sciendo

Carpathian Journal of Electronic and Computer Engineering 14/1 (2021) 20-23

DOI: 10.2478/cjece-2021-0004

Hardware accelerated image processing on FPGA based PYNQ-Z2 board

Ákos Mándi
Intelligent Embedded System Research Laboaratory
University of Debrecen
Debrecen, Hungary

Jeney Máté
Intelligent Embedded System Research Laboaratory
University of Debrecen
Debrecen, Hungary

Dominik Rózsa
Intelligent Embedded System Research Laboaratory
University of Debrecen
Debrecen, Hungary

Stefan Oniga
Intelligent Embedded System Research Laboaratory
University of Debrecen
Debrecen, Hungary

***Abstract*—In this paper we present the partial results of a research in progress made in order to develop a prototype of a self-driving car's controller and processing unit. The framework that we used consisted of a camera for input of visual imagery information (Logitech 720p), a laser range finder for depth and object sensing (Parallax; PulsedLight LIDAR-Lite v2), and the main processing board, an FPGA based accelerator board PYNQ Z2.**

Keywords—FPGA, self-driving car, inference acceleration, PYNQ

I. Introduction

Computer vision is a branch of artificial intelligence (AI) that allows computers and systems to extract useful information from digital pictures, videos, and other visual inputs, as well as execute actions or make suggestions based on that data. If artificial intelligence allows computers to think, computer vision allows them to perceive, observe, and comprehend the world around them. The latest cameras can capture images at resolutions and levels of detail that far beyond what the human eye can see. Computers can also identify and measure color differences with a high degree of accuracy. However, making sense of the information content of those images has been a challenge for machines for decades.

Feature-based methods, when combined with machine learning techniques and complicated optimization frameworks, have witnessed a resurgence in latest studies. The area of computer vision has been given new life thanks to the progress of Deep Learning algorithms. Deep learning algorithms have outperformed previous methods on various benchmark computer vision data sets, including classification, segmentation, and optical flow.

There are a tremendous number of application fields regarding this research topic: Medicine, Autonomous vehicles, Military, Agriculture and Meteorology just to name a few. We can see that this subject can carry endless possibilities and it may be good to learn more about as it can be useful regarding our project.

II. Materials And Methods

A. PYNQ-Z2 accelerator board

The PYNQ-Z2 board developed by TUL Corporation [1], is a board based on Xilinx Zynq SoC, and was designed for the Xilinx University Program in order to support PYNQ (Python Productivity for Zynq) [2] framework for embedded systems development.

ZYNQ is a line of FPGA-CPU hybrid processing unit made by the company Xilinx, it consists of two main parts: the programmable logic and the programmable system, PL and PS for short. The PL is the FPGA part, which is usable in any ways an ordinary FPGA could be configured. The PS is a dual core arm Cortex-A9 series CPU running at 650MHz with full software capabilities just as in other arm Cortex systems. The main innovation is that these systems are conjoined in-die, using an intermediate medium called AMBA Interconnect. This interconnected nature makes it highly flexible, and extremely capable of offloading traditionally really CPU heavy computations onto the FPGA fabric through e.g. DMA (Direct Memory Access).

The exact model on the PNYQ-Z2 is ZYNQ XC7Z020-1CLG400C. It contains 13,300 logic slices, each with four 6-input LUTs and 8 flip-flops, which is equivalent to Xilinx Artix-7 FPGA, 220 DSP slices, 512MB DDR3, 16MB Quad SPI-Flash to name a few specifics.



Fig. 1. PYNQ-Z2 accelerator board



Fig. 2. Vehicles recognition on an image

B. Computer vision on PYNQ-Z2

Reviewing Computer Vison and Image Recognition concepts, research paper we decided to use xfOpenCV [3] library for the FPGA accelerated part of the image processing pipeline, a framework providing premade, openCV derived functions or so-called kernels that can be transformed into HDL modules using the high-level synthesis tool. There are 60+ of these kernels, all of them are optimized for Xilinx FPGA-s and SoC-s. The input to the HLS tool is the xfOpenCV C/C++ code and the output is an IP which is easily, instantly integrable and usable in Vivado Block Design.

The block design we used is the PYNQ-HelloWorld [4] structure, we only changed the "Resize IP" (Fig. 3).

The PL side contains the design we created and seen in the Vivado Block Design view, and on the PS side we can see an important aspect of the ZYNQ ecosystem, the AXI Switch Network, which is the key exchange point between the PL and the PS, whatever information is sent or accepted from or to the PL must go through this integrated AXI controller.

Fig. 4 shows the overlay later to-be exported onto the PYNQ. This block design contains an element, namely the "resize_accel_0" which is the IP generated by the HLS, and our aim is exchanging this IP to an IP that has the same interfaces but performs another type of image transformation.

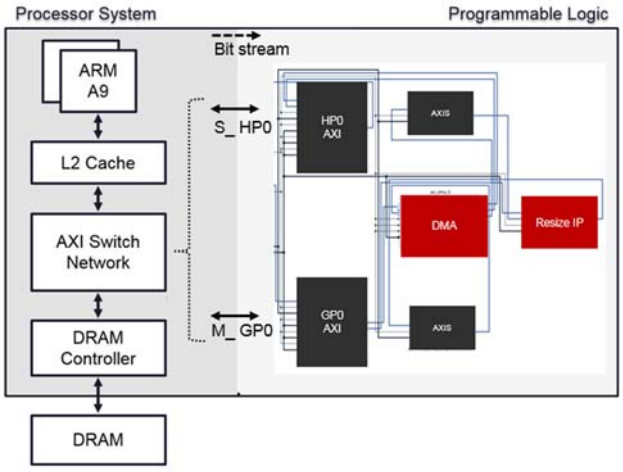

Fig. 3. Architecture of the design

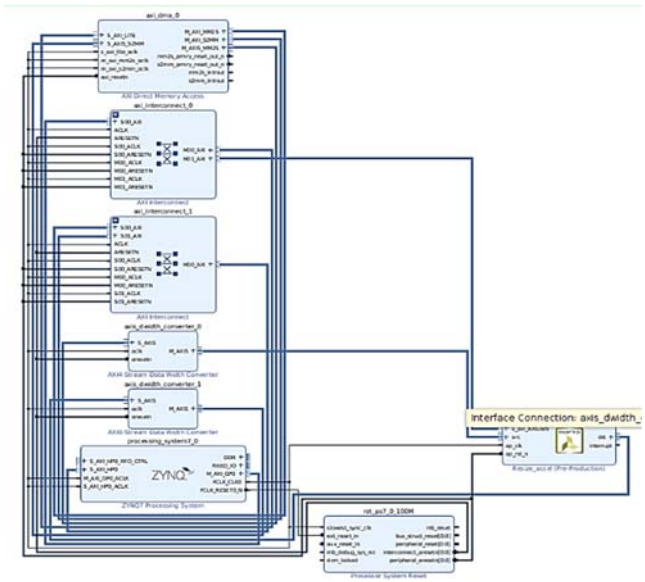

Fig. 4. The overlay to-be exported onto the PYNQ

III. Image Transformation Pipeline

The very first version of a rudimentary pipeline highlighted the following important point: both components (X and Y dimensions) are not needed from the output of the Sobel filter, as we are searching for parallel lane lines in front of the car, we do not need the X dimensioned lines present in the image. Before separating the two, and using only the Y dimensional output, the following styled output was produced Fig. 5.

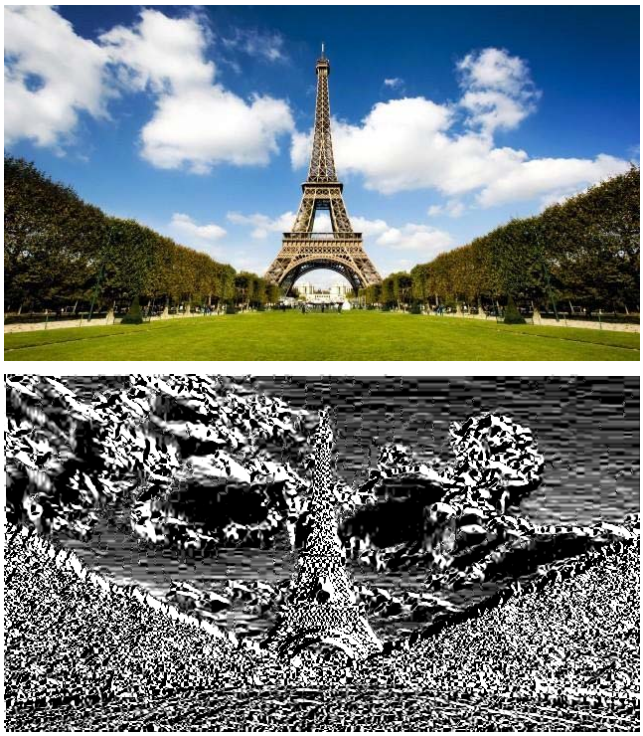

Fig. 5. Edge detection using Sobel filter on Eiffel tower picture

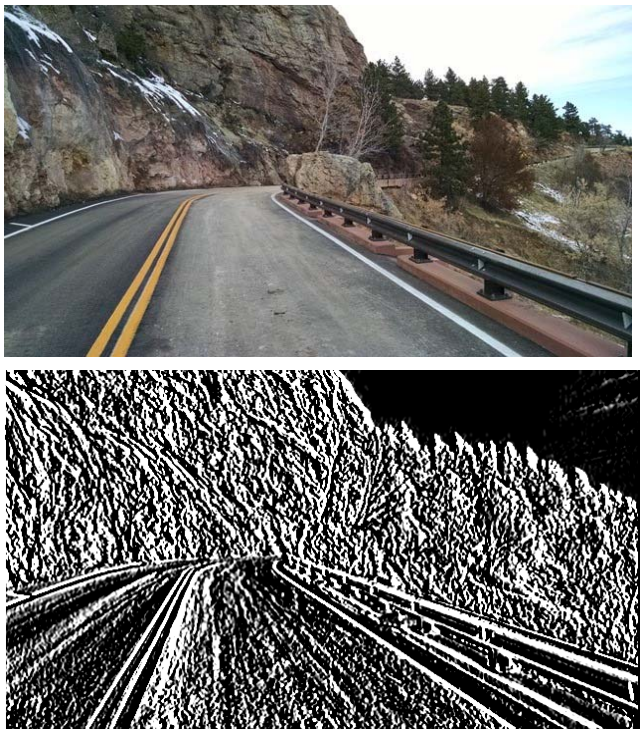

Fig. 6. Result of image transformation without blurring and thresholding

Before reaching the final pipeline, different variations were tried. In Fig. 6 we can see a version where the functions were used with stock parameters, and this version did not contain blurring and thresholding. It can be seen that the edge detection algorithm is too heavily used and this version was impossible to use, the granularity is too high, no definite lines stick out to be used.

The next version contained smoothing and thresholding, Fig. 7.



Fig. 7. Image transformation using smoothing and thresholding

This was a major step forward, as it helped in removing the granular noise present in the previous version, although problems still remained: line consistency issues are still present, less in volume, but high-frequency noise is present in the road surface.

This version underwent tweaking, changing the order of the parts of the pipeline, and the final pipeline became the following:

BGR color space to gray color space conversion -> Thresholding -> Gaussian Blurring -> Sobel Filtering -> gray color space to BGR color space conversion, Fig. 8

```
#pragma HLS dataflow
    axis2xfMat(_src, src, src_rows, src_cols);

    xf::bgr2gray <XF_8UC3,XF_8UC1, HEIGHT, WIDTH, NPC1> (_src, img_1);

    xf::Threshold <XF_THRESHOLD_TYPE_BINARY, XF_8UC1, HEIGHT, WIDTH, NPC1>(img_1, thresholded, 120, 255);

    xf::GaussianBlur<7, XF_BORDER_CONSTANT, XF_8UC1, HEIGHT, WIDTH, NPC1>(thresholded, gaussian_out, 2.8);

    xf::Sobel<XF_BORDER_CONSTANT, 5 , XF_8UC1, XF_8UC1, HEIGHT, WIDTH, NPC1, false>(gaussian_out, sobel_1, sobel_0);

    xf::zero<XF_8UC1, HEIGHT, WIDTH, NPC1>(sobel_0, sobel_zero);
    xf::bitwise_or< XF_8UC1, HEIGHT, WIDTH, NPC1 >(sobel_zero, sobel_1, sobel_out);

    xf::gray2bgr <XF_8UC1,XF_8UC3, HEIGHT, WIDTH, NPC1> (sobel_out, img_2);
    xfMat2axis(img_2, dst, dst_rows, dst_cols);
```

Fig. 8. Sample of final pipeline code

In next section we present in detail the used functions.

As in the example project, before the FPGA processing takes place the image is captured using a camera and that information is processed by PS (Programmable System), then it is passed onto the DDR memory, and from the memory the PL reads the data.

The axis2xfMat function converts the incoming information which is coming through the axi lane into the datatype, so the xfOpenCV functions can work with.

The bgr2gray function converts the color image data into black and white pixel data, as the following functions accept only black and white, 8-bit, unsigned, 1 channel pixel data (XF_8UC1).

The Threshold function gets the converted image from the previous function and performs a thresholding, discarding information we don't need.

The GaussianBlur function gets the thresholded image data from the previous function and performs a Gaussian operation on it, smoothing out unwanted information in the image.

The Sobel function gets the Gaussian filtered image data from the previous function and performs Sobel edge detection. It has two outputs, which correspond to the X and Y dimensioned outputs of a Sobel filter. As we only need the vertical output, we use only the sobel_1 later. This is performed using a zero function, which takes whatever input it has, and zeroes all pixels. We pass the unwanted output of the Sobel filter to this zero function, and after this operation we bitwise OR this totally zeroed image with the wanted, correct, vertically line detected output of the Sobel filter.

The gray2bgr function takes the output of the previous OR function and converts the black and white image data back into BGR color space, so that it can be further processed in software.

The parameters found in the Threshold and GaussianBlur functions are hard coded based on tweaking by hand to achieve the desired output. It can and should be implemented through separate AXI interfaces, doing so they could be set from the software instantaneously. This is a further to-do.

After exporting the HLS generated code as an IP to be used in the Vivado, we don't have to import it again in the Vivado environment, when the HLS generated IP overwrites the previous revision, then the Vivado will automatically offer us an option to update the IP because it is outdated.

IV. Results

This final version, which uses the final pipeline mentioned above produces the output shown in Fig. 9.

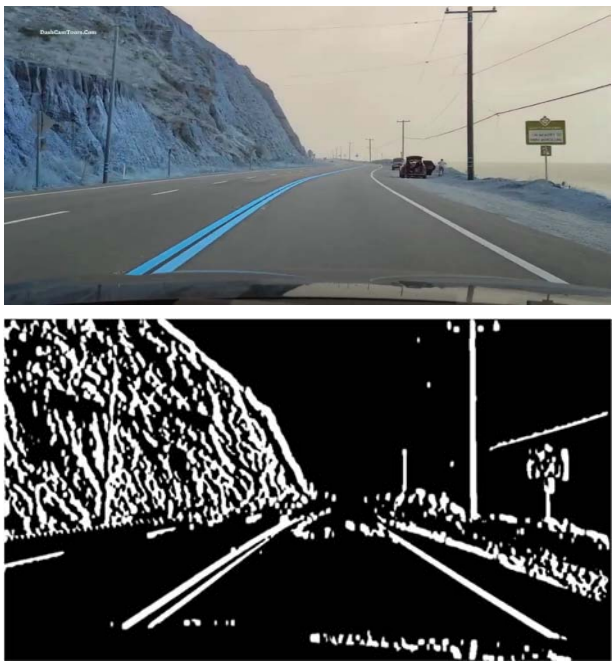

Fig. 9. Image transformation using smoothing and thresholding

Resource utilization report generated by Vivado Design Suite is presented in Table I.

As we can see the design uses only a few resources, still plenty of room for extensions. The speculative reason for the better results compared to the HLS estimates is that Vivado optimized the resources in a way that it uses fewer components in the FPGA fabric compared to the estimates of the HLS tool.

TABLE I. Resource Utilization Report

Resource	Utilization	Available	Utilization %
LUT	6159	53200	11.58
LUTRAM	376	17400	2.16
FF	7807	106400	7.34
BRAM	14	140	10.00
DSP	33	220	15.00
BUFG	1	32	3.13

After exporting the block design, we need to place the two files to the PYNQ's SD card, this newly created overlay should be used instead of the previous one.

The example project uses the PIL library for manipulating images, but this did not prove to be resilient enough to use it for the software side image handling as it was slowing down the pipeline heavily. For solving this problem, we used a library that could capture and prepare the images much faster than the PIL. The final choose was the imutils library, which uses threading to offload the dual core ARM processor.

This proved to be a viable solution, it provided significant improvements in the speed of the image capturing, which previously was a serious bottleneck.

V. Conclusion

We succeeded to design and implement an image transformation pipeline using an PYNQ Z2 board. The time needed to process the image is between 40-50 ms, which equals around 20-24 fps, which is more than enough as a demonstrative case, so we reached our goal.

High frequency artifacts are still present on the road surface sometimes, but this problem with further tweaking should be relatively easy to solve.

Another to-do is using a ROI mask to discard parts of the picture not useful for us, which is basically every other part apart from the road. A trapezoid shape would be the most sufficient, but due to the time constraints of our studies, we only got this far.

Acknowledgment

This work was supported by the construction EFOP-3.6.3-VEKOP-16-2017-00002. The project was co-financed by the Hungarian Government and the European Social Fund.

References

[1] TUL Corporation, TUL PYNQ-Z2 Product Specification, https://www.tul.com.tw/productspynq-z2.html.
Accessed February 20, 2021.

[2] ***, Development Boards, http://www.pynq.io/board.html.
Accessed February 20 2021.

[3] XILINX INC, Xilinx OpenCV User Guide, UG1233 (v2019.1) June 5, 2019.

[4] XILINX INC, Xilinx PYNQ-HelloWorld example project repository at tag v2.5 https://github.com/Xilinx/PYNQ-HelloWorld/tree/v2.5.
Accessed March 15, 2021.

[5] SHAWN HYMEL, LIDAR-Lite v3 Hookup Guide, https://learn.sparkfun.com/tutorials/lidar-lite-v3-hookup-guide/al.
Accessed July 02, 2021.
Mohammad S. Sadri – ZYNQ Training. Accessed 05 February 2021

[6] XILINX INC, Xilinx Zynq-7000 SoC Technical Reference Manual, UG585 (v1.13) April 2, 2021.

[7] XILINX INC, Xilinx SDAccel Development Environment Help for 2019.1 https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/xfopencv-library-api-reference-ycb1504034263746.html.
Accessed March 03, 2021.

[8] ADDISON SEARS-COLLINS, The Ultimate Guide to Real-Time Lane Detection Using OpenCV https://automaticaddison.com/the-ultimate-guide-to-real-time-lane-detection-using-opencv/.
Accessed April 20, 2021.