FPGA accelerated Stewart Platform Stabilization using Xilinx Zynq

Bastian Fuhlenriede*, Nils Stanislawski†
* Participant, † Supervisor
Institute of Microelectronic Systems, Leibniz University Hannover
*bastian.fuhlenriede@stud.uni-hannover.de, †nils.stanislawski@ims.uni-hannover.de

***Abstract*—This work presents an image-processing pipeline that showcases the advantages of custom implementations of image detection systems. Autonomous driving is an application that requires the processing of video streams in real time to identify their surroundings. As such, for a car to be able to respond to its environment, it is crucial to decrease the time between capturing it and making its information available to the control system. The concept's correct function is validated by detecting a ball in an active video stream and using its position to balance it on a Stewart platform. This is done on a PYNQ-Z2 board to show that the basic principles can be implemented with a limited resource budget. We show that this approach can process images significantly faster than comparable solutions.**

***Index Terms*—real-time, image processing, object detection**

I. Introduction

The rise of fields, such as autonomous vehicles, comes with the requirements of real-time image tracking, which is often hard to guarantee for a general-purpose processor. While there are solutions for this with custom hardware extensions, application-specific instruction set processors or specific real-time operating systems, they are all still sequential and often can not use parallelism to its full extent. On the other hand, a solution immediately integrated into the video pipeline guarantees a specific time between updates and allows the fullest extent of parallelism for the given task.

This work proves this through an exemplary implementation of a ball-detection system within the video pipeline of a PYNQ-Z2 [1] board. Furthermore, the real-time effectiveness is demonstrated by using the determined position to balance the ball on a Stewart Platform.

First, the system design will be showcased. Following this, the implementations of the individual filters and modules within our video pipeline will be shown. Afterwards, the pipeline is presented, focusing on including the processing steps. In the end, the system is evaluated by comparing the image processing speeds of our implemented design against similar alternatives.

II. System Design

Implementing the processing of a video stream into its pipeline allows frames to be processed as they are received and removes the overhead of moving them to another processing unit. This is used by our implementation, shown in figure 1. It shows the schematic overview of the system, where the used IP cores are coloured blue, filters yellow, the crop module orange, the detection system purple, and the control logic green. The video stream is recorded by a camera pointing at the Stewart platform. In the programmable logic (PL), it is converted into an AXI4-Stream [2]. The image-tracking system is located after the Video Direct Memory Access (VDMA) and pixel unpack. This is done to allow the control logic to calibrate the system by writing a frame into video memory, which is then processed by the system, giving the initial centre of the frame. Despite knowing the frame size, the centre position is not known as the filters can introduce offsets via their buffers that need to be filled first, which shifts the frame.

The video frames are first processed by smoothing filters, which reduce noise in the image. Four of them can be configured to work in sequence. Multiple filters have been used rather than a singular large one to be able to show the effects of different degrees of smoothing. An edge filter reduces the frames to a binary representation, which only contains object edges. A crop module crops the image to the platform holding the object that has to be detected in the frames. The filtered and cropped frames are routed to the detection system, which extracts the object's position and the video output, to which a monitor is connected showing the processing results.

The control logic for the Stewart platform is implemented in the processing system using Python. It reads the values from the PL via an AXI4 interface. The crop module is also configured via this interface, and the state of the buttons and switches are read from it as well.

III. Filter

In general, all filters follow the same design scheme shown in figure 2. As the filters are 2-dimensional matrices, the implementation is also 2-dimensional, consisting of rows of processing elements. The smallest element of the filter is called a slice, and one exists for each non-zero coefficient in the filter kernel. The zero coefficients are replaced with a register because no processing needs to be done, as multiplication with zero will always return zero.

A slice is built up of a register and a processing element, which depends on the actual filter. N slices are then grouped into a row for a $N \times N$ sized filter. The outputs of the slices are accumulated in the row via a hierarchical pipelined adder. The output of the N rows is then added together via another hierarchical pipelined adder, the result of which serves as the output for the filter operation of the whole filter kernel.

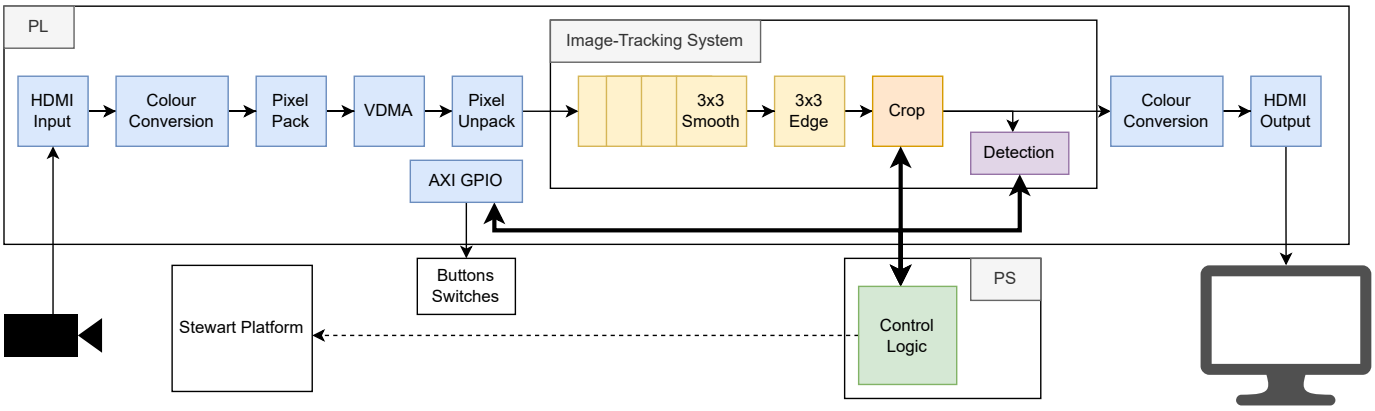


Fig. 1: Diagram showing the system design. The used IP cores are coloured blue, filters are yellow, the crop module orange, the detection system purple, and the processing green. The processing systems (PS), programmable logic (PL), and the integration of the image-tracking systems separation are shown.

The filter operates by moving the filter kernel across the image. This requires buffering N rows of the frame. Each buffer serves as an input stream for a given filter row and is also connected to the buffer in the next row. A buffer is the length of the image, and it is implemented using RAM-based shift registers because implementing it via logic would require a lot of resources for larger frame sizes. The filter kernel is wrapped in a module connecting it to an AXI4-Stream.

The following sections describe the functionalities and purposes of the different implemented filter kernels.

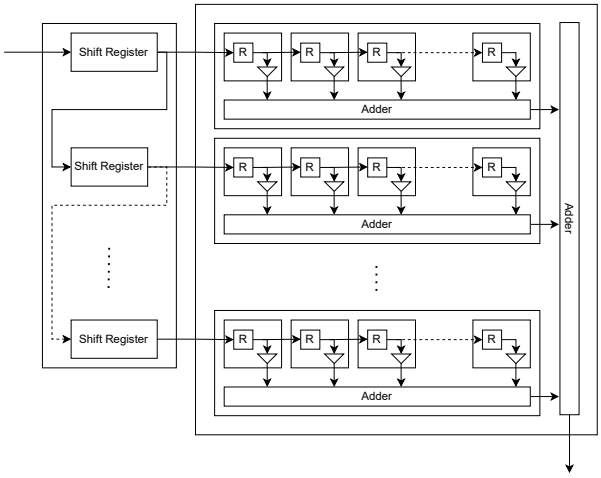


Fig. 2: Diagram of the 2D filter structure.

A. Smoothing Filter

In the first step of the image pipeline the image is smoothed. This acts similar like a low-pass filter and reduces the noise by the camera sensor. Reducing noise in the input image is important, as the edge filter otherwise processes the noise, leading to a dirty output. Also, the edges that should be detected become more prominent as the smoothing effect results in a larger edge detection. The kernel used is based on a Gaussian filter kernel, which is shown in the equation 1. This 3×3 kernel was specifically chosen as all the coefficients can be reduced to a power of two, resulting in an implementation that requires no multiplications but simple shift operations. The filter can be chained multiple times to achieve a more extensive smoothing and low-pass effect, as shown in figure 3.

$$F_s = \frac{1}{16} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{1}$$

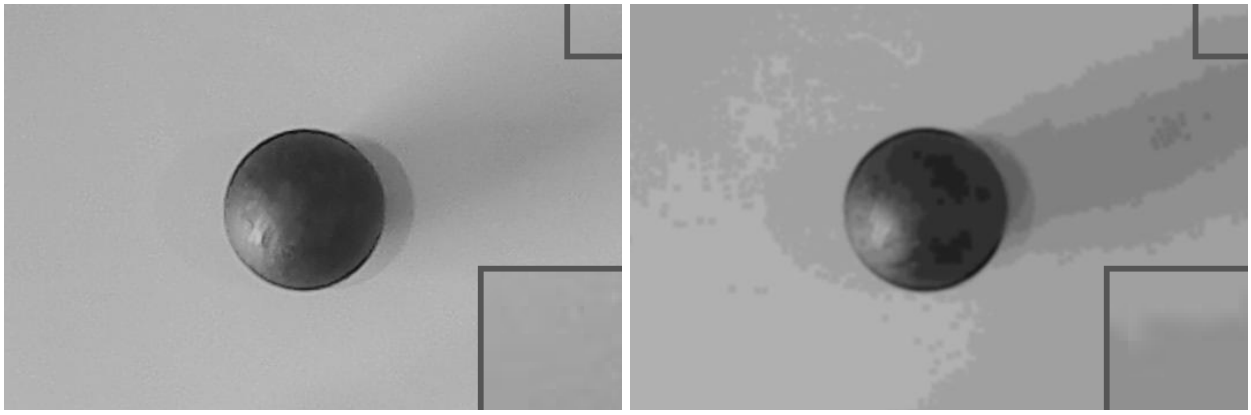
B. Edge Detection

The edge filter is the second step of the image pipeline. It converts the input into an image only containing the edges between high-contrast regions. The Prewitt operator [3] shown in 2 has been used, as it can be implemented by converting the number at the -1 coefficients to their Two's complement representation, saving the multiplication.

$$K_X = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} K_Y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{2}$$

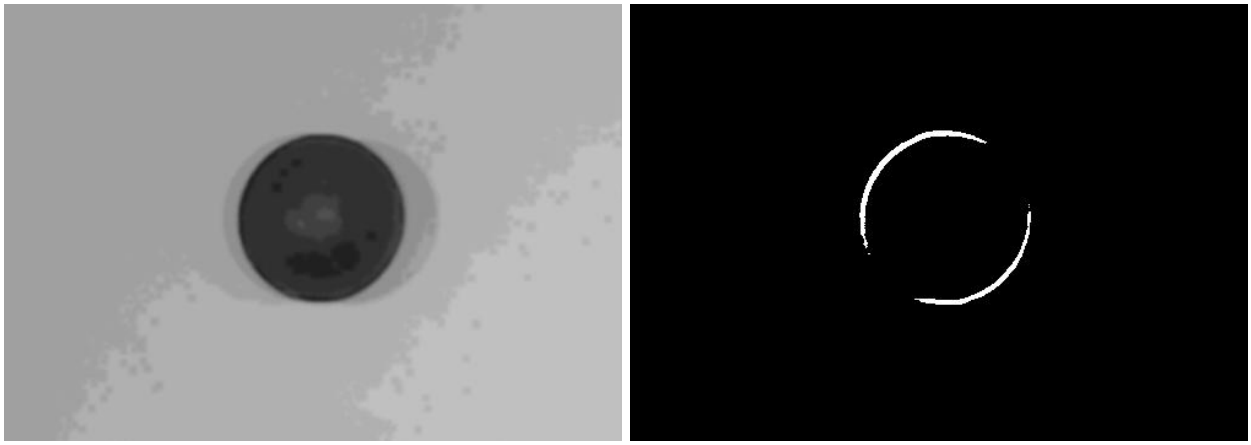
The image is simultaneously filtered along the X and Y axis. This is done by calculating the same filter kernel position in the same filter slice and averaging both results. For example, the slice for the upper right corner would calculate $\frac{c_x \cdot p + c_y \cdot p}{2} = \frac{-p+p}{2}$, were c_x and c_y are the filter coefficients and p is the pixel value. The parallel calculation and early combination prevent redundancies in the design, reducing the hardware required for the filter. The final output of the filter is converted into a binary image by applying a threshold. In this new image every pixel is only represented by a bit. The effect of this filter is shown in figure 4.

s



(a) Initial Image (b) Four applications of the smoothing filter

Fig. 3: The smoothing filter being applied four time to the image a. The upper right corner is displayed enlarged.



(a) Initial Smoothed Image (b) The result of the edge filter

Fig. 4: These images show the input image and the result of the edge filter. The output comes out in binary form. The images are cropped from the originals.

IV. Stream Crop

One additional module that has been implemented is the stream crop module, which can be used to limit the image size in the stream. This is achieved by blanking pixels to a value of zero. This ensures that the video stream can still be displayed. The module limits the frame to include only the area of interest, which, in this case, is the platform with the ball, as the video frames can include other objects, which causes the detection of false edges. The removal of these pixels would have been more complicated and would have used more hardware. The area limits to which the stream should be cropped are specified via four registers, two for the upper and lower bound for the X and Y axis. These registers can be written via a AXI4-Lite [2] interface by the processing system.

V. Position Extraction

The position extraction is the last stage of the pipeline. In this step, the centre position of the ball is calculated from the input image. This can be achieved by accumulating the image in both the X and Y axis, producing two histograms. The histograms are directly stored in dual port block RAMs as the frame is received by the filter. In a second step the centre of the longest line in both histograms is determined. The so calculated pair of points is then returned as the centre of the detected ball. Ideally, this step is performed between two frames, as both read and write pointers are needed for the calculation. However, it is also possible to determine the points while the next frame is already incoming: As the processing takes only the width of the frame in cycles, atmost the first line of the new input image is skipped while the object is expected to be roughly in the centre of the frame. So, the actual position should not be affected by this, meaning no frames need to be skipped, delayed, or more RAM than necessary needs to be used for double buffering.

The final centre point is accessible by the controlling system via an AXI4-Lite interface. For this, the value needs to be transferred from the video clock domain to the AXI clock domain. The transfer is done via a handshake-based clock domain crossing (CDC). The histogram calculation starts once

the CDC signals that the current position has been transferred into the AXI clock domain.

VI. Pipeline Integration

The individual filter modules are integrated into the video pipeline of a PYNQ-Z2 via an AXI4-Stream Switch IP-Core. This allows the dynamic configuration of the filter order and provides the option to disable selected filters. This way, different configurations can be interactively shown.

The video pipeline is separated into three parts: HDMI-in, the frame storage via a Video Direct Memory Access IP core, and the HDMI-out. The HDMI in and out contain the HDMI interfacing, a colour conversion, and a pixel pack/unpack IP. The pixel pack modules are used to store the incoming pixel as 4-byte pairs. The colour conversion allows the conversion of the incoming pixels into different pixel colour formats.

The processing system has been integrated after the HDMI-out sections pixel-unpack and before the colour conversion IP cores. The colour conversion of the HDMI-in section has been configured to convert the input image to grayscale. This reduces the data per pixel from three bytes to one and results in one colour channel that needs to be processed, which simplifies the system and reduces the amount of hardware used.

VII. Evaluation

This section introduces the evaluation platform and then discusses the results of comparing the implemented design against similar systems.

A. Setup

The design was evaluated by detecting the position of a ball, and using it, the ball was balanced on a Stewart platform. This system is shown in figure 5, with labels for each part. The first part is the camera recording the platform for which a Raspberry Pi zero [4] with a camera module [5] was used It is connected to the HDMI input of the PYNQ Z2 board. Some action cameras were also tested, but they were not detected by the design. Presumably, their output is based on a newer HDMI standard than is supported by the used IP core. The second element is the recorded Stewart Platform with a ball place on it's top. Both have a high contrast to each other to improve edge detection. Part three is a PYNQ Z2 board on top of a custom logic board that converts the 3.3 V PWM output to 5 V input for the servos. They are the fourth component and located on a separate base platform. There are six of them, which allows the Stewart platform to be moved in six degrees of freedom. The PYNQs PL contains the implemented design, and its PS runs the platform's control logic. The fifth part is the monitor that is connected to the HDMI output of the PYNQ board and shows the processed video stream. This is used to showcase the effects of the individual filters.

On the PYNQ-Z2 ran a Jupyter Python script, which continuously read the detected ball position from the Pipeline. The position was then low pass filtered to reduce the effect of noisy detections. Lastly, the angles for the platform were

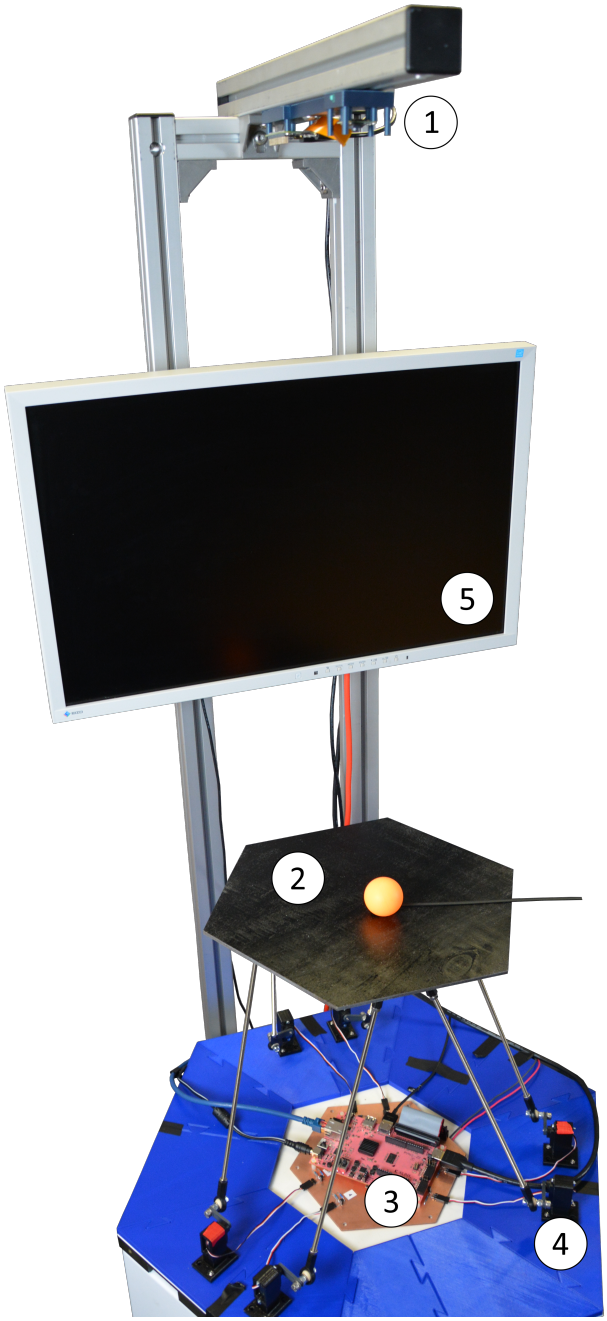


Fig. 5: Evaluation Platform. (1) Raspberry Pi with a camera module, (2) Stewart platform with a ball, (3) PYNQ Z2 board with a custom breakout board, (4) Servos, (5) Monitor.

System	Results
Ryzen 5 2600	0.0589 s
PYNQ-Z2 (PS)	1.329 s
PYNQ-Z2 (PL)	0.0147 s

TABLE I: Processing times between different implementations. Comparable solutions were tested on a Ryzen 5 and the PYNQ-Z2 processing system against the implemented design in the programmable logic.

calculated using a proportional–integral–derivative controller (PID controller) [6] whose input was the filtered value.

B. Results

The implemented design has been compared to one application written in Python using opencv [7]. It has been executed on a machine with an AMD Ryzen 5 2600 running at 3.4 GHz and the PS of the PYNQ-Z2 board. This gives a comparison to a much more powerful system than the PL and a comparable system with the PYNQ-Z2.

The programs process a single frame. It is first processed by applying a smoothing filter followed by an edge filter. The circle is then detected via a circle Hough transform [8]. This transformation is a fast method for detecting circles in a sequential system. The run time was captured and averaged over a 1000 runs. These results can be seen in table I.

The pipeline processing time depends on the input image's actual size. A Full HD (1920x1080) video stream was used for this work. The video pipeline uses a 142.857 132 MHz clock. Assuming a continuous pixel stream, it would take 0.0145 s to receive the image. Assuming four Smooth and one edge detection filters were used, an additional 28 825 cycles are needed to compensate for the buffer and processing delay, which results in 0.0002 s delay. The processing of the histogram would take another 1920 cycles or 0.000 013. The resulting time is around 0.014 713 s per frame. This shows that the processing time is dominated by the time it takes to receive the frame. In conclusion, the proposed design can process every frame faster than the compared systems. Additionally, the time between updates is guaranteed to be the same, while it can vary on the other systems, and that might not be desired depending on the application.

VIII. Conclusion

This work implemented an image processing system into the AXI4-Stream video pipeline of a PYNQ-Z2. For this, efficient filters and a detection system based on determining the longest line in the histograms for the X and Y axes were implemented. The final comparison of the processing times between a Ryzen 5, the PYNQ-Z2 PS, and this pipeline showed that the FPGA solution is around 90 times faster than the alternative on the PS and 4 times faster than the Ryzen 5.

IX. Acknowledgement

We would like to express our deepest gratitude to several individuals whose invaluable assistance made this project possible. Their expertise and support were crucial in various aspects of this work, from technical skills to general guidance.

My heartfelt thanks go to Jakob Marten, Christian Fahnemann, Jonas Hollmann, and Finn Venema. Their collective contributions in 3D modeling and printing, video recording and editing, and general help were instrumental in the success of this project.

Without the dedication and support of these individuals, this project would not have been possible. Thank you all for your invaluable contributions.

References

[1] "Pynq-z2 website," https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html, accessed: 16.09.2023.

[2] "Axi-4 axi documentation," https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide, accessed: 18.09.2023.

[3] J. M. Prewitt *et al.*, "Object enhancement and extraction," *Picture processing and Psychopictorics*, vol. 10, no. 1, pp. 15–19, 1970.

[4] R. P. Ltd, "Buy a Raspberry Pi Zero 2 W." [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-zero-2-w/

[5] "Camera - Raspberry Pi Documentation." [Online]. Available: https://www.raspberrypi.com/documentation/accessories/camera.html

[6] M. A. Johnson and M. H. Moradi, *PID control*. Springer, 2005.

[7] "opencv-python: Wrapper package for OpenCV python bindings." [Online]. Available: https://github.com/opencv/opencv-python

[8] S. J. K. Pedersen, "Circular hough transform," in *Encyclopedia of Biometrics*, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:17799110