

A Novel Real-Time Image Processing Verification Framework Targeted to the Zynq SoC for Drone Localization Applications

By

Alberto Santos-Castro

A Research Paper Submitted

in

Partial Fulfillment

of the

Requirements for the Degree of

MASTER OF SCIENCE

in

Computer Engineering

Approved by:

PROF _____
(Dr. Daniel S. Kaputa, Research Advisor)

PROF _____
(Dr. Marcin Lukowiak, Department Advisor)

DEPARTMENT OF COMPUTER ENGINEERING

KATE GLEASON COLLEGE OF ENGINEERING

ROCHESTER INSTITUTE OF TECHNOLOGY

ROCHESTER, NEW YORK

December 2017

Table of Contents

1. Introduction.....	1
1.1. UAVs for Indoor Applications.....	2
1.2. Challenge: GPS-Denied Navigation.....	2
1.3. Proposed Sensing Solution to GPS-Denied Navigation.....	3
1.4. Proposed Hardware Solution to GPS-Denied Navigation.....	5
2. Real-Time Image Processing Verification Framework Topology.....	8
2.1. Current Work vs. Future Work	10
2.2. Video Processing.....	11
2.3. Required Modules for Video Processing	11
3. Design Methodology.....	12
3.1. FPGA.....	13
3.1.1. FPGA/ μ p Communication	2
3.1.2. FPGA Image Operations.....	17
3.2. Camera	21
3.3. Graphical User Interface	21

3.3.1. Program Operation.....	24
3.1. Connection with the FPGA	25
4. Limitations	25
5. Results.....	26
5.1. Streaming Operations	26
5.2. Complete Frame	28
5.3. Analysis.....	30
5.3.1. Regular Bram Controller:	30
5.3.2. Improved Bram Controller:.....	32
5.4. FPGA Utilization.....	34
6. Literature Review.....	35
7. Conclusions.....	37
7.1. Future Work	38
8. References.....	39
Appendix.....	41

Table of Figures

Figure 1 - Zynq 7000 representation.....	5
Figure 2 - Snickerdoodle block diagram.....	6
Figure 3 - General inter-connections.	9
Figure 4 - FPGA image processing path.....	10
Figure 5 - 3D Imaging technologies comparison [14]......	Error! Bookmark not defined.
Figure 6 - Basic blocks required for localization. Used for x,y and z detection in [13]......	12
Figure 7 - FPGA interconnection between main modules.....	14
Figure 8 - command_and_control FPGA block.....	2
Figure 9 - bram_ctrl FPGA modules. (left original, right improved).....	16
Figure 10 - operation_select constructed FPGA module.	17
Figure 11 - pixel size conversion modules.	18
Figure 12 - Grayscale and Invert, first two image procesing modules in pipeline.	19
Figure 13 - rgb_to_hsv FPGA module.....	19
Figure 14 - Three constructed threshold modules.....	20
Figure 15 - Image operation modules (grayscale, invert, threshold) connected in a pipeline fashion.	20

Figure 16 - GUI operation example.....	22
Figure 17 - Firmware operation example, black and white images.....	23
Figure 18 - GUI flow diagram.	24
Figure 19 - GUI - FPGA communication registers.....	25

List of Graphs

Graph 1 - Comparison between the processor and the FPGA in terms of total time (image loadtime + runtime). Comparison from the image apple.tif (262x240).....	27
Graph 2 - Runtime comparison between processor and FPGA. Image fruits.tif (280x186)	28
Graph 3 - Comparison of image size between processor and FPGA.....	29
Graph 4 - Comparison between sizes.....	30

Nomenclature

AXI:	Advanced eXtensible Interface
BRAM:	Block RAM
CPU:	Central Processing Unit
FPGA:	Field Programmable Gate Array
FPS:	Frames Per Second
GNSS:	Global Navigation Satellite System
GPS:	Global Positioning System
GUI:	Graphical User Interface
IEEE:	Institute of Electrical and Electronics Engineers
IMU:	Inertial Measuring Unit
LUT:	Look Up Tables
MUX:	Multiplexer
RAM:	Random Access Memory
RSSI:	Received Signal Strength Indicator
RGB:	Red, Green, Blue

SoC: System On Chip

UAV: Unmanned Aerial Vehicle

1. Introduction

UAVs (Unmanned Aerial Vehicles) commercially known as drones, are in simple terms mini helicopters that fly without someone on board. Whether designed to drive autonomously or to be remotely controlled, drones need at least four motors (quadrotors) to be capable to fly and maintain stability. IEEE (Institute of Electrical and Electronics Engineers) described drone development as one of the decade's top 11 new technologies [1]. Global drone industry has grown rapidly in the past few years, reaching \$8 billion in 2015, and is expected to reach \$12 billion by 2021. [2] This great evolution in drones is due to the large number of applications that they have; such as recreational use, film and video recording, or even toys for kids and grown-ups. But, they also have more essential applications as technology tools for disaster rescue mission, agriculture, and security. A good example of a drone application would be the disaster recovery operations and humanitarian actions after the Nepal earthquakes and Cyclone Pam in Vanuatu [3], where drones were used to locate the most affected areas after the earthquake, and to deliver supplies to sectors that were not reachable or difficult to access.

To explain the behavior of a drone it is necessary to introduce the IMUs. Inertial Measurement Units (IMUs) are devices containing the basic inertial measuring sensors (accelerometer, gyroscope, magnetometers, etc.) to keep drone stability during flight. These measuring devices are commonly used because of their accuracy along with their inexpensive cost; however, they also have limitations. The main problem is that the sensors inside an IMU are imprecise due to noise and calibration errors. The IMU processor time-integrates these noise and calibration errors, which accumulate over time producing a drift effect (drone position is shifted over time) [4, 5]. When connected to drones, the constant vibration from the motors increases even more reading errors in the IMU causing the drift to increase more [6-8].

To compensate for the drift, drones need to obtain position, velocity and attitude from external measurements [8]. In other words, IMUs are integrated with a GPS (GNSS or a similar positioning sensor), that provides them with an objective position/location; making the drone localization highly dependent on GPS.

1.1. UAVs for Indoor Applications

Most of the current drone development is geared towards outdoor environments (adding to previous examples we include agriculture, 3-D building mapping, geographic mapping, etc.) [2]. UAVs obtain the location information from satellite navigation systems like the US Global Positioning System (GPS) or the Russian Global Navigation Satellite System (Glonass) [3]. Global navigation sensors allow inexpensive hand-held receivers to determine earth-relative position to within a few meters; but they only work in outdoor environments with sufficient sky visibility, and they are not precise around large building structures, forests, valleys, etc. [9]. This becomes a problem in conditions when the GPS signal is not available; examples include indoors, underground, underwater, and obstructed outdoor environments. Limiting drone usage to GPS-accessed areas also limits the range of applications where drones can be used, restraining users and developers to explore the full potential of drones. To mention a few, possible indoor applications include automated inventory management, construction reporting, personal home assistance, in-building delivery and search/rescue operations. Thus, it is important to find new ways to control drones that do not require data from global positioning navigation systems.

1.2. Challenge: GPS-Denied Navigation

Designing a reliable GPS-independent localization method for indoor environments has become a technical challenge. Some approaches are as simple as using optical sensors as positioning trackers, other

approaches are more complex and require to know the physical location of access points and routers to map an estimate of where the drone is currently located, based on the strength of the signals coming from these Wi-Fi devices. Computer Vision seems to be the best localization method for our purposes, it is affordable for low-budget drones, independent from external sensors and hardware and easy to understand. However, an affordable drone based on a small processor (like the Raspberry Pi), would not be able to analyze the necessary frames per seconds due to the lack of on-board resources. For this reason, some strategies opt to use external sources to analyze the video from the drone and send it back, however the problem with this approach is that there is a delay caused by the video transmission. We propose an improvement of this approach, by using the FPGA advantages of on-board processing and parallel processing to achieve greater FPS (frames per second) and thus estimate the drone position. With all intermediate values in local line-buffer storage, eliminating unnecessary communication with off-chip DRAM [10]. To test this theory, we will focus on the creation of the infrastructure necessary to test FPGA image processing algorithms, construct basic FPGA image processing algorithms, and estimate the resources needed to successfully implement the camera-controlled FPGA drones. The results from this research will be used to construct drones that improve their location and stability based mainly on camera inputs.

1.3. Proposed Sensing Solution to GPS-Denied Navigation

Machine Vision is the ability of a computer system to extract visual data from the environment. A computer vision system needs at least three basic components to function efficiently, the sensor, a processor, and an analysis unit. After picking the right sensing strategy, the processing and analysis can be done with the FPGAs. It is necessary to find the sensing technology that will provide the view

information. There are multiple technologies and sensing devices in the machine vision field, we present a few of these technologies next, also discuss the advantages of our picked strategy:

Structured-Light is the process of projecting a known pattern onto a surface (this pattern is usually grids or horizontal bars). The basic theory behind this method is that when a light is projected in a surface (using a laser or a projector) it produces a pattern that appears distorted, the distorted pattern then is used to recreate the shape of the object that it is being reflected to.

CONSIDERATIONS	STEREO VISION	STRUCTURED-LIGHT	TIME-OF-FLIGHT (TOF)
Software Complexity	High	Medium	Low
Material Cost	Low	High	Medium
Compactness	Low	High	Low
Response Time	Medium	Slow	Fast
Depth Accuracy	Low	High	Medium
Low-Light Performance	Weak	Good	Good
Bright-Light Performance	Good	Weak	Good
Power Consumption	Low	Medium	Scalable
Range	Limited	Scalable	Scalable
APPLICATIONS			
Game		X	X
3D Movies	X		
3D Scanning		X	X
User Interface Control			X
Augmented Reality	X		X

Figure 1 - 3D imaging technologies comparison [11].

Time-of-Flight works by illuminating the scene with a modulated light source (usually a solid-state laser or a LED operating in the near-infrared range) and observing the reflected light; then the phase shift between the illumination and the reflection is measured and translated to distance. When the light is transmitted an imaging sensor receives the light and converts the photonic energy to electrical current [11].

Stereo Vision is the idea of using two 2-D cameras to obtain extra information from the images, such as the depth of a specific object. Stereo Vision is possible due to Triangulation, which is the process of determining the location of a third point, based on the information from the first two points. As can be seen in Figure 1, stereo vision has the most compact technology and the cheapest one, not to mention also low power consumption compared to the other two technologies discussed.

1.4. Proposed Hardware Solution to GPS-Denied Navigation

Our proposal includes an FPGA-based image processing structure, instead of a processor-based model; the FPGAs allow additional pixel operations to be performed while the pixels are being read, as it is not necessary to load the pixel in order to perform the operations. Using FPGAs will decrease the overall pixel operation time, next we explain our FPGA selection choice:

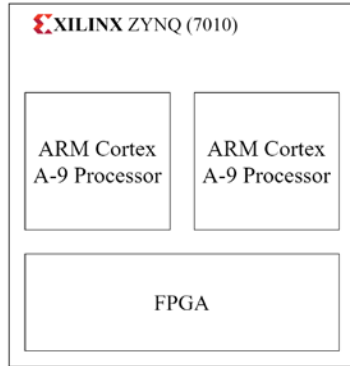


Figure 2 - Zynq 7000 representation

The Zynq 7000 All Programmable SoC family contains a multicore Cortex-A9 system with programmable logic in the same chip. It integrates the software programmability of an ARM-based processor and the hardware programmability of an FPGA [12]. The processor in the chip is a Dual-Core ARM Cortex-A9 that runs at 667MHz in the Z-7010 model and 866 MHz in the Z-7020. This SoC also

comes with an Artix-7 FPGA that has a total of 430K gates/17.6K LUTs for the Z-7010 model and 1.3M gates/53.2K LUTs for the Z-7020 model. Making this SoC chip sufficient to handle the Real-Time Image Processing Verification Framework processing operations.

There are several development boards constructed with Zynq SoCs (examples include microzed and picozed), but the most convenient for our purposes is the Snickerdoodle. Described by its creators as “A microprocessor you can customize”. Snickerdoodle is a development board targeted to handle high definition image processing and video and is ideal for GPS-denied applications. Besides its image processing capabilities, we chose this board due to its advantages over the other Zynq-based boards. It is small and light weight, which is perfect for UAV applications, for customizations it has high I/O, an affordable price, and finally it comes with Bluetooth and Wi-Fi connections.

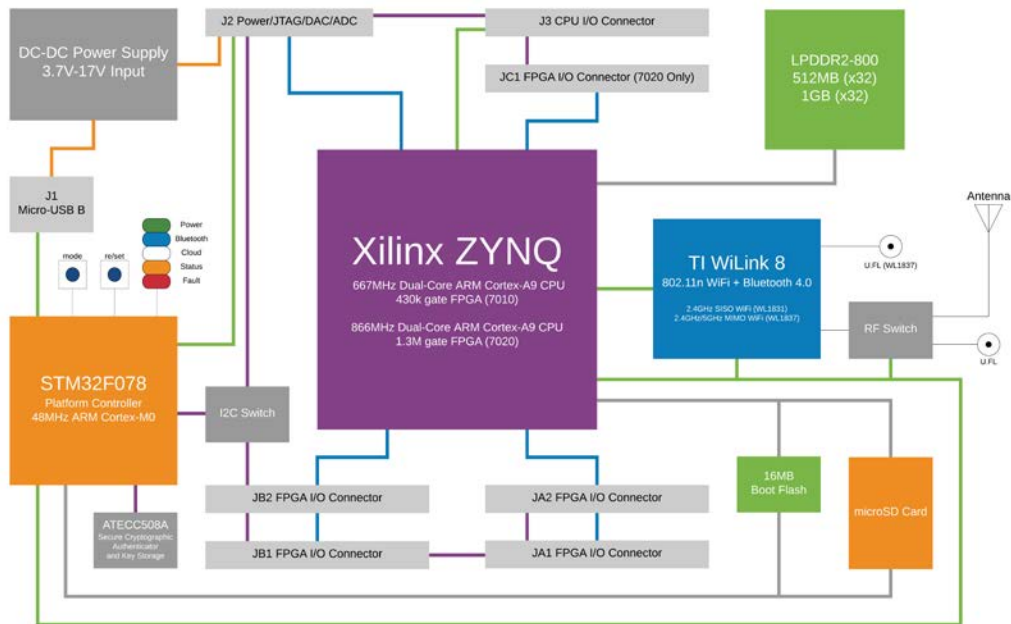


Figure 3 - Snickerdoodle block diagram

The snickerdoodle Wi-Fi capability allows one to remotely connect to a graphical interface (using tools like Windows Remote Desktop Connection or Putty with Xming) in order to test the Real-Time Image Processing Verification Framework. The Wi-Fi connectivity and remote connection to the snickerdoodle provides an easy way to visualize the framework, manipulate images and graphically compare the real-time image processing operations. This remote access tool gives the snickerdoodle an advantage against similar development boards. Figure 2 represents a block diagram of the main components of the Snickerdoodle board that was used.

There are two different Snickerdoodle boards that can be used for

- Model 1 (Z-7010): Model used in this project. 2.1Mb (60 x 36Kb Blocks) of Dedicated BRAM (Block RAM). This BRAM would theoretically be capable of storing a full frame of 320x240 16-bit pixels, one single camera (320x240 times 16 bits per pixel). Although its maximum image is actually 320x200 (more on this in the limitations section). This model also comes with a total of 17.6k LUTs which is sufficient for the image processing mathematical operations.
- Model 2 (Z-7020): For future expansion of the project and incorporation into the flying drone module. Capable of storing a total of 3.3Mb of Dedicated Block RAM. Allowing one to store two full frames of 320x240 16-bit pixels, two cameras. Also expanding the LUTs to 53k.

Snickerdoodle General Specifications:

Chipset: Xilinx Zynq – 7010/7020

Clock Speed: 50MHz

FPGA: 430K gates/17.6K LUTs (or 1.3M/53.2K)

Flash: 16MB NOR + microSD

Wi-Fi: 2.4GHz 802.11n SISO (or 5GHz 2x2 MIMO)

Bluetooth: Classic & BLE

Power input: 3.7V to 17V

Dimensions: 2in x 3.5in (50.8mm x 88.9mm)

2. Real-Time Image Processing Verification Framework Topology

The project can be divided in two different parts. On one side, we have our FPGA construction of the basic image and video processing algorithms; these algorithms perform the operations to the image saved in the BRAM. On the other side, a processor running the friendly Graphical User Interface (Python based) that will perform the same image operations as the FPGA, and then analyze and compare the result of these operations. Everything will be run on the snickerdoodle and accessed via remote desktop from a computer.

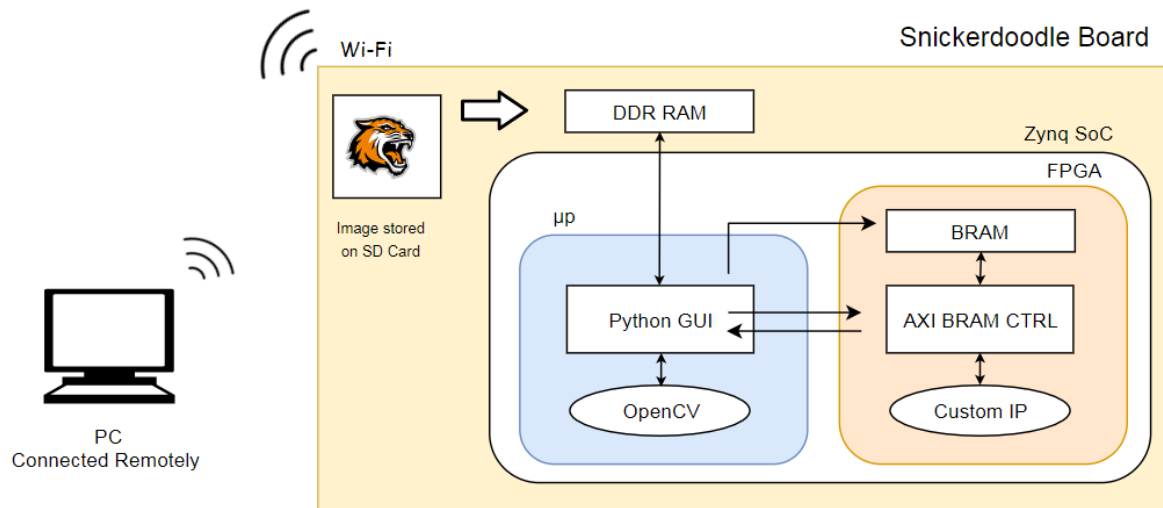


Figure 4 - General inter-connections.

With the application running on the snickerdoodle the user is able to locally browse for an image (stored in the Snickerdoodle SD card) which will then be loaded to the FPGA BRAM. At the same time, the user can have the image loaded into the DDR RAM to have the snickerdoodle processor perform the same operations. The user would then proceed to select from a list the image processing operations (grayscale, invert, threshold RGB, threshold HSV) that will be performed simultaneously on both sides (processor and FPGA). On the FPGA side, once the user presses the button, the FPGA IP bram_ctrl it will start sending the pixels simulating a camera and performing operations via the pipeline. The result of these operations is saved back into BRAM and read by the firmware. On the processor side, the same image processing blocks were constructed using OpenCV (open source computer vision tools). Once both sides are complete, the program compares the time it takes to perform the operation in both sides. Also, when the images are completed the program proceeds to realize a pixel-by-pixel comparison of both resulting images. The result from this comparison is represented as a difference percentage. The images created with the processor are compared to the FPGA in terms of quality and processing time; and the results are stored and displayed to the user.

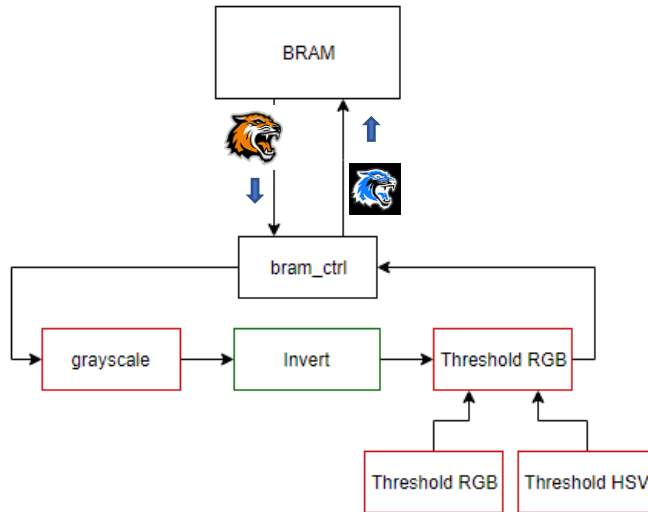


Figure 5 - FPGA image processing path

2.1. Current Work vs. Future Work

The Image Processing Verification Framework is designed to load and compare one image at the time; this is because the image has to first be loaded into BRAM before performing the operations. A future optimization for this implementation is to send the pixels directly through the FPGA image processing pipeline from the camera without the need of using the BRAM storage. This implementation could emphasize the advantages of the pixel streaming operations of the FPGA over the processor. The goal is to have a firmware qualified for video processing working on the pixels as they go through the pipeline.

2.2. Video Processing

Video processing is the base step for UAV localization in our proposal. Some of the main approaches to control a drone's indoor applications include triangulation, trilateration, time of arrival, time difference of arrival, and round-trip time of flight [13]. However, before working with video it is necessary to verify the algorithms can be performed on the frames individually. The most essential requirement for video analysis for localization is based on the capacity of performing image operations. Image processing can be defined as the collection of algorithms necessary to obtain information from an image. Many of the image and video processing systems that are used nowadays are based on specialized tools like OpenCV. There are also software tools specialized in the construction of FPGA and ASIC image/video processing algorithms. Some examples of these tools are the MATLAB Vision HDL Toolbox, the Darkroom high level language, and Rigel. These tools are geared to translate a code from a familiar language to a more specialized set of algorithms for image processing, making it easy for engineers with no background in computer vision to construct very specialized tools. Mah et al. [14] provided an example that demonstrated the capacity of a drone to detect its location from the input cameras; using the MATLAB image processing tools. In the next section we will compare the modules that were used in his approach to the constructed FPGA image processing IPs.

2.3. Required Modules for Video Processing

Mah [14] demonstrated a successful localization technique based on the detection of a drone's position relative to a color-defined object (red ball). With his implementation, he was able to determine a drone's distance in the X, Y, and Z coordinates. Taking his model as a reference point, we can estimate the basic FPGA blocks required for self-localization.

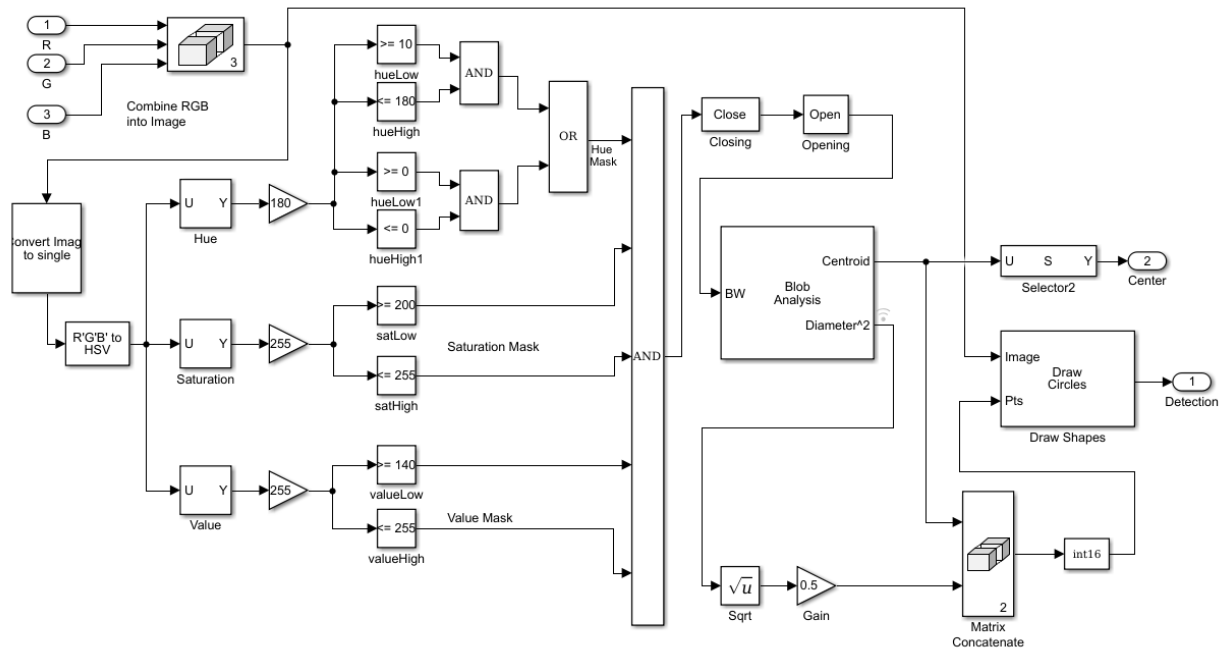


Figure 6 - Basic blocks required for localization. Used for x,y and z detection in [14].

From the basic image processing blocks represented in Figure 6, the Image Processing Verification Framework has the three basic algorithms for object detection. The *RGB555_to_RGB888* IP block to change the color format to a more precise one. *RGB_to_HSV* conversion, since the HSV color format has the colors separated by degrees (Hue: red 0°, green 120°. blue 240°) provides for an easy way to detect the desired color. The last module is the HSV based thresholding. From Mah [14] block diagram we did not include *Close_Image*, *Open_Image*, *Blop_Analysis*, and drawing shapes. These modules are beyond the scope of the Image Processing Verification Framework.

3. Design Methodology

Before the construction of the framework it was necessary to develop a compilation of VHDL programs geared to simulate the complete operation of the camera (OV5640). The main purpose of these modules was to read a real image from the local drive (simulated as an image from the camera), send the pixels with the camera

synchronization protocol, perform operations on the images, and save them back to the drive. Image_sync was a simulator of the camera by generating the video synchronization signals (href, hsync, vsync and pclk) and outputting the 16-bit pixel data. Image_handler is a package containing the VHDL code to read an uncompressed TIFF image file and convert it into an array of 24-bit pixels. Also, this package is capable of performing some image processing operations and save the resulting image as an uncompressed TIFF file. Combining both packages, image_handler as the camera sensing the image and image_sync as the camera sending the data to the FPGA. This pre-design phase involved a total of 8 modules including image_sync, bram_invert, image_handler, image_invert, image_sync, rgb_to_hsv and threshold. All modules were constructed, tested, and simulated using the ModelSim software. The next stage of the process is to construct the actual FPGA IP blocks that would be generated as a bitstream to be loaded into the Snickerdoodle.

3.1. FPGA

The Figure 7 below represents the basic data path of the pixels that are stored in the BRAM. The microprocessor stores the image directly into BRAM; the communication between the microprocessor and the FPGA modules is possible via the AXI_bram_controller. This block indicates the image properties, the operations selected, and when to start the operation. Bram_control is the middle point between the image stored in BRAM and the pipeline of operations that will be performed to the image. It is important to note that two different bram_ctrl blocks were constructed. First, the original bram_ctrl that behaves like the camera by sending only one pixel at the time through the pipeline. Second, a slight improvement named bram_ctrl_plus. This module is capable of processing 2x the pixels from the original bram_ctrl by taking advantage of the parallel processing characteristic of the FPGA. Two pixels are read from the BRAM and run through simultaneously through two different data paths. Because of the construction of the two similar bram_ctrl IPs two different Vivado Firmware were constructed, one with each bram_ctrl. Figure 7 shows a representation of the original bram_ctrl.

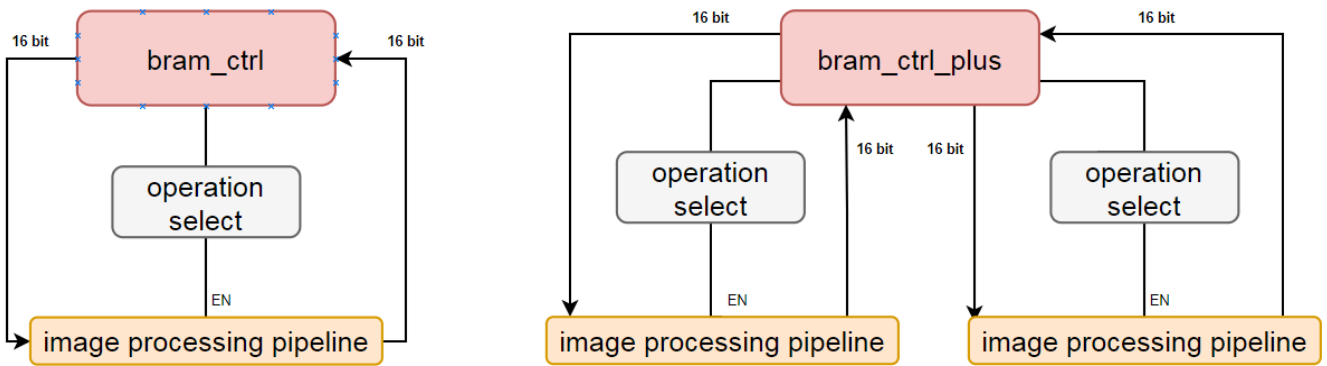


Figure 7- *bram_ctrl* (left) vs improved version *bram_ctrl_plus* (right).

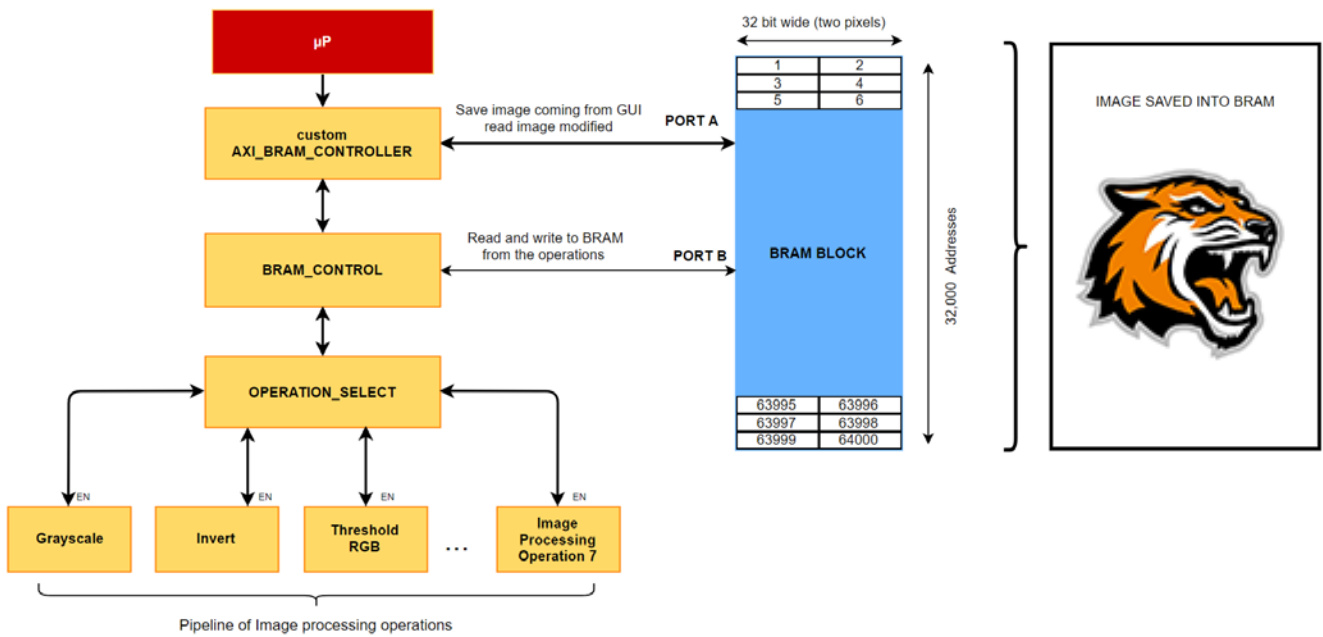


Figure 8 - *FPGA interconnection between main modules*

Including the processor, a built-in invert IP and other blocks, the completed design has a total of 12 IP modules for the regular version, and 27 modules for the improved version. The main constructed modules will be explained in the next section, these constructed IP modules are:

- Bram_ctrl
- Bram_ctrl_plus
- Bram_invert
- Command_and_control
- Operation_select
- Rgb_to_grayscale
- Rgb_to_hsv
- Rgb565_to_rgb888
- Rgb888_to_rgb565
- Threshold_enable
- Threshold_rgb
- Threshold_hsv

3.1.1. FPGA/μp Communication

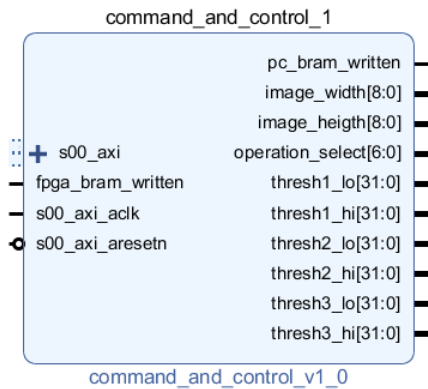


Figure 9 - Command_and_control FPGA block.

Command_and_control is a customized AXI-lite Slave peripheral IP that interfaces with the Zynq processing system providing direct read/write access to the registers. This custom IP connects its input/output pins to specific registers giving our programmable logic direct access to read and write from the registers. This block was constructed to allow the communication between the processor and our

constructed programmable logic blocks. The *pc_bram_written* is a self-clearing bit that indicates to our FPGA modules that a synchronization (start-of-frame-analysis) has been requested by the micro-processor. The same way *fpga_bram_written* concludes the communication by indicating when the FPGA blocks are done, and the operation/s requested have been completed. The rest of the customized output pins indicate the nature of the operation that will be performed, always starting at address 0x00. *image_width*, *image_height*, *operation_select* indicate the image size and the operations; in other words, until what *bram_address* they will go and what bit-computation (grayscale, invert, threshold) they will perform along these addresses. The threshold bits indicate the high and low boundaries for the types of threshold that will be performed (*red_low-red_high*, *hue_low-hue_high*, etc.).

The *bram_ctrl* module is the next module and it is directly connected to the Block RAM where the image is being stored. This module starts the operation when it receives the synchronization request coming from the *command_and_control*. Once the synchronization has started the *bram_ctrl* reads the pixels from BRAM and supplies them through the operation pipeline *pipe_data_out*. The pixels come back from the pipeline already converted *pipe_data_in* and they are stored back into BRAM.

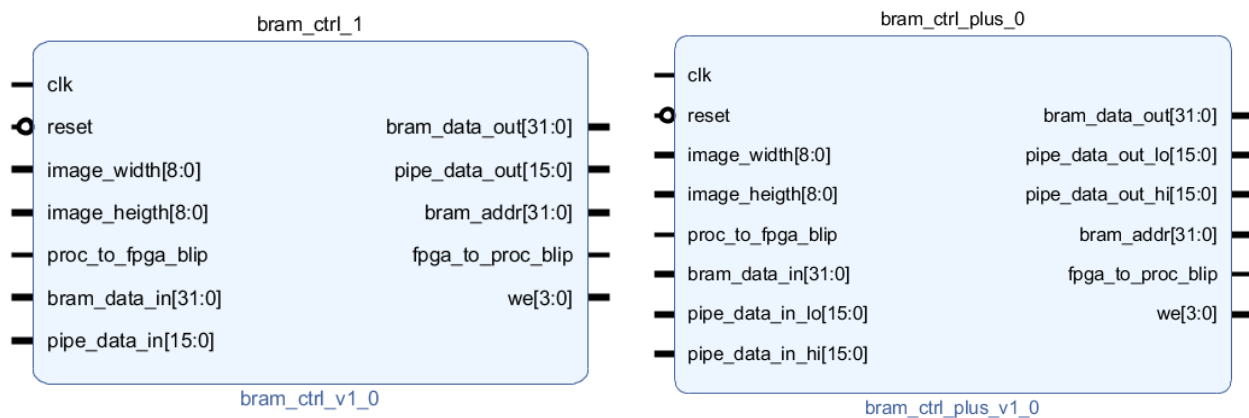


Figure 10 - Bram_ctrl FPGA modules. (left original, right improved)

The image on the left represents the original *bram_ctrl*. This block simulates a real camera supplying only one 16-bit pixel at the time and it is mainly designed for streaming operations. The image on the right *bram_ctrl_plus* is a slightly modified version of the *bram_ctrl* and was constructed to obtain 2x the frame performance. Taking advantage of the parallel programming qualities of FPGAs this block reads one complete BRAM address (32-bit) at a time and sends two pixels to run the same operation simultaneously. The 32 bits from BRAM are divided into two 16-bit (one lo and one hi) and sent through the pipeline in one cycle, the coming bits are concatenated and stored back into BRAM.

3.1.2. FPGA Image Operations

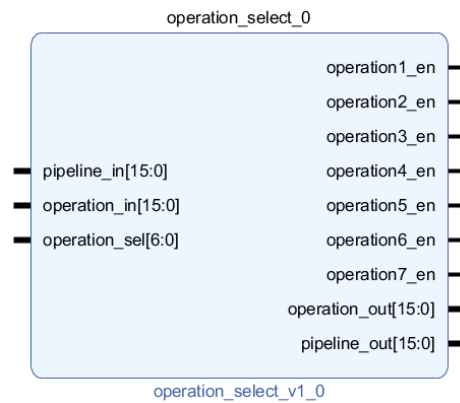


Figure 11 - Operation_select constructed FPGA module.

Operation_select simply controls the enable/disable bit from the operation modules; it selects individually what image processing operations are active depending on the data coming from command_and_control. The data is obtained through the *operation_sel* pins.

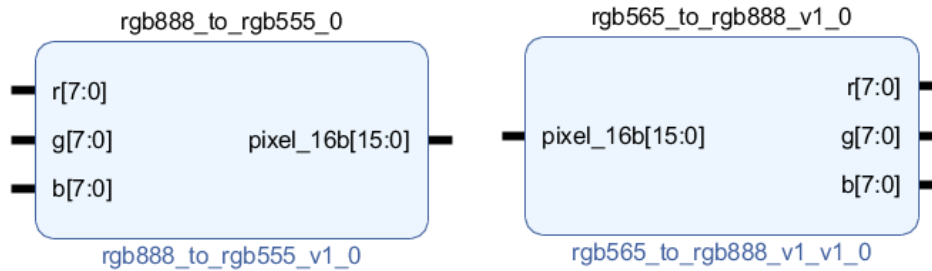


Figure 12 - Pixel size conversion modules.

All the pixel operations are performed using 24-bit pixels; for this reason the 16-bit pixels coming from BRAM have to be converted to 24-bit and then converted back to 16-bit before storing them back into BRAM. This will give us the most precise calculation and is the reason why the operations between the FPGA and the processor have a 100% match. However these modules also increase the resources needed in the FPGA since all the computations after the conversion is done with 24-bit.

The first image processing operation is the convert to grayscale. It simply takes the RGB data and converts it to its similar Grayscale value ($0.3R + 0.59G + 0.11B$). In most image processing tools, a 24-bit RGB would be converted into a single 8-bit grayscale containing the addition between the three colors. However, since the FPGA pipeline was constructed to work similarly regardless of the operations that are active, we output the same calculated RGB value in the three 8-bit outputs (ro, go, bo). This way if the module is not enabled, the output would be the regular red, green and blue values. The second module *bram_invert* simply realizes a bit-wise inversion of the pixels.

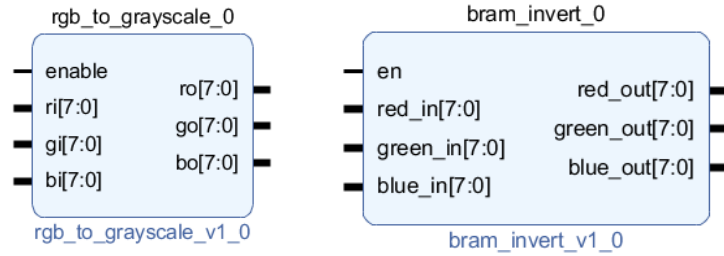


Figure 13 - Grayscale and invert, first two image processing modules in pipeline.

The *rgb_to_hsv* converter that takes the RGB modules and outputs a different color format HSV. This format makes it easier for color detection and using it in collaboration with the threshold block it can easily identify objects with regard to their color.

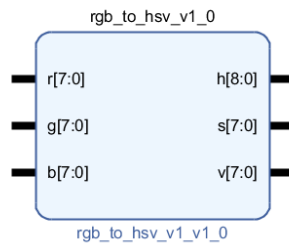


Figure 14 - Rgb_to_hsv FPGA module.

There are three modules geared to threshold operations in our customized blocks. First a simple RGB threshold that does not require additional blocks. This block simply takes the pixel colors from the previous modules and the threshold boundaries coming from *command_and_control* and applies the threshold. The output is a single bit indicating whether the values are in range or not. The second threshold block works the same way as the first block, but using a different color format. Finally the *threshold_enable* block that, depending on the information coming from *command_and_control*, this block will pick a threshold output. Since threshold are pure black and white images, each pixel should be represented with only one bit (1-white, 0-black); however, (just like the grayscale module) the FPGA

output size will not change regardless of the number of operations selected. Thus our threshold output still contains the 24-bit pixels. A future improvement of this modules could include an extra-bit representing whether the image is color or black and white which can significantly reduce the operation time; however, since this project was geared to work with a camera in a color environment it is unlikely to have a pure black and white image for our purposes.

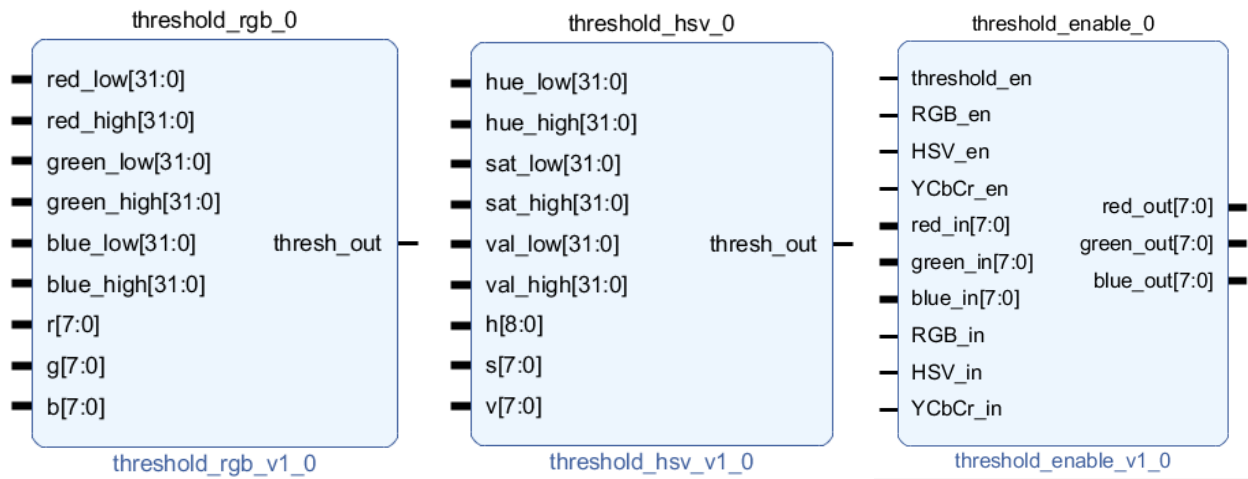


Figure 15 - Three constructed threshold modules.

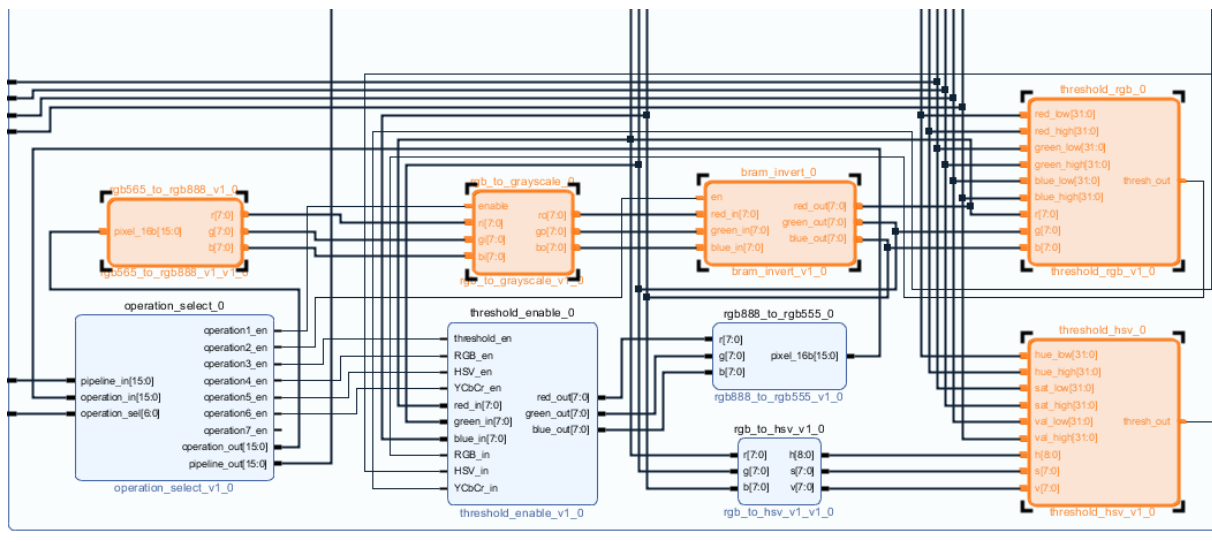


Figure 16 - Image operation modules (grayscale, invert and threshold) connected in a pipeline fashion.

3.2. Camera

Even though in this project we do not use real inputs from a digital camera, the motivation was to make it able to synchronize with the OmniVision (OV5640) camera. The modules in this project have been created to work in collaboration with the VGA input provided with 16 bits per pixel (RGB565) at 90 FPS. All the modules in this project follow the 16-bit RGB convention and were successfully tested with the ModelSim tools.

3.3. Graphical User Interface

The created interface gives one the capability of real-time testing the implemented algorithm running on the FPGA. It was created with the objective to compare the FPGA created image processing algorithms with an algorithm created in OpenCV (Open Source Computer Vision Library). In simple terms it provides an easy way to connect real-time with the FPGA, load an image to the BRAM, and realize different image processing operations. For the construction of the GUI program we used the utilities of PyQt for its user-friendly capabilities.

As mentioned before, OpenCV is an image processing tool with python binding that has proven to be capable to perform location analysis operations, object detection, line tracking, motion detection, and even face recognition. We use this image processing tool as reference to compare the images processed by the FPGA modules. The program loads an image to the snickerdoodle accessing the BRAM memory located at 0x43C00000 using the mmap (memory map) library. This library allows a direct connection between registers and the program. The program was constructed for multiple platforms and detects when it is running on a different OS like windows. However only on the snickerdoodle it can access the registers.

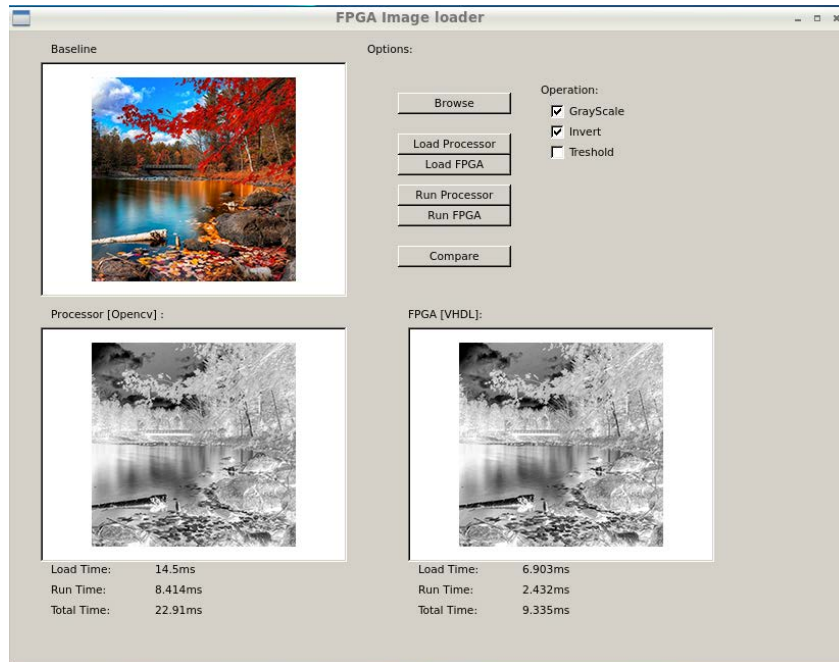


Figure 17 - GUI operation example.

The Figure 16 and 17 show an output (of the program operation) that was taken by remotely connecting to the snickerdoodle via Remote Desktop Connection. It is important to mention that even though a simulation of the program can be run on windows all the outputs are taken from the actual implementation running on the snickerdoodle. The software starts by allowing one to browse the image one would like to perform the operations upon (baseline, initial image located on the top left). Once the image is selected one can load it into the processor and into the FPGA separately, to study their results individually. The next step would be to select the basic image-processing operations that will be performed; the firmware allows for multiple operations to be done at once. The operations on the processor are done using OpenCv; once the button *run* is pressed the program calculates the individual time loading the image, time running the images, and the total time. The last step is to perform a comparison of the two resulting images (FPGA and OpenCV) in terms of quality. The compare tool runs pixel by pixel printing the error percentage between both images.

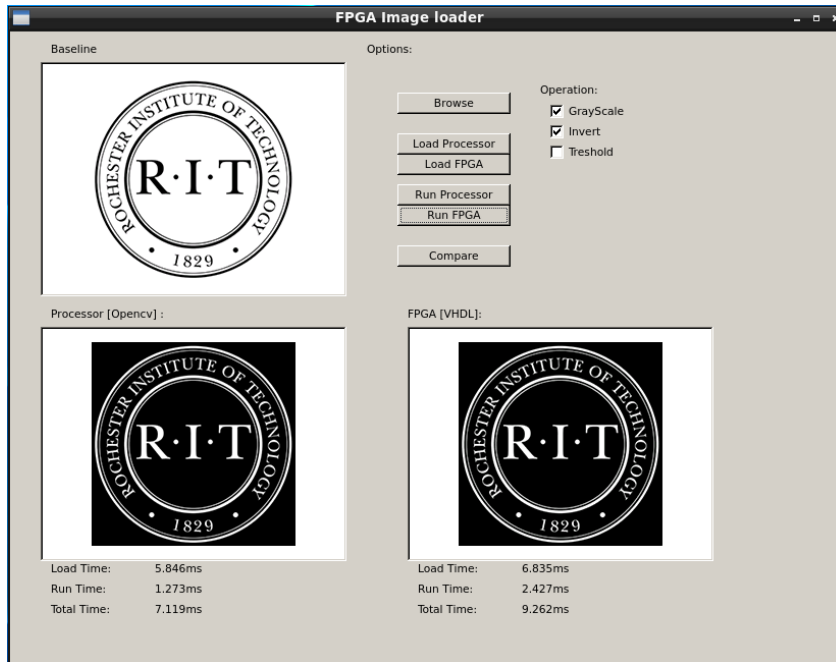


Figure 18 - Firmware operation example, black and white images.

Figure 18 shows an example of how the time taken in the software is very dependent on the characteristics of the image. For instance, black and white images take less time in the processor because they are smaller (one bit per pixel instead of 24 bits per pixels), unlike the FPGA measurements that are always constant.

3.3.1. Program Operation

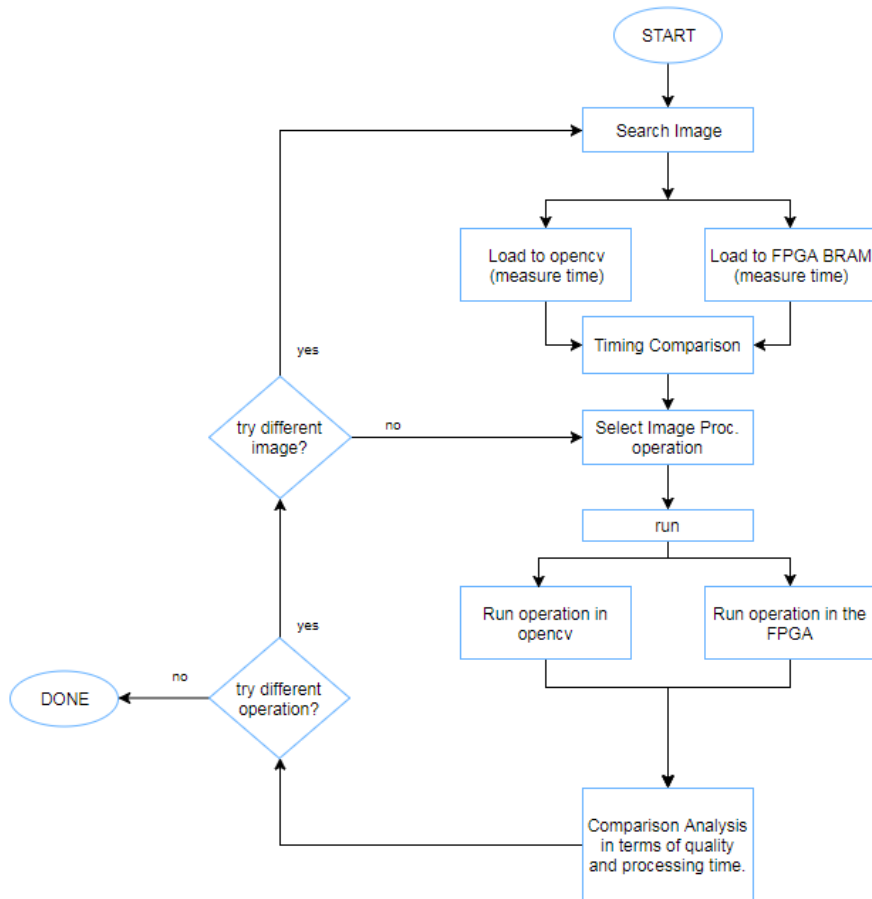


Figure 19 - GUI flow diagram.

Figures 17 and 18 showed a regular operation of the GUI program. The images are in the snickerdoodle Linux directories and loaded into the program. From here each step takes two paths, one running in python using the image processing tools and the other one on the FPGA with the constructed image processing modules. In each step, the program calculates the image loading time and then performs the user-selected operation/s. For a visual representation of the program behavior we can see the GUI flow diagram on Figure 19.

3.1. Connection with the FPGA

The communication between the constructed python program and the BRAM is done through a customized AXI Block (on the FPGA side) and with the *mmap* (memory map, function of the python based program) on the GUI. The next blocks represent how the communication is made.

Base Address + 0				
9 bits (31 - 23)	9 bits (22 - 14)	8 bits (13 - 7)	6 bits (6 - 1)	1 bit (0-self clear)
width	height	unused bits	operation select	synchronization

Base Address + 1	
30 bits (31 - 1)	1 bit
Info from FPGA to PC. Currently unused	FPGA done bit

Base Address + from 2 to 7
threshold boundaries information: hue_low, hue_high, sat_low, sat_high, val_low, val_high red_min, red_max, green_min, green_max, blue_min, blue_max

Figure 20 - GUI - FPGA communication registers.

4. Limitations

The Snickerdoodle board that was used for the project has a Zynq-7000 XC7Z010, which comes with a dedicated block RAM of 2.1Mb. However, the range of the BRAM size can only have 2^n increments starting at 4KB (8KB, 16KB, 32KB, 64KB, 128KB, 256KB and so on). A 320x200 image (maximum tested size) has a total of 128KB. 320x200 pixels times 2 bytes per pixel equals 128KB. Including the custom AXI both represent a 53% of the dedicated BRAM space. Given that the next available range is 256KB, over the 47% that is left it is not possible to upgrade to a 320x240 image. Also, the project does

not include more advanced image processing techniques (that involved two cameras) geared to localization, such as triangulation, this is beyond the scope of the project.

5. Results

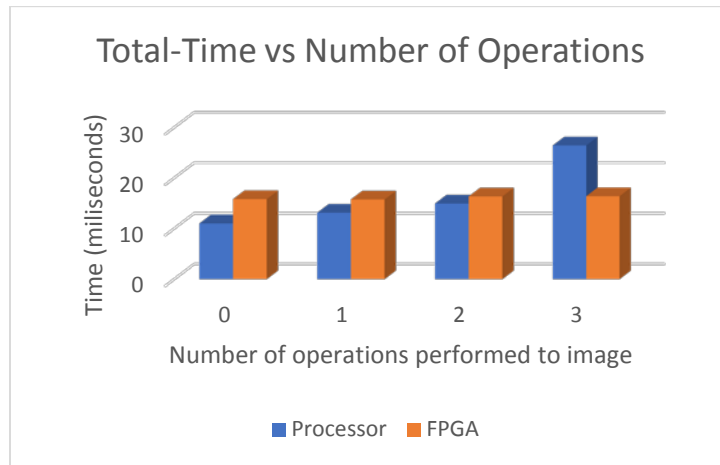
For the comparison, we averaged the results from 150 samples of 43 different images (different sizes and color) and compared these results between the FPGA and the processor. For the size comparison we analyzed the same image stored in 9 different sizes (1x1, 5x5, 10x10, 20x20, 50x50, 100x100, 150x150, 200x200 and 250x250). Example image outputs can be seen in the Appendix section after the references.

5.1. Streaming Operations

Even though the timing measurements were only taken on the whole frame and not pixels individually, all the processing operations were performed as the pixel entered the FPGA from the BRAM. The `bram_ctrl` module supplied pixel by pixel data and this data was analyzed as the pixel entered the pipeline. All of the image processing operations were computed in a stream-wise fashion. The fact that the images presented a 100% match between the processor and the FPGA demonstrates that the FPGA was successful at performing these streaming operations.

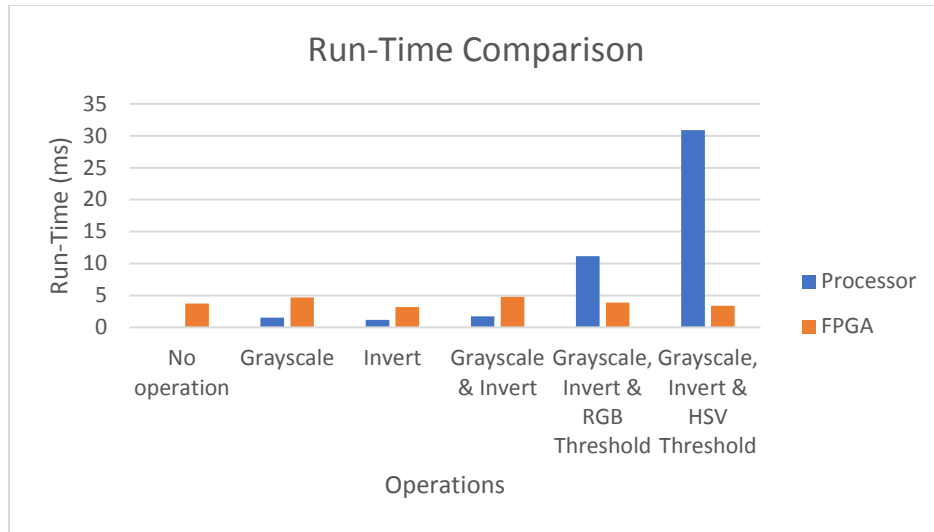
It is also important to note that the runtime measurements of the processor lacked consistency. An image could initially take 10ms to process in one run and for the second measurement increase its runtime by over five times the value of the first measurement. The reason is that the processor operations are affected by background parallel processes that run simultaneously on the processor. The variations in

reading from the FPGA however, were not as noticeable and the small range was due to the cyclic executive variations in the read-time of the BRAM data and the nature of the loop.



Graph 1 - Comparison between the processor and the FPGA in terms of total time. Image: apple.tif (262x240).

In graphs 1 and 2 we can see how the processor is greatly affected by the amount of operations performed consecutively to one image. The more operations performed, the greater total computation time on the processor. Conversely, the FPGA does not change its runtime regardless of the number of operations, matching our initial hypothesis. The pixel data stream will flow through all of the FPGA operations regardless if a specific operation is going to be performed or not. The various enable bits sent down from the processor control whether an operation is turned on or turned off.

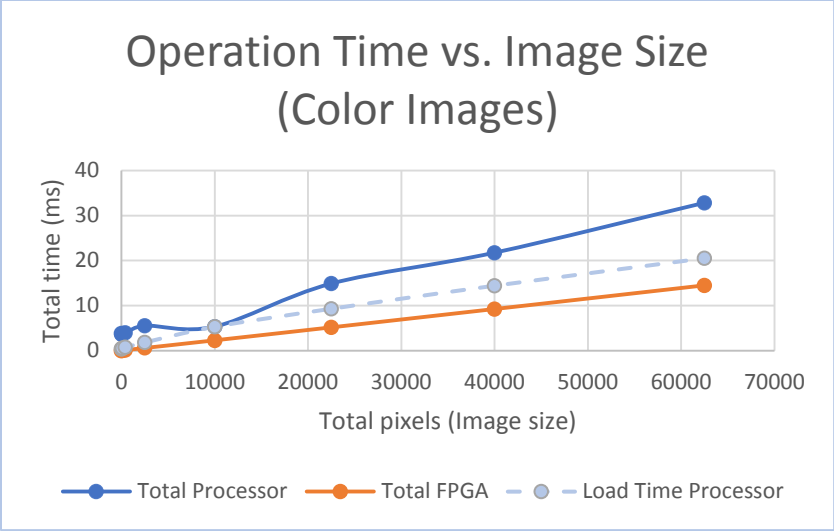


Graph 2 - Runtime comparison between processor and FPGA. Image: fruits.tif (280x186)

5.2. Complete Frame

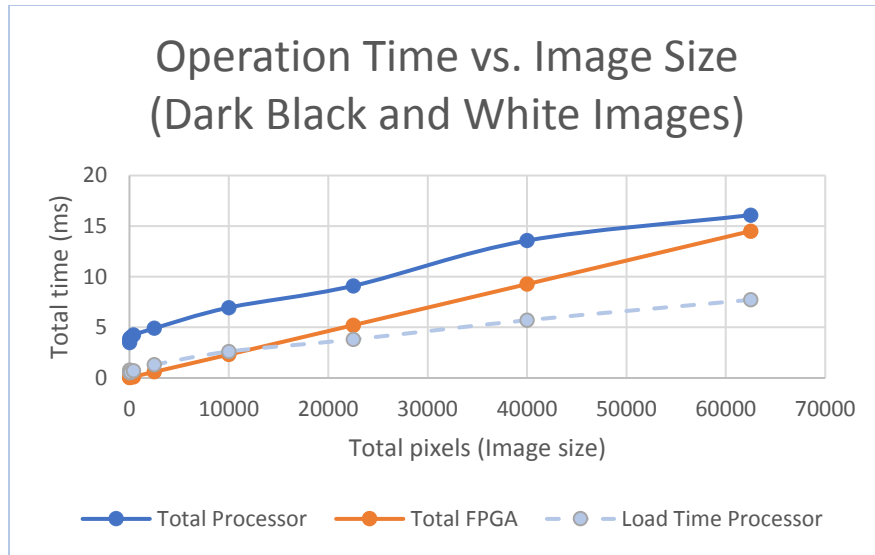
Before explaining our results, we will briefly introduce the components of the graph. On the FPGA, load-time is defined as the time it takes the program to save the image into the FPGA BRAM, and run-time is defined as the time it takes the FPGA to load each pixel from BRAM, run it through the modules, store the resulting operation back in BRAM, and send the signal back to the python program. On the processor side, load-time is the time it takes the program to load an image and store it in a temporal variable (`self.image = cv2.imread(self.image_path)`). And the run-time is the time it takes to perform the specific operation.

```
(for grayscale: gray_img = cv2.cvtColor(self.image, cv2.COLOR_BGR2GRAY)).
```



Graph 3 - Comparison of image size between processor and FPGA.

On graphs 3 and 4 we can see how the total time (load-time + run-time) increases in both (processor and the FPGA) along with the image size. On graph 3 is the performance in color images, where the FPGA total time is even faster than just the load-time on the processor. The black and white images have a speed up on the processor compared to their color counterparts as the processor treats each pixel will less bits whereas the FPGA always treats each pixel as 24 bits even if it is black and white. That is why there is no noticeable speed up on the black and white FPGA processing.



Graph 4 - Comparison between sizes

5.3. Analysis

The advantage of the FPGA is that it allows extra computations without sacrificing the timing. With a processor, adding image or video-feed computations to the pixels also adds processing time; conversely in the FPGA all the streaming computations can be done as soon as the pixel enters the pipeline, with no additional charge.

5.3.1. Regular Bram Controller:

In the FPGA every two-pixels stored in the BRAM go through a state machine of 9 cycles. Running the FPGA clock at 50MHz produces a clock period of 20ns. Due to timing constrains, it was necessary to wait for one additional cycle on each step of the process. Even though this can be improved, it did not affect the performance of the FPGA over the processor. The definition of the states of the pixel operation, including the additional cycles just mentioned are described next:

Change_address:	Disables write and increments the address counter
Read_cycle_1_low:	Reads the first pixel bits and sends them through the pipeline.
Read_cycle_2_low:	Second reading clock required for a successful read of the pixel.
Write_cycle_1_low:	Enables write and start saving the first pixel into BRAM.
Write_cycle_2_low:	Second clock required for a successful 16-bit write to BRAM.
Read_cycle_1_high:	Reads the second pixel bits and sends them through the pipeline.
Read_cycle_2_high:	Second reading clock required for a successful read of the pixel.
Write_cycle_1_high:	Enables write and start saving the second pixel into BRAM.
Write_cycle_2_high:	Second clock required for a successful 16-bit write to BRAM.

9 stages x 20ns = 180 ns every two pixels. An image of 320 x 240 would take a total of 6.912ms per frame. Allowing for a total of 144.7 FPS. The formula is described below:

$$frame\ operation\ time = \left(\frac{total\ pixels}{pixels\ per\ address} \right) \cdot (stages\ per\ pixel) \cdot (clk\ period) \quad (1)$$

The 145 FPS is not affected by the number of operations that will be performed to the pixels. Our results proved being fully capable of performing 3 different operations in the pixels within one clock cycle.

In the basic version of the BRAM_CTRL. Following this equation, we obtain a 3.18% timing percent error between the calculation analysis and the experimental results (having the largest allowable image

for the BRAM – 320x200). This way we can estimate the operation speed of this module in 640x480 images to be 36 FPS.

$$frame\ operation\ time = \left(\frac{320 \cdot 200}{2} \right) 180 \cdot 10^{-9}$$

5.3.2. Improved Bram Controller:

Taking the advantage of FPGA parallel programming, we can improve the pixel computation time by analyzing two pixels at the same time. Given that each BRAM address has a width of 32 bit, there are two pixels per address; thus, we can practically divide in half the state machine steps. There are only five states every two pixels (change_address, read _1, read_2, write_1, write_2).

5 stages x 20ns = 100 ns every two pixels.

An image of 320 x 240 would take a total of 3.84ms per frame. Allowing for a total of 260 FPS.

$$frame\ operation\ time = \left(\frac{total_pixels}{2} \right) 100 \cdot 10^{-9}$$

This formula has a 0.72% timing error percentage between the calculated value from the analysis and the obtained experimental results for the values for the maximum allowed picture size for this BRAM (320x200). This small difference between the calculations and the experimental results is due to the communication, background processes and the nature of the while loop that is constantly checking. Following this analysis, we can project that a full image of 320x240 would be able to process up to 260 FPS in the improved version of the BRAM_CTRL. But a small 65 FPS for images of 640x480.

The following tables demonstrate the difference between our prior analysis and the actual data obtained from experimental results after running the project. It is possible to see how the readings from the python program cannot be lower than 51 μ s; thus, the experimental values for small images that take less than 0.051ms.

Frame computation time on FPGA- Original BRAM (one pixel at the time)			
Width/Height	Size (pixels)	Experimentation (ms)	Theory (ms)
1x1	1	0.051	0.00009
5x5	25	0.051	0.00225
10x10	100	0.051	0.009
20x20	400	0.68	0.036
50x50	2500	0.253	0.225
100x100	10000	0.926	0.900
150x150	22500	2.052	2.025
200x200	40000	3.626	3.600
250x250	62500	5.654	5.625
320x200	64000	5.778	5.760

Table 1- Analysis vs run-time results for the original BRAM.

Tables 1 and 2 indicate how accurate our calculations were in comparison with the actual real-life timing measurements. This information can help us determine how accurate our calculation will be for larger images 640x480 and how good will be the performance of the system. Due to the nature of the while loop that reads the memory information, our measurements cannot go below 51 μ s; so, there is a greater difference between the analysis and experimentation in the earlier readings (as can be seen on Tables 1 and 2). However, as the image grows we can see how our calculations and the obtained measurements match.

Frame computation time on FPGA - Improved version (more than one pixel at the time)			
Width/Height	Size (pixels)	Experimentation (ms)	Theory (ms)
1x1	1	0.052	0.00005
5x5	25	0.051	0.00125
10x10	100	0.055	0.005

20x20	400	0.058	0.020
50x50	2500	0.15	0.125
100x100	10000	0.528	0.500
150x150	22500	1.151	1.125
200x200	40000	2.026	2.000
250x250	62500	3.157	3.125
320x200	64000	3.223	3.200

Table 2- Analysis vs run-time results for the Improved BRAM.

5.4. FPGA Utilization

The largest resource consumption came from the mathematical operations used for the conversions (rgb565_to_rgb888, rgb_to_hsv) that caused a total of 86% of the designated LUT space. Given that the output below reflects the measurements of the improved version, in this output the mathematical blocks are repeated twice. The mathematical operations occupied a large amount of the LUTs, but the VHDL code performing these color-conversion operations can be still improved. There are researches geared to this kind of improvement, such as demonstrates Hanumantharaju [15] with a fast and accurate color-conversion algorithm that does not take many resources. The other challenge was the BRAM, that with 320x200 images we utilized a 53% of the designate Block RAM space.

Utilization			
		Post-Synthesis	Post-Implementation
Resource	Utilization	Available	Utilization %
LUT	15191	17600	86.31
LUTRAM	1410	6000	23.50
FF	5494	35200	15.61
BRAM	32	60	53.33
DSP	4	80	5.00
BUFG	1	32	3.13

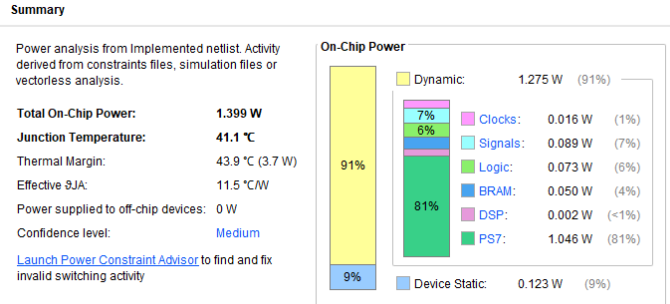


Figure 21 - bram_ctrl_plus implementation Power and Utilization from Vivado.

6. Literature Review.

The development of this research involved the combination of several sources.

Mah et al [14] Demonstrates with the development of his research that it is possible to use image triangulation as a base of video processing to determine the location of the UAV with respect to a fixed point. The experiment was capable to successfully determine the position of the drone with a distance approximation error percentage of 1.29% based on the distance between the centers of the two camera lenses. The problem however was the processing time required for each frame. The Raspberry Pi used for this development was only capable of delivering at most 10 frames per second. This paper is the base for the construction of GPS independent drone localization and is the motivation to construct an improvement based on FPGA.

Gao et al [16]. Uses a similar strategy to demonstrate the advantages of two image sensing devices for indoor localization; however, this methodology is based on the substitution of one of the camera by an optical flow sensor. An optical flow sensor is a device that calculates average movement estimates along the x and y directions. These estimates are calculated based on the movement of pixels from frame to frame (100 FPS in article). Like a camera, the optical flow sensor obtains the information from input

images, but they usually have a lower resolution (64x64). Allowing them to have higher frames per second.

RSSI (Received Signal Strength Indicator) is another way to measure the distance and localization of a drone or UAV. RSSI is based in measuring how well the wireless signal from a router (or access point) is received, and with this information determine its position. Mao et al [3] utilizes a method that they called NIL-RSSI (Novel Algorithm for Indoor localization based on RSSI). In this approach they use MDS (multidimensional scaling), an algorithm that allows the creation of a map, based on an array of distances. The problem with this approach is that is highly dependent on the Wi-Fi service and it is not always precise. Ideally this strategy should be implemented in conjunction with other methods. Very similar to this idea Seller et al [17] utilizes the distance from the source by measuring the time it takes for the signal to go and return, however this specific application is for outdoor drones.

There are also studies geared toward the construction of user friendly software that will create the FPGA algorithms for image/video processing. This way, instead of constructing the algorithms from scratch, engineers can use the software and load it to the FPGA or ASIC that will be used for the image processing. For instance, Darkroom [10] allows the programmer to write a code in a high-level language, model it and load it into the required FPGA or ASIC. In a test using Darkroom they compared the processing time of an Image Processing Algorithm running on an FPGA to a similar code written in C++/GCC running on a CPU (on a single core). Their results prove the advantages of the parallel processing of FPGAs over the C++ code. The code running on the FPGA (runtime of 0.33 seconds) proved a speedup efficiency 6.7 better than the C++ code (2.2 seconds).

Another example of these image processing tools is Rigel. As explained by the author, Rigel takes the computer vision pipelines specified in a multi-rate architecture and converts them into FPGA

implementations [18]. In their experiments they compare the performance of a Zynq 7020 (Artix-7 same FPGA that comes in one of the snickerdoodle modules) against the four-core ARM Cortex A15 with 2.32GHz for Computer Vision applications. Their results indicate that the Zynq SoC is between 3 and 55 times faster than the A15 in two different applications.

Other studies focus in the memory management and organization for image processing. As Mefenza et al [19] they focus on developing an improvement of the BRAM allocation algorithm for image processing on the FPGAs of the Zynq SoC, same used for this project.

7. Conclusions

This work has demonstrated the efficiency of developing image processing algorithms on an FPGA and its advantages over CPU based image processing algorithms for UAVs. The FPGA demonstrated to be faster, more reliable, and being capable of performing the same result on the image processing operations that were performed on the processor. We also provided a friendly Image Processing Verification Framework to test the image and video processing codes for future projects; this interface allows for real-time comparison of image operations between the FPGA and a processor running Python with OpenCV. In addition, our research included the description of image processing software that simplifies the construction of FPGA/ASIC computer vision algorithms. Among the found tools we can mention MATLAB's Vision HDL Toolbox, Darkroom, and Rigel that will facilitate the construction of future projects.

7.1. Future Work

With our results we have opened the door for many future improvements and alterations of this project. In terms of software, the next step would be to construct a framework that not only compares images, but real-time video feed to the FPGA as well. This will allow for a deeper comparison between the processor and the FPGA by including the streaming operations advantages of the FPGA into the formula. On the FPGA portion, for future work it would be ideal to develop more complex algorithms to have a drone that can fully operate autonomously and only based on the information provided by the input cameras. An improvement of the FPGA algorithm can be done different ways, the first would be following a single object [14] (recognizing a red ball) or with the use of AprilTags [20]. However, from the two mentioned approaches, the first one would be closer to our results since we have already constructed many of the required modules.

8. References

- [1] J. E. Gomez-Balderas, G. Flores, L. R. García Carrillo, and R. Lozano, "Tracking a Ground Moving Target with a Quadrotor Using Switching Control," *Journal of Intelligent & Robotic Systems*, vol. 70, pp. 65-78, April 01 2013.
- [2] W. Mao, Z. Zhang, L. Qiu, J. He, Y. Cui, and S. Yun, "Indoor Follow Me Drone," presented at the Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, Niagara Falls, New York, USA, 2017.
- [3] B. R. Stojkoska, J. Palikrushev, K. Trivodaliev, and S. Kalajdziski, "Indoor localization of unmanned aerial vehicles based on RSSI," in *IEEE EUROCON 2017 -17th International Conference on Smart Technologies*, 2017, pp. 120-125.
- [4] A. M. Sabatini, "Quaternion-based extended Kalman filter for determining orientation by inertial and magnetic sensing," *IEEE Transactions on Biomedical Engineering*, vol. 53, pp. 1346-1356, 2006.
- [5] S. A. M. Lajimi and J. McPhee, "A comprehensive filter to reduce drift from Euler angles, velocity, and position using an IMU," in *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2017, pp. 1-6.
- [6] M. Gowda, J. Manweiler, A. Dhekne, R. R. Choudhury, and J. D. Weisz, "Tracking drone orientation with multiple GPS receivers," presented at the Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking, New York City, New York, 2016.
- [7] Y. S. Suh, "Attitude Estimation by Multiple-Mode Kalman Filters," *IEEE Transactions on Industrial Electronics*, vol. 53, pp. 1386-1389, 2006.
- [8] K. Abdulrahim, C. Hide, T. Moore, and C. Hill, "Increased error observability of an inertial pedestrian navigation system by rotating IMU," *Journal of Engineering and Technological Sciences*, vol. 46, pp. 211-225, 2014.
- [9] J. g. Park, E. D. Demaine, and S. Teller, "Moving-Baseline Localization," in *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, 2008, pp. 15-26.
- [10] J. Hegarty, *et al.*, "Darkroom: compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, vol. 33, pp. 1-11, 2014.
- [11] L. Li, "Time-of-flight camera—an introduction."
- [12] Xilinx. (2017, *All Programmable SoC with Hardware and Software Programmability*. Available: <https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>

- [13] C. E. Palazzi, "Drone Indoor Self-Localization," presented at the Proceedings of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use, Florence, Italy, 2015.
- [14] B. Mah, "Model-Based Design for Visual Localization via Stereoscopic Video Processing," Electrical Engineering, Department of Electrical and Microelectronic Engineering, Rochester Institute of Technology, 2017.
- [15] D. M. a. R. V. G. a. H. S. a. B. S. S. Hanumantharaju, "A Novel FPGA Based Reconfigurable Architecture for Image Color Space Conversion," vol. 270, 2011.
- [16] Q. Gao, Y. Wang, and D. Hu, "Onboard optical flow and vision based localization for a quadrotor in unstructured indoor environments," in *Proceedings of 2014 IEEE Chinese Guidance, Navigation and Control Conference*, 2014, pp. 2387-2391.
- [17] S. Á, R. Seller, D. Rohács, and P. Renner, "Multilateration based UAV detection and localization," in *2017 18th International Radar Symposium (IRS)*, 2017, pp. 1-10.
- [18] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan, "Rigel: flexible multi-rate image processing hardware," *ACM Trans. Graph.*, vol. 35, pp. 1-11, 2016.
- [19] M. Mefenza, N. Edwards, and C. Bobda, "Interface Based Memory Synthesis of Image Processing Applications in FPGA," *SIGARCH Comput. Archit. News*, vol. 43, pp. 64-69, 2016.
- [20] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 3400-3407.

Appendix

A – Images and formats used for comparison

Image: colorful_landscapes.tif

Sizes: 1x1, 5x5, 10x10, 20x20, 50x50, 100x100, 150x150, 200x200, 250x250



Image: rit_logo.tif

Sizes: 1x1, 5x5, 10x10, 20x20, 50x50, 100x100, 150x150, 200x200, 250x250



Image: rit_logo_invert.tif

Sizes: 1x1, 5x5, 10x10, 20x20, 50x50, 100x100, 150x150, 200x200, 250x250



Image: apple.tif

Sizes: 262x240



Image: apple.tif

Sizes: 262x240



Image: colorful.tif

Sizes: 40x40

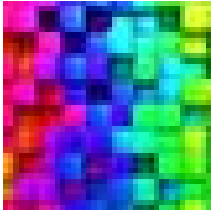


Image: fruits.tif

Sizes: 280x186



Image: imagine_rit.tif

Sizes: 60x60, 200x200



Image: RIT_logo_invert.tif

Sizes: 40x40, 100x100



Image: RIT_logo_invert.tif

Sizes: 40x40, 100x100



Image: max_image.tif

Sizes: 320x200



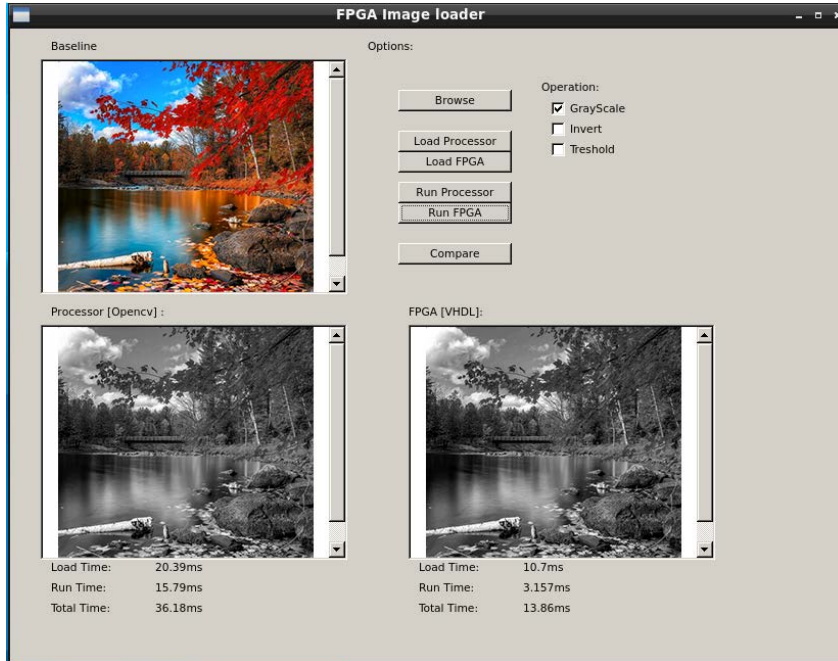
Image: test_image.tif

Sizes: 280x158

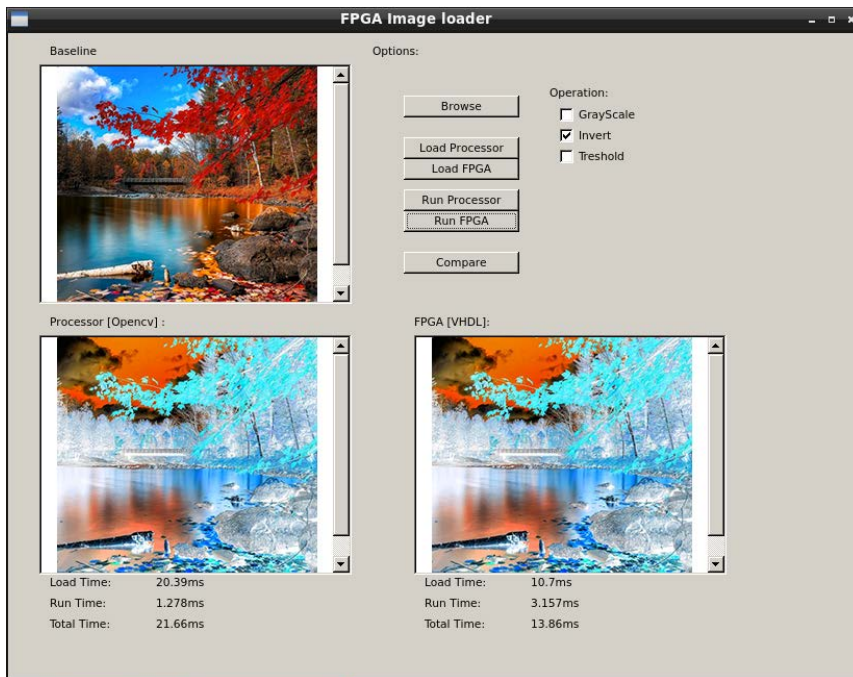


B - Program execution examples:

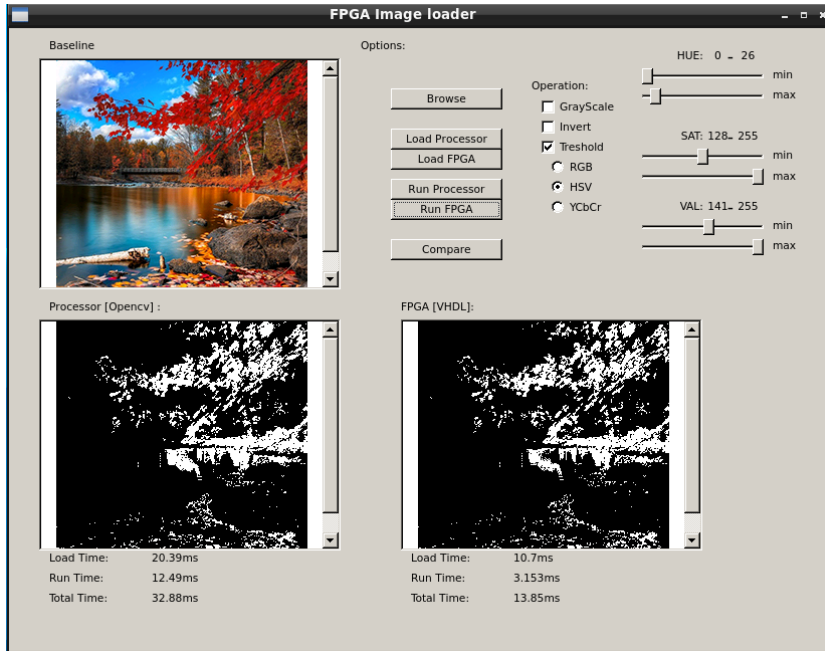
One operation: grayscale. Image: colorful_landscapes.tif



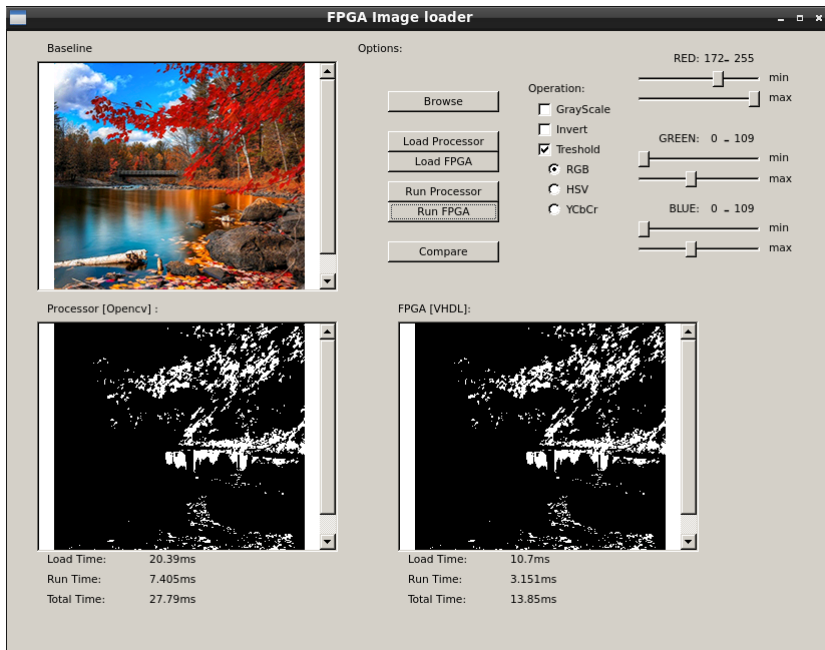
One operation: invert. Image: colorful_landscapes.tif



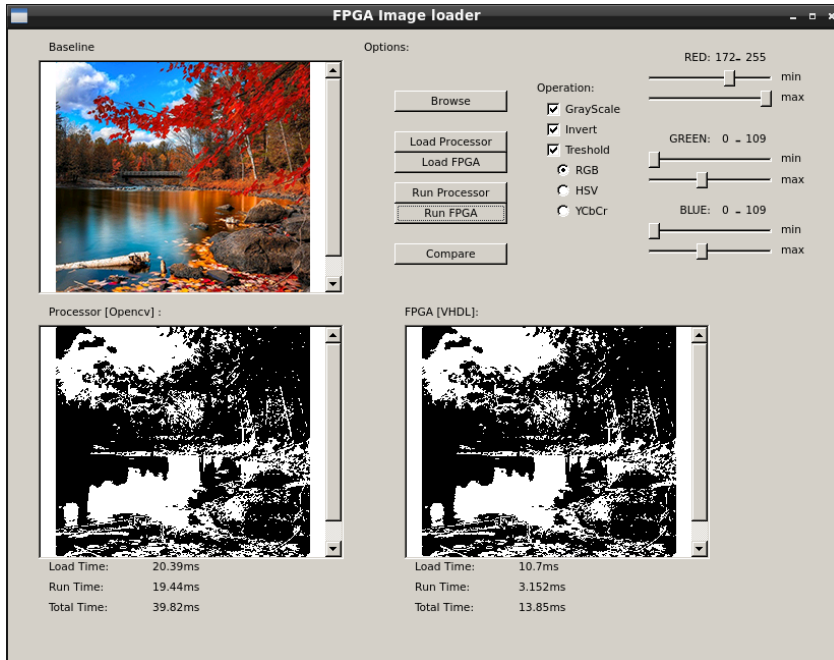
One operation: HSV threshold. Image: colorful_landscapes.tif



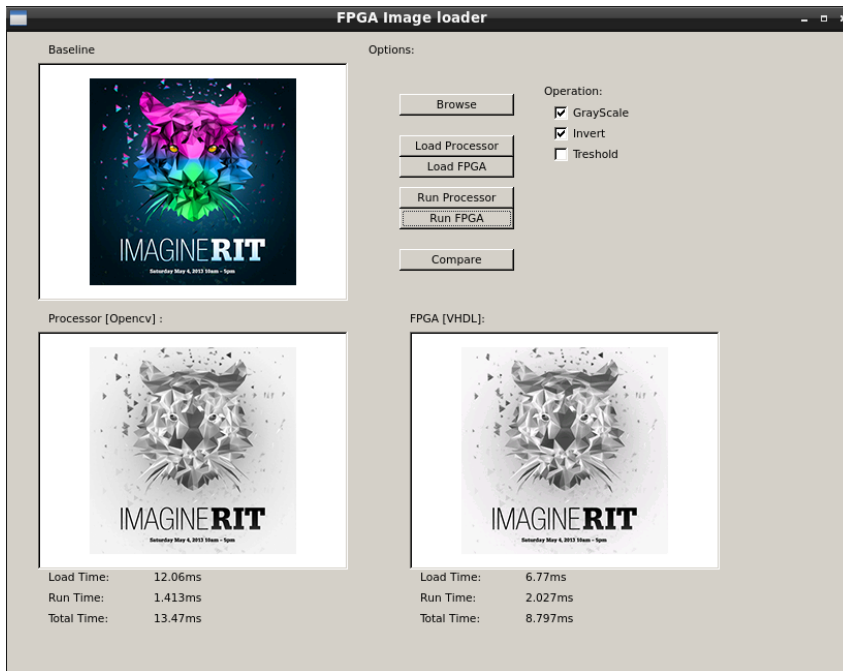
One operation: RGB threshold. Image: colorful_landscapes.tif



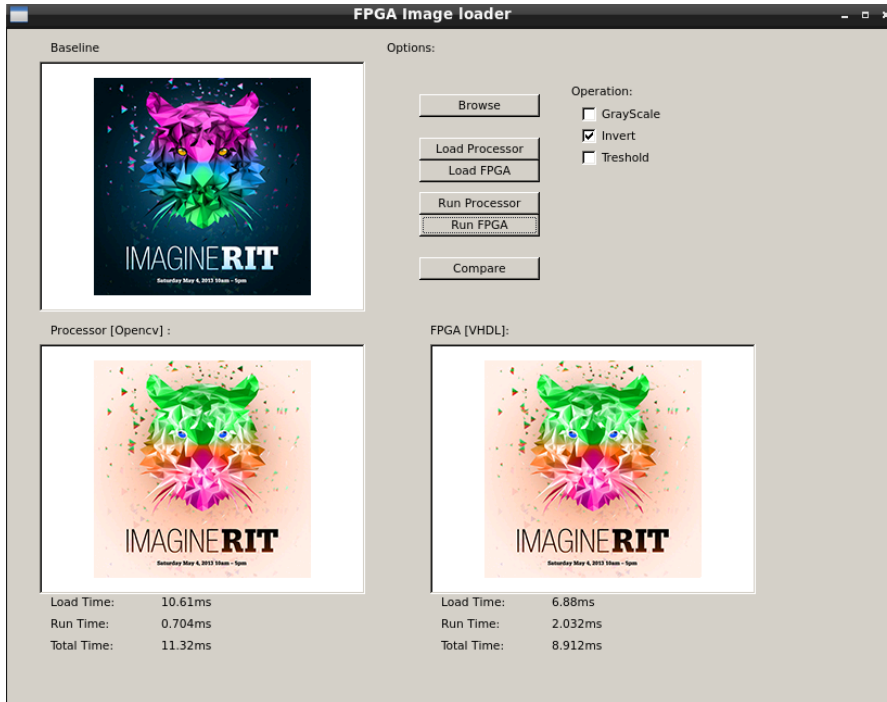
Three operations: grayscale, invert and RGB Threshold



Two operations: grayscale, invert. Image: imagine_rit.tif



One operation: invert. Image: imagine_rit.tif



One operation: HSV threshold. Image: imagine_rit.tif

