EasyChair Preprint

№ 8997

Computer-Vision Applied to Embedded Real Time Traffic Control System

Hazzem Abdelhalim, Ziet Lahcene, Mouaz Mohamed Amine and Radjah Fayçal

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

October 5, 2022

1st International Conference on Innovative Academic Studies

https://www.icias.net/

September 10 - 13, 2022, Konya, Turkey

Computer Vision Applied to Embedded Real Time Traffic Control System

Hazzem Abelhalim*[1],Ziet Lahcene[1], Mouaz Mohamed Amine [1], and Radjah Fayça1[1]

[1]*Electronics department/technology faculty,Ferhat Abbas Setif1 University, Algeria*

(abdelhalim.hazzem@univ-setif.dz) Email of the corresponding author

Abstract – In this work, the capabilities offered by the Pynq Z2 development board are used for designing computer vision applications. The FPGA part of the system with its basic overlay IPs were performed by Python language in addition to the Open-CV library to firstly process the video road traffic (detection and vehicles counting) and then show how the mixed design for an embedded real complex system is proceeded in a System On Chip (SoC). Some other components such as grayscale conversion edge detection by Sobel filter have been developed using Xilinx-Vivado Heigh Level Synthesis HLS and inserted into the base overlay of the Logic Part of the Zynq-7000 processor to increase the video processing speed approach used, the principal results and major conclusions.

Keywords – FPGA, ZYNQ, HLS, SOC, Embedded computer vision

I. Introduction

This Applications such as streaming video, recognition, image processing and highly interactive services place new demands on the computing units that implement these applications.

Hardware accelerated operations are the most appropriate way to increase performance using dedicated hardware architectures performing parallel processing. With the advent of Field Programmable Gate Array (FPGA) technology, hardware architectures can be implemented at lower cost. In fact, FPGAs are the basis of Reconfigurable Computing [37], a technology that offers great flexibility as well as unprecedented levels of performance. Indeed, performance can be significantly increased through the use of custom computing units running in parallel. These units are mapped onto a reconfigurable structure to achieve the benefits of an application-specific approach at the cost of a general-purpose product. Costs and time-to-market are also reduced as the manufacturing phase is replaced by field programming. Finally, energy consumption can be reduced as the circuits are optimized for the application.

- *Experimentation platform*:

In this work we have used the PYNQ-Z2 which is a very good card, equipped with a Zynq 7020 processor, that can be used in several fields and applications such as image processing or sound processing thanks to its various input and output ports like HDMI-IN an HDMI-out that we used in our application.

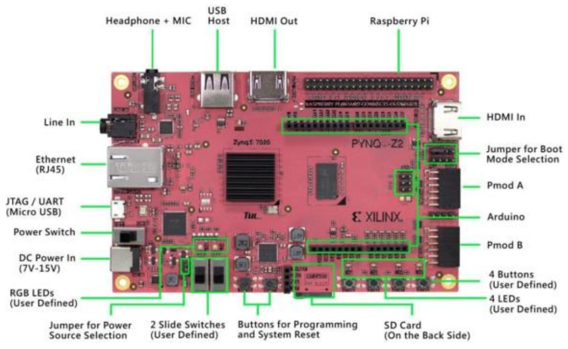

Fig. 1 The prototype board

- *PL-PS Interface:*

The PS-PL interface contains all the signals available to the PL designer to integrate PL and PS based functions. There are two types of interfaces between the two:

1. Functional interfaces that include AXI (Advanced eXtensible Interface) interconnect, MIO Extended EMIO (Extended Multiplexed Input/Output) interfaces for most I/O devices, interrupts, DMA (Direct Memory Access) flow control, clocks, and debug interfaces. These signals are available for connection to user-designed IP blocks in the PL.

2. Configuration signals that include Processor Configuration Access Point (PCAP), configuration status, Single Event Upset (SEU), and schedule/delete/reset signals [26].

- *Working environment*:

The environment PYNQ is a project from Xilinx-AMD® that makes the use of Xilinx platforms easier and by using the libraries of python and the libraries provided by Xilinx, the designer can benefit both de advantages of programmable logic (PL) and programming software (PS), so by using this environment we managed to build some computer vision applications.

- *Overlay notion*:

Overlays are reconfigurable architectures that can be implemented on FPGAs. They are regular designs described using the structural HDL language, but with reconfiguration capabilities. They can be considered as "softcore FPGA IPs". Thus, an overlay can be seen from two sides:

Firstly, it can be seen as the functional architecture or functional view is the top layer, it is the set of reconfigurable elements available for the applications targeting the overlay.

Secondly, the implementation is the view from below, which is how the functional architecture is implemented and synthesized (like an ordinary IP) on the host FPGA.

- *PYNQ libraries*:

Hardware IP libraries are included in Vivado and can be used to connect to a wide range of interface standards and protocols. PYNQ provides a Python API for a number of common devices including video (HDMI input and output), GPIO devices (buttons, switches, LEDs), sensors and actuators. The PYNQ API can also be extended to support additional IPs.

PYNQ also supports low-level control of an overlay, including reading/writing memory-mapped I/O, memory allocation (e.g., for use by a master implemented in the PL part), control and management of an overlay (downloading an overlay, reading IP from an overlay), and low-level control of the PL via the "bitstream"

Since the video library is the subject of the example implementation that will be presented later, we will present the hardware subsystem.

The Video sub-package contains a collection of drivers for reading from the HDMI-In port, writing to the HDMI-Out port, data transfer, interrupt configuration and video image manipulation.

The video hardware subsystem consists of an HDMI-In block, an HDMI-Out block and a video DMA. The HDMI-In and HDMI-Out blocks also support color space conversions and changing the number of channels in each pixel.

Video data can be captured from the HDMI-In and passed to the DRAM using the video DMA, or passed from the PS DRAM to the HDMI-Out, diagram in Figure 2.

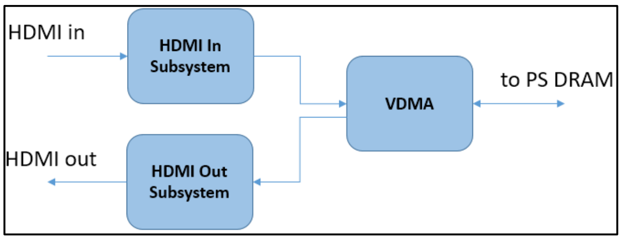

Fig. 2 HDMI bloc Diagram

II. SURVEILLANCE SYSTEM :

In this section we will describe in detail the different steps and operations used in a traffic monitoring system. The following figure represents the system used for the application:

It is a prototyping board with a ZYNQ processor embedding a mixed hard-soft application. The video signal of the road traffic captured by the HDMI source or webcam will undergo various operations of filtering, extraction and overlay to display the information concerning the traffic.

To achieve this application, we first considered necessary to master the video capture. This operation can be done in two different ways. The first one uses the Fswebcam function, the second one uses functions of the computer vision library "Open-CV".

B. *Fswebcam:*

The fswebcam function is very useful for an introduction to image capture. It offers different arguments (options) that help us to better capture and manipulate images for the rest of the work.

The Structure of the function is as follows:

``fswebcam [<options>] <filename> [[<options>] <filename> ... ]``

Capture acquisition options can be:

-d Sets the source or device to be used. The source module is selected automatically unless specified in the prefix.

-f, --frequency <frequency>: Sets the frequency of the selected input or tuner. The value can be read as KHz or MHz depending on the input or tuner.

-p, --palette <name>: specifies the color palette exp: RGB32, RGB24, BGR32, BGR24, GRAY

-r, --resolution <dimensions>: Sets the resolution of the source or device image. The actual resolution used may differ if the source or device cannot capture at the specified resolution, The default is "384x288".

--fps <frames per second>: Sets the frame rate of the capture device.

-F, --frames <number>: Sets the number of frames to capture. More frames means less noise in the final image.

-D, --delay <delay>: Inserts a delay after the source or device has been opened and initialized, and before the capture begins. Some devices require this delay to allow the image to stabilize after a parameter has been changed. The delay is specified in seconds.

C. *Capturing a video by using the OpenCV library:*

In this section we will show how to capture and display the image and then the video using computer vision functions.

Reading and displaying the image:

To use the OpenCV library in python, we need to install and import the following libraries as prerequisites:

* NumPy library (Necessary, because OpenCV uses it in the background).

* OpenCV python

To read the images, the method cv2.imread(argument) is used with the steps below. This method loads an image from the file specified in the argument. It returns an empty array in case of errors.

Steps to follow:

- Import the 3 libraries **NumPy, cv2** and **Matplotlib**

- Load the image using **cv2.imread**

- Use **plt.imshow ()** to display the image.

D. *Implementation of a Road Traffic Flow Control Application:*

Once the video stream is retrieved, different processing steps are required:

a) Conversion to grayscale is done by the function: cv2.cvtColor(). This reduces the amount of data to be processed. The original 3 channel frame (RGB) becomes a single channel gray level matrix, see the following figure

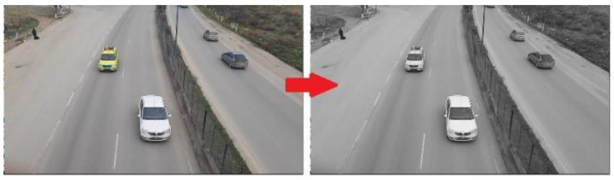

Fig3. Color space conversion

b) Define regions of interest in order to choose only one part of the frame to process, which reduces the calculations made by the CPU (one part of the frame versus a complete frame). In the following example, two areas of interest

(ROI) of 300 pixels length and 100 pixels width, represented by rectangles, will be used to detect the passage of objects (vehicles in our case). This task will be performed by the commands: cv2.rectangle and cv2.putext (figure 4)

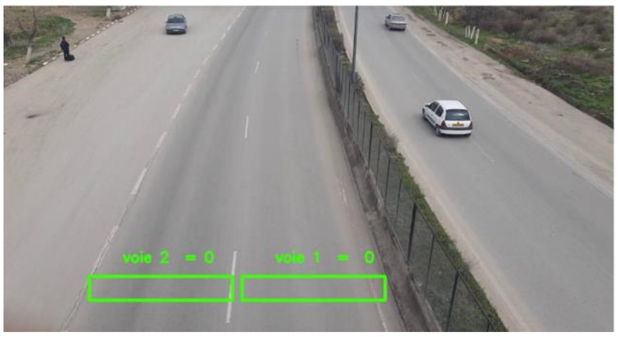

Fig4. Region of interest.

c) The detection of the cars is done in two phases: the first one subtracts the first frame from the next ones, the second one binarizes (in black and white) the image with a certain configurable threshold, used to adjust the sensitivity of the capture. The white color in the image, indicates the existence of motion. These two are provided by the function cv2.BackgroundSubstractor

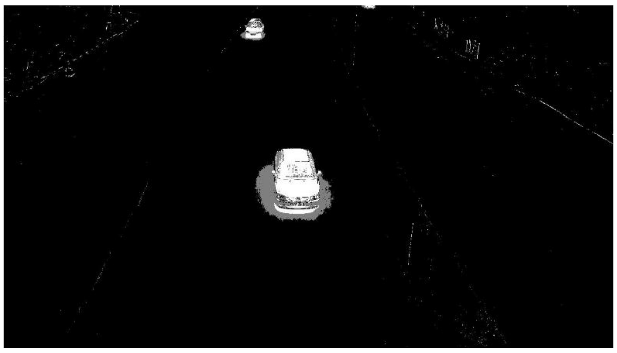

Fig5. The result of the subtraction and the binarization of the images.

The inhomogeneity of the surface of the objects pushes to use morphological transformations of dilation and erosion by a Kernel whose parameters were determined by several experimental trials. These tests led to the use of a 15x15 filter (see method cv2.Blur paragraph 1.6.1).

The following figure represents (a) the original image (b) the noisy binarized image, (c) the erosion filtered image and (d) the image with the dilated useful areas.

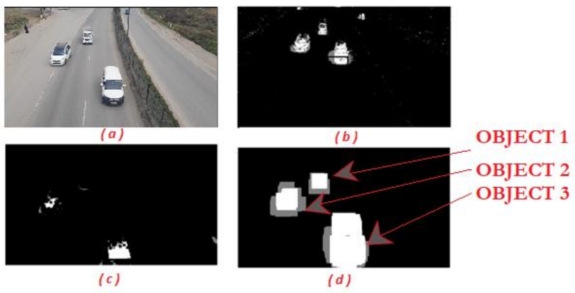

Fig6. The preprocessing operation for detection

Figure 6 The preprocessing operation for detection

d) After binarization and noise filtering, the edge detection step will be performed by the function **cv2.FindContours () :** It defines all the points along the image boundary. These points are mainly useful for the analysis of the shape of the image provided, for the detection of the size and dimension of the object to be detected, and for the detection of specific objects in our case is the cars.

e) After drawing the contours, a condition on the radius by the function cv2.minenclousringCircle is imposed in order to decide on the interpretation on the object. This function returns the center and the minimal radius of the circle to consider. Figure 7 shows the experimental values (the values are between 80 pixels and 130 pixels in most cases).

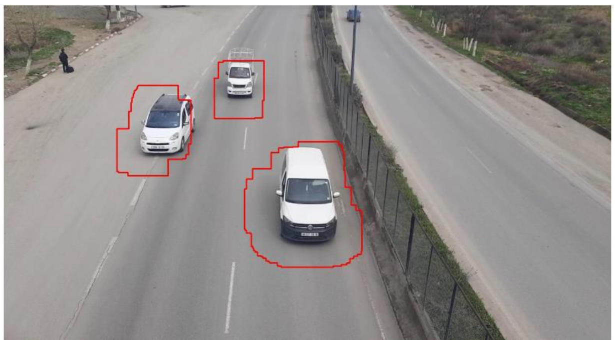

Fig7. Drawing the contours around cars.

The value (80 pixels) was chosen as the minimum radius to decide the presence or not of a vehicle.

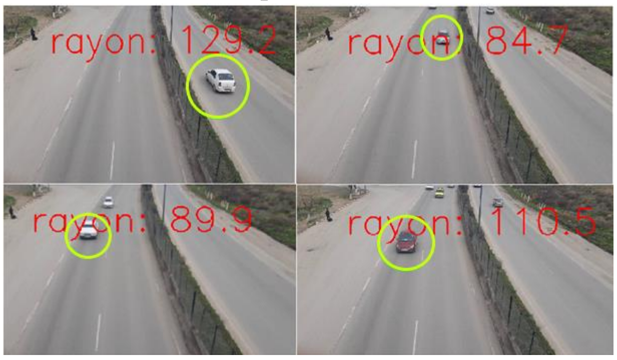

Fig8. Radius of different cars.

All these treatments are applied only on the areas of interest (way 1 and way 2) in order to reduce the excess of calculations on the processor the result is shown in this figure 9 is way 1 and figure 10 is way 2:

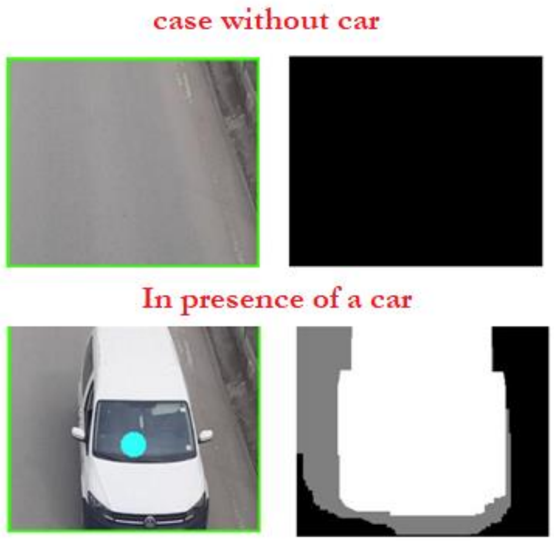

Fig9. Region of interset 1.

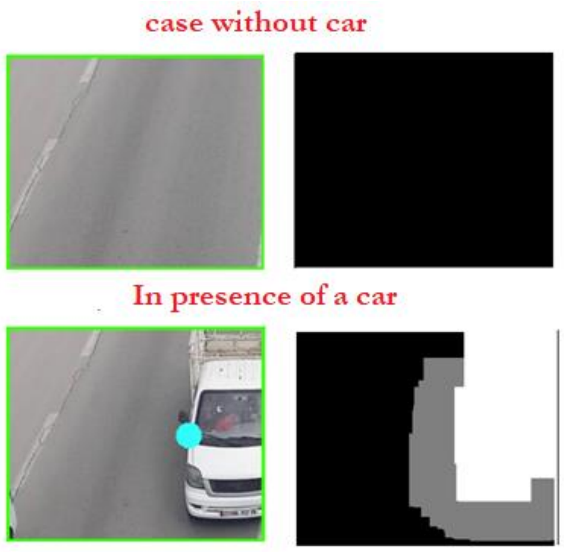

Fig10. Region of interset 2.

The display of the information recovered by the detection and the number of frames processed per second in the video will be embedded in the video itself, using respectively the state of two counters incremented at each vehicle detection pass, and the inverse of the difference in time between the two successive frames processed.

Note: Other information such as date and time may be included in the display, if necessary. Results in figure 11

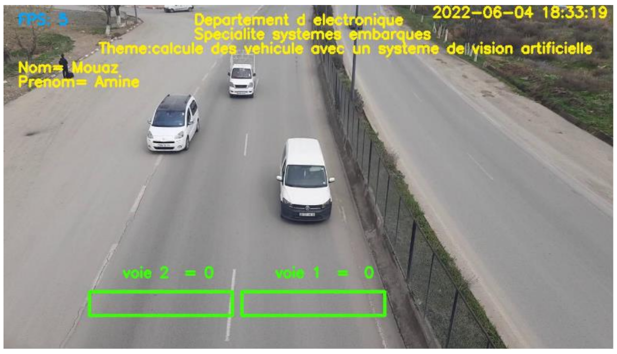
FPS: 5
Departement d electronique
Specialite systemes embarques
Theme:calcule des vehicule avec un systeme de vision artificielle
Nom= Mouaz
Prenom= Amine
2022-06-04 18:33:19
voie 2 = 0
voie 1 = 0

Fig11. The final result of the Road Traffic Flow Control Application.

E. *Comparison of soft application and accelerated application:*

The second application suggested in this project consists in highlighting the improvement of the real time performances in the processing systems particularly those related to the video by the use of the task parallelism approach by introducing hardware gas pedals in the calculation parts.

For that, we exploited the example of edge detection in a video where the filtering by the SOBEL method was attributed to an IP implemented in the FPGA part of the Zynq 7020 processor.

To show the effect of the acceleration, the application will first be run on the system without IP (i.e. the filtering is executed by a Python program using CV2), then on the same system but this time if the filtering is done by a circuit.

The first case is by using OpenCV built-in function

And in the second case we used a hardware accelerated function for the same Sobel filter

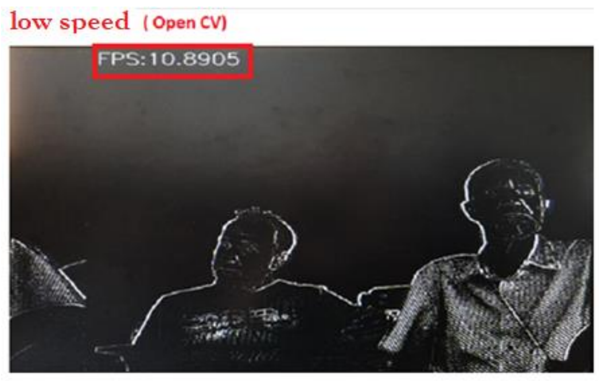

Fig12. Sobel filter using OpenCV.

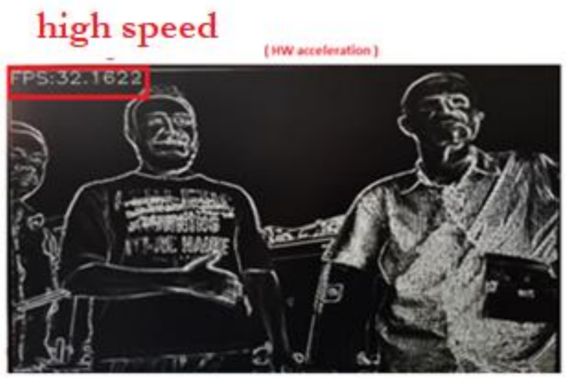

Fig13. Sobel filter using hardware acceleration

We notice that in the second case, the Sobel hardware filter has significantly improved 10 to 30 frames per second (FPS) the speed of the video.

F. *Conclusion and perspectives:*

The field of video surveillance in general remains one of the most complex problems, in spite of current active research. Indeed, many real conditions, difficult to model and to predict, limit the evolution in the different treatments. In addition to all this, the EDA tools delivered by the major equipment and component manufacturers have bugs in their operation.

In spite of all these problems we have managed, using a Pynq Z2 card, to design and realize a prototype of a video surveillance system intended for the control of the traffic flow of a highway. The realization was successfully tested on the dual carriageway El Bez-Setif , Algeria. A video capture of a few minutes will represent the video signal at the input of the system emulating the real flow that must be presented either at the HDMI-in. or via a USB webcam. The first attempt revealed a failure in the number of FPS frames better than the current

References

[1] OpenCV Face Detection HDMI url:https://github.com/Xilinx/PYNQ/blob/v2.0/boards/PynqZ1/base/notebooks/video/opencv_face_detect_hdmi.ipynb

[2] T. Wu, Y. Wang, W. Shi and J. Lu, "HydraMini: An FPGA-based Affordable Research and Education Platform for Autonomous Driving," 2020 International Conference on Connected and Autonomous Driving (MetroCAD), Detroit, MI, USA, 2020, pp. 45-52, doi: 10.1109/MetroCAD48866.2020.00016.

[3] . Xilinx Inc., “Zynq-7000 SoC product.” 2020,

[4] V. Y. Çambay, A. Uçar and M. A. Arserim, "Object Detection on FPGAs and GPUs by Using Accelerated Deep Learning," 2019 International Artificial Intelligence and Data Processing Symposium (IDAP), Malatya, Turkey, 2019, pp. 1-5, doi: 10.1109/IDAP.2019.8875870.M.

[5] BNN-PYNQ PIP INSTALL Package. url: https://github.com/Xilinx/BNN-PYNQ/

[6] . QNN-MO-PYNQ PIP INSTALL Package. url:https://github.com/Xilinx/QNN-MOPYNQ

[7] OpenCV Face Detection HDMI url:https://github.com/Xilinx/PYNQ/blob/v2.0/boards/PynqZ1/base/notebooks/video/opencv_face_detect_hdmi.ipynb

[8] OpenCV Filters Webcam. Url: https://github.com/Xilinx/PYNQ/blob/v1.4/Pynq