|  |
| --- |
| University of South Florida |

Image Processing on FPGA

April 20th 2015

Juan Lopez Marcano

U18567779

Rapid Systems Prototyping

EEL 6936

Professor: Dr. Jorge Galvis

1. Abstract

While using FPGAs, building a computer vision application that processes live video in real time requires a complex architecture. This architecture includes an FPGA, a camera module, a microcontroller, a display, and the interfaces to merge all these components. Therefore, the scope of this project is to perform image processing on static images which are stored in distributed ROMs.

This project will focus on detecting features of images. Pixel value features will be used to detect glasses in static pictures. To achieve this goal, a conventional computer vision approach will be taken: the code will include a training phase and a detecting phase. The training phase will be in charge of extracting the features to be searched in the static image and the detecting phase will be in charge of loading the images and performing the pixel computations to decide whether or not elements in the static image match the extracted features. A manually-set threshold (tolerance) will be used to determine if there is a match.

1. Theory Background

When using computer vision for feature detection, the approach to be taken depends on the gesture to detect or recognize. For instance, an algorithm suitable for detecting a corner might not be suitable for detecting a face. However, all the algorithms contain at least two fundamental stages: training and recognition [2][4].

The training stage is where the computing unit is taught what to look for in an image or in a stream of images. This can be done by uploading pre-processed classifiers [1] or databases with images to be processed[4] . When using a custom database, it is common practice to upload images that contain the gesture or object to be tracked under different conditions in order to make the system robust to occlusion, lighting conditions, backgrounds, among other factors that make computer vision challenging. Unfortunately, the larger the dataset used for training, the longer the computing unit takes to detect and/or recognize.

Another approach used for training is template matching. It consists of extensive feature extraction of typically one image. Rich features, such as direction, lines, corners, shapes, etc., are extracted from the training image and compared with those of a test image.

One of the most popular detection algorithms is the HOG feature detection algorithm. In this algorithm, a NxM image is segmented in nxm blocks. Each block is taken to grayscale and then convolved with itself, and the result of that operation is the gradient of the block, which can provide information on orientation, corners, shapes, among others.

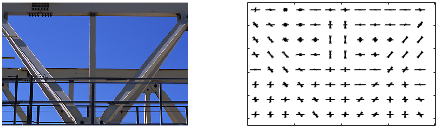
 Figure 1: Input image (left), output image (right). Image courtesy of MathWorks.com (See link 1 for more information)

Figure 1 shows a Matlab simulation of a HOG feature extractor. On the left is a 384x256 image, which was passed through a HOG feature extractor that computes features of blocks of 32x32 pixels. On the right is the output of the feature extractor, which contains the features. If we compare both images, we can see that the input image contains an inclined, gray metal bar, and the output image contains inclined black lines in a white background, which validates what was explained in the previous paragraph.

The detection stage depends heavily on what it is that the system is trying to recognize and what approach the designers wish to pursue. For instance, the algorithm proposed in [6] performs skin color segmentation to improve the accuracy of hand detection. Similarly, the algorithm proposed in [10] performs skin color segmentation and finds the centroid of the hand to recognize gestures. Further, the authors of [2] used median filters and skin color detection to improve accuracy of hand detection. Another approach is by defining region of interests and perform intensive computing operations in the area where the hand is known to be. Nevertheless, this can be performed only the object to be detected is always at the same position within the image, which is the case of [6].

Regardless of what approach is taken, the image should be taken to grayscale to reduce computational cost. When RGB images are converted to grayscale, the result is a vector of values that go from 0 to 255 which synthesizes the three channels of RGB images [1]. Grayscaling is also common for the training stage, but it could be omitted because training is done only once. In contrast, testing images may come in the system in an ongoing basis. After the image is grayscaled, computations are made to decide whether or not the test image is similar to the target the system trying to detect.

1. Proposed Approach

The feature extraction approach used in this project is a less sophisticated combination of template matching and HOG features. In our approach, a dataset of 5 images of 240x160 pixels each. The training stage consists of the following steps:

1. Pixel averaging: In this step, the average value of the pixels of each image is computed. Thus, we obtain 5 different averages.
2. Threshold: In this step, the value of pixels is compared to the average of its respective image. Therefore, a pixel Pi , from an image Ii whose average is Ai , is compared to Ai . If the average value is greater than the pixel value, it is assigned 0 (denoting a dark pixel), otherwise, it is assigned 1 (denoting a light pixel). This could be seen as an implementation of a median filter, or, as commonly known in computer vision, a threshold.
3. Feature extraction: Similar to our example of HOG feature extraction, our method uses blocks of 24x16 pixels. Thus, every picture is segmented into 10x10 blocks, which totals 100 features. To convolve the pixels, the number of ones is counted. If more than half of the pixels is 1, then the weight of the feature, or orientation, is 1. Otherwise, it is 0. This is done for every image in the training dataset. Hence, we create 500 features. Unlike figure n1, the output of this implementation will look like a checkerboard rather than a rich vector of features.

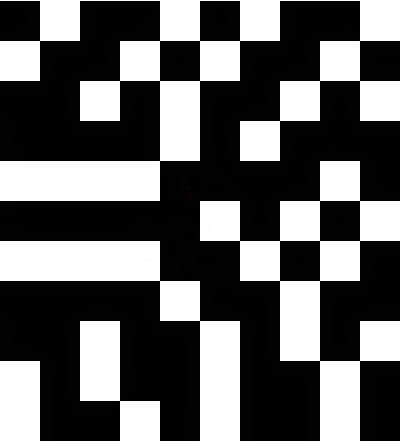
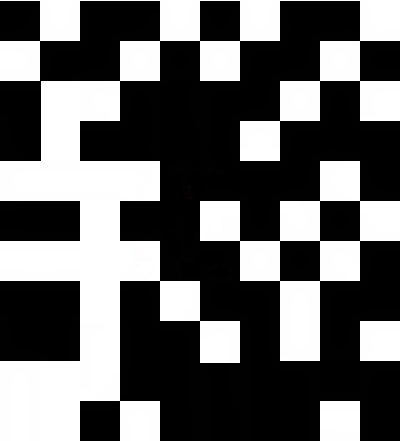


Figure 2: Mockup images of the feature detector

For the detection stage, steps 1 through 3 are done on the test images. Then, the feature of vectors of the test image is compared to the features of the training dataset. If there is a 85% percent match with one of the 5 feature vectors, then the image contains glasses.

Figure 3: Pictures used in the training dataset

Figure 4: Test images

1. Project Description

Although the previous sections of this report have been concerned with the algorithms developed and implemented, this section will begin by briefly describing how the display was accomplished and will elaborate on the hardware implementation of the object detection algorithm.

* 1. Display:

Because we are using an Altys XC6SLX45 board from the Spartan-6 family, connecting to a monitor is possible through a DVI interface. Then, depending on the frequency what frequency is used and what format the screen is compatible with, we can implement formats as low as VGA or as high as HDTV. For this project, HDTV was used.

The code can be found in the appendix. Most of the code came from link 2, which implements a DVI-D interface for a Pipistrello board. To make this code compatible with the Altys board, the constrains file was made from scratch and three new cores were generated:

1. IBUF: This core was created by using the core generator. The input to this core is the clock of the board and the output is a buffered clock signal.
2. BUFIO2: This core divides the frequency of a buffered clock signal by a factor of 2 and a duty cycle of 50%. For our design, the output was a buffered clock signal of 50MHz. Additional outputs, such as a locked clock, could have been made available, but they were not relevant for the purpose of this design.
3. TMDS\_fifo: An instance of a TMDS\_ fifo was referenced in the code. It was generated with all the signals specified in the component declaration in order to obtain the desired results.

Items 1) and 2) were necessary because the Pipistrello board contains a 50MHz clock signal with 50% duty cycle which does not need to be buffered. On the other hand, the XC6SLX45 contains a 100MHz clock that needs to be buffered.

File vga\_gen.vhd was edited as well so that it could display the test image and the result of classification test. In order to do so, vga\_gen.vhd instantiates all the test images in ROM so that the images can be displayed. In addition, these display regions are defined in this file. The last relevant addition to this file is a process called addresser, which synchronizes the data from the ROMs to the 240x160 region. In lines 112 through 115, we can see the manipulations necessary to properly synchronize the ROM address to be read. The computations performed in this lines can be summarized in the following formula:

Addr = 240\*Ypix + Xpix

This is done so that ROM addresses are displayed in the region deterministically. In other words, the value stored at ROM address ADDR will always be displayed at pixel XY.

The top level implementation of this code, file dvid\_serdes contains the following instances:

1. dvid\_out.vhd: It is the interface between the FPGA and the DVI. It contains 4 TMDS encoders: one for the red channel, one for the blue channel, one for the green channel, and one for the pixel clock. It receives signals from the display generator (vga\_gen) which indicate the color data (through signals red\_p, green\_p, and blue\_p); whether or not the video should be synced; whether or not it should be blank; the three clocks of the system; and the strobe flag to serialize the data. It contains 3 data serializers that encode each channel.
2. clocking.vhd: This instance receives the 50MHz clock created by BUFIO2 and generates a 25MHz clock, a 125MHz clock, and a 250MHz clock. It uses a PLL and 4 BUFG instances to buffer the new clocks;
3. vga\_gen.vhd: This unit creates a pattern of colors to be displayed in a monitor. It outputs flags that indicate whether or not the screen should be blank; whether or not the horizontal pixels for the current line have been exceeded; whether or not the vertical pixels for the current line have been exceeded; and creates a 240x160 pixels region where the image to be analyzes is displayed and a 256x256 region where the output of the classification process is displayed. All this is done by constantly checking a horizontal and a vertical line counter and making sure that they do not exceed the resolution. If they do, the color data sent to the encoder is black.

Below are the box diagrams of the code explained in this section:

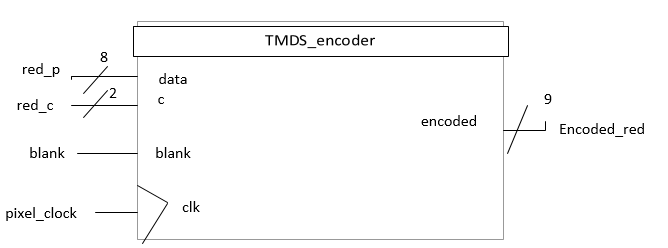
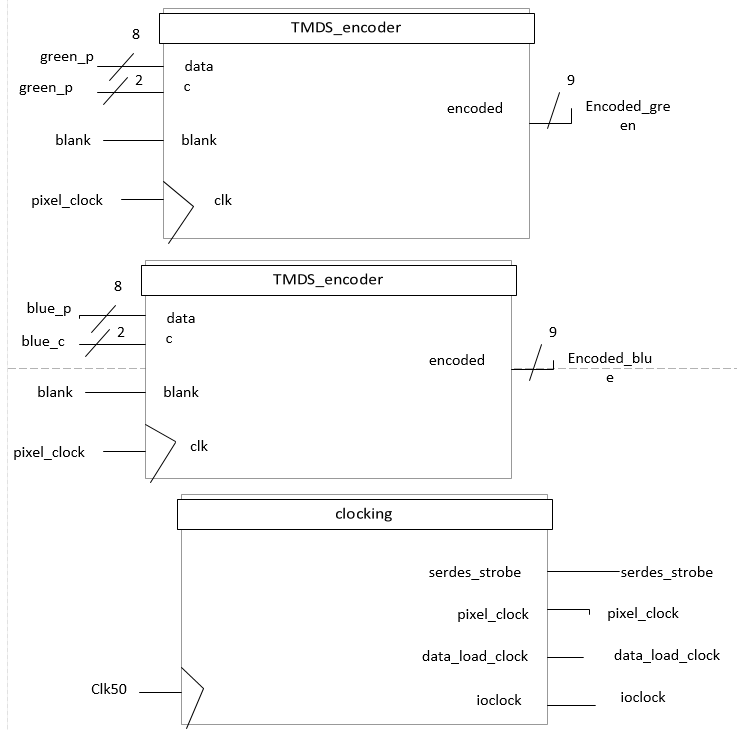
 

Figure 5: TMDS encoders (top through middle), clocking (bottom)

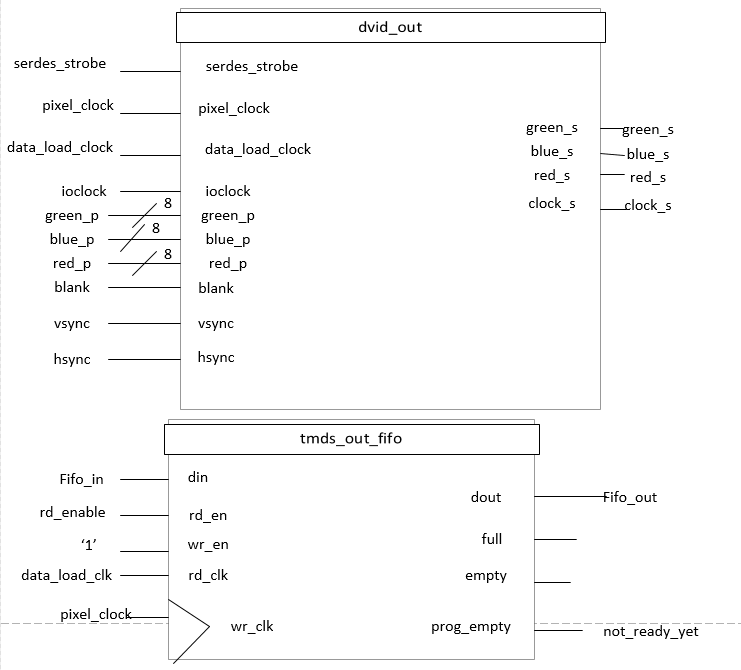
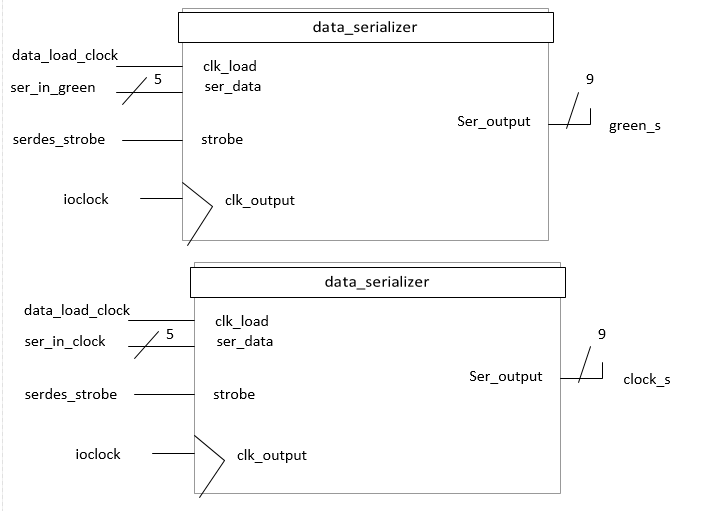
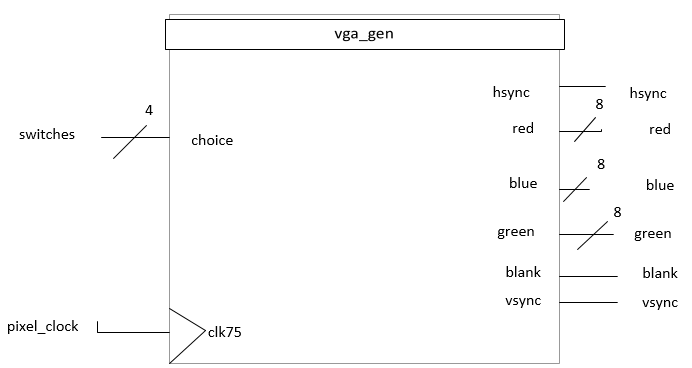
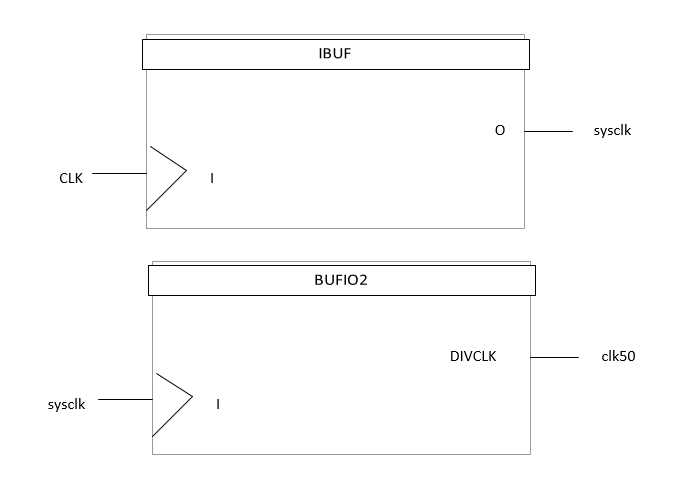
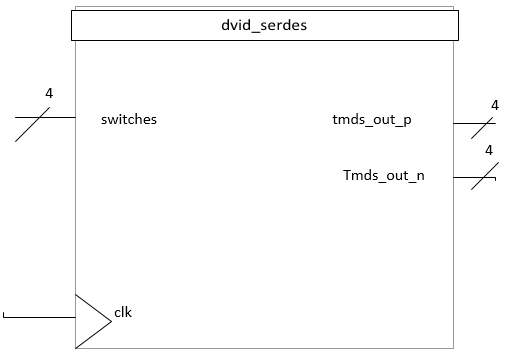


Figure 6: dvid\_out (top), tmds\_out\_fifo (bottom). Note: there is no source for tmds\_out\_fifo because it is a core.



Figure 7: Data serializers for green channel and clock (top) and vga\_gen (bottom). Note the remaining 2 data serializers (for red and blue) are not shown.

  Figure 8: IBUF (top), BUFIO2 (center), and dvid\_serdes – top level – (bottom).

* 1. Training and Detection in VHDL.

Section 4.1 was explained from a high-level perspective because most of its code was taken from a third party. In contrast, section 4.2 will be explained more thoroughly because it contains the most important part of the project.

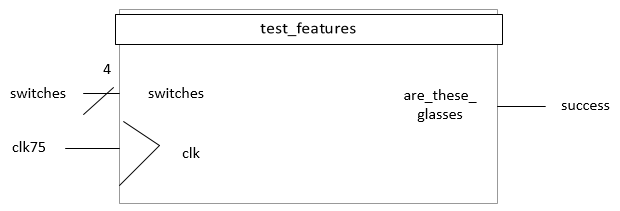
To train the system, one file was created: pix\_avg.vhd. It contains an algorithmic state machine that drives all the operations. As dataset, 5 images were used (see Figure n3). These images were downloaded online and preprocessed so that they could be used by the FPGA. In order to access the images via the FPGA, the following procedure was done:

1. The images were resized to 240x160 pixels, which yields 38400 pixels.
2. The images were converted to .coe files so that they could be stored in distributed ROMs. Distributed ROMs were used because they consume LUTs instead of actual on-chip memory.
3. Using Core Generator, distributed ROMs of 38400 of 8-bit values were created for each image. The input to the ROMs is the address, which is a 16-bit std\_logic\_vector. The output is the pixel value, which is an 8-bit vector of std\_logic.
4. The five images were instantiated in pix\_avg.vhd, and their input address is set by an address counter

Figure 9 shows the box diagram of training.vhd. As input, it has a clock and an enable bit, and as outputs it has a done bit and 5 100-bit vectors of std\_logic\_vector. The training stage should be done only once after the program starts running, thus, the enable bit is usually set to ‘0’ (active-low) so that the system can be trained. Once the system is trained, the done bit will be set to ‘1’, which will set the enable bit to 1 and preventing the system from training itself. The 5 std\_logic\_vectors are the 5 vectors of features mentioned in item 3) of section 3.

 Figure 9: training.vhd

Figure 10 shows the pseudo-ASM for training.vhd. It is called pseudo because every state is controlled by a process, and a process runs only if certain flags are high or low. The first stage is the averaging stage (see process avg). It is sensitive to train\_flag, th\_flag, and enable. Train\_flag is 0 at the beginning of process avg and is set to 1 once the p. Similarly, th\_flag is 1 when the threshold process is over and zero otherwise. In this process, the sum of all the pixels of every image are added and registered. Then, it jumps to process div, where the sums are divided by the number of pixels. The output of this division is the average pixel value.

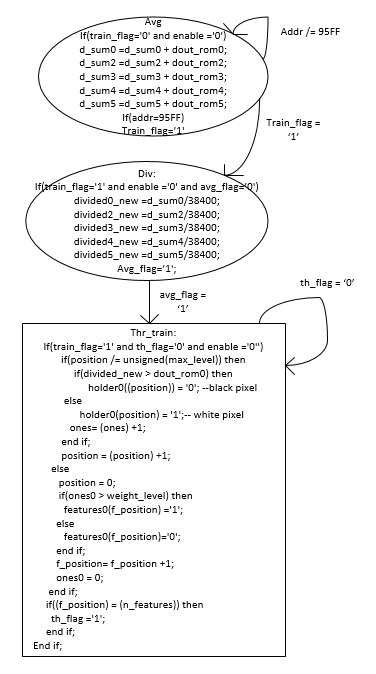
 Figure 10: test\_features.vhd

The remaining steps consist of applying the threshold to the image and creating the feature vectors. For simplicity, these two tasks are done together in the same process. There is a process in charge of doing this for every image. These five processes are thr\_train, thr\_train2, thr\_train3, thr\_train4, and thr\_train5. The objective was to parallelize slightly to improve performance. For each image, pixels are compared with the average threshold value. If the current pixel value is smaller than that of the average, a temporary std\_logic\_vector that holds up to 384 elements and another signal counts the number of ones. Once the vector is filled, if the number of ones in the vector is greater than 192 (50%), the value of the feature is 1, otherwise, it is 0. This procedure is repeated 100 times for every image. Thus, we obtain five feature vectors of 100 features each.

Once the feature vectors are computed, the done bit is activated and the system goes to “IDLE” state. Once the idle state is reached, the state machine will not advance, thus, preventing the system from retraining itself.

There is a file called test\_features.vhd. It instantiates pix\_avg.vhd and the three test images used in this program. Figure n10 illustrates the box diagram for this design file.

It reuses the code of pix\_avg to find the average pixel value of the test image, apply the threshold, and find the feature vector of the test image. In order to choose the test image, the user has to utilize the slide switches of the board. Thus, it works as a mux.

 Figure 10.a: Pseudo-ASM

This design file also contains five processes that compare the feature vector of the test image with those of the images in the dataset.

Something to note from processes of file pix\_avg is that all the files are sensitive to a signal called reset. This reset is a flag that is set to 0 every time the input from the switches changes. When the reset is set to 0, all the flags that control the processes that find the pixel average, apply the threshold, and compare with the classifier are set to 0. Thus, the processes will run again if a new image is tested. This reset signal was necessary so that all the processes could run more than once.

1. Experiment and Results

The following test images were used for the testing phase of the system.

Figure n11: Images used for testing. Glasses (left), boots (center), and an apple (middle).

As we can see in Figure n11, an image taken from the dataset was used for testing. The objective was to test a trivial case. On the middle is a pair of boots and on the left is an apple.

Because the program takes too long to synthesize, two tests were run for each image: The tolerance was set to 85 matches and 10 matches for the first and second test respectively. The results are shown in the table below.

|  |  |  |
| --- | --- | --- |
|  | Tolerance set to 85 | Tolerance set to 10 |
| Glasses | Positive | Positive |
| Boots | Negative | Positive |
| Apple | Negative | Positive |

Table 1: Results of the tests

Table 1 summarizes the results of the experiments. As expected, the glasses returned positive for both tests. This was expected because it completely matches one of the feature vectors of the dataset.

Boots and apples returned negative for a tolerance of 85, but positive for a tolerance of 10. It was unknown how much the selectivity had to be reduced to get false-positives, so we tested with 10 matches because it is a low tolerance.

Figures n12 through n14 show the tests run for 10% of tolerance.

 Figure n12: True-positive

 Figure n13: False-positive due to low tolerance

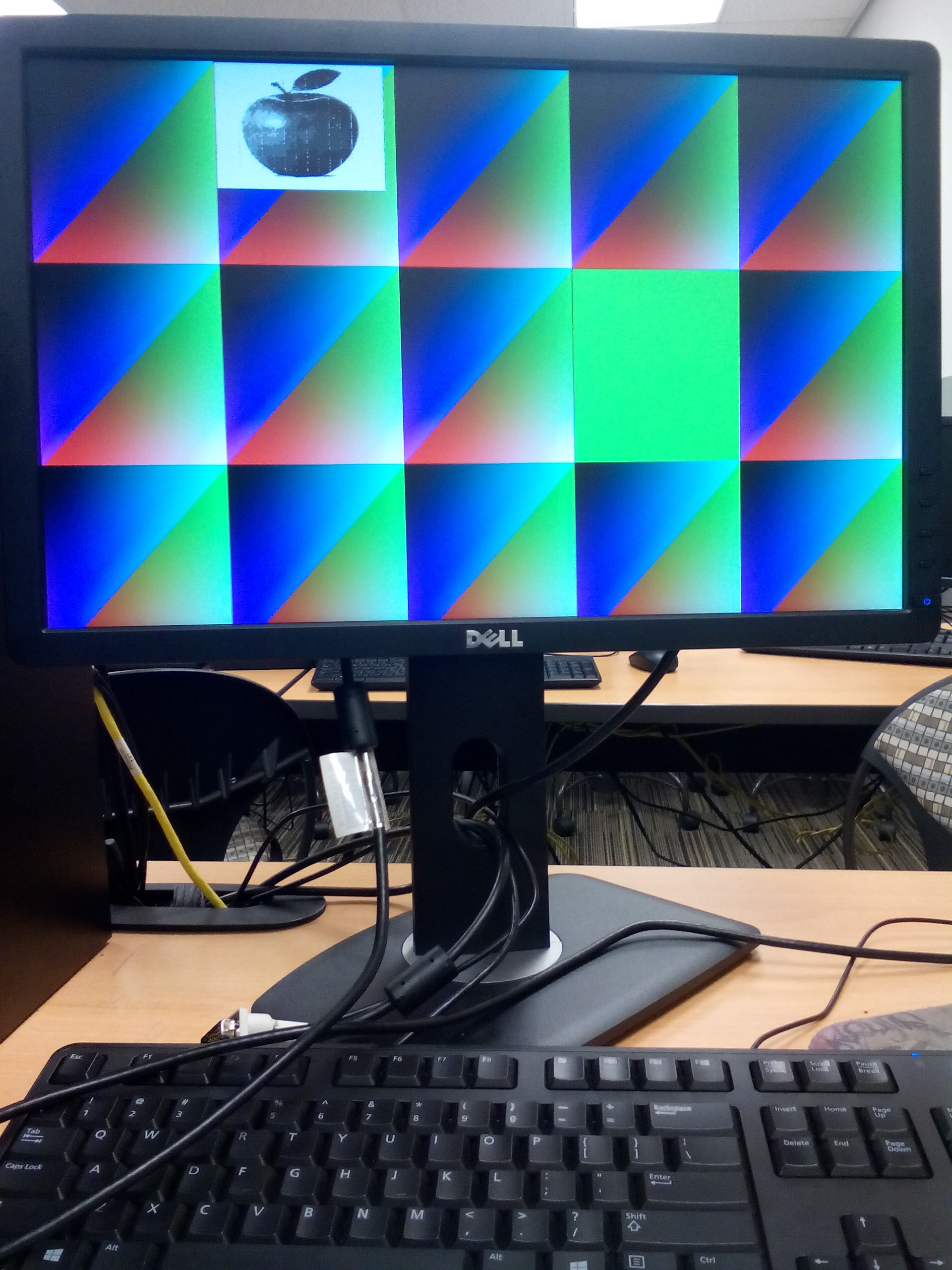


Figure n13: False-positive due to low tolerance

The background has a color pattern that was obtained from the code of link1. Close to the top left corner, is the grayscale image obtained using Matlab. On the right is a 256x256 block that denotes the result of the tests. The block is green when there is at least 10 matches and red otherwise.

1. Performance and FPGA Consumption

If we ignore combinational delay, which Xilinx ISE reported to be 3.15ns, we can conceptually calculate the delay of the training stage and the detection stage by computing the number of clock cycles and dividing by the frequency of the clock, which was 75MHz.

For the training stage, the processes that take the largest amount of cycles are the process that performs pixel averaging and the process that computes the threshold and creates the feature vector. Because the images contain 38400 pixels, we know that these two processes take 76800 clock cycles to be completed. Therefore, the training stage takes 1.024 seconds (if we ignore path delay).

The performance of the training stage can be considered acceptable. Classifiers built using C++ or Matlab can take several seconds, but they are more sophisticated: We can obtained the feature vector of Figure n1.

Something to note is that in this architecture, the time it takes to train a system with 5 images, is the same as that necessary to train N images. In this architecture, all five feature vectors are in parallel. Even though the pixel averaging process is done sequentially, it could be easily be broken into N number of parallel processes. It was not done this way, however, because it requires keeping track of too many flags.

The time to run a test is slightly higher than that used to train the system. The same number of cycles is needed to extract the features, and once they are extracted, they are compared with the features of the classifier, which take 100 cycles. Thus, the time to test an image is 1.025 seconds.

As with the training stage, the testing stage is highly parallelized. The comparisons are run in parallel processes, which means that increasing the number of feature vectors from 5 to N does not affect the theoretically computed time of 1.025 seconds. Unfortunately, increasing the number of processes increases LUT consumption.

|  |  |
| --- | --- |
| Slices | 6612 |
| FF’s | 162 |
| BRAMs | 0 |
| LUTs | 20800 |
| IOBs | 13 |
| Multi/DSP48s | 0 |

Table 2: Device utilization

Table 2 summarizes the device utilization of the architecture developed in this project. Similarly, table illustrates the device utilization reported by the authors of [10]. The architecture proposed in this project and that proposed in [10] differ greatly. To begin with, the authors of [10] utilized Xilinx System Generator and DSP cores. Also, they used an external memory to save the images of their dataset. Lastly, they created an interface to a camera unit.

|  |  |
| --- | --- |
| Slices | 14053 |
| FF’s | 56085 |
| BRAMs | 0 |
| LUTs | 39540 |
| IOBs | 463 |
| Multi/DSP48s | 459 |

Table 3: Device utilization of [10]

We can see that the combinational logic of [10] doubles that of the architecture of this proposed. We can find that by comparing the number of LUT’s and slices. However, part of our device utilization is due to the distributed ROMs we created to hold the dataset of the training system and the testing images.

An achievement of this architecture is its low flip-flop (FF) usage. As explained previously in this section, the architecture proposed in this project is highly parallel and only required 162 FF’s. On the other hand, the architecture of [10] uses approximately 350 times more LUTs than the proposed architecture.

In terms of IOBs and DSP cores, our architecture has a much shorter usage. Firstly, no DSP cores were used; secondly, the I/O of the FPGA was barely used. It was used for the clock, the four slide switches, and the 8 TMDS encoders for the DVI interface. In contrast, the architecture of [10] implemented an interface for a camera module, in addition to, perhaps, a few slide switches/buttons and encoders.

1. Issues and possible solutions

If LUT consumption is considered an issue for this design, a simple way to solve that problem is by making the system more sequential than parallel. In addition, an external memory could be used in order to store the images.

As far as performance goes, the display has some imperfections. Figure 15 shows imperfections in a test image. There should be some glasses and a white background, but there are black pixels in the black background. This occurs because of the path delay and the number of clock cycles described in section 6. A solution, which could not be synthesized, is to use a higher frequency clock, such as the 250MHz, to drive the computation-intensive portions training stage and the classification stage. Unfortunately, Xilinx threw errors when trying to synthesize it.

 Figure n15: Imperfections in the display

1. Conclusions

When using FPGAs, there are many different ways to create a computer vision application. The architecture of the application will not only depend on the features to detect, track, and/or recognize, but also on the author’s preferences on how much filtering they wish to perform.

The most fundamental components of computer vision applications are training and detection/classification. In this project, training and classification were broken into a few steps, such as pixel averaging, applying the threshold, comparing, and deciding whether or not the test image contains glasses based on a tolerance factor.

Computer vision also require grayscaling to downgrade the image and perform computations for a single channel instead of three. This and many other operations are performed with the intent to reduce computational cost, but they add extra challenges to the long list situations that may make a computer vision application fail.

The accuracy of a computer vision application based on FPGAs will depend not only on the application, but also on the board to be used. Some applications, such as face recognition, have a very discouraging performance, even when using the most sophisticated algorithms. Also, not every architecture can be implemented on any board. We learned that in this project, where 96% of the LUTs were used.

Besides LUT consumption, the other issue found in this project has to do with imperfections in the display. As mentioned in the previous section, this issue could be solved by using a different clock frequency and/or pipelining. For the former, the challenge is that the synthesis tools may not allow this. For the latter, the challenge is to handle flushing and stalling.

References:

1. Bradski, G., & Kaehler, A. (2008). Learning OpenCV. [electronic resource] : Computer Vision with the OpenCV Library. Sebastopol : O'Reilly Media, 2008.
2. Cho, P. C., Li, C. T., & Chen, W. H. (2012, June). Implementation of low-cost vision-based gesture recognition systems based on fpga approach. In *Computer, Consumer and Control (IS3C), 2012 International Symposium on*(pp. 329-332). IEEE.
3. González, A., Hayashibe, M., & Fraisse, P. (2012, October). Estimation of the Center of Mass with Kinect and Wii balance board. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on* (pp. 1023-1028). IEEE.
4. Gu, Q., Takaki, T., & Ishii, I. (2013). Fast FPGA-based multiobject feature extraction. *Circuits and Systems for Video Technology, IEEE Transactions on*,*23*(1), 30-45.
5. Gultekin, G. K., & Saranli, A. (2013). An FPGA based high performance optical flow hardware design for computer vision applications. *Microprocessors and Microsystems*, *37*(3), 270-286. Elsevier.
6. Gupta, A., Sehrawat, V. K., & Khosla, M. (2012). FPGA Based Real Time Human Hand Gesture Recognition System. *Procedia Technology*, *6*, 98-107.
7. Kalarot, R., & Morris, J. (2010, June). Comparison of FPGA and GPU implementations of real-time stereo vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on* (pp. 9-15). IEEE.
8. Matai, J., Irturk, A., & Kastner, R. (2011, May). Design and implementation of an fpga-based real-time face recognition system. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on* (pp. 97-100). IEEE.
9. Park, In-Kwon, Jung-Hyun Kim, and Kwang-Seok Hong. "An implementation of an FPGA-based embedded gesture recognizer using a data glove."*Proceedings of the 2nd international conference on Ubiquitous information management and communication*. ACM, 2008.
10. Raj, S. A., Sreelatha, G., & Supriya, M. H. (2012, March). Gesture recognition using field programmable gate arrays. In *Devices, Circuits and Systems (ICDCS), 2012 International Conference on* (pp. 72-75). IEEE

Links

1. <http://www.mathworks.com/help/vision/ref/extracthogfeatures.html>
2. <http://hamsterworks.co.nz/mediawiki/index.php/DVI-D_Serdes>