Study of Simulatable SoCs containing ML and Computation Accelerators and their Power and Architecture Analysis via Case Studies in HOG-SVM and Haar-Cascade-based-Face-Detection

M.Tech. Dissertation

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

Electronic Systems

by

Topomoy Dhar

Roll No. 193070039

under the guidance of **Prof. Sachin B. Patkar**

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

Powai, Mumbai - 400076

June 2021

Dissertation Approval

The dissertation entitled

Study of Simulatable SoCs containing ML and Computation Accelerators and their Power and Architecture Analysis via Case Studies in HOG-SVM and Haar-Cascade-based-Face-Detection

by

Topomoy Dhar

(Roll No. : 193070039)

is approved for the degree of

Master of Technology in Electrical Engineering

(Examiner)	(Examiner)
(Chairperson)	Prof. Sachin Patkar Dept. of Electrical Engineering (Supervisor)

Date: 29th June 2021

Place: IIT Bombay.

Declaration

I declare that this written submission represents my ideas in my own words and where others ideas or words have been included, I have adequately cited and referenced the sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Topomoy Dhar
Roll No. 193070039
IIT BOMBAY

June 29, 2021

Acknowledgments

I would like to express my deep gratitude to Prof. Sachin Patkar for his valuable guidance all through. I also extend thanks to the research scholar of the HPC lab, Niraj N. Sharma for his constant help and support.

Topomoy Dhar

Abstract

In the modern world, applications running on embedded systems have very high requirements. The most important metric in these requirements is the need for high performance with minimum power dissipation and cost. To meet these demands, the industry is adopting more heterogeneous architectures that have reconfiguration capabilities. Unfortunately, this increases the complexity of the system and overshadows the advantages of such architectures. This thesis address performance enhancement by introducing the concept of hardware-software co-design and systems implemented on reconfigurable and heterogeneous platforms. We focus on performance enhancement for dynamically reconfigurable FPGA-based systems.

Histogram of Oriented Gradients(HOG) is a well-known computer vision algorithm for detection of objects and then using the Support Vector Machine(SVM) ML algorithm to classify those objects. Similarly, the Haar Cascade classifier is another method for object detection. This method was proposed by Paul Viola and Michael Jones in their paper "Rapid Object Detection using a Boosted Cascade of Simple Features". Haar Cascade is an ML approach where we train the classifier using a standard image database. Here, both of these algorithms are used as case studies to look into the increase in performance metrics once incorporated as Hardware-Software heterogeneous systems.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

In the current situation, the demand for high-performance computing are increasing every passing day, and to increase the performance, the industry initially resorted to increasing the clock frequency of the single-core processor, before it hit the power wall in early 2000. However, the need for increasing hardware performance to meet the software demands is still rising and one such way to handle such performance requirements is through heterogeneous computing. It refers to the use of different types of processors in a single computer system, also referred to as Hardware-Software Co-Design. There are 2 ways to implement an algorithm, the first way is to implement complete functionality on the software side but that will involve sequential execution of data and parallelism won't be exploited until multi-core architectures are used. The second way is to implement complete functionality on the hardware side. This option could exploit parallelism algorithm-specific and it will not be generic. However, it will incur the cost of increased design time and complexity.

1.2 Objective

The goal of the project is to implement the HOG SVM algorithm used for vehicle detection and Viola-Jones face detection algorithm. Here we are comparing the algorithm by implementing it in 2 different ways. First, the entire algorithm is built purely on software, which involves sequential execution of the algorithm. In the second part, the algorithm is built using Hardware-Software Co-design, i.e., we are using an accelerator IP for parallel processing of the algorithm,

to enhance the performance.

1.3 Organization of the thesis

In the first part of the Thesis, both the algorithms are implemented. For this, first, the algorithm is built-in initially in python and from there, after training images we are able to get the features for classification, which is needed to implement the algorithm in Vivado HLS. After, getting those features, the entire flow of creating the hardware IP and integrating the IP with the Microblaze softcore processor available in Xilinx Vivado ,and then verifying the algorithm is done.

In the second part, the entire algorithm is code purely on software, i.e., without using the IP block ,and then integrated with Microblaze and executed the entire flow to verify the algorithm. Both these codes would be compared based upon the power consumption and latency in Vivado, thereby creating a baseline for this system. Then the entire flow will be performed on the Zedboard to cross verify the algorithm.

Chapter 2

Theoretical Background

2.1 Case Study 1 : Car detection using HOG Algorithm with SVM Classification

2.1.1 Histogram of oriented gradients

The histogram of oriented gradients (HOG) is a feature descriptor used in computer vision and image processing for object detection. This algorithm was introduced by Navneet Dalal and Bill Triggs in their paper "Histograms of Oriented Gradients for Human Detection".The technique counts occurrences of gradient orientation in localized portions of an image. This method is similar to that of edge orientation histograms, scale-invariant feature transform descriptors, and shape contexts, but differs in that it is computed on a dense grid of uniformly spaced cells and uses overlapping local contrast normalization for improved accuracy.

2.1.2 Object Detection using HOG SVM flow

The HOG descriptor mainly provides the structure of an object, with edge direction pointing to the sharp changes. It is done by calculating the gradient and orientation of the edges. Here the complete image is broken down into smaller regions of particular rectangular boxes and for each region, the gradient vector i.e., gradients and orientations. are calculated. We will discuss this in much more detail in the upcoming sections. Ultimately a Histogram is separated for each of these regions separately. Since, we are creating a histogram of the gradient vectors of image pixels for each region, hence the name ‘Histogram of Oriented Gradients’.

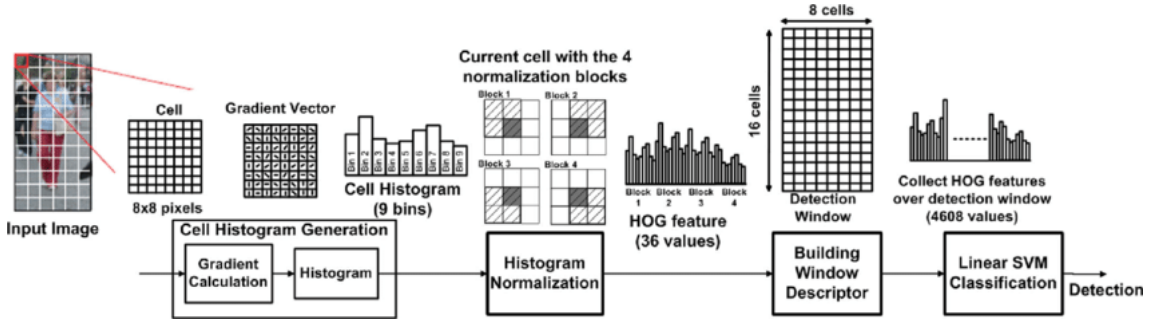


Figure 2.1: Object detection using HOG SVM algorithm

In this paper, we are detecting a car using HOG considering a sliding window of 40x100 pixels with a stride of 10 pixels in the horizontal direction and the vertical direction scans the entire image. For each such window, HOG generates certain features which are passed to the SVM classifier to determine whether there is a car in the given window. SVM classifier predicts the existence of a car.

The detection algorithm is divided into the following steps:

- Preprocessing
- Gradient Vector Calculation
- HOG feature Extraction
- Normalization
- SVM Classification

Preprocessing

In this implementation, the gamma (power-law) compression technique is used for either computing the square root or the log of each color channel. In this work square root is used to do the normalization, which helps to reduce the effects of local shadowing and illumination variations in the image.

Gamma Normalization in HOG is Power Law Transformation

$$s = cr^{\gamma}$$

where s is output pixel, r is input pixel, c is constant and γ is the exponent. This technique is used in various devices to correct image intensity by using the power-law transformation also

known as gamma correction. In short, gamma normalization in HOG is the same as gamma correction.

Gradient Vector Calculation

In image processing, we want to capture the sudden change in color like edges (i.e. black to white on a grayscale image). Hence, we need to calculate "gradient" on pixels of colors which is discrete because each pixel is independent and cannot be further split.

The gradient vector is termed as a metric that denotes pixel color changes in both the x-axis and y-axis, to each pixel.

Suppose f(x, y) records the color of the pixel at location (x, y), the gradient vector of the pixel (x, y) is defined as follows:

$$\nabla f(x,y) = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} f(x+1,y) - f(x-1,y) \\ f(x,y+1) - f(x,y-1) \end{bmatrix}$$

Thus the gradient vector for a particular pixel comprises of 2 terms:
Magnitude is resultant of the vector, g=$\sqrt{g_x^2 + g_y^2}$
Direction is the tan inverse of the ratio between the partial derivatives,i.e., phase $\theta = arctan(g_y/g_x)$.

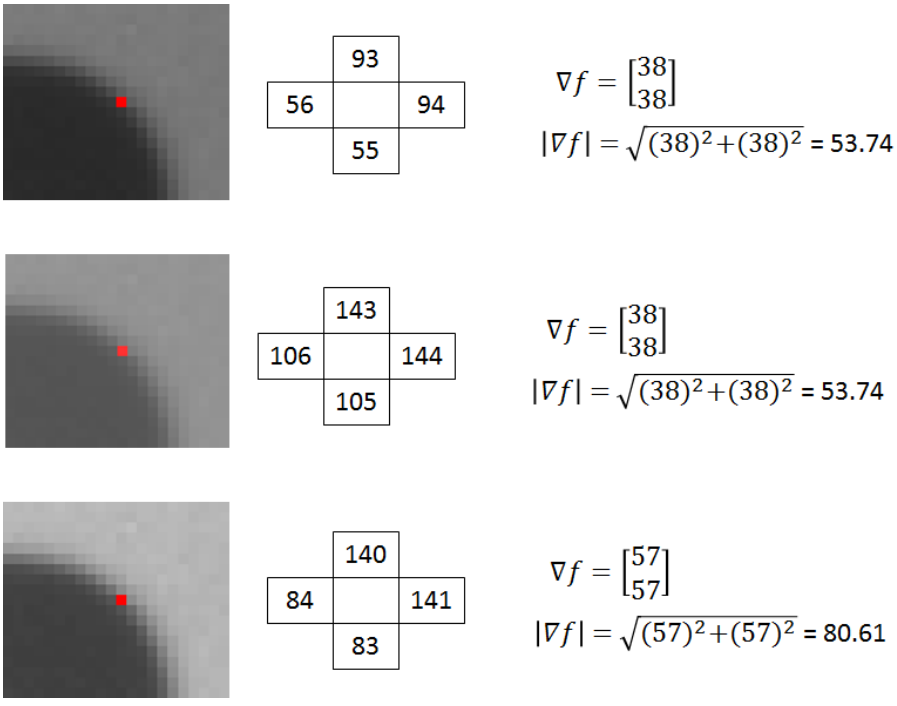


Figure 2.2: Gradient Vector Calculation

HOG feature Extraction

In our case study, the sliding window is divided into 5x5 pixels which form a cell, and then 3x3 cells in turn form a block. Each cell of 5x5 pixels is converted into a histogram consisting of 9 bins each of size 20°(following the size used by Dalal and Triggs in their paper). For each pixel its bin is found out by taking modulus with 180 of θ:

$\theta = arctan(g_y/g_x)$

The histogram is generated by adding the magnitude of gradient corresponding to this pixel to its assigned bin.HOG descriptor is obtained by adding all these histograms bins for all cells in a given block for a given image, called orientation histogram. Here a sliding window of size 40x100 pixel is composed of 6x18 blocks. Each block consists of 3x3 cells and a cell has a histogram of 9 discrete bins.

Thus,

$$Totalno.ofHOGfeatures = 6*18*3*3*9 = 8748$$

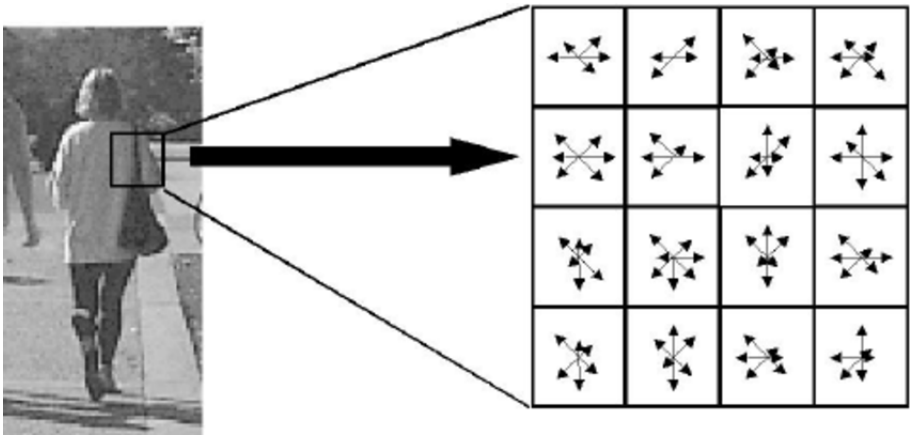


Figure 2.3: HOG Feature Extraction

Histogram Normalization

After generating the histogram, normalization is performed. This normalization step introduces better invariance to illumination, shadowing, and edge contrast.

2.1.3 Support Vector Machine

Support Vector Machine or SVM is a supervised ML algorithm for two-group classification problems. After giving an SVM model set of labeled training data for each category, they're able to categorize new text.

LinearSVC implementation is used as an SVM-classifier in this car detection work. Equation which is used to form the hyperplane is given by:

$$y = w^T x + b$$

where, w = SVM coefficients, x = HOG Feature vector, b = Bias term

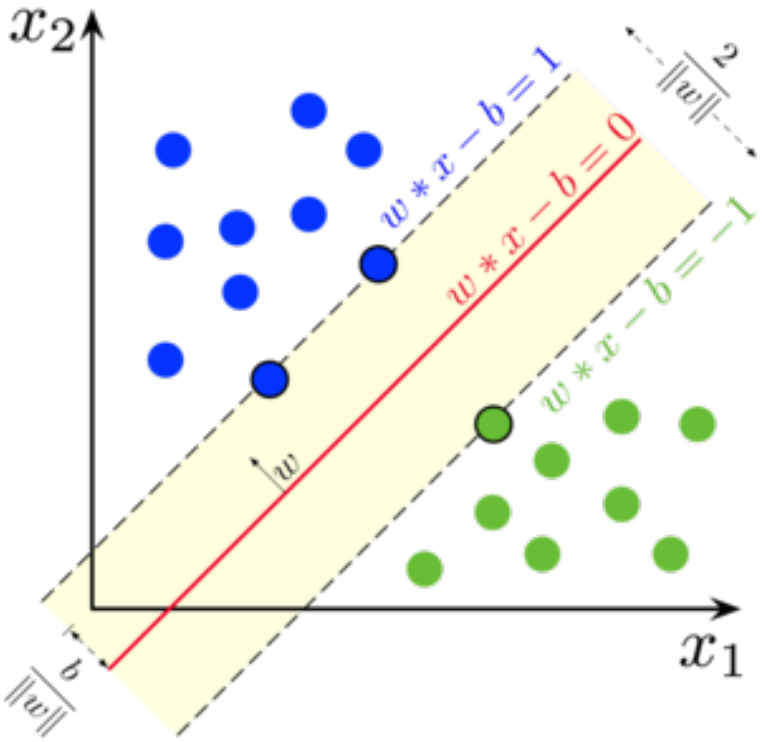


Figure 2.4: Linear SVM

Thus we are substituting the HOG features that we extract using the HOG algorithm, in the above equation to determine the value. If the value obtained from the above equation is above 0 then it is considered as a window consisting of a car otherwise not.

$$y > 0; Detected$$

$$y <= 0; NotDetected$$

2.1.4 Non maximum suppression

History of Oriented Gradients(HOG) combined with Support Vector Machines(SVM) has been pretty successful for detecting objects in images, however multiple bounding boxes surrounding the objects in the image after detection. To replace all the rectangles with a single rectangle that is the best fit, we follow Non-maximum suppression(NMS) given by detectors.

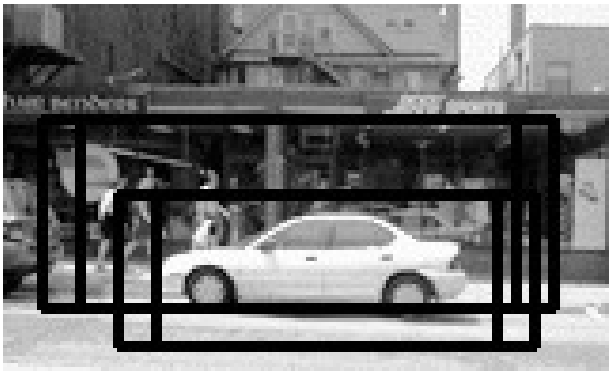


Figure 2.5: Multiple sliding windows that detected a car inside an image

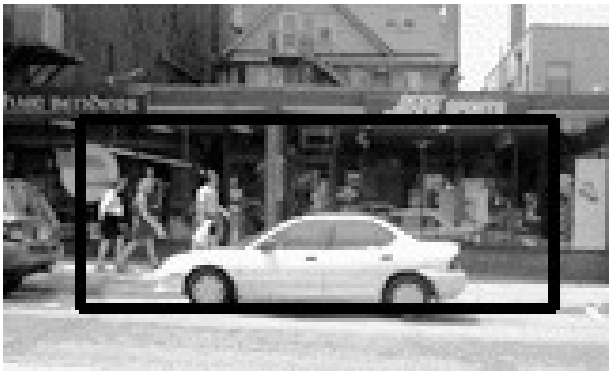


Figure 2.6: Filtered detected car sliding window using NMS

2.1.5 Linear interpolation of the arctangent function

Here I have used the technique followed by Neelam in her RnD project, where the orientation of the gradients are computed using the arctangent function:

$$arctan(y,x) = \begin{cases} atan(y/x) & \text{if } x > 0 \\ atan(y/x) + 180° & \text{if } x < 0 and y \geq 0 \\ atan(y/x) - 180° & \text{if } x < 0 and y < 0 \\ 90° & \text{if } x = 0 and y > 0 \\ -90° & \text{if } x = 0 and y < 0 \\ undefined, & \text{if } x = 0 and y = 0 \end{cases}$$

We use Linear interpolation which uses an LUT-based approach of a limited number of entries to consume less space. Each entry of LUT stores the value of arctan(x) (it gives principal angle) in degrees where x lies in [0,1] with a resolution of 0.01 in x.

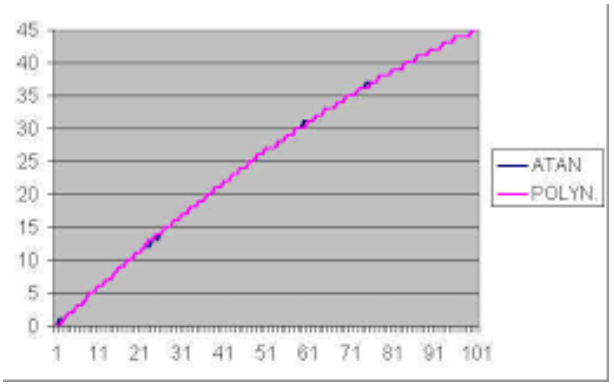


Figure 2.7: Linear Interpolation of the arctangent function

The arctangent computation can be done using Linear Interpolation as:

$$\begin{cases} input = |y/x| \\ input\_1 = round(input * 100) \\ index = input\_1 + 1, \\ angle\_new = LUT(index) + (input * 100 - input\_1)(LUT(index+1) - LUT(index)) \end{cases}$$

This is one of the fast methods of approximating arctangent computation in heterogenous

embedded applications.

2.2 Case Study 2: Face Detection using Viola-Jones algorithm

Viola-Jones algorithm is named after two computer vision researchers who proposed the method in 2001, Paul Viola and Michael Jones in their paper, “Rapid Object Detection using a Boosted Cascade of Simple Features”. The Viola-Jones is quite powerful, and the algorithm works exceptionally well in real-time face detection.

Given a grayscale input test image, the algorithm takes into consideration small regions and generates specific features in that region to detect a face. The small regions can be anywhere in that image and size is also scalable since an image can contain many faces of various sizes The algorithm used Haar-like features to detect faces in the test images.

The Viola-Jones algorithm has four main steps as follows:

- Selection of Haar-like features
- Integral image Generation
- AdaBoost Algorithm training
- Cascading Classifiers

2.2.1 Selection of Haar-like features

Certain specific image features are used in object recognition in an image. We know that faces share some unique properties, for example in a human face, the eyes region is darker than its neighbor pixels, and the nose region is brighter than the eye region. A Haar-like feature comprises light and dark regions, which produces a single value by taking the sum of the intensities of the light regions and subtracting that by the sum of the intensities of dark regions. There are many various types of Haar-like features but the Viola and Jones Framework only uses the ones in Figure 2.8. The various types of these features help us to extract useful information from an image such as edges, straight lines, and diagonal lines that can be used to detect a human face.

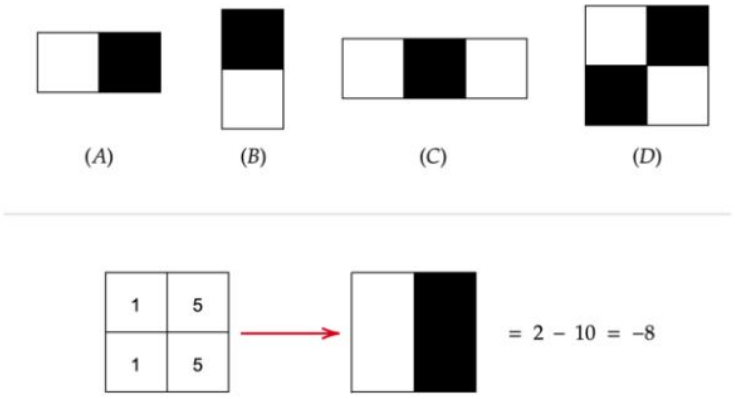


Figure 2.8: Haar-like features (top) and how to calculate them (bottom)

2.2.2 Integral image Generation

Integral Image is a type of image representation where the pixel value at location (x, y) on the integral image equals the sum of the pixels above and to the left (inclusive) of the (x, y) location on the original image (Viola and Jones, 2001). It is essential since it allows fast calculation of rectangular regions while extracting the haar features.

For example, Figure 2.9 shows that the sum of the red region D can be calculated very fast using the integral image. We know that the process of extracting haar features consists of summing up of dark/light rectangular regions, the introduction of Integral Images greatly reduces down the time needed to calculate those features.

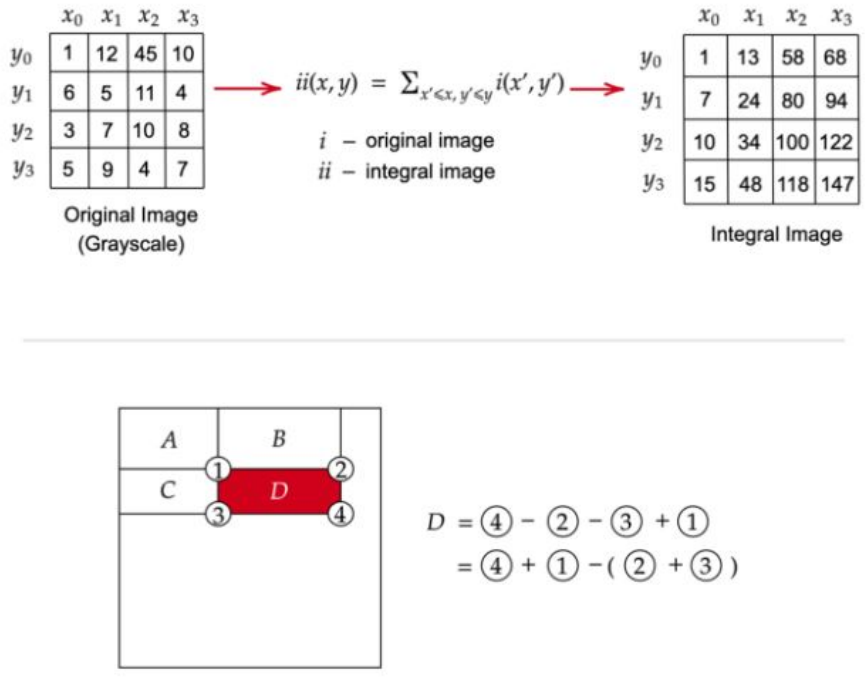


Figure 2.9: Conversion of the original image to integral image (top) and how to calculate a rectangular region using an integral image (bottom)

2.2.3 AdaBoost Algorithm training

The AdaBoost Algorithm is an ML algorithm that selects the best subset of features among all available features. We know that while selecting a small subregion and calculating the haar features, for example nearly 160,000 features are present for a 24×24 detector window, but only a handful of these features are important to identify a face. So we use the AdaBoost algorithm to identify the best features in those 160,000 features. The output of the algorithm is a classifier called a "Strong Classifier". A Strong Classifier is made up of linear combinations of "Weak Classifiers".

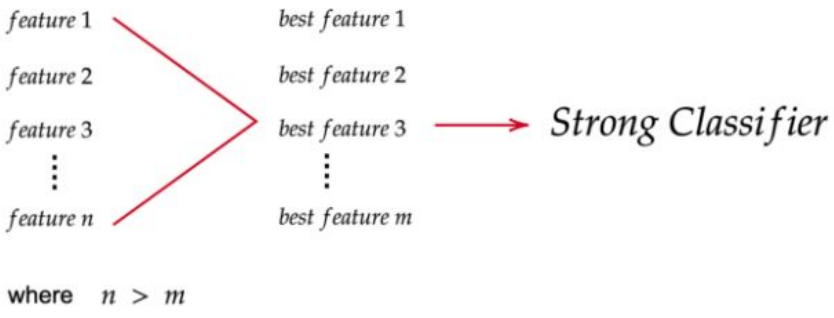


Figure 2.10: The AdaBoost algorithm is to extract the best features from n features

2.2.4 Cascading Classifiers

Although the number of features is reduced drastically by using the Adaboost algorithm, still there will be a significant amount of features, for example, let's consider 3000 best features are extracted from those 160,000 features. Hence, it will take a significant amount of time to process those features. Thus in order to speed it up, we divide those features into multiple stages and cascade each stage sequentially. If a classifier for a specific stage outputs a negative result, the input is discarded immediately otherwise, the input is forwarded onto the next stage. This way, we can save time specifically for a negative image.

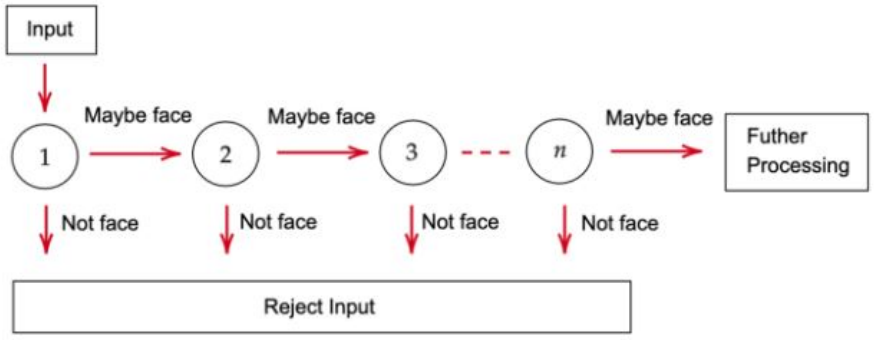


Figure 2.11: Cascade Classifier

Chapter 3

Implementation

In this chapter, we shall discuss the implementation of both the algorithm to extract the features for a given input image. Then we shall use those image features as input to the classifier to detect whether there is a car/face in that image. Initially, the algorithm is implemented in Python to extract the necessary features needed to compute the confidence value.

3.1 Case Study 1: Car detection using HOG Algorithm with SVM Classification

3.1.1 Python Implementation

Initially, python script[1] is used to perform the HOG-SVM algorithm to detect cars in the test image and extract the SVM coefficients and intercepts needed. The python script will first download images from a standard database, the UIUC Image Database for Car Detection is used to detect cars in an image. The SVM model files will be stored in the directory so that they can be reused later on.

There are mainly 5 modules for this implementation. Initially, the HOG algorithm is applied over the images to extract the HOG features. Then, a script is used to train the classifier using the sliding window technique to detect a car within an image and surround it with a rectangle. As multiple boxes can detect the car, hence we use NMS to remove all boxes and keep a single box that detects the car in the image.

[1] Appendix 7.1.1

We know the equation of a hyperplane -

$$y = w^T x + b$$

where, w = SVM coefficients, x = HOG Feature vector, b = Bias term

On successfully executing the python scripts, we can deduce the values of SVM coefficients(w) and the bias term(b) for detecting the car from the respective hog features.

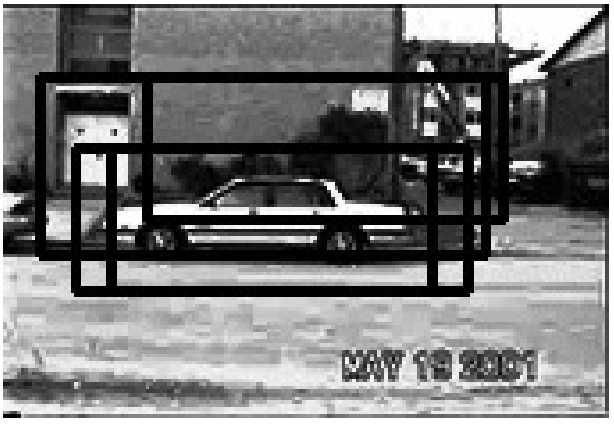


Figure 3.1: Car detection before applying NMS

Figure 3.2: Car detection after applying NMS

3.1.2 Design Implementation

After executing the python script we get the necessary coefficients needed in the hyperplane equation. Then we implement the algorithm in Xilinx Vivado to simulate it on softcore Microblaze processor. This part of the design implementation is followed from the RnD work of Neelam.

HOG-SVM accelerator algorithm Implementation

Algorithm 1 HOG-SVM accelerator algorithm flow

1: **procedure** HOGHLS(*image_base_address*) ▷ Returns the confidence of a car detected in a sliding window
2: *DEFINE image_arr* ▷ Stores the pixels of size 40×100 from BRAM
3: *image_arr* $\leftarrow$ *Pixel values copied from memory with image_base_address*
4: *COMPUTE g_row g_col* ▷ Compute the gradients in x- and y-directions
5: *COMPUTE gradient_magnitude and gradient_orientation* ▷ Values are computed for each pixel in at (i,j)
6: *COMPUTE orientation_histogram* ▷ Bins the magnitude to their corresponding orientation range bins
7: *PERFORM normalization to orientation_histogram*
8: *OBTAIN confidenceScore from SVM − classification* ▷ SVM-classification is performed using orientation_histogram as features
9: **return** *confidenceScore*

The above[2] showcases the steps of the algorithm needed to detect the presence of a car in an image. Here, the image pixels stored in the memory are given as the input, from which the hog features are extracted by using the image gradient and orientation and plotting it on a histogram of 9 bins. Then, the features are given as the input to the SVM classifier to detect confidence score for the presence of a car.

[2]Source code borrowed from Neelam Sharma's RnD Project

Implementation of approximate arctan2

Algorithm 2 Approximation of arctangent2() algorithm

```
1:  procedure ARCTAN2(x, y)                ▷ Returns the approximate value of arctan2 function
2:      if x == 0 then
3:          if y > 0 then
4:              return 90°
5:          else if y < 5 then
6:              return -90°
7:          else
8:              return 0°
9:      tan_inv_lut[101] ← Initialization
10:     ratio ← y/x
11:     abs_ratio ← |ratio|
12:     is_inverted ← False
13:     if abs_ratio > 1 then
14:         is_inverted ← True
15:         ratio ← 1/ratio
16:         abs_ratio ← |ratio|
17:     lut_index ← [100 * abs_ratio]
18:     lut_index_1 ← lut_index + 1
19:     COMPUTE principal_angle                ▷ Done using linear interpolation
20:     arcTanResult ← principal_angle         ▷ Stores the result of arctan2() of this function
21:     if ratio < 0 then
22:         arcTanResult ← 90° − principal_angle
23:     else
24:         arcTanResult ← −90° − principal_angle
25:     if x < 0 and y >= 0 then
26:         arcTanResult ← 180° + arcTanResult
27:     else
28:         arcTanResult ← −180° + arcTanResult
```

The above[3] describes the arctangent algorithm necessary to get the orientation of the gradients by using linear interpolation. It is one of the fast methods used to determine the phase in embedded applications.

[3]Source code borrowed from Neelam Sharma's RnD Project

3.1.3 Vivado Block Design

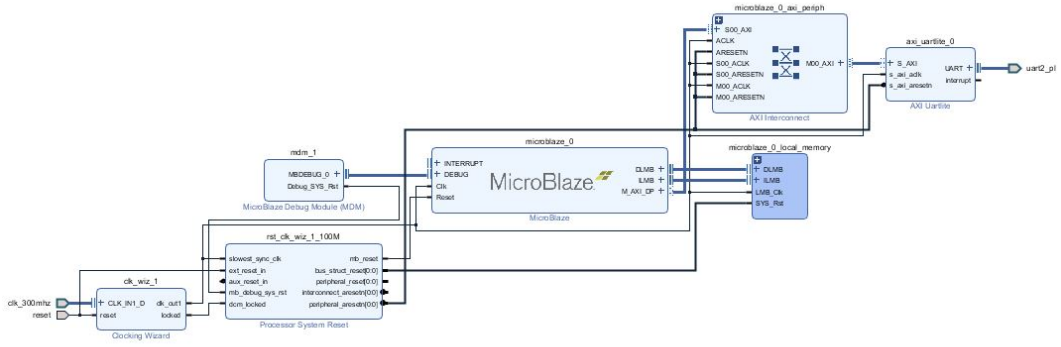


Figure 3.3: Block Design in Vivado

So, the entire algorithm is implemented in Vivado and integrated with the Microblaze softcore processor. Here, the algorithm is running completely on the soft core processor, just as a normal sequential C code program. At the next step, we need to determine the power consumption that it takes.

3.2 Performance analysis based on different modifications to pragmas

3.2.1 Power Analysis in Vivado

Having designed the algorithm in python and then implementing the algorithm in Xilinx Vivado, integrating with softcore processor Microblaze, comes the part where we will take into consideration of power dissipation involved.

Before going into the power analysis of the algorithm, I did a power analysis of a simple algorithm integrated with the Microblaze soft core processor.

The steps followed to execute and determine the power analysis:

- Initially create the block design and run the connection automation to add all the components.
- Then created a HDL wrapper class for the design and exported the hardware.
- Then launched the SDK and wrote the code corresponding to the execution in the design.

- Generated the elf file corresponding to the code and created a testbench for the execution and associated the .elf file.
- Ran the Behavioral simulation and got the respective output in the TCL window of Vivado and executed the Power Analysis

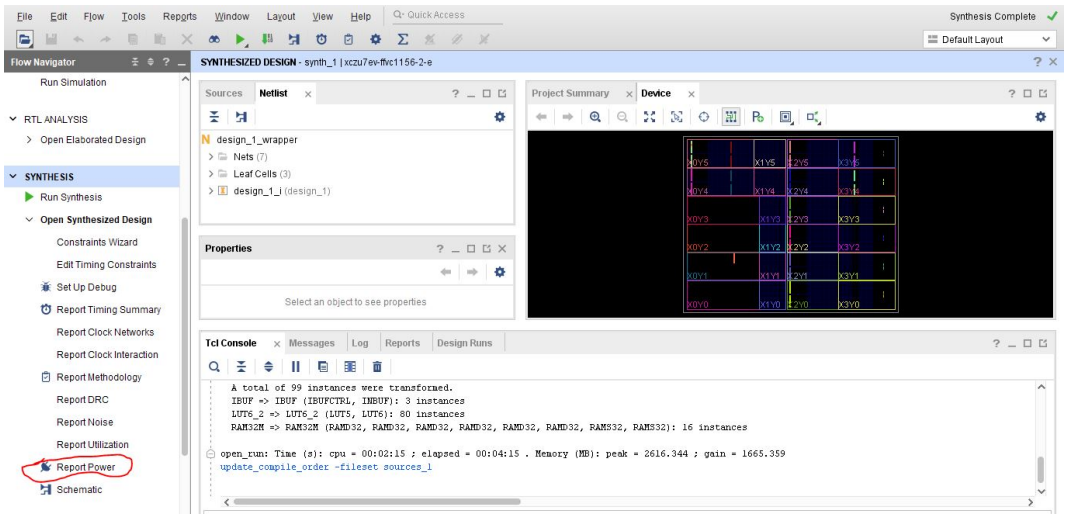


Figure 3.4: Synthesized Design in Vivado

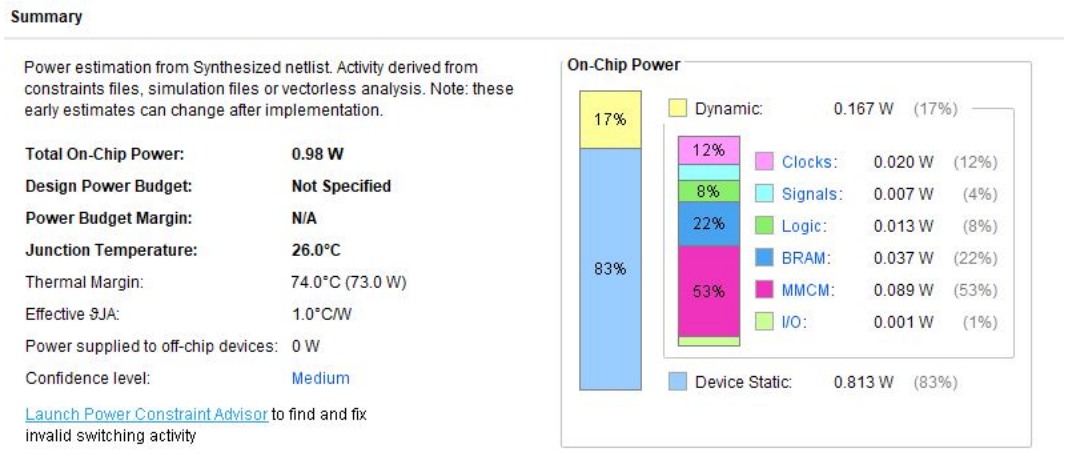


Figure 3.5: Power Analysis in Vivado

3.2.2 Performance metrics for various pragmas in Vivado HLS

The general behavior of Vivado HLS is to execute functions and loops in a sequential manner such that the hardware is an accurate copy of the C++ code snippet. Various directives can be used to enhance the performance of the hardware, allowing pipelining which significantly

increases the performance of the hardware. The main is to create a design with the highest possible performance, by reducing latency or resources.

These performance enhancements can be further accomplished by the use of pragmas in HLS. Vivado HLS provides pragmas that can be used to optimize the design: reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code. These pragmas can be added directly to the source code for the kernel.

The performance factors are as follows:

1. Design Latency: The latency of the design is the number of cycles it takes to output the result.

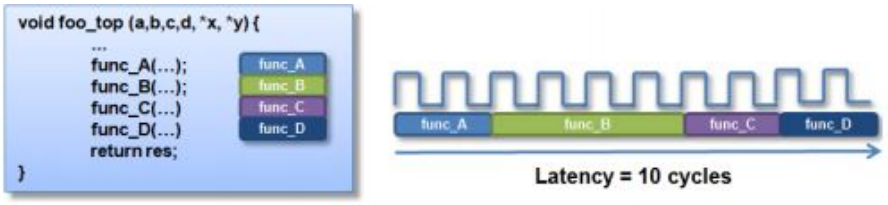


Figure 3.6: Design Latency

2. Design Throughput – The throughput of the design is the number of cycles between new inputs. By default (no concurrency) this is the same as the latency.

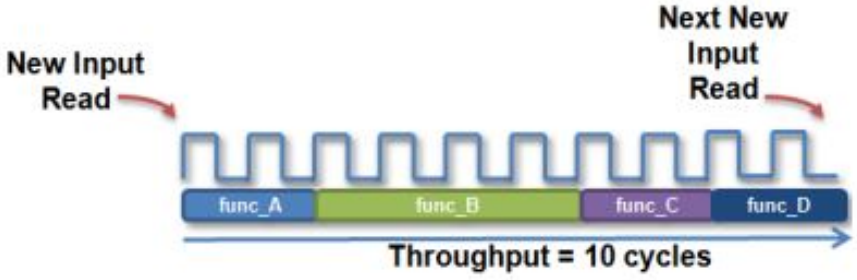


Figure 3.7: Design Throughput

In the design of HOG-SVM HLS IP, various modifications of the pragmas were made and compared to check for the performance metric of our system. The following pragmas were used in our design:

- pragma HLS array_partition
- pragma HLS interface
- pragma HLS unroll
- pragma HLS pipeline

pragma HLS array_partition

This pragma partitions an array into smaller arrays or individual elements which results in RTL with multiple small memories instead of one large memory. This improves the throughput of the design although requires more memory instances or registers.

```
    static float tan_inv_lut[101] =
            { 0, 0.5729387, 1.14576284, 1.718358, 2.29061004, 2.86240523,
                    3.43363036, 4.00417294, 4.57392126, 5.14276456, 5.71059314,
                    6.27729849, 6.84277341, 7.40691213, 7.96961039, 8.53076561,
                    9.09027692, 9.64804532, 10.20397372, 10.75796709,
                    11.30993247, 11.85977912, 12.40741853, 12.95276451,
                    13.49573328, 14.03624347, 14.5742162, 15.10957512,
                    15.64224646, 16.17215902, 16.69924423, 17.22343619,
                    17.74467163, 18.26288994, 18.77803322, 19.29004622,
                    19.79887635, 20.30447371, 20.80679101, 21.30578362,
                    21.80140949, 22.29362916, 22.78240573, 23.26770481,
                    23.74949449, 24.22774532, 24.70243023, 25.17352452,
                    25.64100582, 26.10485401, 26.56505118, 27.02158159,
                    27.47443163, 27.92358972, 28.36904629, 28.81079374,
                    29.24882634, 29.68314018, 30.11373315, 30.54060485,
                    30.96375653, 31.38319106, 31.79891282, 32.21092772,
                    32.61924307, 33.02386756, 33.42481118, 33.82208522,
                    34.21570213, 34.60567555, 34.9920202, 35.37475184,
                    35.75388725, 36.12944414, 36.50144112, 36.86989765,
                    37.23483398, 37.59627115, 37.95423088, 38.30873557,
                    38.65980825, 39.00747255, 39.35175263, 39.69267315,
                    40.03025927, 40.36453657, 40.69553104, 41.02326902,
                    41.34777722, 41.66908262, 41.9872125, 42.30219437,
                    42.61405597, 42.92282523, 43.22853026, 43.53119929,
                    43.83086067, 44.12754288, 44.42127443, 44.71208393, 45 };
#pragma HLS ARRAY_PARTITION variable=tan_inv_lut complete dim=0

    float ratio = (y / x);
    bool inverted = false;
    float abs_ratio = hls::abs(ratio);
    if (abs_ratio > 1) {
        inverted = true;
        ratio = 1 / ratio;
        abs_ratio = hls::abs(ratio);
    }
```

Figure 3.8: pragma HLS array_partition

pragma HLS interface

The INTERFACE pragma specifies how RTL ports are created from the function during interface synthesis. In C-based design, all input and output operations are performed, through function arguments. In an RTL design, these same input and output operations must be performed through a port in the design interface.

```
    if (y >= 0 && x < 0) {
        angle_result = 180 + angle_result;
    } else if (y < 0 && x < 0) {
        angle_result = -180 + angle_result;
    }

    return angle_result;
}


float hog_hls(float *image) {

    #pragma HLS INTERFACE m_axi depth=4020 port=image offset=slave
    #pragma HLS INTERFACE s_axilite port=return

    static float svm_coef[8748] = {0.012377273871147993,-0.008348374911842356,-0.0034249563195780

    #pragma HLS ARRAY_PARTITION variable=svm_coef cyclic factor=9 dim=1
```

Figure 3.9: pragma HLS interface

pragma HLS unroll

The UNROLL pragma transforms loops by creating multiple copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel. Using the UNROLL pragma you can unroll loops to increase data access and throughput. The UNROLL pragma allows the loop to be fully or partially unrolled.

```
//**************************** normalize_block_array() starts
float sum = 1e-10;
    for (int i = 0; i < 3; i++) {
#pragma HLS UNROLL factor=3
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < orientations; k++) {
                float val = block[i][j][k];
                val = val * val;
                sum += val;
            }
        }
    }

    sum = hls::sqrt(sum);
    float sum2 = 1e-10;
    for (int i = 0; i < 3; i++) {
#pragma HLS UNROLL factor=3
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < orientations; k++) {
            float block_val = hls::min(block[i][j][k] / sum, zero_pt_two);
                sum2 += (block_val * block_val);
                block[i][j][k] = block_val;
            }
        }
    }
```

Figure 3.10: pragma HLS unroll

pragma HLS pipeline

The PIPELINE pragma reduces the initiation interval for a function or loop by allowing the concurrent execution of operations. Pipelining a loop allows the operations of the loop to be implemented concurrently as shown in the following figure.

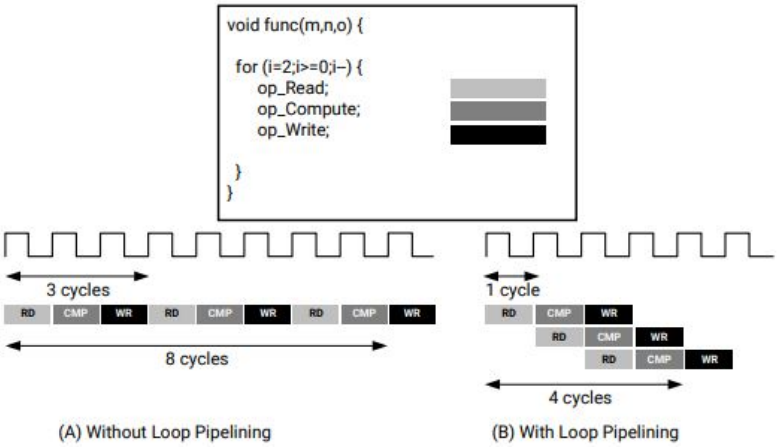
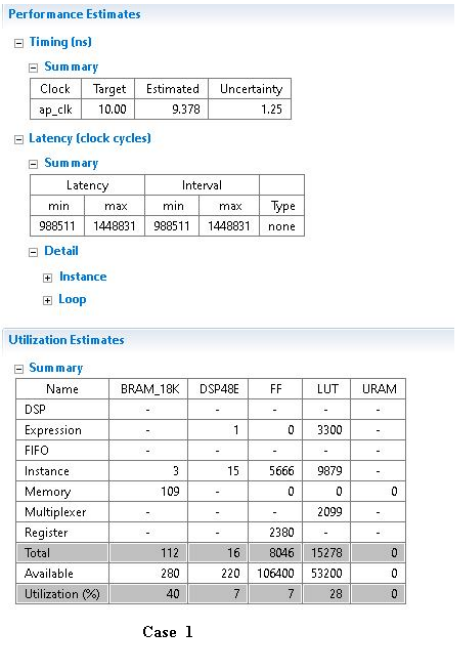


Figure 3.11: Loop pipeline

```
    for (cell_row = range_rows_start; cell_row < range_rows_stop; cell_row++) {
        cell_row_index = r + cell_row;
        if (cell_row_index >= 0 && cell_row_index < s_row) {


        for (cell_column = range_columns_start; cell_column < range_columns_stop; cell_column++) {
#pragma HLS PIPELINE
            cell_column_index = c + cell_column;
            if (cell_column_index >= 0 && cell_column_index < s_col && orientation[cell_row_index][ce:

                total = total + magnitude[cell_row_index][cell_column_index];
            }


        }
        }

    }

    total = (total / (c_row * c_col));
```

Figure 3.12: pragma HLS pipeline

Thus using those above pragmas within our HLS code, we performed various experiments to determine an IP with the best performance benefits and low power requirements.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.378	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
988511	1448831	988511	1448831	none

Detail

Instance

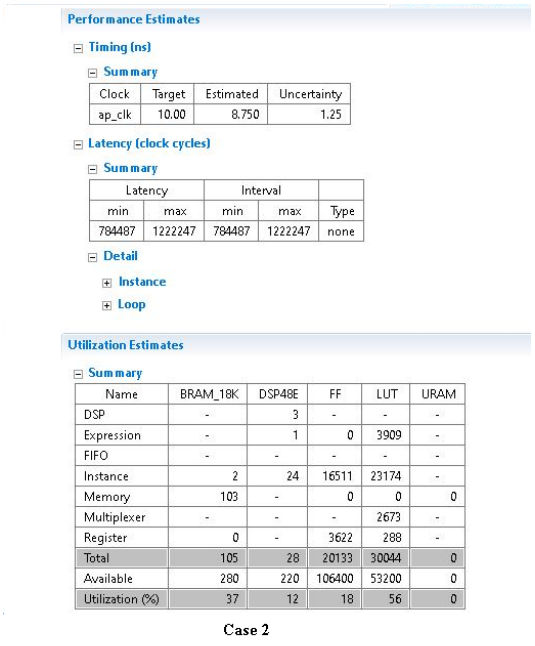
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	1	0	3300	-
FIFO	-	-	-	-	-
Instance	3	15	5666	9879	-
Memory	109	-	0	0	0
Multiplexer	-	-	-	2099	-
Register	-	-	2380	-	-
Total	112	16	8046	15278	0
Available	280	220	106400	53200	0
Utilization (%)	40	7	7	28	0

Case 1



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.750	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
784487	1222247	784487	1222247	none

Detail

Instance

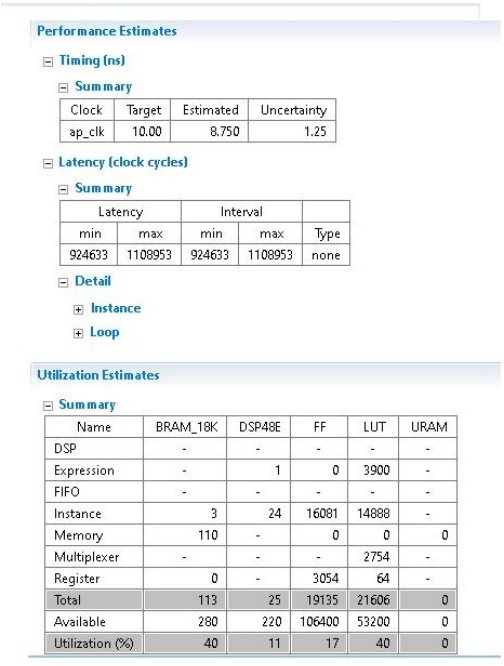
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	3	-	-	-
Expression	-	1	0	3909	-
FIFO	-	-	-	-	-
Instance	2	24	16511	23174	-
Memory	103	-	0	0	0
Multiplexer	-	-	-	2673	-
Register	0	-	3622	288	-
Total	105	28	20133	30044	0
Available	280	220	106400	53200	0
Utilization (%)	37	12	18	56	0

Case 2



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.750	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
924633	1108953	924633	1108953	none

Detail

Instance

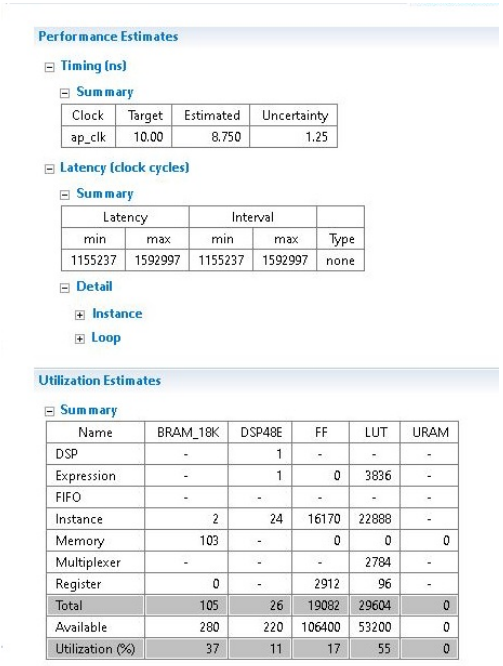
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	1	0	3900	-
FIFO	-	-	-	-	-
Instance	3	24	16081	14888	-
Memory	110	-	0	0	0
Multiplexer	-	-	-	2754	-
Register	0	-	3054	64	-
Total	113	25	19135	21606	0
Available	280	220	106400	53200	0
Utilization (%)	40	11	17	40	0

Case 3



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.750	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1155237	1592997	1155237	1592997	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	1	-	-	-
Expression	-	1	0	3836	-
FIFO	-	-	-	-	-
Instance	2	24	16170	22888	-
Memory	103	-	0	0	0
Multiplexer	-	-	-	2784	-
Register	0	-	2912	96	-
Total	105	26	19082	29604	0
Available	280	220	106400	53200	0
Utilization (%)	37	11	17	55	0

Case 4

In the above experiments, Case1 and case 2 are the extreme configurations. In case 1, the synthesis report corresponds to the HLS code in which there are no pragmas, other than the interface pragma, which can be easily noticed looking into the number of FFs and LUTS . Whereas in Case 2, it corresponds to the best configuration obtained in which the latency is the minimum of all, which corresponds to the performance metrics. Case 3 and are a few variations/experiments of case 2.

3.2.3 Integrating TCM memory and BRAM with Microblaze

Tightly coupled memories(TCM) are memories that have a dedicated interface to the microprocessor, and possess the high speed, low latency properties of cache memory.

The MicroBlaze processor uses a form of TCM with a LOW latency through a dedicated interface, the Local Memory Bus (LMB). As shown below, the processor has three external interfaces. M_AXI_DP is the data port to access the various peripheral registers. The other two, DLMB and ILMB are LMBs for data and instruction fetches by the CPU core, to which the local memory region is connected.

The BRAM connected through an AXI interface is an option as shown below. Two new buses are connected to an AXI SmartConnect, the MicroBlaze data and instruction cache ports.

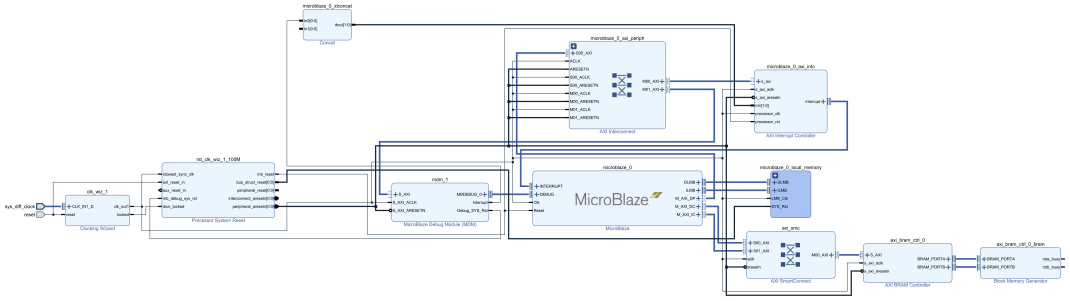


Figure 3.13: MicroBlaze with AXI block RAM memory

3.3 Integrating the AXI-Stream FIFO with Microblaze SOC

3.3.1 Theory

This design includes the use of the AXI Stream interface to simplify the data receiving and sending processes. The AXI4-Stream protocol is used as a standard interface to connect components that wish to exchange data. The interface can be used to connect a single master, that generates data, to a single slave, that receives data. The protocol can also be used when connecting larger numbers of master and slave components.

The stream protocol minimizes overhead by removing the need for addressing :

• Bus signals indicate when data is available, TVALID

• Receiver can optionally specify ready, TREADY

• Data is sent using TDATA

• Signal end of packet of data with TLAST

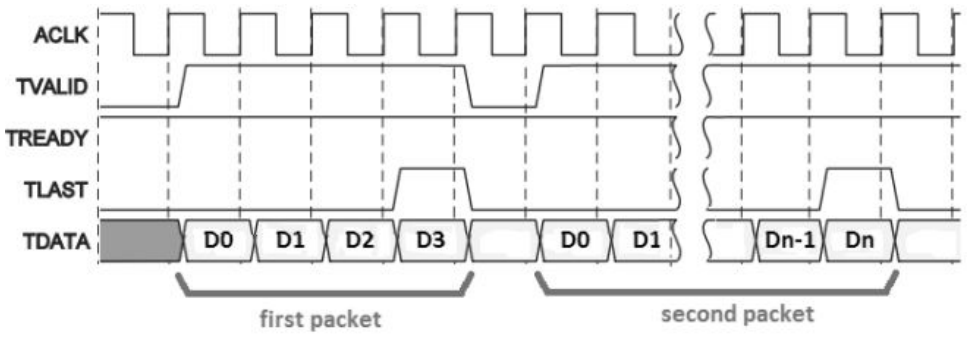


Figure 3.14: AXI Stream Signals

3.3.2 Design Flow

In this section, we will be integrating an AXI- Stream FIFO with a Microblaze processor along with an IP that does the HOG-SVM operation along with uart-receiver.

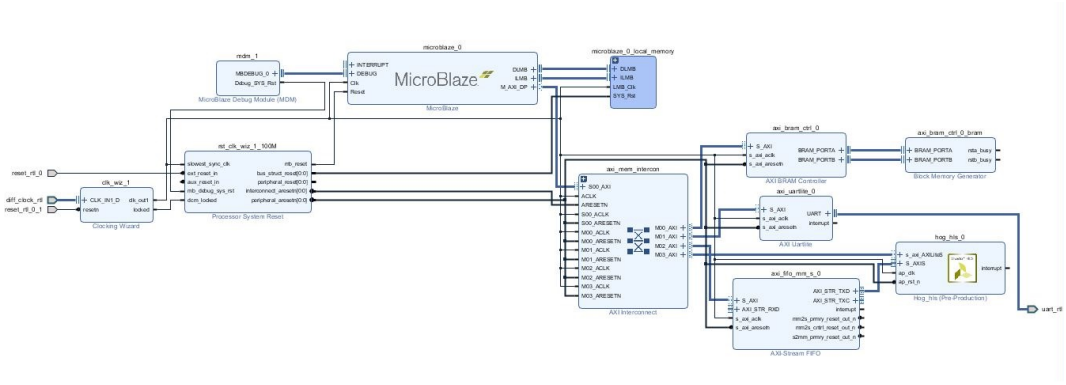


Figure 3.15: Block Diagram

3.4 Integrating the AXI-DMA with Microblaze SOC

3.4.1 Theory

The AXI Direct Memory Access (AXI DMA) IP core provides high-bandwidth direct memory access between the AXI4 memory mapped and AXI4-Stream IP interfaces. The primary high-speed DMA data movement between system memory and stream target is through the AXI4 Read Master to AXI4 memory-mapped to stream (MM2S) Master, and AXI stream to memory-mapped (S2MM) Slave to AXI4 Write Master.

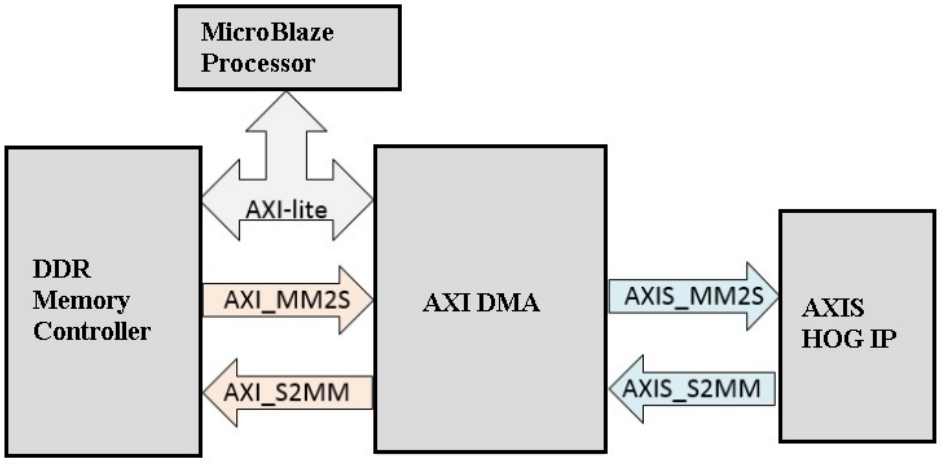


Figure 3.16: Block Diagram

Here, the AXI-lite bus allows the processor to communicate with the AXI DMA to setup, initiate and monitor data transfers. The AXI_MM2S and AXI_S2MM are memory-mapped AXI4 buses and provide the DMA access to the DDR memory. The AXIS_MM2S and AXIS_S2MM are AXI4-streaming buses, which source and sink a continuous stream of data, without addresses.

3.4.2 Design Flow

In this section, we will be integrating an AXI-DMA with a Microblaze processor along with an IP that does the HOG-SVM operation along with uart-receiver.

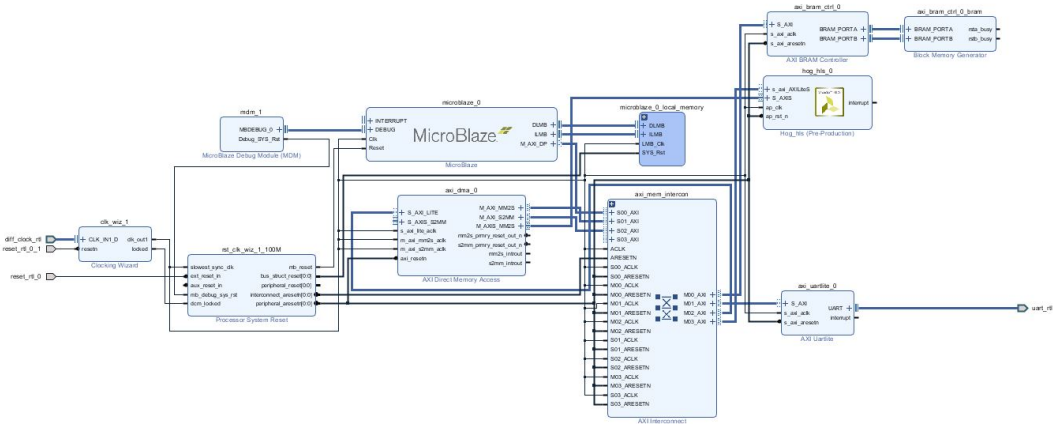


Figure 3.17: Vivado Block Design Diagram

3.5 Case Study 2: Face Detection using Viola-Jones algorithm

The algorithm is implemented assuming pixel-wise image data. Initially, we need to use the OpenCV haar classifiers XML file to extract the features using a python script. The algorithm flow is done by following a Swedish Thesis paper 'Design Exploration of an FPGA Based Face Detection Processing Core Utilizing High-Level Synthesis'.

3.5.1 Haar Feature Extraction

Using a python script[4] we extract the features present in the Haar classifiers XML file and store it in the BRAMs used in the design. The extracted XML data is then stored into variables based on their structure, and a C file is created and the data gets stored as a C array structure.

```
const int height = 24;
const int width = 24;
const int maxStageSize = 211;
const int stageNum = 25;

const int stageOrga[25] = {9, 16, 27, 32, 52, 53, 62, 72, 83, 91, 99, 115, 127, 135, 136, 137, 159, 155, 169, 196, 197, 181, 199,
 211, 200};

const float stageThresholds[25] = {-5.0425500869750 98, -4.9842400550842285, -4.6551899909973145, -4.453158855438232, -4
 .386458873748779, -4.129930019378662, -4.021809101104736, -3.883208990097046, -3.8424909114837646, -3.6478610038757324, -3
 .870048999786377, -3.7160909175872803, -3.5645289421081543, -3.702599048614502, -3.4265899658203125, -3.5125269889831543,
 -3.593964099884033, -3.3933560848236084, -3.2396929264068604, -3.2103500366210938, -3.2772979736328125, -3.31964111328125,
 -3.2573320865631104, -3.370300054550171, -2.992827892303467};

const float nodes[8739] = {-0.0315119996666690826, 2.087538003921509, -2.217210054397583, 0.012396000325679779, -1.863394021987915,
 1.327204942703247, 0.021927999332547188, -1.5105249881744385, 1.062572956085205, 0.005752999801188707, -0.8746389746665955,
 1.1760339736938477, 0.015014000236988068, -0.7794569730758667, 1.260841965675354, 0.09937100112438202, 0.5575129985809326,
```

Figure 3.18: Final output C file after XML extraction

3.5.2 Integral Image Calculation

The Integral Image calculation is done on the values that flow through three major buffers: the line buffer, the column buffer, and the Integral Image buffer. Initially, the buffers are given 0 value. The line buffer contains the column sum of 24 lines of the original image. The column buffer lies between the line buffer and the Integral Image buffer[5].

[4] Appendix 7.1.2

[5] Source code borrowed from Design Exploration of an FPGA Based Face Detection Processing Core Utilizing HighLevel Synthesis, by HUMAN SAMII MOGHADAM , STOCKHOLM, SWEDEN 2018

```
oldvalue = linebuff[0][newpixcol];
// loop through all rows but last (from top to bottom, or from old to new)
int row;
for(row = 0; row < 24; row++){
colBuffer[row]=linebuff[row][newpixcol];
// all rows content are based on the row below them (one higher index), except the last row
linebuff[row][newpixcol]=(row==23) ? (linebuff[row][newpixcol]-oldvalue)+newPixel:
linebuff[row+1][newpixcol]-oldvalue;
}
```

Figure 3.19: Procedure for updating line and column buffer

The Integral Image buffer calculates the integral image of the current window and is stored as a 1D array. The buffer gets updated when a detection window starts moving from the top left of the input image in a way that the first pixel of the image provides the bottom right pixel of the detection window.

```
for(row = 0; row < 24; row++){
oldinrow = ii[24*row];
int col;
// Loop through all values in a row except the last one
for(col = 0; col < 23; col++){
ii[(24*row) + col] = ii[(24*row) + col + 1] - oldinrow;
 }
ii[(24*row) + col] = ii[(24*row) + col] + colBuffer[row] - oldinrow;
}
```

Figure 3.20: Procedure for updating Integral Image buffer

3.5.3 Cascade Classification

Here it is checked whether the current Integral Image buffer contains a face. In this step of evaluation, the sum stored in the integral image buffer is compared to the node threshold, if it is smaller, then the left value is stored in a temporary variable, otherwise, the right value is stored . At the end of the evaluation of all the nodes, the variable is compared to the stage threshold, which detects a face if the temporary variable(stageTmp) reaches the stage threshold[6].

[6]Source code borrowed from Design Exploration of an FPGA Based Face Detection Processing Core Utilizing HighLevel Synthesis, by HUMAN SAMII MOGHADAM , STOCKHOLM, SWEDEN 2018

```
// calculate the area sum according to Viola-Jones and adding result to sum
sum += ((sumA + sumB) * weight);
}// looping through rects done

// evaluate current node
if(sum < nodeThresh){
stageTmp += lValue; // add left value to tmp if node threshold was not reached
}else{
stageTmp += rValue; // add right value to tmp if node threshold was not reached
}
} // looping through nodes done
// check if stageTmp reaches threshold, if yes a face is detected
if(stageTmp < stage24Thresh){
//printf("no Face!\n");
return 0;
}else{
//printf("face detected!\n");
return 1;
}
}
```

Figure 3.21: Procedure for detecting a face in the image

3.5.4 Functional Verification

Initially, a python script is used to provide a 1D array of the pixel of a test image in C format. The script can be called in the terminal by typing python pixel.py png-file. Another Python script is used to create golden coordinates which use the opencv-python library to detect faces in the image, and is implemented using the same XML file of the Haar features extraction. The result is appended to an array of the format (x1,y1,x2,y2,...). After that, all the above components are verified with a self-checking test bench, implemented in C[7] to maintain compatibility with Vivado HLS. The test bench calls the function for each pixel of the VGA test image and checks whether the return value detects a face. Whenever a face is detected, it stores the coordinates of the face in a stack and then compared them to "golden" coordinates.

```
int img[62500]={0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Figure 3.22: Sample pixel values of a test image

```
const int coordGoldFaces[2] = {181, 181};
```

Figure 3.23: Golden coordinates of the face detected in the image

[7]Source code borrowed from Design Exploration of an FPGA Based Face Detection Processing Core Utilizing HighLevel Synthesis, by HUMAN SAMII MOGHADAM , STOCKHOLM, SWEDEN 2018

Figure 3.24: Input image was given to the VJ algorithm

Figure 3.25: Obtained output after processing

3.6 Implementing the Viola-Jones algorithm in Intel Quartus

Here, the algorithm is implemented in Intel Quartus Design software, where the input image is taken from a camera module. The image format that is captured is of the format RGB(4:4:4) , ie., of the color depth of 12 bit per pixel, of 340x240 pixels. The captured image gets stored in an image buffer which has 76,800 addressable memory locations to store the 12 bit captured color image of resolution 320x240. Then the processing of the algorithm starts to take place in the fpga, i.e., start from the integral image generation to the output detection, where the RGB values can be found and can be displayed on a VGA display. Here, for the software simulation

part, I have replaced the camera module with a testbench, in which the image is directly loaded in the image frame buffer and output is given in the RGB format, which can be seen in the simulation[8].

3.6.1 System Design block

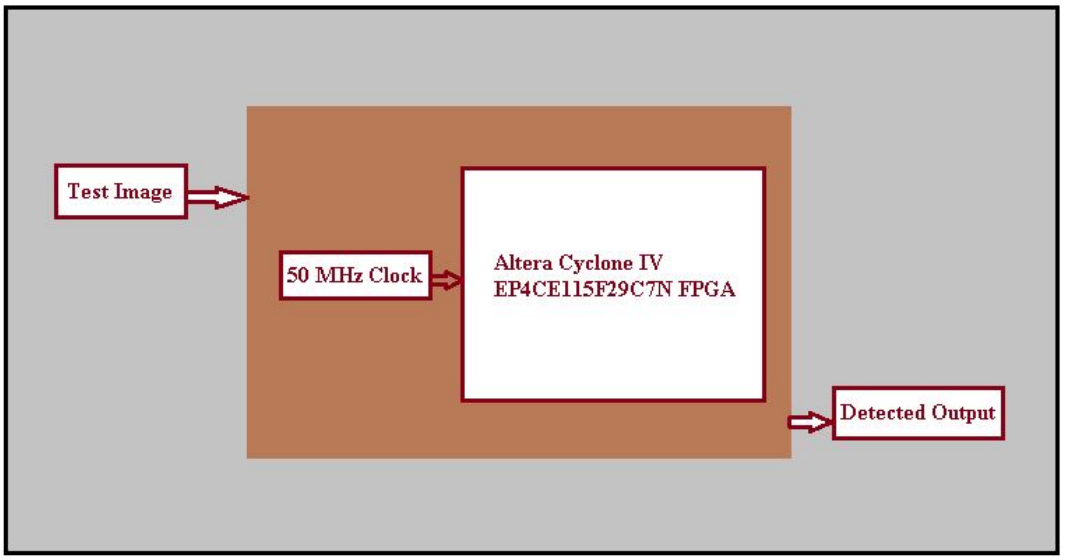


Figure 3.26: Block Diagram

3.6.2 Synthesized Netlist

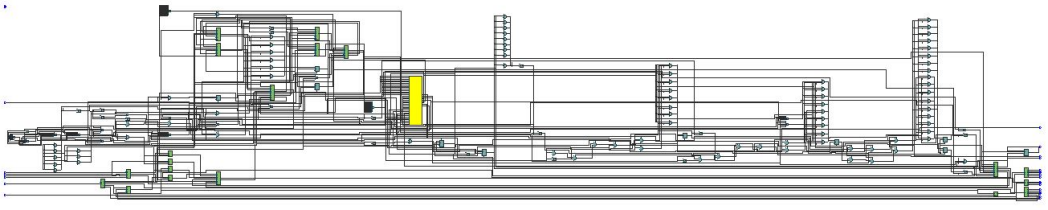


Figure 3.27: RTL Netlist

[8]Source Code borrowed from An efficient and cost effective FPGA based implementation of the Viola-Jones face detec-tion algorithm Peter Irgens, Curtis Bader, Theresa Lé, Devansh Saxena, Cristinel Ababei

3.6.3 Image buffer

As said before, we are backdoor loading the image buffer with a test image for the algorithm to start processing the image. Since, we are taking 340x240 images, hence the buffer has 76800 locations each of 12 bits.

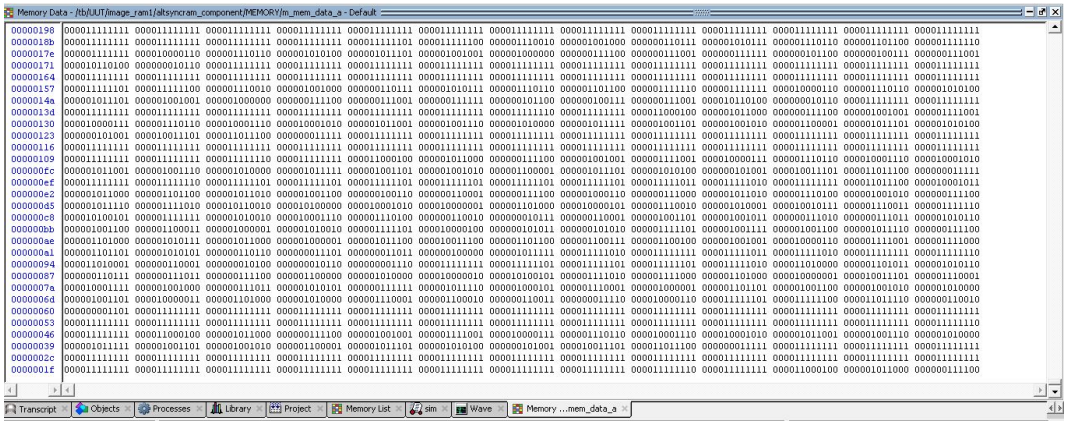


Figure 3.28: Image frame buffer

Chapter 4

Results

4.1 Simulation

4.1.1 C- Simulation

It provides a check on whether C++ code works correctly as per behavior. HLS performs this simulation considering it as a normal C++ program.

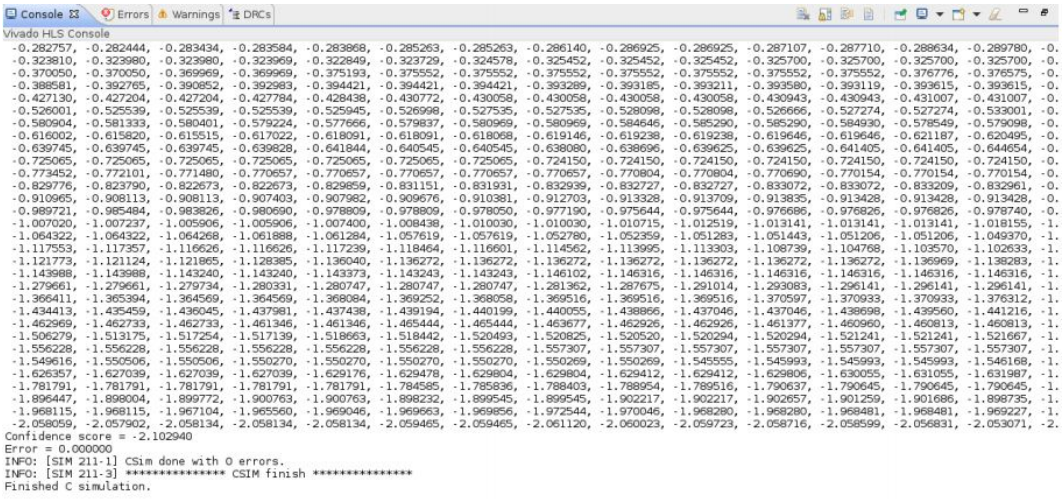
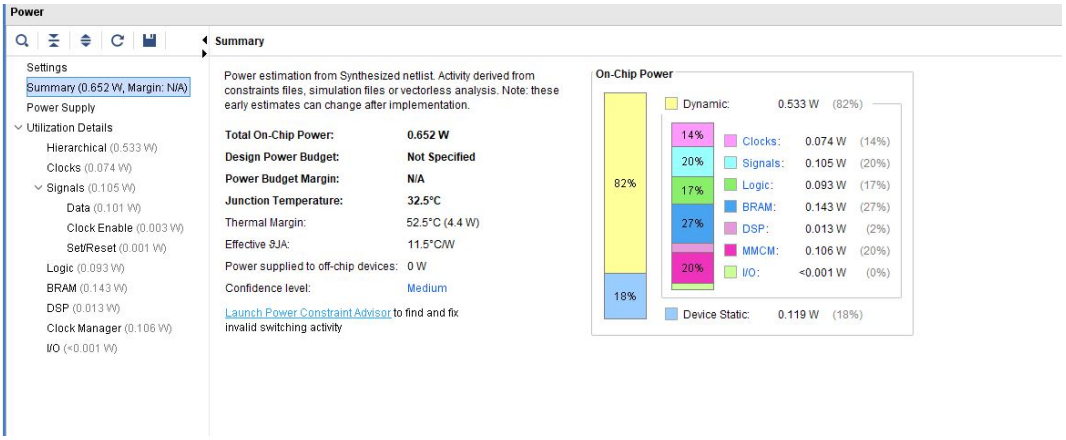


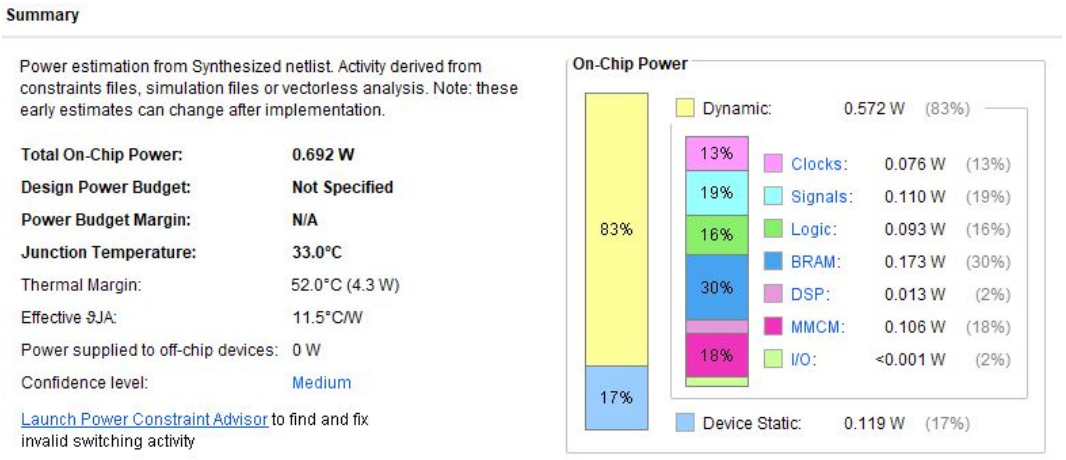
Figure 4.1: HLS C-simulation output of the HOG - SVM accelerator

4.1.2 Co-Simulation

It is a simulation result of the synthesized hardware from HLS and simulation is being performed using a C-testbench.

4.1.3 Power Analysis of few HOG- SVM pragma variations





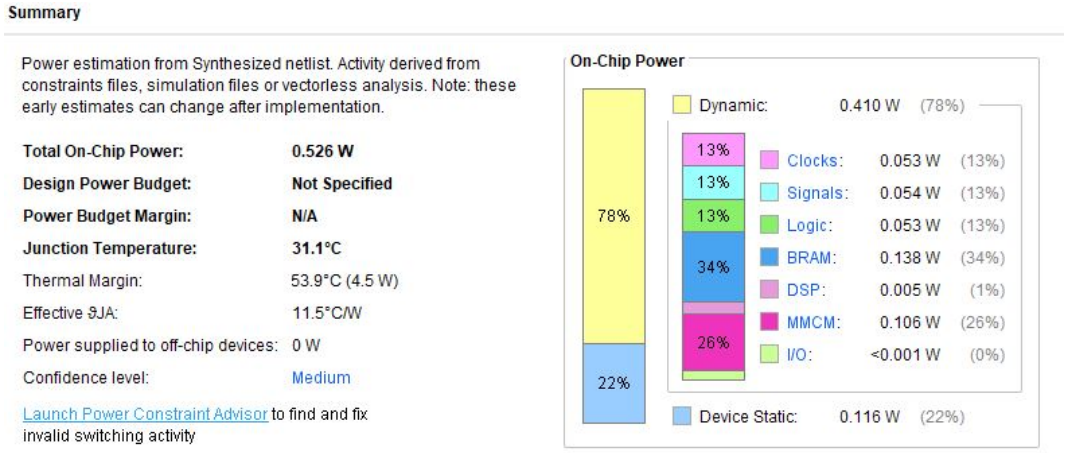


Figure 4.2: Case 3

4.1.4 Microblaze RTL Simulation

Provides simulation result of overall design using .elf file written for software that is expected to run on Microblaze.

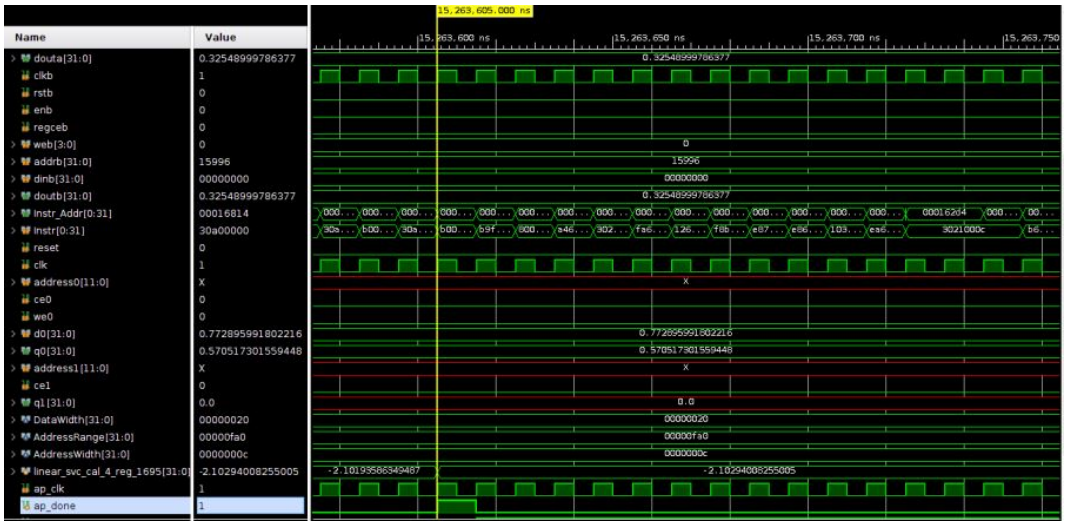


Figure 4.3: Microblaze RTL Behavioural simulation

4.1.5 Microblaze RTL Simulation of HOG-SVM with AXI- DMA

Provides simulation result of overall design integrated with stream HOG-SVM IP with AXI-DMA and using .elf file written for software that is expected to run on Microblaze.

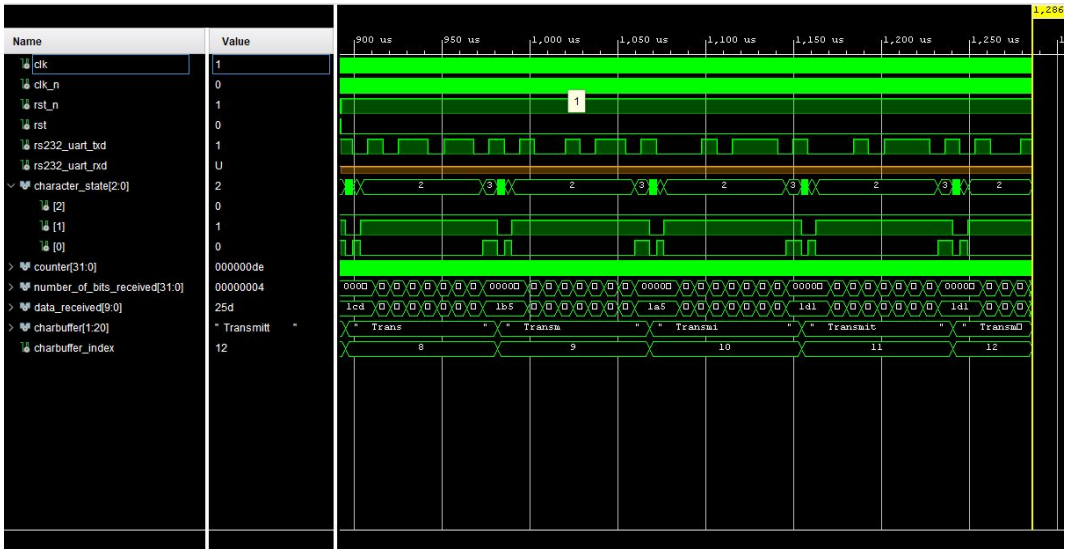


Figure 4.4: Microblaze RTL Simulation of HOG-SVM with AXI- DMA

4.1.6 Synthesis Report for HOG SVM Design

Name	BRAM 18K	DSP48E	FF	LUT
DSP	-	3	-	-
Expression	-	1	-	4074
FIFO	-	-	-	-
Instance	2	24	16713	18633
Memory	103	-	-	-
Multiplexer	-	-	-	2765
Register	-	-	3580	256
Total	105	28	20293	25701
Utilization	14	3	7	19

Figure 4.5: Utilization summary for HOG SVM

4.1.7 Synthesis Report for Viola-Jones Design

Name	BRAM 18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	1229	874
FIFO	-	-	-	-
Instance	-	6	775	698
Memory	41	-	64	12
Multiplexer	-	-	-	434
Register	-	-	752	-
Total	41	6	2820	2018
Utilization	5	~0	1	1

Figure 4.6: Utilization summary for Viola Jones

4.1.8 RTL Simulation Output of the Quartus project

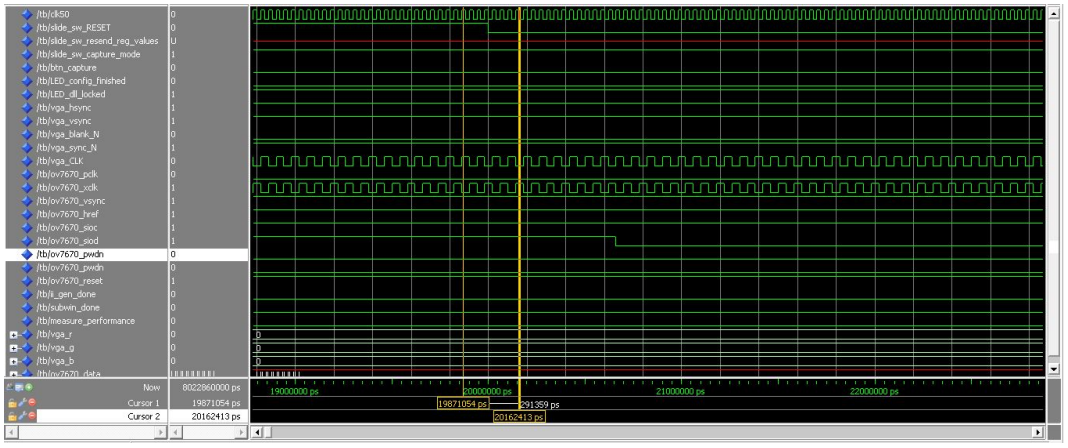


Figure 4.7: Simulation starts when Reset goes to 0

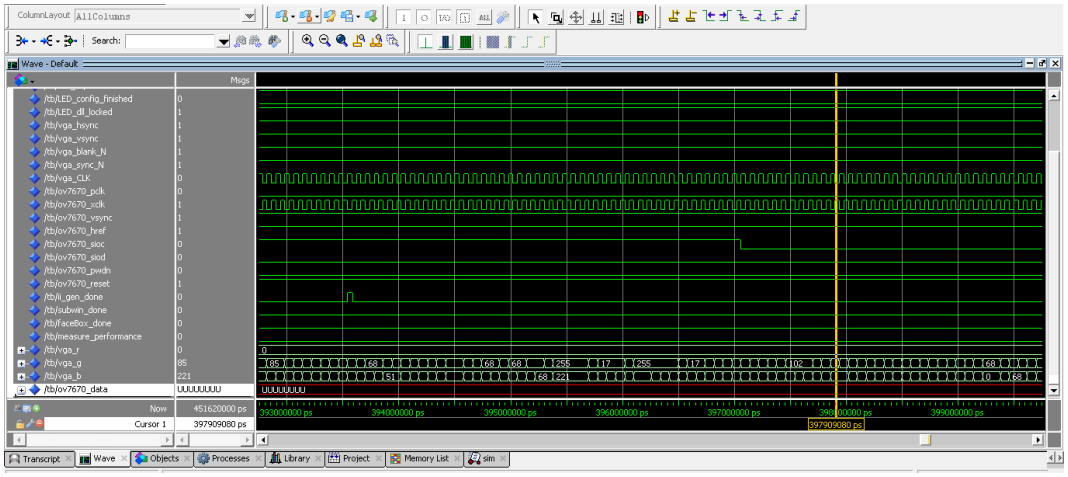


Figure 4.8: Processing going on as ii_gen done flag gets high for a single subwindow

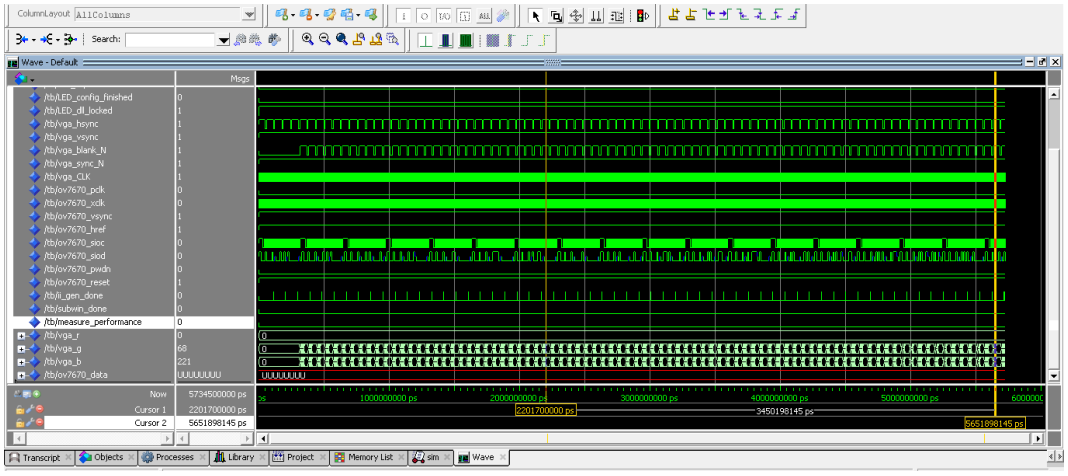


Figure 4.9: Output processing with multiple subwindows done

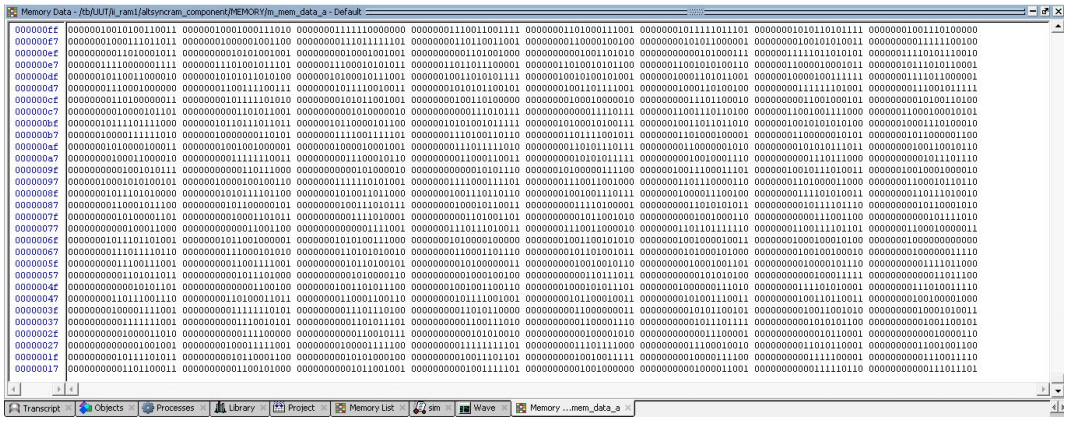


Figure 4.10: Integral Image buffers getting populated as processing goes on

Chapter 5

Tools and Programming Language

Initially, as a starting point dealt with various machine language terminology and used python to implement and understand those. After that, studied the HOG SVM and Viola-Jones algorithm, how to extract features, and then use those features in the classifier to detect the presence of a car. The entire algorithm is initially implemented in python and tested. From there, we get the necessary feature vectors along with the coefficients and the intercept. After that, the algorithm is completely rewritten in System C programming language and it is integrated with the Microblaze softcore processor in the Vivado Design Suite. After that, we create the block design in Vivado and run the Power Analysis. After that, the design is re-structured by integrating TCM memory and FIFO in the entire design.

Chapter 6

Conclusion and Future Work

This work presents the implementation of the HOGS-VM accelerator and Viola Jones accelerator using High-level-Synthesis (HLS) considering hardware and software co-design approach. HLS although providing one higher level of abstraction can provide comparable hardware performances to that provided by design using a Hardware Description Language(HDL). Both of them are then re written as completely on the software and compared to determine the differences in power and latency in Vivado 2019.1. Later on, integrated one of the designs with Fifos and TCM memory to make the design more robust and faster. In, future work, we can incorporate a DMA functionality integrated into the accelerator. The CPU will issue additional commands to the accelerator asking it to get images into its TCM before issuing commands to process said images. And then the entire design can be tested in the Xilinx FPGA.

Chapter 7

Appendix

7.1 Python Scripts

I have included the python source code in the appendix for reference.

7.1.1 HOG SVM Car Detection

HOG Feature extraction [1]

```
1  from skimage.feature import local_binary_pattern
2  from skimage.feature import hog
3  from skimage.io import imread
4  import joblib
5  import argparse as ap
6  import glob
7  import os
8  from config import *
9
10 def feature_extractor():
11
12     args = {}
13     args["path"]='../TrainImages'
14     args["descriptor"]="HOG"
15     path = args["path"]
16     cnt=0
17     des_type = args["descriptor"]
18
19     pos_feat = './features/pos'
20     neg_feat = './features/neg'
21
22     # If directories don't exist, create them
24
```

[1] Source code borrowed and modified from jianlong-yuan/HOG-SVM-python github repo available online

```
23      if not os.path.isdir(pos_feat):
25          os.makedirs(pos_feat)
26
27      if not os.path.isdir(neg_feat):
28          os.makedirs(neg_feat)
29
30
31      for i in os.listdir(path):
32
33       count=count+1
34        if os.path.isfile(os.path.join(path,i)) and 'pos' in i:
35
36          image_act_path=path+'/'+i
37          im = imread(image_act_path, as_gray=True)
38          if des_type == "HOG":
39              fd = hog(im, orientations, pixels_per_cell, cells_per_block, visualize=
                    visualize,transform_sqrt=transform_sqrt)
40          fd_name = os.path.split(i)[1].split(".")[0] + ".feat"
41          fd_path = os.path.join(pos_feat, fd_name)
42          joblib.dump(fd, fd_path)
43
44        elif os.path.isfile(os.path.join(path,i)) and 'neg' in i:
45          image_act_path=path+'/'+i
46          im = imread(image_act_path, as_gray=True)
47          if des_type == "HOG":
48              fd = hog(im, orientations, pixels_per_cell, cells_per_block, visualize=
                    visualize,transform_sqrt=transform_sqrt)
49          fd_name = os.path.split(i)[1].split(".")[0] + ".feat"
50          fd_path = os.path.join(neg_feat, fd_name)
51          joblib.dump(fd, fd_path)
52
53      print("Positive features saved in {}".format(pos_feat))
54
55      print("Negative features saved in {}".format(neg_feat))
56
57
58
59      print("Calculated features from training images")
60      print(count)
61
62  feature_extractor()
```

Training the Algorithm [2]

```
1  import joblib
2  import argparse as ap
3  import glob
4  import os

6
```

[2] Source code borrowed and modified from jianlong-yuan/HOG-SVM-python github repo available online

```
from config import *
import numpy as np
from sklearn.svm import SVC
from skimage.feature import local_binary_pattern
from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression

def trainer():

    args={"posfeat":'./features/pos', "negfeat": './features/neg','classifier':'LIN_SVM'}

    pos_feat_path =  args["posfeat"]
    neg_feat_path = args["negfeat"]

    # Classifiers supported
    clf_type = args['classifier']

    model_path = './svm_models.model'
    # model_path = 'svm.model'

    fds = []
    labels = []
    # Load the positive features
    for feat_path in glob.glob(os.path.join(pos_feat,"*.feat")):
        fd = joblib.load(feat_path)
        fds.append(fd)
        labels.append(1)

    # Load the negative features
    for feat_path in glob.glob(os.path.join(neg_feat,"*.feat")):
        fd = joblib.load(feat_path)
        fds.append(fd)
        labels.append(0)

    if clf_type == "LIN_SVM":
        clf = LinearSVC()
        print("Training a Linear SVM Classifier")
        clf.fit(fds, labels)
        # If feature directories don't exist, create them
        if not os.path.isdir(os.path.split(model_path)[0]):
            os.makedirs(os.path.split(model_path)[0])
        joblib.dump(clf, model_path)
        print("Classifier saved to {}".format(model_path))

trainer()
```

Non Minimum Suppression [3]

[3] Source code borrowed and modified from jianlong-yuan/HOG-SVM-python github repo available online

```
def overlapping_area(detection_1, detection_2):
     # Calculate the x-y co-ordinates of the rectangles
    x1_tl = detection_1[0]
    x2_tl = detection_2[0]
    x1_br = detection_1[0] + detection_1[3]
    x2_br = detection_2[0] + detection_2[3]
    y1_tl = detection_1[1]
    y2_tl = detection_2[1]
    y1_br = detection_1[1] + detection_1[4]
    y2_br = detection_2[1] + detection_2[4]
    # Calculate the overlapping Area
    x_overlap = max(0, min(x1_br, x2_br)-max(x1_tl, x2_tl))
    y_overlap = max(0, min(y1_br, y2_br)-max(y1_tl, y2_tl))
    overlap_area = x_overlap * y_overlap
    area_1 = detection_1[3] * detection_2[4]
    area_2 = detection_2[3] * detection_2[4]
    total_area = area_1 + area_2 - overlap_area
    return overlap_area / float(total_area)

def nms(detections, threshold=.5):

    if len(detections)==0:
        return []

    # Sort the detections based on confidence score
    detections = sorted(detections, key=lambda detections: detections[2],
            reverse=True)
    # Unique detections will be appended to this list
    new_detections=[]
    # Append the first detection
    new_detections.append(detections[0])
    # Remove the detection from the original list
    del detections[0]

     for index, detection in enumerate(detections):
        for new_detection in new_detections:
            if overlapping_area(detection, new_detection) > threshold:
                del detections[index]
                break
        else:
            new_detections.append(detection)
            del detections[index]
    return new_detections

if __name__ == "__main__":
    # Example of how to use the NMS Module
    detections = [[31, 31, .9, 10, 10], [31, 31, .12, 10, 10], [100, 34, .8,10, 10]]
    print("Detections before NMS = {}".format(detections))
    print("Detections after NMS = {}".format(nms(detections)))
```

Testing the Algorithm [4]

```
import sklearn
from skimage.transform import pyramid_gaussian
from skimage.io import imread
from skimage.feature import hog
import joblib

import argparse as ap
from nms import nms
from config import *


import numpy
import numpy as np
from sys import maxsize
from numpy import set_printoptions

set_printoptions(threshold=maxsize)

import cv2

file1 = open('hog_features_test.txt','w')

def print_like_array(im_window):

    print('{',end='')
    for i in range(im_window.shape[0]):
        # print('{',end='')
        for j in range(im_window.shape[1]):
            print(im_window[i][j],end='')

            if j!=im_window.shape[1]-1:
                print(',',end='')
            # else:
                # print('}',end='')
        if i!=im_window.shape[0]-1:
            print(',',end='')
        # else:
    print('}',end='')

def write_to_file(fd,coefficients,intercept):


    for i in range(fd.shape[0]):
        file1.write(str(fd[i]))
        file1.write('\n')


```

[4] Source code borrowed and modified from jianlong-yuan/HOG-SVM-python github repo available online

```
47      file1.write('coefs = \n')
49      for i in range(coefficients.shape[0]):
50          file1.write('{')
51          for j in range(coefficients.shape[1]):
52              file1.write(str(coefficients[i][j]))
53              if j!= coefficients.shape[1]-1:
54                  file1.write(',')
55              else:
56                  file1.write('}')
57          if i!=coefficients.shape[0]-1:
58              file1.write(',')
59
60      file1.write('\nintercept = '+str(intercept[0]))
61
62
63
64  def sliding_window(image, window_size, step_size):
65
66      for y in range(0, image.shape[0], step_size[1]):
67          for x in range(0, image.shape[1], step_size[0]):
68              yield (x, y, image[y:y + window_size[1], x:x + window_size[0]])
69
70  def main_func():
71
72      args={}
73      args["image"]='../TestImages/test-1.pgm'
74      args['downscale']=1.25
75      args['visualize']=False
76      model_path = 'svm_models.model'
77
78      im = imread(args["image"], as_gray=False)
79      # print('im: ',im)
80      np.savetxt('image_array.txt',im.flatten(),delimiter=',',newline=',')
81      # min_wdw_sz = (100, 40)
82      # step_size = (10, 10)
83      downscale = args['downscale']
84      visualize_det = args['visualize']
85
86      # Load the classifier
87      clf = joblib.load(model_path)
88
89      # List to store the detections
90      detections = []
91      # The current scale of the image
92      scale = 0
93
94      cnt=0
95      # Downscale the image and iterate
96      for im_scaled in pyramid_gaussian(im, downscale=downscale):
```

```
97          # This list contains detections at the current scale
98          cd = []
99          print('im.shape',im.shape)
100
101         if im_scaled.shape[0] < min_wdw_sz[1] or im_scaled.shape[1] < min_wdw_sz[0]:
102             break
103         for (x, y, im_window) in sliding_window(im_scaled, min_wdw_sz, step_size):
104             if im_window.shape[0] != min_wdw_sz[1] or im_window.shape[1] != min_wdw_sz
                    [0]:
105                 continue
106             # Calculate the HOG features
107             fd =hog(im_window, orientations, pixels_per_cell, cells_per_block, visualize=
                    visualize, transform_sqrt=transform_sqrt)
108
109             if cnt==0:
110                 print('Hog features shape: ',fd.shape[0])
111
112                 write_to_file(fd,clf.coef_,clf.intercept_)
113                 # print(im_window.shape)
114                 print_like_array(im_window)
115                 cnt+=1
116
117             fd = fd.reshape(1,-1)
118             pred = clf.predict(fd)
119             if pred == 1:
120                 print ("Detection:: Location -> ({}, {})".format(x, y))
121                 print('Decision function will now be called')
122                 print("Scale ->  {} | Confidence Score {} \n".format(scale,clf.
                        decision_function(fd)))
123                 detections.append((x, y, clf.decision_function(fd),
124                     int(min_wdw_sz[0]*(downscale**scale)),
125                     int(min_wdw_sz[1]*(downscale**scale))))
126                 cd.append(detections[-1])
127             # If visualize is set to true, display the working
128             # of the sliding window
129             if visualize_det:
130                 clone = im_scaled.copy()
131                 for x1, y1, _, _, _  in cd:
132                     # Draw the detections at this scale
133                     cv2.rectangle(clone, (x1, y1), (x1 + im_window.shape[1], y1 +
134                         im_window.shape[0]), (0, 0, 0), thickness=2)
135                 cv2.rectangle(clone, (x, y), (x + im_window.shape[1], y +
136                     im_window.shape[0]), (255, 255, 255), thickness=2)
137                 cv2.imwrite("Sliding_Window_in_Progress.pgm", clone)
138
139         # Move the the next scale
140         scale+=1
141
142     # Display the results before performing NMS
```

```
143     clone = im.copy()
144     for (x_tl, y_tl, _, w, h) in detections:
145         # Draw the detections
146         cv2.rectangle(im, (x_tl, y_tl), (x_tl+w, y_tl+h), (0, 0, 0), thickness=2)
147     cv2.imwrite("Raw_Detections_before_NMS.pgm", im)
148 
149     # Perform Non Maxima Suppression
150     detections = nms(detections, threshold)
151 
152     # Display the results after performing NMS
153     for (x_tl, y_tl, _, w, h) in detections:
154         # Draw the detections
155         cv2.rectangle(clone, (x_tl, y_tl), (x_tl+w,y_tl+h), (0, 0, 0), thickness=2)
156     cv2.imwrite("Final_Detections_after_applying_NMS.pgm", clone)
157 
158 main_func()
```

7.1.2 Viola Jones Face Detection

Parsing the Haar Cascade Classifiers XML file[5]

```
1  import sys # needed for open/close files
2  from string import maketrans
3  import xml.etree.ElementTree as ET # needed for xml parsing
4 
5  trans = maketrans('[]','{}')
6  if len(sys.argv) != 2:
7      print("Invalid number of arguments (",len(sys.argv),").\nUsage: python casconverter.
           py haar_cascade.xml")
8      sys.exit(-1)
9  xmlFilePath = sys.argv[1]
10 
11 tree = ET.parse(xmlFilePath)
12 root = tree.getroot()
13 
14 def createC():
15     print("Creating new file...")
16     cFilename = xmlFilePath[:-4] # use xml filename as template for generated file name
17     extension = "c"
18     try:
19         cFilename = cFilename + "." + extension
20         file = open(cFilename,"w")
21         file.write("// Hi there, this file is generated by the python script! \n\"\n\n")
22         file.close()
23         print("File created.")
24 
25 
```

[5]Source code borrowed and modified from Design Exploration of an FPGA Based Face Detection Processing Core Utilizing HighLevel Synthesis, by HUMAN SAMII MOGHADAM , STOCKHOLM, SWEDEN 2018

```
24      except:
26          print("Error: Could not create file.")
27          sys.exit(0)
28
29  #--------------------------------------------------------------------------------
30  # Creating File
31  createC()
32  cFilename = xmlFilePath[:-4] + ".c"
33  print("Opening C file: " + cFilename)
34  try:
35      file = open(cFilename,"a")
36      print("Inside CreateC")
37  except:
38      print("Error: Could not open file.")
39  #--------------------------------------------------------------------------------
40  TMP = root.find("./cascade/height")
41  height = int(TMP.text) # gets text of TMP and casts it as int and stores in "height"
42  file.write("const int height = "+str(height)+";\n")
43  TMP = root.find("./cascade/width")
44  width = int(TMP.text)
45  file.write("const int width = " + str(width) + ";\n")
46  #--------------------------------------------------------------------------------
47  # getting meta data
48  # maximum no. of nodes in a stage:
49  TMP = root.find("./cascade/stageParams/maxWeakCount")
50  maxStageSize = int(TMP.text)
51  file.write("const int maxStageSize = " + str(maxStageSize) + ";\n")
52  print("XML file reports that each stage has max " + str(maxStageSize) + " nodes.")
53  # maximum number of stages:
54  TMP = root.find("./cascade/stageNum")
55  stageNum = int(TMP.text)
56  file.write("const int stageNum = " + str(stageNum) + ";\n")
57  print("XML file reports number of stages: " + str(stageNum))
58  rectCountAll = 0 # count the number of total rects
59  nodes = []
60  nodesAdded = 0
61  stageOrga = []
62  stageCounter = 0
63  stageThresholds = []
64  featOrga = []
65  rectangles = []
66  #--------------------------------------------------------------------------------
67
68  # loop through all stages and store stage thresholds and nodes
69  for child in root.findall("./cascade/stages/"):
70      stageThresh = child.find('stageThreshold')
71      stageThresholds.append(float(stageThresh.text))
72      stageMaxWeakCount = child.find('maxWeakCount')
73      stageOrga.append(int(stageMaxWeakCount.text))
```

```
74      stageCounter += 1
75      for wClassifier in child.findall("./weakClassifiers/_"):
76          node = wClassifier.find('internalNodes').text
77          node = node.lstrip() # get rid of whitespace on the left side
78          node = node.split(' ') # seperate elements by whitespace (node becomes an array)
79          featIndex = int(node[2]) # extract feature index
80          nodeThresh = float(node[3]) # extract node threshold
81          leafVals = wClassifier.find('leafValues').text
82          leafVals = leafVals.lstrip() # Get rid of Whitespace on left side
83          leafVals = leafVals.split(' ') # Seperate elements by whitespace (leafVals
                becomes an array)
84          lVal = float(leafVals[0])
85          rVal = float(leafVals[1])
86          nodes.append(nodeThresh)
87          nodes.append(lVal)
88          nodes.append(rVal)
89          nodesAdded += 1
90
91  file.write("\nconst int stageOrga[" + str(stageCounter) + "] = " + str(stageOrga).
        translate(trans) + ";\n")
92  file.write("\nconst float stageThresholds[" + str(len(stageThresholds)) + "] = " + str(
        stageThresholds).translate(trans) + ";\n")
93  file.write("\nconst float nodes[" + str(nodesAdded*3) +"] = " + str(nodes).translate(
        trans) + ";\n")
94    # Debug Messages:
95  print("Step 1: " + str(stageCounter) + " stages processed.")
96  print("Step 2: " + str(len(stageThresholds)) + " stage thesholds found.")
97
98  if stageCounter != len(stageThresholds):
99      print("ERROR: The number of stages processed does not match the number of stage
            thresholds!")
100
101 print("Step 3: " + str(nodesAdded) + " nodes added.")
102  # loop through all features
103 rectCountInFeature = 0 # count the number of rects found in current feature
104
105 for currentFeature in root.findall("./cascade/features/"):
106     for currentRect in currentFeature.findall("./rects/"):
107         rectCountInFeature += 1
108         rectCountAll += 1
109         temp = currentRect.text
110         temp = temp.lstrip() # remove all whitespace on leftside
111         tempList = temp.split(' ') # seperate number by splitting at whitespace and store
                result to tempList
112         for i in range(len(tempList)):
113             try:
114                 tempList[i] = tempList[i].translate(None, '.')
115                 tempList[i] = int(tempList[i])
116             except ValueError:
```

```
117                 print("Problem with this String: " + tempList[i])
118         rectangles.append(tempList)
119     featOrga.append(rectCountInFeature)
120     rectCountInFeature = 0
121
122 file.write("\nconst int featOrga[" + str(len(featOrga)) + "] = " + str(featOrga).
        translate(trans) + ";\n")
123 file.write("\nconst int rectangles[" + str(rectCountAll) + "][5] = " + str(rectangles).
        translate(trans)+ ";\n")
124  # Debug Messages:
125 print("Step 4: " + str(len(featOrga)) + " features processed.")
126 print("Step 5: " + str(rectCountAll) + " rectangles added.")
127
128 file.close()
129 print("C file generated. Done!")
```

Capturing the golden coordinates of a face[6]

```
1  import sys # needed for open/close files
2  import numpy as np
3  import cv2
4
5  trans = {91:123,93:125}
6
7  if len(sys.argv) != 2:
8      print("Invalid number of arguments (",len(sys.argv),").\nUsage: python face2file.py
           VGAImage.png")
9      sys.exit(-1)
10
11 pngFilePath = sys.argv[1]
12 if pngFilePath[-4:] != ".png":
13     print("invalid argument: This script only works on files ending in .png")
14     sys.exit(-1)
15
16  ## Defintion of function: creating C-file for output
17 def createC():
18     print("Creating new file...")
19     cFilename = pngFilePath[:-4] # use png filename as template for generated file name
20     extension = "c"
21     try:
22         cFilename = cFilename + "." + extension
23         file = open(cFilename,'w')
24         file.write("// Hi there, this file is generated by the python script! \n\n")
25         file.close()
26         print("File created.")
27     except:
28         print("Error: Could not create file.")
29
30
```

[6]Source code borrowed and modified from Design Exploration of an FPGA Based Face Detection Processing Core Utilizing HighLevel Synthesis, by HUMAN SAMII MOGHADAM , STOCKHOLM, SWEDEN 2018

```
29          sys.exit(0)
31  ##-------------------------------------------------------------------------------
32
33 goldenFaces = []
34  # read Haar cascade and create classifier
35 face_cascade = cv2.CascadeClassifier('file.xml')
36  # read image
37 img = cv2.imread(pngFilePath)
38
39 if len(img.shape) > 2:
40     gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
41     print(gray)
42     print("Image was converted to grayscale.")
43 else:
44     gray = img
45     print("Image did not need conversion to greyscale.")
46
47  ## Creating File-----------------------------------------------------------
48 createC()
49 cFilename ="golden_coor_"+pngFilePath[:-4] + ".c"
50 print("Opening C file: " + cFilename)
51 try:
52     resFile = open(cFilename,"w")
53 except:
54     print("Error: Could not open file.")
55  ##-------------------------------------------------------------------------------
56
57 face = face_cascade.detectMultiScale(gray, scaleFactor = 1.1, minNeighbors = 1, minSize
       =(24,24))
58 print(gray[181][181])
59 counter = 0
60 for (x,y,w,h) in face:
61     roi_gray = gray[y:y+h, x:x+w]
62     filename = pngFilePath[:-4] + "_face" + str(counter) + ".png"
63     cv2.imwrite(filename, roi_gray)
64     goldenFaces.append(x+(w-1))
65     goldenFaces.append(y+(h-1))
66     counter = counter + 1
67
68 print(counter)
69 # print all coordinates of faces in form array = {x1,y1,x2,y2,...}
70 resFile.write("const int coordGoldFaces[" + str(len(goldenFaces)) + "] = " + str(
       goldenFaces).translate(trans) + ";\n")
71
72 resFile.close()
73
74 print("C file generated. Done!")
```

Converting the input image in a pixel file

```
import sys
import numpy as np
from PIL import Image

trans = {91:123,93:125}

if len(sys.argv) != 2:
    print("Invalid number of arguments (",len(sys.argv),").\nUsage: python face2file.py
        VGAImage.png")
    sys.exit(-1)
# Read file path of xml file provided
pngFilePath = sys.argv[1]

cFilename = "pixel_"+pngFilePath[0:-4]+ ".c"
print("Opening C file: " + cFilename)
try:
    file = open(cFilename,"w")
    print("Inside CreateC")
except:
    print("Error: Could not open file.")

im = np.array(Image.open(pngFilePath).convert('L'))

pngmod= "mod_"+pngFilePath
gr_im= Image.fromarray(im).save(pngmod)

result = im.flatten()
list1 = result.tolist()
print(len(list1))
print(np.size(im))
print(im.shape)

file.write(str(list1).translate(trans))
file.close()
```

Chapter 8

Bibliography

[1] N. Dalal and B. Triggs, Histograms of oriented gradients for human detection," in 2005 IEEE computer society conference on computer vision and pattern recognition(CVPR'05), vol. 1, pp. 886893, IEEE, 2005(paper available online).

[2] Paul Viola and Michael Jones. Robust real-time object detection." In: International Journal of Computer Vision 4.3447 (2001)(paper available online).

[3] Junguk Cho et al.Fpga-based Face Detection System Using Haar Classifiers." In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays. FPGA '09. Monterey, California, USA: ACM, 2009, pp. 103(paper available online).

[4] Design Exploration of an FPGA Based Face Detection Processing Core Utilizing High Level Synthesis, by HUMAN SAMII MOGHADAM, STOCKHOLM, SWEDEN 2018(paper available online).

[5] An efficient and cost effective FPGA based implementation of the Viola-Jones face detection algorithm Peter Irgens, Curtis Bader, Theresa Lé, Devansh Saxena, Cristinel Ababei, Published date - 2017 (paper available online).

[6] https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients.

[7] Joseph Howse. Haar Cascade for Face Recognition. Ed. by Andrey Pavlenko, Dmitry Kurtaev, and GitHub.

[8] https://www.mygreatlearning.com/blog/viola-jones-algorithm/.

[9] G. Bradski and A. Kaehler. Learning OpenCV: Computer Vision with the OpenCV Library. O'Reilly Media, 2008.

[10] https://monkeylearn.com/blog/introduction-to-support-vector-machines-svm.

[11] Neelam Sharma. RnD Project, IIT Bombay, June 2020.

[12] https://towardsdatascience.com/the-intuition-behind-facial-detection-the-viola-jones-algorithm-29d9106b6999.

[13] S. Agarwal, A. Awan, and D. Roth, UIUC Image Database for Car Detection, 2004.

[14] https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/.

[15] https://www.pyimagesearch.com/2014/11/10/histogram-oriented-gradients-object-detection/.

[16] https://medium.com/@mithi/vehicles-tracking-with-hog-and-linear-svm.

[17] A. Ukil, V. H. Shah, and B. Deck, computation of arctangent functions for embedded applications: A comparative analysis," in 2011 IEEE International Sym-posium on Industrial Electronics, pp. 12061211, IEEE, 2011.