herwig, Christian, Hirschauer, James, Jindariani, Sergo, Tran, Nhan, Carloni, Luca P., Di Guglielmo, Giuseppe, Harris, Philip, Krupa, Jeffrey, Rankin, Dylan, Valentin, Manuel Blanco, Hester, Josiah, Luo, Yingyi, Mamish, John, Orgrenci-Memik, Seda, Aarrestad, Thea, Javed, Hamza, Loncar, Vladimir, Pierini, Maurizio, Pol, Adrian Alan, Summers, Sioni, Duarte, Javier, Hauck, Scott, Hsu, Shih-Chieh, Ngadiuba, Jennifer, Liu, Mia, Hoang, Duc, Kreinar, Edward, Wu, Zhenbin, 2021. Hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices. arXiv:2103.05579. URL https://api.semanticscholar.org/CorpusID:232168344.

Ge, Zheng, Liu, Songtao, Wang, Feng, Li, Zeming, Sun, Jian, 2021. YOLOX: Exceeding YOLO series in 2021. arXiv:2107.08430.

Gholami, Amir, Kim, Sehoon, Dong, Zhen, Yao, Zhewei, Mahoney, Michael W., Keutzer, Kurt, 2022. A survey of quantization methods for efficient neural network inference. In: Thiruvathukal, George K., Lu, Yung-Hsiang, Kim, Jaeyoun, Chen, Yiran, Chen, Bo (Eds.), Low-Power Computer Vision: Improve the Efficiency of Artificial Intelligence, first ed. Chapman and Hall/CRC, http://dx.doi.org/10.1201/9781003162810.

Google, Coral, 2023. Edge TPU. (Accessed 20 September 2023). URL https://coral.ai/docs/edgetpu/faq/#what-is-the-edge-tpu.

Gunay, Bestami, et al., 2022. LPYOLO: Low precision YOLO for face detection on FPGA. In: 8th World Congress on Electrical Engineering and Computer Systems and Science. EECSS 2022, Prague.

He, K., Zhang, X., Ren, S., Sun, J., 2015. Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition. CVPR.

Hosang, Jan, Benenson, Rodrigo, Schiele, Bernt, 2017. Learning non-maximum suppression. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. CVPR.

Howard, A., Sandler, M., Chen, B., Wang, W., Chen, L., Tan, M., Chu, G., Vasudevan, V., Zhu, Y., Pang, R., Adam, H., Le, Q., 2019. Searching for MobileNetV3. In: 2019 IEEE/CVF International Conference on Computer Vision. ICCV, pp. 1314–1324. http://dx.doi.org/10.1109/ICCV.2019.00140.

Howard, Andrew G., Zhu, Menglong, Chen, Bo, Kalenichenko, Dmitry, Wang, Weijun, Weyand, Tobias, Andreetto, Marco, Adam, Hartwig, 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861. URL https://arxiv.org/abs/1704.04861.

Iandola, Forrest N., Han, Song, Moskewicz, Matthew W., Ashraf, Khalid, Dally, William J., Keutzer, Kurt, 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. http://dx.doi.org/10.48550/arXiv.1602.07360, arXiv:1602.07360. URL https://arxiv.org/abs/1602.07360.

Jones, T., Neogi, N., Krishnakumar, K., 2023. Deep learning and vision-based obstacle detection and classification for autonomous off-road navigation. IEEE Trans. Robot. Autom. 29, 45–60.

Jones, E., Nguyen, L., 2024. Comparative study on power efficiency of GPUs, TPUs, and FPGAs for AI inference in embedded systems. IEEE Trans. Sustain. Comput. 10 (2), 102–115. http://dx.doi.org/10.1109/TSC.2024.3355447.

Krizhevsky, Alex, 2014. One weird trick for parallelizing convolutional neural networks. arXiv:1404.5997. URL http://arxiv.org/abs/1404.5997.

Lin, T., Dollar, P., Girshick, R., He, K., Hariharan, B., Belongie, S., 2017. Feature pyramid networks for object detection. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition. CVPR, pp. 936–944. http://dx.doi.org/10.1109/CVPR.2017.106.

Lin, Tsung-Yi, Maire, Michael, Belongie, Serge J., Bourdev, Lubomir D., Girshick, Ross B., Hays, James, Perona, Pietro, Ramanan, Deva, Dollár, Piotr, Zitnick, C. Lawrence, 2014. Microsoft COCO: Common objects in context. In: Computer Vision – ECCV 2014. pp. 740–755.

Liu, S., Deng, W., 2015. Very deep convolutional neural network based image classification using small training sample size. In: 2015 3rd IAPR Asian Conference on Pattern Recognition. ACPR.

Liu, Zhiqiang, Dou, Yong, Jiang, Jingfei, Wang, Qiang, Chow, Paul, 2017. An FPGA-based processor for training convolutional neural networks. In: 2017 International Conference on Field Programmable Technology. ICFPT, pp. 207–210. http://dx.doi.org/10.1109/FPT.2017.8280142.

Liu, C., Layland, J., 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. J. ACM 20 (1), 40–61.

Ma, Yufei, Zheng, Tu, Cao, Yu, Vrudhula, Sarma, Seo, Jae-sun, 2018. Algorithm-hardware co-design of single shot detector for fast object detection on FPGAs. In: 2018 IEEE/ACM International Conference on Computer-Aided Design. ICCAD, pp. 1–8. http://dx.doi.org/10.1145/3240765.3240775.

Nane, Razvan, Sima, Vlad-Mihai, Pilato, Christian, Choi, Jongsok, Fort, Blair, Canis, Andrew, Chen, Yu Ting, Hsiao, Hsuan, Brown, Stephen, Ferrandi, Fabrizio, Anderson, Jason, Bertels, Koen, 2016. A survey and evaluation of FPGA high-level synthesis tools. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 35 (10), 1591–1604. http://dx.doi.org/10.1109/TCAD.2015.2513673.

Pestana, D., Miranda, P.R., Lopes, J.D., Duarte, R.P., Véstias, M.P., Neto, H.C., Sousa, J.T. De, 2021. A full featured configurable accelerator for object detection with YOLO. IEEE Access http://dx.doi.org/10.1109/ACCESS.2021.3081818.

Qasaimeh, Murad, Denolf, Kristof, Lo, Jack, Vissers, Kees, Zambreno, Joseph, Jones, Phillip H., 2019. Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels. In: 2019 IEEE International Conference on Embedded Software and Systems. ICESS, pp. 1–8. http://dx.doi.org/10.1109/ICESS.2019.8782524.

Redmon, Joseph, Farhadi, Ali, 2018. YOLOv3: An incremental improvement. arXiv:1804.02767.

Ren, S., He, K., Girshick, R., Sun, J., 2017. Faster R-CNN: Towards real-time object detection with region proposal networks. IEEE Trans. Pattern Anal. Mach. Intell..

Rodriguez, M., Martinez, L., 2024. FPGA-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: The ARTICo3 framework. MDPI Electron. 13 (4), 9418–9439. http://dx.doi.org/10.1109/ACCESS.2024.3352266.

Rosero-Montalvo, P.D., Tozun, P., Hernandez, W., 2024. Optimized CNN architectures benchmarking in hardware-constrained edge devices in IoT environments. IEEE Internet Things J. 11, 20357–20366. http://dx.doi.org/10.1109/JIOT.2024.3369607.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L., 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition. CVPR, pp. 4510–4520. http://dx.doi.org/10.1109/CVPR.2018.00474.

Seshadri, Kiran, Akin, Berkin, Laudon, James, Narayanaswami, Ravi, Yazdanbakhsh, Amir, 2022. An evaluation of edge TPU accelerators for convolutional neural networks. In: 2022 IEEE International Symposium on Workload Characterization. IISWC, pp. 79–91. http://dx.doi.org/10.1109/IISWC55918.2022.00017.

Siemens, 2023. Catapult high-level verification solutions. (Accessed 20 September 2023). URL https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls-verification/.

Smith, J., Lee, A., 2024. High-level power estimation techniques in embedded systems hardware: An approach for efficient power management. J. Low Power Electron. Appl. 12 (1), 23–45. http://dx.doi.org/10.3390/jlpea12010023.

Stone, John E., Gohara, David, Shi, Guochun, 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. 12 (3), 66–73. http://dx.doi.org/10.1109/MCSE.2010.69.

Tan, Mingxing, Le, Quoc V., 2020. EfficientNet: Rethinking model scaling for convolutional neural networks. http://dx.doi.org/10.48550/arXiv.1905.11946, arXiv:1905.11946. URL https://arxiv.org/abs/1905.11946.

Tian, Z., Shen, C., Chen, H., He, T., 2019. FCOS: Fully convolutional one-stage object detection. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. ICCV.

Wulfert, L., Kühnel, J., Gembaczka, P., 2024. AIfES: A next-generation edge AI framework. IEEE Trans. Pattern Anal. Mach. Intell. 46, 4519–4533. http://dx.doi.org/10.1109/TPAMI.2024.3355495.

Xing, J., Cioffi, G., Hidalgo-Carrió, J., Scaramuzza, D., 2023. Autonomous power line inspection with drones via perception-aware MPC. arXiv:2304.00959. URL https://arxiv.org/abs/2304.00959.

Ye, H., Hao, C., Cheng, J., Jeong, H., Huang, J., Neuendorffer, S., Chen, D., 2022. ScaleHLS: A new scalable high-level synthesis framework on multi-level intermediate representation. In: 2022 IEEE International Symposium on High-Performance Computer Architecture. HPCA, pp. 741–755. http://dx.doi.org/10.1109/HPCA53966.2022.00060.

Zhang, Xiangyu, Zhou, Xinyu, Lin, Mengxiao, Sun, Jian, 2018. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition. CVPR, pp. 6848–6856. http://dx.doi.org/10.1109/CVPR.2018.00716.

Zhou, Yan, Chen, Shaochang, Wang, Yiming, Huan, Wenming, 2020a. Review of research on lightweight convolutional neural networks. In: 2020 IEEE 5th Information Technology and Mechatronics Engineering Conference. ITOEC, pp. 1713–1720. http://dx.doi.org/10.1109/ITOEC49072.2020.9141847.

Zhou, Daquan, Hou, Qibin, Chen, Yunpeng, Feng, Jiashi, Yan, Shuicheng, 2020b. Rethinking bottleneck structure for efficient mobile network design. In: Computer Vision – ECCV 2020. pp. 680–697.