Study of Hardware-Software Co-design of Vehicle(Car) detection using HOG and SVM

RnD Report

by

Neelam Sharma

(Roll No. 18307R030)

Under the guidance of

Prof. Sachin B. Patkar

Department of Electrical Engineering
Indian Institute of Technology Bombay

June 2020

Acknowledgement

I express my gratitude to my guide Prof. Sachin B. Patkar and High Performance Computing (HPC) Lab members for providing me with the opportunity to work on this topic and supporting me throughout the course of this project.

Neelam Sharma
Electrical Engineering
IIT Bombay

Abstract

HOG (Histogram of Oriented Gradients) along with SVM(Support Vector Machine) has been employed in many of the computer vision problems for object detection. HOG coarsely bins the gradients of the images to form the features. This method has outperformed many of the feature extraction techniques at lower computational complexity and lesser energy consumption. This work provides a study of design methodology for accelerating Hardware/Software co-design techniques to obtain better performance with flexibility for a given amount of resources. Implementation of a HOG-SVM accelerator for car detection in an image targeted for FPGA's with Zynq processing systems is done in this work.

Contents

List of Figures

Chapter 1

Introduction

Vehicle detection is becoming important in the field of self-driving cars and intelligent traffic management. For example, there are vehicle detection systems that make use of various sensors to determine the presence and movement of vehicles in traffic applications. Irresponsible vehicle drivers are a major cause of fatalities that occur on roads theses days. Vehicle detection can help in reducing these traffic accidents to an extent so it forms one of the most important research topics in computer vision. It is important to implement such real-world problems in real-time.
Histogram of oriented gradients, proposed by N. Dalal and B. Triggs [4] is one of the most effective features extraction algorithms when used along with SVM for classification at a lower computational complexity in comparison to convolutional neural networks.
Due to the parallelizable nature of HOG algorithm, it is more suitable to be implemented in hardware leaving pre-processing work to the software which consumes much lower time in comparison to that of algorithm.
This work provides a study of a systematic approach to be followed for a Software/Hardware Co-design using HOG with SVM as a study implemented on UIUC Image Database for Car Detection [5], downloaded from GitHub[6]. Original software implementation of code is taken from GitHub[7]. An accelerator is designed using Vivado HLS 2019.1 for implementing HOG and SVM flow of the algorithm and simulation results are shown for the same.

Chapter 2

Background

2.1 Histogram of oriented gradients (HOG)

HOG is a computer vision algorithms which is used for object localization and detection. Input to HOG algorithm is an image and it provides feature descriptors for a sliding frame inside an image. These feature descriptors are then fed to SVM to determine the presence of an object.

2.1.1 Overall flow

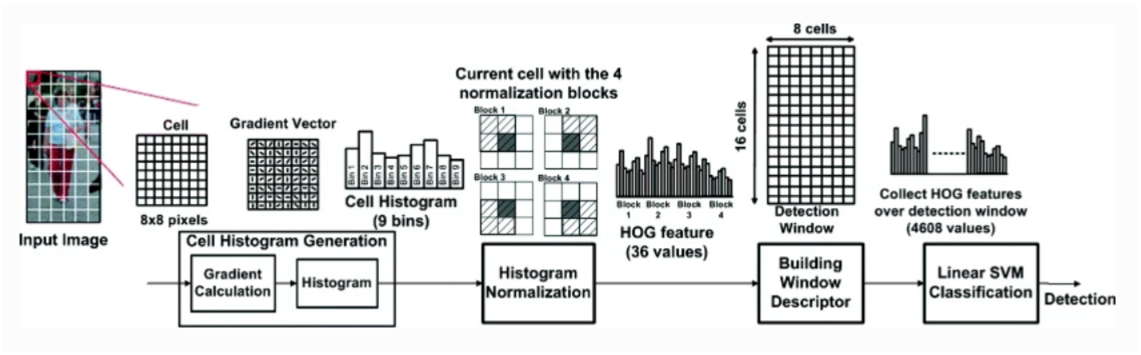


Figure 2.1: Object detection algorithm using HOG features[1]

In this implementation, a sliding window of size 40 pixels horizontally and 100 pixels vertically with a stride of 10 pixels in the horizontal direction and 10 pixels in the vertical direction scans the entire image as shown in Figure 2.2. For each window, it generates 8748 features which are passed to SVM classifier to determine whether there is a car in the given window. SVM classifier predicts the existence of a car with a confidence score.

Given the input, sliding image is successively passed through four stages of:

1. Gradient calculation
2. Histogram extraction
3. Histogram normalization
4. SVM classification

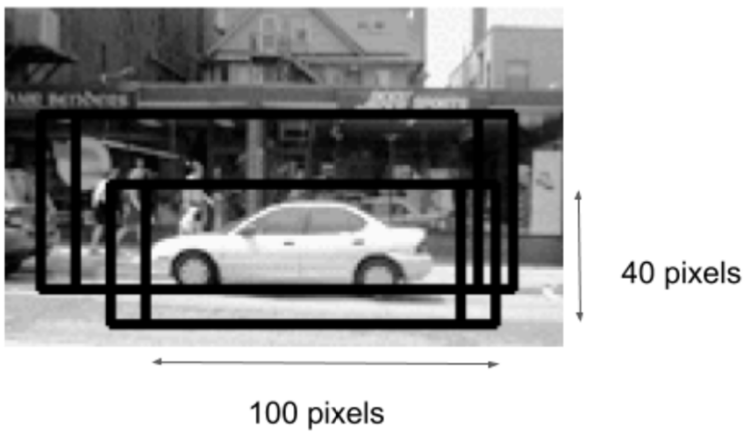


Figure 2.2: Sliding windows that detected car in an image

2.1.2 Preprocessing

The first stage applies an optional global image normalization equalisation that is designed to reduce the influence of illumination effects. In practice, gamma (power-law) compression is used for either computing the square root or the log of each colour channel. In this work square root is used to do the normalization. Image texture strength is typically proportional to the local surface illumination so this compression helps to reduce the effects of local shadowing and illumination variations.

2.1.3 Gradient vector of pixel magnitudes

The second stage computes the first-order image gradients. These capture contour, silhouette and some texture information, while providing further resistance to illumination variations.
A gradient points in the direction of the greatest rate of increase of the function. In computer vision problems we want to determine the magnitude and direction of the maximum change in pixel's intensity. The gradients in x and y directions are computed by convoluting with 1×3 and 3×1 sized kernels respectively to obtain $\mathrm{G_x}$ and $\mathrm{G_y}$.

$$K_x = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \quad K_y = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}^T$$

Magnitude and gradients are computed as:

$$\mid G(x,y) \mid = \sqrt{G_x^2 + G_y^2}$$

$$tan(x,y) = \frac{G_y}{G_x}$$

2.1.4 Histogram extraction

The sliding window is divided into 5×5 pixels forms a cell and 3×3 cells combine to form a block. Each cell of 5×5 pixels is converted into a histogram consisting of 9 bins each of size 20°. For each pixel its bin is found out by taking modulus with 180° of θ:

$$\theta = \arctan \frac{G_y}{G_x} \tag{2.1}$$

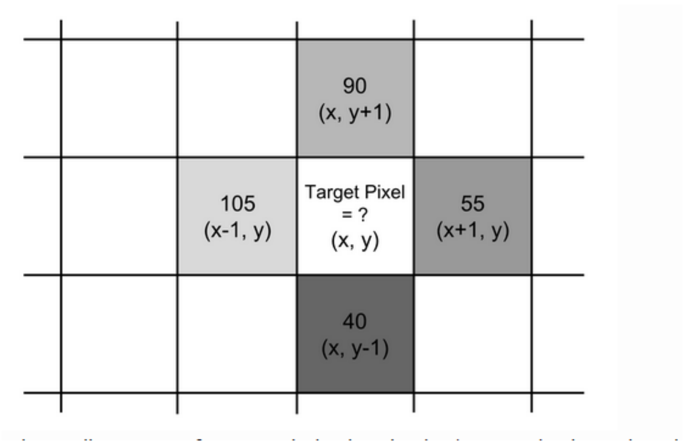


Figure 2.3: To compute the gradient vector of a target pixel at location (x, y), we need to know the colours of its four neighbours[2].

Contribution of a pixel to a bin is done by adding the magnitude of gradient corresponding to this pixel to its assigned bin obtained from Equation 2.1. HOG descriptor is obtained by concatenation of these histograms bins for all cells in a given block, called orientation histogram. Blocks overlap with each other with a block stride of (1,1).
A sliding window of size 40×100 pixel is composed of 6×18 blocks. Each block consists of 3×3 cells and a cell has a histogram of 9 discrete bins. Thus,

$$Total\ no.\ of\ features = 6 \times 18 \times 3 \times 3 \times 9 = 8748$$

2.1.5 Histogram Normalization

This is the final computation step of the HOG algorithm in which the orientation histograms are normalized over the blocks to which they belong. This normalization step introduces better invariance to illumination, shadowing, and edge contrast.

2.2 Support Vector Machine (SVM)

Support Vector Machine (SVM) is a supervised method in Machine Learning for classification and regression.
Classification using SVM is done in this work. Classification is done by determining a hyperplane that is used to differentiate among the classes.

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

$$y = w^T x + b \tag{2.2}$$

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

HOG descriptor which is obtained from the HOG algorithm is used to form the features (vector x in Equation 2.2). Values of w and b in Equation 2.2 are determined at the time of training. Result of the above equation is used to determine the confidence score of classifying it between window consisting of a car or not by comparing it with a threshold.

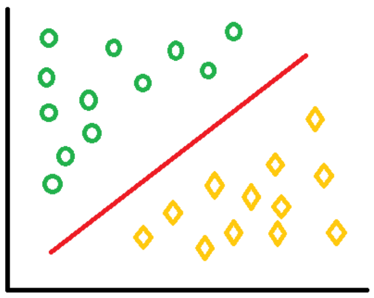


Figure 2.4: Linearly separable data points with two classes

If the confidence obtained from the above equation is above 0 then it is considered as a window consisting of a car otherwise not.

$$\begin{aligned} y > 0;&\ \ Detected\ car(s) \\ y <= 0;&\ \ Detected\ no\ car \end{aligned} \tag{2.3}$$

2.3 Non-maximum Suppression

Object detectors have a problem of detecting multiple bounding boxes surrounding an object in an image. Non-maximum suppressing is used to refine bounding boxes for these object detectors.

HOG algorithm scans over the image with a stride of (10,10) resulting in a car detected by many sliding windows as shown in Figure 2.5.

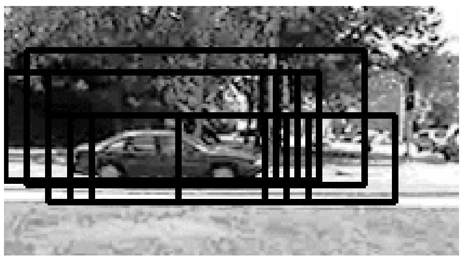


Figure 2.5: Multiple sliding windows that detected a car inside an image

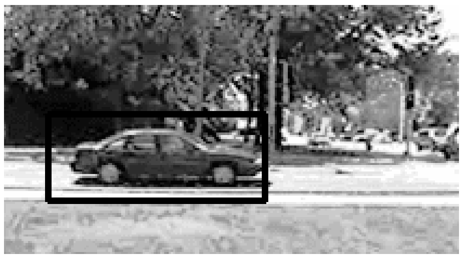


Figure 2.6: Filtered detected car sliding window using NMS

Non-maximum Suppression (NMS) is used to remove redundant sliding windows that detected a car with an overlapping of more than 30% (as per Equation 2.4) with other

detected sliding windows as shown in Figure 2.6.
Overlapping of two sliding windows with areas area1, area2 and an overlap area represented as overlap_area is given by

$$Overlapping = \frac{overlap\ area}{area1 + area2 - overlap\ area} \tag{2.4}$$

2.4 Approximation of arctangent function

The orientation of gradients is computed using the arctan function as shown in Equation 2.5.

$$arctan2(y, x) = \begin{cases} atan(\frac{y}{x}), & \text{if } x > 0 \\ atan(\frac{y}{x}) + 180°, & \text{if } x < 0\ and\ y \geq 0 \\ atan(\frac{y}{x}) - 180°, & \text{if } x < 0\ and\ y < 0 \\ +90°, & \text{if } x = 0\ and\ y > 0 \\ -90°, & \text{if } x = 0\ and\ y < 0 \\ \text{undefined}^{1}, & \text{if } x = 0\ and\ y = 0 \end{cases} \tag{2.5}$$

where $atan(\frac{y}{x})$ gives the principal angle.

In HLS these are implemented using CORDIC algorithms. Operations required by CORDIC algorithms are addition, subtraction, bit-shift and table lookup. This algorithm is suitable for applications where there is no actual hardware multiplier available e.g., FPGA and microcontroller based embedded applications. However, the CORDIC type algorithms are in general sequential in nature, making them rather slow [3]. This approximation of arctangent is originally taken from a paper[3]. In this work a comparative study of hls::atan2() function implemented in HLS with the implementation of arctangent as per the method mentioned in the paper.

2.4.1 Linear Interpolation

This approximation uses a LUT-based approach, so the no. of entries in the LUT are kept to be 101 such that it does not consume too much space and it can be partitioned completely in HLS for faster access. Each entry of LUT stores the value of arctan(x) (it gives principal angle) in degrees where x ∈ [0,1] with a resolution of 0.01 in x.

For better accuracy for gaps, in-between values of x in LUT values are filled up by using Linear Interpolation because the curve shown in Figure 2.7 is a monotonically increasing curve.

Using linear interpolation the value of y_2 for x_2 in Figure 2.8 can be determined by fitting the curve between (x_1, y_1) and (x_2, y_2) linearly as:

$$y_2 = y_0 + (x_2 - x_0)\frac{(y_1 - y_0)}{(x_1 - x_0)} \tag{2.6}$$

In Equation 2.6 $(x_1 - x_0)$ is the difference between two adjacent indices, i.e., 1. x_2 and x_0 are the real number input and the nearest integer respectively. y_0 is the value stored

1. In this case, as per mathematical definition of arctan2() is undefined but in the software implementation of NumPy arctan2() it is taken to be as zero[8].

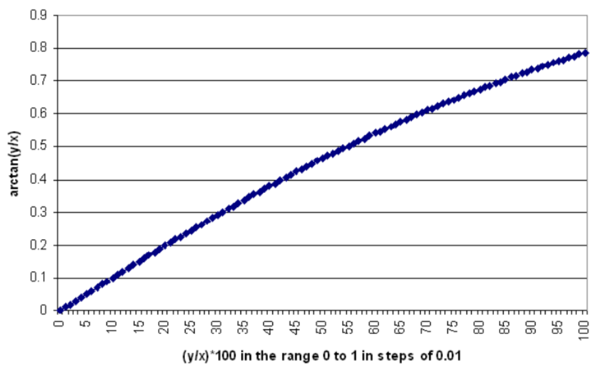


Figure 2.7: arctan(x) vs 100*(x) [3]

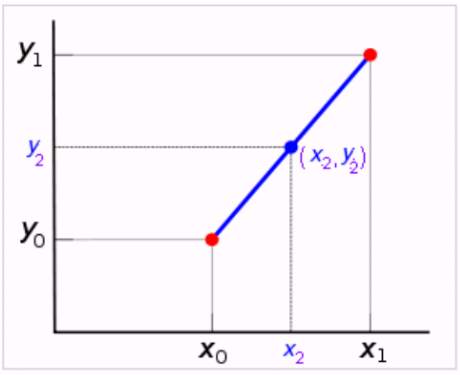


Figure 2.8: Linear interpolation between two points [3]

in the LUT corresponding to x_0 value.
arctan computation can be done using Linear Interpolation as:

$$\begin{cases} \widehat{input} = round(input \times 100), \\ index = \widehat{input} + 1, \\ angle_{new} = LUT(index) + (input \times 100 - \widehat{input})(LUT(index+1) - LUT(index)) \end{cases} \tag{2.7}$$

$$where\ input = \mid \frac{y}{x} \mid$$

Using Equation 2.7 $atan(input) = angle_{new}$ is computed. For $arctan(\frac{y}{x})$ calculation refer to Algortihm 1.

This method provided a fast method of approximating *arctangent* computation in embedded applications.

Chapter 3

Hardware-Software Co-design Approach

The first option, implement complete functionality in software side but that will involve sequential execution of data and parallelism won't be exploited until multi-core architectures are used. The second option, implement complete functionality in the hardware side. This option could exploit parallelism algorithm-specific and it will not be generic. It will also incur the cost of increased design time and complexity. This work shows how the hardware/software co-design approach can be used to partition an application on an FPGA fabric with some embedded microprocessor.

3.1 Algorithm partitioning

An experiment is performed to determine the time taken by each portion of the entire algorithm to identify the critical operations that can be implemented on processing logic(PL) or in the processing system(PS). This profiling is done using a PyPI package line_profiler[9]. It is a module for doing line-by-line profiling of functions. The profiler is implemented in C via Cython to reduce the overhead of profiling.

Function	**# Hits**	**Total time**	**Time per hit**	**% Time**
HOG feature computation	217	9379319.0	43222.7	82.9
Prediction using LinearSVC	217	292502.0	1347.9	2.6
Non-maximum suppression	1	403	403	0

Table 3.1: line_profile results of top 3 most time consuming parts of the application

The output of line_profiler can be explained as:

- **# Hits:** The number of times that line was executed.
- **Total time:** The total amount of time spent executing the line in the timer's units. Timer units are 1e-06 s in my case.
- **Time per hit:** The average amount of time spent executing the line once in the timer's units.
- **% Time:** The percentage of time spent on that line relative to the total amount of recorded time spent in the function.

From results as shown in Table 3.1 it is clear that HOG algorithm is one of the most time consuming part of the entire algorithm. Being a parallelized algorithm it is also suitable for implementation in hardware.

To speed up the execution in hardware many computations in the hardware are pipelined so that the accelerator can accept more computation when running time is amortized over a large data, thus improving the throughput.

3.2 Communication constraints

Algorithmic partitioning should be also done considering communication constraints to determine the optimal partitioning and not focusing only upon the parallelizable and most time-consuming part of the algorithm to be implemented on hardware. For example, HOG feature computation is found out in results obtained after time profiling as shown in Table 3.1. Communication between PS and PL can become a bottleneck if communication overheads are not considered in the overall design process.

Considering HOG-SVM study of this work, after 8748 HOG-features computation it is required them to be passed to Linear-SVM classification stage, which if implemented on PS will cause a communication overhead. To solve this issue, the SVM classification part is also pushed towards the PL. Now, PL can provide a single confidence score of classification result corresponding sliding window.

Shared memory is also used for addressing communication constraints between PS and PL for improving performance. It reduces the amount of data that needs to be transmitted between PS and PL. PS only needs to transmit the start address of the memory from where the pixels of the image are stored. Even now, accessing memory can become a performance bottleneck for PL. Since for sliding window compute only 10 pixels in the horizontal and 10 pixels in the vertical direction are new. So, only new pixels are required to be re-fetched rather than fetching new sliding window for all pixels.

3.3 Accelerator-level parallelism

HOG-feature extraction algorithm dictates that computation of HOG-features is independent for all sliding windows. Thus, more than one accelerators can be used for speeding up the execution. Each accelerator can be supplied with different sliding windows in parallel. This method can improve performance for sure but at the cost of increased resource utilization.

The three approaches that are described above are used to provide a generic Hardware-Software Co-design techniques that can be effective to improve the performance at a much lower effort compared to making a complete hardware design.

Chapter 4

Design

4.1 System-level Overview

Figure 4.1 shows the abstract representation of the overall system implemented for Nexys-Video (Xilinx part number XC7A200T-1SBG484C) FPGA.

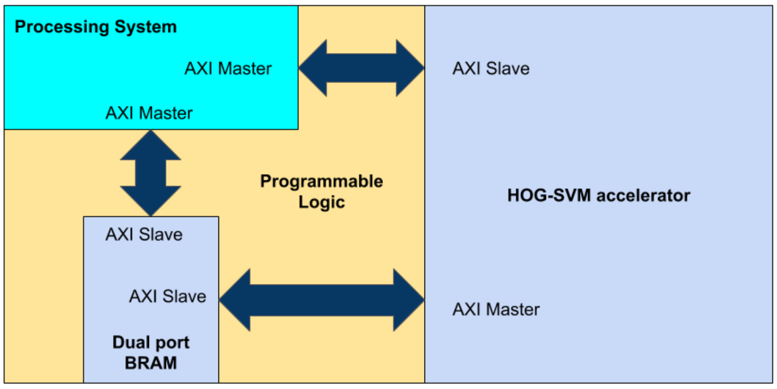


Figure 4.1: System-level view of the accelerator

The design has a standalone Microblaze system which provides Microblaze soft-core processor with memory access control, ICAP (internal configuration access port) access, and a reconfigurable evolvable section. The overall flow of the design can be expressed as a loop of three steps described as:

- Processing logic (Microblaze core) initializes BRAM for sliding window image pixel values, through its AXI-Master interface.
- Microblaze starts the HOG-SVM accelerator once it is done initializing values in BRAM. HOG-SVM then transfers the initialized sliding window from BRAM to its internal memory.
- HOG-SVM computes the confidence score of the presence of a car in an image and returns its value through AXI slave interface connected to the Processing logic.

4.2 Accelerator Design

4.2.1 Implementation of approximate arctan2

Algorithm 1 Approximation of arctangent2() algorithm

```
1: procedure ARCTAN2($x, y$)   ▷ Returns the approximate value of arctan2 function as per Equation 2.5
2:     if $x == 0$ then
3:         if $y > 0$ then
4:             return $90°$
5:         else if $y < 0$ then
6:             return $-90°$
7:         else
8:             return $0°$
9:     $tan\_inv\_lut[101] \leftarrow \{Static\ initialization\}$   ▷ Static initialization of principal tan-inverse values in degrees for arguments from 0 to 1
10:    $ratio \leftarrow \frac{y}{x}$
11:    $abs\_ratio \leftarrow |\ ratio\ |$
12:    $is\_inverted \leftarrow False$
13:    if $abs\_ratio > 1$ then
14:        $is\_inverted \leftarrow True$
15:        $ratio \leftarrow \frac{1}{ratio}$
16:        $abs\_ratio \leftarrow |\ ratio\ |$
17:    $lut\_index \leftarrow \lfloor 100 \times abs\_ratio \rfloor$
18:    $lut\_index\_1 \leftarrow lut\_index + 1$
19:    $COMPUTE\ principal\_angle$   ▷ Done using linear interpolation as described in Equation 2.6
20:    $arcTanResult \leftarrow principal\_angle$ ▷ Stores the result of arctan2() of this function
21:    if $ratio < 0$ then
22:        $arcTanResult \leftarrow 90° - principal\_angle$
23:    else
24:        $arcTanResult \leftarrow -90° - principal\_angle$
25:    if $x < 0 \&\& y >= 0$ then
26:        $arcTanResult \leftarrow 180° + arcTanResult$
27:    else if $x < 0 \&\& y < 0$ then
28:        $arcTanResult \leftarrow -180° + arcTanResult$
29:    return $arcTanResult$
```

Section 2.4 describes the idea behind the algorithm and it has been implemented using its idea as per Algorithm 1.

4.2.2 Block Design Generation

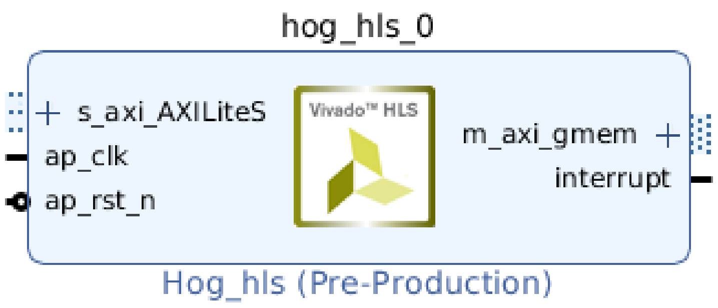


Figure 4.2: Block diagram of an accelerator

Algorithm 2 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 ← 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
```

Algorithm 2 describes the algorithmic flow of the accelerator. Two external interfaces are provided from accelerator:

- Master-AXI interface that requests the data from memory providing the address of the location to be accessed.
- Slave-AXI interface, connected to Microblaze core to return the confidence_score obtained.

Different **pragmas**, provided by Vivado HLS are included inside accelerator are used to increase the performance of the hardware generated.

1. **PIPELINE:** It reduces the initiation interval(II) of operations and allows their concurrent execution. Since, HLS-C code of accelerator is loop dominated, by making perfect nested loops it was possible to pipeline the loops with lowest possible achievable II.
2. **ARRAY_PARTITION:** Accelerator requires many array storages. To make sure that accessing these arrays does not become any performance bottleneck, it was used. This pragma increases the number of read and write ports because it partitions a large memory into small RTL memories. It improves the throughput of the design by allowing potential concurrent accesses to this memory.

4.2.3 Overall block Design

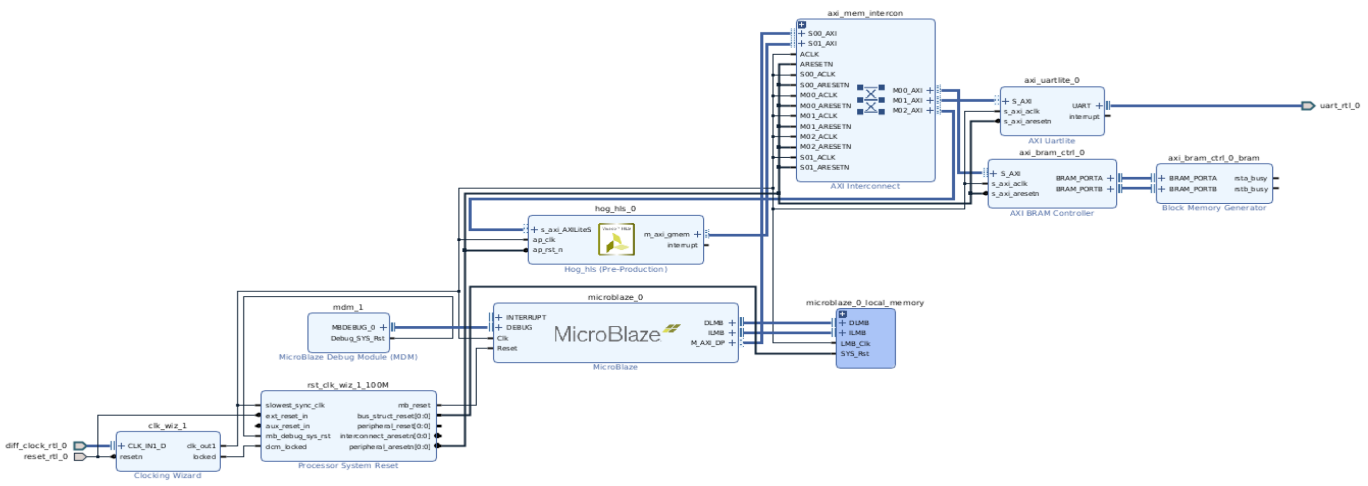
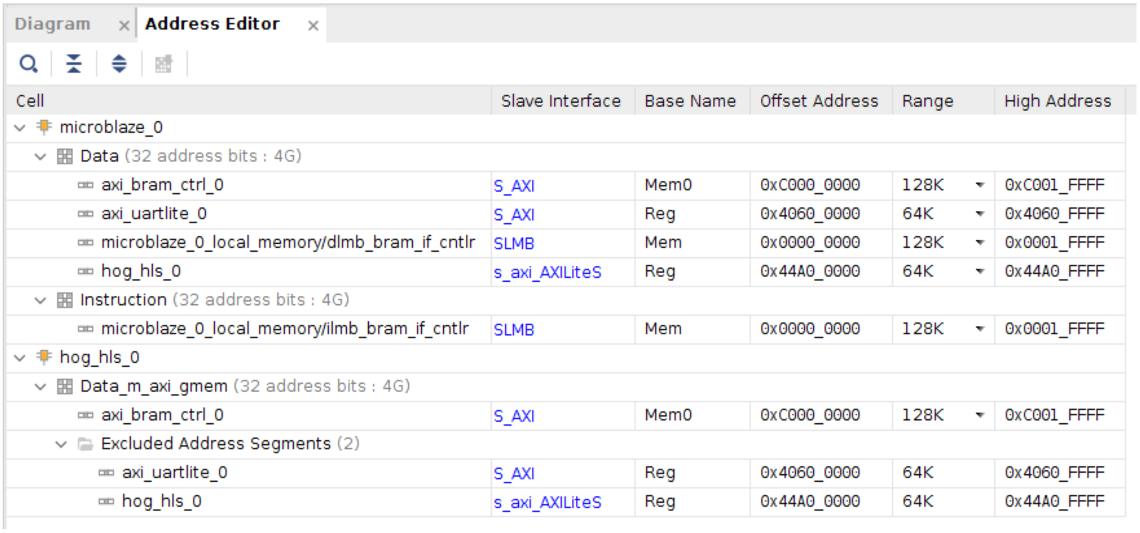


Figure 4.3: Block diagram of the entire design



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	128K	0xC001_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	128K	0x0001_FFFF
hog_hls_0	s_axi_AXILiteS	Reg	0x44A0_0000	64K	0x44A0_FFFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	128K	0x0001_FFFF
hog_hls_0					
Data_m_axi_gmem (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	128K	0xC001_FFFF
Excluded Address Segments (2)					
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
hog_hls_0	s_axi_AXILiteS	Reg	0x44A0_0000	64K	0x44A0_FFFF

Figure 4.4: Address Editor Layout

For complete integration of accelerator developed, Microblaze core along with AXI memory interconnect and BRAM is used.

From Address Map as shown in Figure 4.4 it can be seen that hog_hls accelerator is connected as a memory mapped block to Microblaze. To accomodate large size of .elf file for source code run on Microblaze size of local memory is taken as 128K X 32 bits. AXI BRAM controller(memory mapped to Microblaze) is used to store sliding window data as provided from Microblaze and it has 128K locations.
s_axi_AXILiteS interface is used for creating a slave interface to hog_hls accelerator. Using it accelerator can be started and values can be written to the accelerator interfaces e.g., *image_address* using Xil_Out32() command, given that address to the memory mapped block is known.

Chapter 5

Results

5.1 Synthesis

5.1.1 Comparison of approximate arctangent function with CORDIC implementation

- Timing Summary

Parameter	**Approximate arctan()**	**HLS arctan()**
Estimated clock frequency	9.378 ns	8.552 ns

Table 5.1: Synthesized timing summary results of two arctan() implementations

Larger clock frequency of HLS arctan() can be explained by pipelined implementation of CORDIC sequential algorithm whereas approximate arctan() tries to implement it in minimum cycles possible. LUT access is another reason for smaller clock frequency of Approximate arctan().

- Latency

Parameter	**Approximate arctan()**		**HLS arctan()**	
Latency (in cycles)	**min**	**max**	**min**	**max**
	3	274	41	196

Table 5.2: Latency (in cycles) comparison

From Algorithm 1 it can be seen that for base cases it can return the computed angle faster, which is why it has a lower minimum latency.

- Utilization

Since approximate version of arctan() requires float point division and multiplication which results in more DSP block utilization

Parameter	**Approximate arctan()**	**HLS arctan()**
BRAM_18K	1%	1%
DSP48E	3%	2%
FF	2%	2%
LUT	9%	9%

Table 5.3: Utilization Summary comparison

5.1.2 Accelerator

This section provides the synthesis comparison results obtained with two designs of accelerator in Vivado HLS as:

1. **Design A:** Design not optimized with any of the pragmas.
2. **Design B:** Design optimized with pragmas provided by Vivado HLS.

- Timing Summary

Clock	**Target (in ns)**	**Estimated (in ns)**		**Uncertainity (in ns)**
		Design A	**Design B**	
ap_clk	10.00	9.342	8.75	1.25

Table 5.4: Timing summary comparison

- Latency Summary

Latency (Design A)		**Latency (Design B)**		**Interval (Design A)**		**Interval (Design B)**	
min	**max**	**min**	**max**	**min**	**max**	**min**	**max**
1517922	2932532	792487	1253287	1517972	2932532	792487	1253287

Table 5.5: Comparison of latency in clock cycles

- Utilization Summary

Name	**BRAM_18K**		**DSP48E**		**FF**		**LUT**	
	Design A	**Design B**	**Design A**	**Design B**	**Design A**	**Design B**	**Design A**	**Design B**
DSP	-	-	-	3	-	-	-	-
Expression	-	-	2	1	0	0	3224	4047
FIFO	-	-	-	-	-	-	-	-
Instance	5	2	46	24	18980	16713	21683	18633
Memory	170	103	-	-	0	0	0	0
Multiplexer	-	-	-	-	-	-	1478	2765
Register	-	0	-	-	2249	3580	-	256
Total	175	105	48	28	21229	20293	26385	25701
Available	730		740		269200		134600	
Utilization (%)	23	14	6	3	7	7	19	19

Table 5.6: Utilization summary comparison

From HLS synthesis results as been shown in Tables 5.4, 5.5, 5.6 performance and utilization improves with the use of pragmas provided by HLS.

5.1.3 Overall Block Design

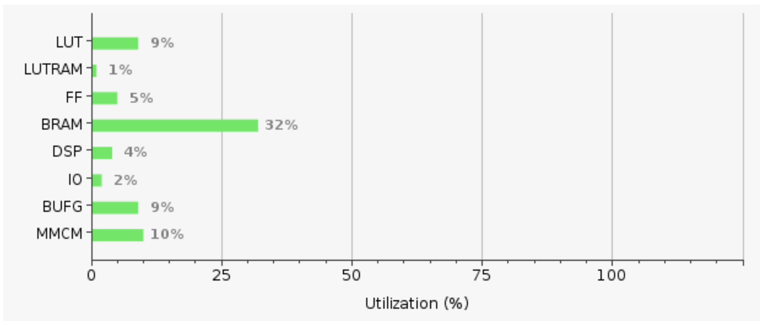


Figure 5.1: Utilization of overall design, post-implementation of Design B

Resource	**Utilization**	**Available**	**Utilization %**
LUT	11541	133800	8.63
LUTRAM	405	46200	0.88
FF	12461	267600	4.66
BRAM	116	365	31.78
DSP	28	740	3.78
IO	5	285	1.75
BUFG	3	32	9.38
MMCM	1	10	10.00

Table 5.7: Post-Implementation Utilization Summary of Design B

5.2 Simulation

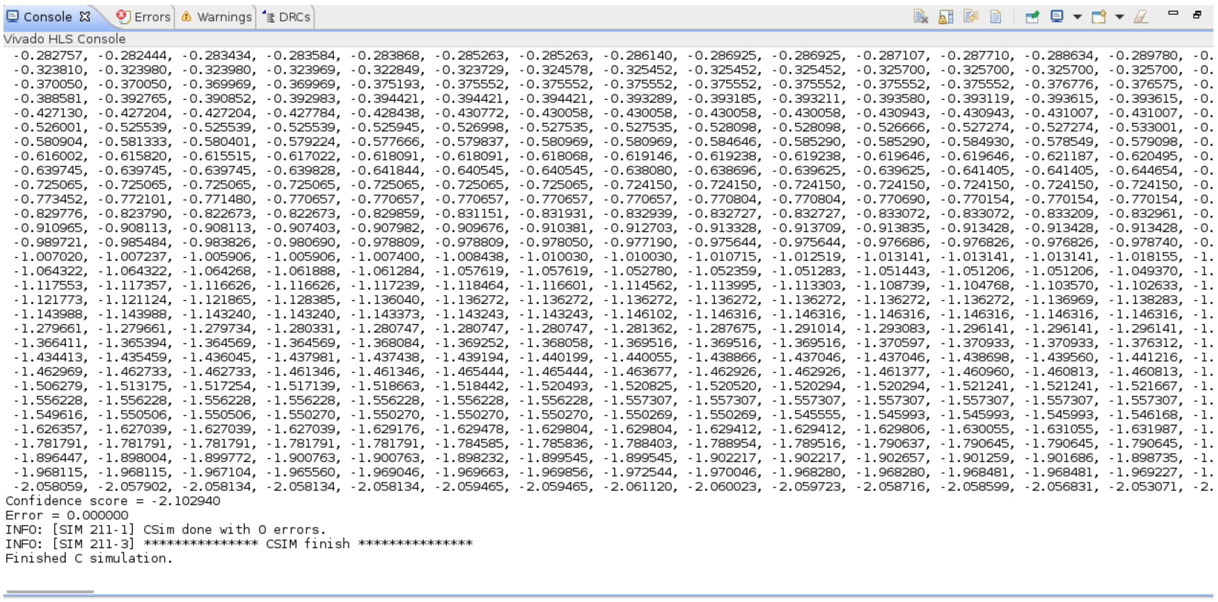
This section provides three different simulation results for the accelerator designed.

1. **C-simulation:** It provides a check on whether C++ code works correctly as per behaviour. HLS performs this simulation considering it as a normal C++ program.

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

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

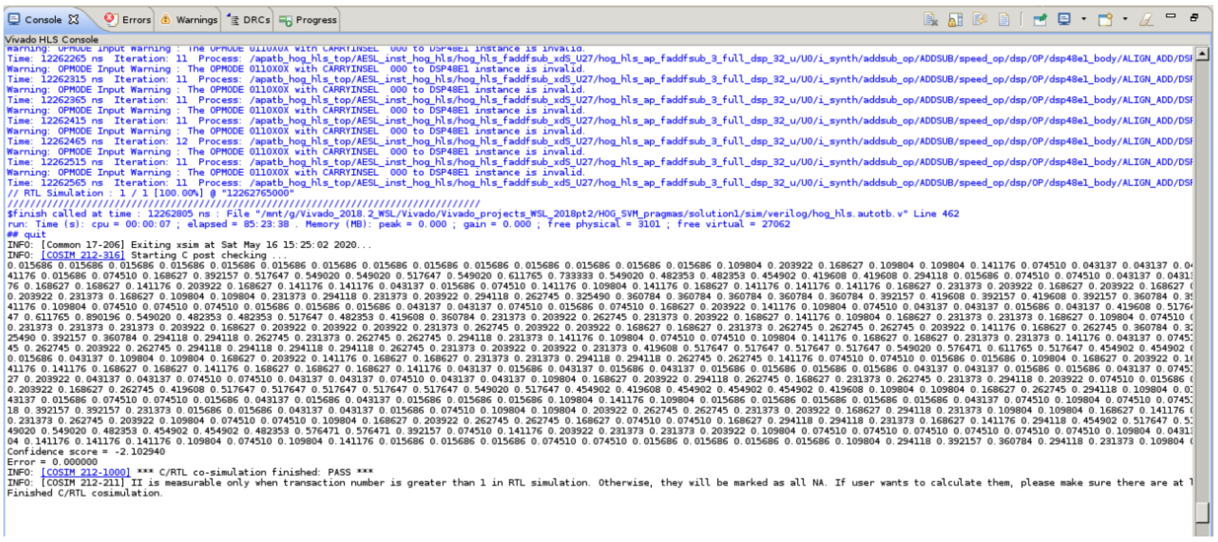
5.2.1 C-simulation



```
 -0.282757, -0.282444, -0.283434, -0.283584, -0.283868, -0.285263, -0.285263, -0.286140, -0.286925, -0.286925, -0.287107, -0.287710, -0.288634, -0.289780, -0.
 -0.323810, -0.323980, -0.323980, -0.323969, -0.322849, -0.323729, -0.324578, -0.325452, -0.325452, -0.325452, -0.325700, -0.325700, -0.325700, -0.325700, -0.
 -0.370050, -0.370050, -0.369969, -0.369969, -0.375193, -0.375552, -0.375552, -0.375552, -0.375552, -0.375552, -0.375552, -0.375552, -0.376776, -0.376575, -0.
 -0.388581, -0.392765, -0.390852, -0.392983, -0.394421, -0.394421, -0.394421, -0.393289, -0.393185, -0.393211, -0.393580, -0.393119, -0.393615, -0.393615, -0.
 -0.427130, -0.427204, -0.427204, -0.427784, -0.428438, -0.430772, -0.430058, -0.430058, -0.430058, -0.430058, -0.430943, -0.430943, -0.431007, -0.431007, -0.
 -0.526001, -0.525539, -0.525539, -0.525539, -0.525945, -0.526998, -0.527535, -0.527535, -0.528098, -0.528098, -0.526666, -0.527274, -0.527274, -0.533001, -0.
 -0.580904, -0.581333, -0.580401, -0.579224, -0.577666, -0.579837, -0.580969, -0.580969, -0.584646, -0.585290, -0.585290, -0.584930, -0.578549, -0.579098, -0.
 -0.616002, -0.615820, -0.615515, -0.617022, -0.618091, -0.618091, -0.618068, -0.619146, -0.619238, -0.619238, -0.619646, -0.619646, -0.621187, -0.620495, -0.
 -0.639745, -0.639745, -0.639745, -0.639828, -0.641844, -0.640545, -0.640545, -0.638080, -0.638696, -0.639625, -0.639625, -0.641405, -0.641405, -0.644654, -0.
 -0.725065, -0.725065, -0.725065, -0.725065, -0.725065, -0.725065, -0.725065, -0.724150, -0.724150, -0.724150, -0.724150, -0.724150, -0.724150, -0.724150, -0.
 -0.773452, -0.772101, -0.771480, -0.770657, -0.770657, -0.770657, -0.770657, -0.770657, -0.770804, -0.770804, -0.770690, -0.770154, -0.770154, -0.770154, -0.
 -0.829776, -0.823790, -0.822673, -0.822673, -0.829859, -0.831151, -0.831931, -0.832939, -0.832727, -0.832727, -0.833072, -0.833072, -0.833209, -0.832961, -0.
 -0.910965, -0.908113, -0.908113, -0.907403, -0.907982, -0.909676, -0.910381, -0.912703, -0.913328, -0.913709, -0.913835, -0.913428, -0.913428, -0.913428, -0.
 -0.989721, -0.985484, -0.983826, -0.980690, -0.978809, -0.978809, -0.978050, -0.977190, -0.975644, -0.975644, -0.976686, -0.976826, -0.976826, -0.978740, -0.
 -1.007020, -1.007237, -1.005906, -1.005906, -1.007400, -1.008438, -1.010030, -1.010030, -1.010715, -1.012519, -1.013141, -1.013141, -1.013141, -1.018155, -1.
 -1.064322, -1.064322, -1.064268, -1.061888, -1.061284, -1.057619, -1.057619, -1.052780, -1.052359, -1.051283, -1.051443, -1.051206, -1.051206, -1.049370, -1.
 -1.117553, -1.117357, -1.116626, -1.116626, -1.117239, -1.118464, -1.116601, -1.114562, -1.113995, -1.113303, -1.108739, -1.104768, -1.103570, -1.102633, -1.
 -1.121773, -1.121124, -1.121865, -1.128385, -1.136040, -1.136272, -1.136272, -1.136272, -1.136272, -1.136272, -1.136272, -1.136272, -1.136969, -1.138283, -1.
 -1.143988, -1.143988, -1.143240, -1.143240, -1.143373, -1.143243, -1.143243, -1.146102, -1.146316, -1.146316, -1.146316, -1.146316, -1.146316, -1.146316, -1.
 -1.279661, -1.279661, -1.279734, -1.280331, -1.280747, -1.280747, -1.280747, -1.281362, -1.287675, -1.291014, -1.293083, -1.296141, -1.296141, -1.296141, -1.
 -1.366411, -1.365394, -1.364569, -1.364569, -1.368084, -1.369252, -1.368058, -1.369516, -1.369516, -1.369516, -1.370597, -1.370933, -1.370933, -1.376312, -1.
 -1.434413, -1.435459, -1.436045, -1.437981, -1.437438, -1.439194, -1.440199, -1.440055, -1.438866, -1.437046, -1.437046, -1.438698, -1.439560, -1.441216, -1.
 -1.462969, -1.462733, -1.462733, -1.461346, -1.461346, -1.465444, -1.465444, -1.463677, -1.462926, -1.462926, -1.461377, -1.460960, -1.460813, -1.460813, -1.
 -1.506279, -1.513175, -1.517254, -1.517139, -1.518663, -1.518442, -1.520493, -1.520825, -1.520520, -1.520294, -1.520294, -1.521241, -1.521241, -1.521667, -1.
 -1.556228, -1.556228, -1.556228, -1.556228, -1.556228, -1.556228, -1.556228, -1.557307, -1.557307, -1.557307, -1.557307, -1.557307, -1.557307, -1.557307, -1.
 -1.549616, -1.550506, -1.550506, -1.550270, -1.550270, -1.550270, -1.550270, -1.550269, -1.550269, -1.545555, -1.545993, -1.545993, -1.545993, -1.546168, -1.
 -1.626357, -1.627039, -1.627039, -1.627039, -1.629176, -1.629478, -1.629804, -1.629804, -1.629412, -1.629412, -1.629806, -1.630055, -1.631055, -1.631987, -1.
 -1.781791, -1.781791, -1.781791, -1.781791, -1.781791, -1.784585, -1.785836, -1.788403, -1.788954, -1.789516, -1.790637, -1.790645, -1.790645, -1.790645, -1.
 -1.896447, -1.898004, -1.899772, -1.900763, -1.900763, -1.898232, -1.899545, -1.899545, -1.902217, -1.902217, -1.902657, -1.901259, -1.901686, -1.898735, -1.
 -1.968115, -1.968115, -1.967104, -1.965560, -1.969046, -1.969663, -1.969856, -1.972544, -1.970046, -1.968280, -1.968280, -1.968481, -1.968481, -1.969227, -1.
 -2.058059, -2.057902, -2.058134, -2.058134, -2.058134, -2.059465, -2.059465, -2.061120, -2.060023, -2.059723, -2.058716, -2.058599, -2.056831, -2.053071, -2.
Confidence score = -2.102940
Error = 0.000000
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

Figure 5.2: HLS C-simulation output of the accelerator

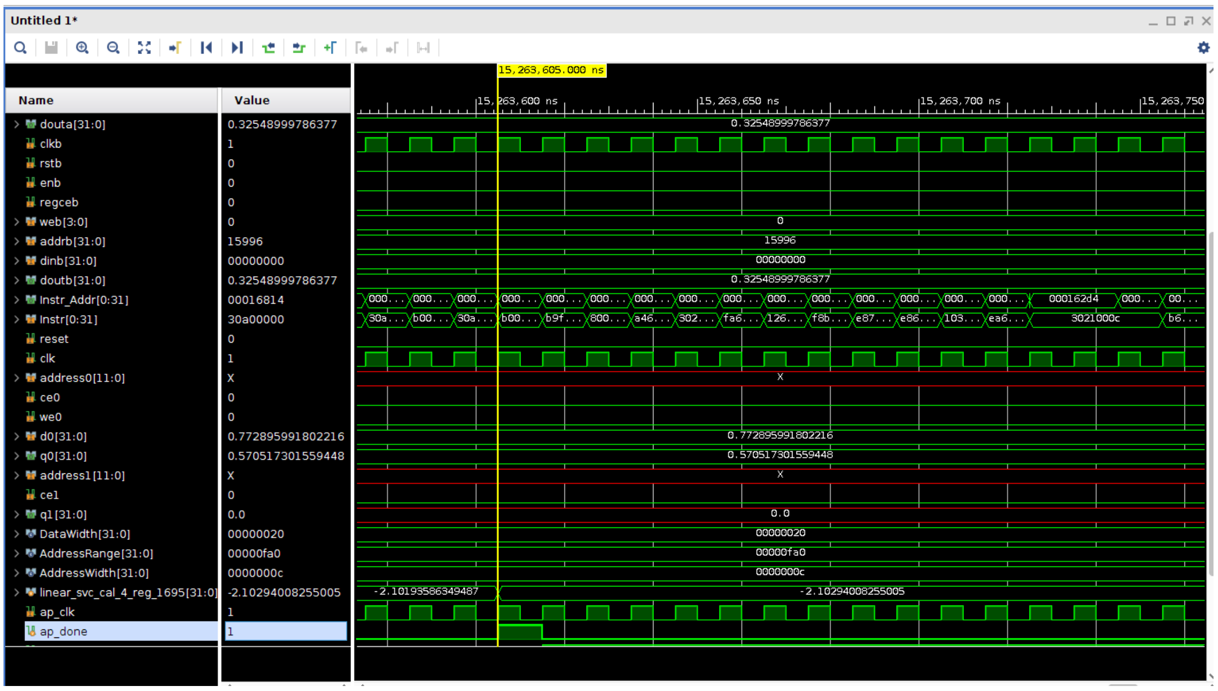
5.2.2 Co-simulation



```
Warning: OPMODE Input Warning : The OPMODE 0110X0X with CARRYINSEL  000 to DSP48E1 instance is invalid.
Time: 12262265 ns  Iteration: 11  Process: /apatb_hog_hls_top/AESL_inst_hog_hls/hog_hls_faddfsub_xdS_U27/hog_hls_ap_faddfsub_3_full_dsp_32_u/U0/i_synth/addsub_op/ADDSUB/speed_op/dsp/OP/dsp48e1_body/ALIGN_ADD/DS
Warning: OPMODE Input Warning : The OPMODE 0110X0X with CARRYINSEL  000 to DSP48E1 instance is invalid.
Time: 12262315 ns  Iteration: 11  Process: /apatb_hog_hls_top/AESL_inst_hog_hls/hog_hls_faddfsub_xdS_U27/hog_hls_ap_faddfsub_3_full_dsp_32_u/U0/i_synth/addsub_op/ADDSUB/speed_op/dsp/OP/dsp48e1_body/ALIGN_ADD/DS
Warning: OPMODE Input Warning : The OPMODE 0110X0X with CARRYINSEL  000 to DSP48E1 instance is invalid.
Time: 12262365 ns  Iteration: 11  Process: /apatb_hog_hls_top/AESL_inst_hog_hls/hog_hls_faddfsub_xdS_U27/hog_hls_ap_faddfsub_3_full_dsp_32_u/U0/i_synth/addsub_op/ADDSUB/speed_op/dsp/OP/dsp48e1_body/ALIGN_ADD/DS
Warning: OPMODE Input Warning : The OPMODE 0110X0X with CARRYINSEL  000 to DSP48E1 instance is invalid.
Time: 12262415 ns  Iteration: 11  Process: /apatb_hog_hls_top/AESL_inst_hog_hls/hog_hls_faddfsub_xdS_U27/hog_hls_ap_faddfsub_3_full_dsp_32_u/U0/i_synth/addsub_op/ADDSUB/speed_op/dsp/OP/dsp48e1_body/ALIGN_ADD/DS
Warning: OPMODE Input Warning : The OPMODE 0110X0X with CARRYINSEL  000 to DSP48E1 instance is invalid.
Time: 12262465 ns  Iteration: 12  Process: /apatb_hog_hls_top/AESL_inst_hog_hls/hog_hls_faddfsub_xdS_U27/hog_hls_ap_faddfsub_3_full_dsp_32_u/U0/i_synth/addsub_op/ADDSUB/speed_op/dsp/OP/dsp48e1_body/ALIGN_ADD/DS
Warning: OPMODE Input Warning : The OPMODE 0110X0X with CARRYINSEL  000 to DSP48E1 instance is invalid.
Time: 12262515 ns  Iteration: 11  Process: /apatb_hog_hls_top/AESL_inst_hog_hls/hog_hls_faddfsub_xdS_U27/hog_hls_ap_faddfsub_3_full_dsp_32_u/U0/i_synth/addsub_op/ADDSUB/speed_op/dsp/OP/dsp48e1_body/ALIGN_ADD/DS
Warning: OPMODE Input Warning : The OPMODE 0110X0X with CARRYINSEL  000 to DSP48E1 instance is invalid.
Time: 12262565 ns  Iteration: 11  Process: /apatb_hog_hls_top/AESL_inst_hog_hls/hog_hls_faddfsub_xdS_U27/hog_hls_ap_faddfsub_3_full_dsp_32_u/U0/i_synth/addsub_op/ADDSUB/speed_op/dsp/OP/dsp48e1_body/ALIGN_ADD/DS
// RTL Simulation : 1 / 1 [100.00%] @ "12262765000"
////////////////////////////////////////////////////////////////////////////////////
$finish called at time : 12262805 ns : File "/mnt/g/Vivado_2018.2_WSL/Vivado/Vivado_projects_WSL_2018pt2/HOG_SVM_pragmas/solution1/sim/verilog/hog_hls.autotb.v" Line 462
run: Time (s): cpu = 00:00:07 ; elapsed = 85:23:38 . Memory (MB): peak = 0.000 ; gain = 0.000 ; free physical = 3101 ; free virtual = 27062
## quit
INFO: [Common 17-206] Exiting xsim at Sat May 16 15:25:02 2020...
INFO: [COSIM 212-316] Starting C post checking ...
0.015686 0.015686 0.015686 0.015686 0.015686 0.015686 0.015686 0.015686 0.015686 0.015686 0.015686 0.015686 0.015686 0.015686 0.015686 0.109804 0.203922 0.168627 0.109804 0.109804 0.141176 0.074510 0.043137 0.043137 0.04
41176 0.015686 0.074510 0.168627 0.392157 0.517647 0.549020 0.549020 0.517647 0.549020 0.611765 0.733333 0.549020 0.482353 0.482353 0.454902 0.419608 0.419608 0.294118 0.015686 0.074510 0.074510 0.043137 0.0431
76 0.168627 0.168627 0.141176 0.203922 0.168627 0.141176 0.141176 0.043137 0.015686 0.074510 0.141176 0.109804 0.141176 0.168627 0.141176 0.141176 0.141176 0.168627 0.231373 0.203922 0.168627 0.203922 0.168627 0
0.203922 0.231373 0.168627 0.109804 0.109804 0.231373 0.294118 0.231373 0.203922 0.294118 0.262745 0.325490 0.360784 0.360784 0.360784 0.360784 0.360784 0.392157 0.419608 0.392157 0.419608 0.392157 0.360784 0.39
41176 0.109804 0.074510 0.074510 0.074510 0.015686 0.015686 0.015686 0.043137 0.043137 0.074510 0.015686 0.074510 0.168627 0.203922 0.141176 0.109804 0.074510 0.043137 0.043137 0.015686 0.043137 0.419608 0.5176
47 0.611765 0.890196 0.549020 0.482353 0.482353 0.517647 0.482353 0.419608 0.360784 0.231373 0.203922 0.262745 0.231373 0.203922 0.168627 0.141176 0.109804 0.168627 0.231373 0.231373 0.168627 0.109804 0.074510 0
0.231373 0.231373 0.231373 0.203922 0.168627 0.203922 0.203922 0.203922 0.231373 0.262745 0.203922 0.203922 0.168627 0.168627 0.231373 0.262745 0.262745 0.262745 0.203922 0.141176 0.168627 0.262745 0.360784 0.32
25490 0.392157 0.360784 0.294118 0.294118 0.262745 0.231373 0.262745 0.262745 0.294118 0.231373 0.141176 0.109804 0.074510 0.074510 0.109804 0.141176 0.168627 0.168627 0.231373 0.231373 0.141176 0.043137 0.0745
45 0.262745 0.203922 0.262745 0.294118 0.294118 0.294118 0.294118 0.262745 0.231373 0.203922 0.203922 0.231373 0.419608 0.517647 0.517647 0.517647 0.517647 0.549020 0.576471 0.611765 0.517647 0.454902 0.454902 0
0.015686 0.043137 0.109804 0.109804 0.168627 0.203922 0.141176 0.168627 0.168627 0.231373 0.231373 0.294118 0.294118 0.262745 0.262745 0.141176 0.074510 0.074510 0.015686 0.015686 0.109804 0.168627 0.203922 0.1
41176 0.141176 0.168627 0.168627 0.141176 0.168627 0.168627 0.168627 0.141176 0.043137 0.015686 0.043137 0.015686 0.043137 0.015686 0.015686 0.015686 0.015686 0.043137 0.043137 0.015686 0.015686 0.043137 0.0745
27 0.203922 0.043137 0.043137 0.074510 0.074510 0.043137 0.043137 0.074510 0.043137 0.043137 0.109804 0.168627 0.203922 0.294118 0.262745 0.168627 0.231373 0.262745 0.231373 0.294118 0.203922 0.074510 0.015686 0
0.203922 0.168627 0.262745 0.419608 0.517647 0.517647 0.517647 0.517647 0.517647 0.517647 0.549020 0.517647 0.454902 0.419608 0.454902 0.454902 0.454902 0.419608 0.109804 0.109804 0.168627 0.262745 0.294118 0.109804 0.03
43137 0.015686 0.074510 0.074510 0.015686 0.043137 0.015686 0.043137 0.015686 0.015686 0.015686 0.109804 0.141176 0.109804 0.015686 0.015686 0.015686 0.015686 0.015686 0.043137 0.074510 0.109804 0.074510 0.0745
18 0.392157 0.392157 0.231373 0.015686 0.015686 0.043137 0.043137 0.015686 0.074510 0.109804 0.109804 0.203922 0.262745 0.262745 0.231373 0.203922 0.168627 0.294118 0.231373 0.109804 0.109804 0.168627 0.141176 0
0.231373 0.262745 0.203922 0.109804 0.074510 0.074510 0.109804 0.168627 0.203922 0.262745 0.262745 0.168627 0.074510 0.074510 0.168627 0.294118 0.294118 0.231373 0.168627 0.141176 0.294118 0.454902 0.517647 0.5
49020 0.549020 0.482353 0.454902 0.454902 0.482353 0.576471 0.576471 0.392157 0.074510 0.141176 0.203922 0.231373 0.231373 0.203922 0.109804 0.074510 0.074510 0.074510 0.074510 0.074510 0.074510 0.109804 0.0431
04 0.141176 0.141176 0.141176 0.109804 0.074510 0.109804 0.141176 0.015686 0.015686 0.015686 0.074510 0.074510 0.015686 0.015686 0.015686 0.015686 0.109804 0.294118 0.392157 0.360784 0.294118 0.231373 0.109804 0
Confidence score = -2.102940
Error = 0.000000
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
INFO: [COSIM 212-211] II is measurable only when transaction number is greater than 1 in RTL simulation. Otherwise, they will be marked as all NA. If user wants to calculate them, please make sure there are at l
Finished C/RTL cosimulation.
```

Figure 5.3: HLS RTL Co-simulation output of the accelerator

5.2.3 Microblaze RTL Simulation



Untitled 1*

Name	Value
douta[31:0]	0.32548999786377
clkb	1
rstb	0
enb	0
regceb	0
web[3:0]	0
addrb[31:0]	15996
dinb[31:0]	00000000
doutb[31:0]	0.32548999786377
Instr_Addr[0:31]	00016814
Instr[0:31]	30a00000
reset	0
clk	1
address0[11:0]	X
ce0	0
we0	0
d0[31:0]	0.772895991802216
q0[31:0]	0.570517301559448
address1[11:0]	X
ce1	0
q1[31:0]	0.0
DataWidth[31:0]	00000020
AddressRange[31:0]	00000fa0
AddressWidth[31:0]	0000000c
linear_svc_cal_4_reg_1695[31:0]	-2.10294008255005
ap_clk	1
ap_done	1

Figure 5.4: Microblaze RTL Behavioural simulation

Chapter 6

Conclusion

This work presents the implementation of HOG-SVM 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). Time taken and efforts required to make a design is much lesser in case of HLS compared to HDL designs. Overall accelerator integration into the system has been tested using behavioural simulation using Vivado 2019.1.

Bibliography

[1] C. Bagavathi and O. Saraniya, "Hardware designs for histogram of oriented gradients in pedestrian detection: A survey," in *2019 5th International Conference on Advanced Computing & Communication Systems (ICACCS)*, pp. 849–854, IEEE, 2019.

[2] L. Weng, "Object Detection for Dummies Part 1: Gradient Vector, HOG, and SS." `https://lilianweng.github.io/lil-log/2017/10/29/object-recognition-for-dummies-part-1.html`. Accessed: 2020-06-04.

[3] A. Ukil, V. H. Shah, and B. Deck, "Fast computation of arctangent functions for embedded applications: A comparative analysis," in *2011 IEEE International Symposium on Industrial Electronics*, pp. 1206–1211, IEEE, 2011.

[4] 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. 886–893, IEEE, 2005.

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

[6] M. Ishan, "UIUC Image Database for Car Detection." `https://github.com/Menuka5/Data-sets-for-opencv-classifier-training/tree/master/Other%20Image%20Datasets%20collected`. Accessed: 2020-06-04.

[7] J. Yuan, "HOG-SVM-python." `https://github.com/jianlong-yuan/HOG-SVM-python`. Accessed: 2020-06-04.

[8] NumPy Documentation, "numpy.arctan2() Documentation." `https://numpy.org/doc/stable/reference/generated/numpy.arctan2.html`. Accessed: 2020-06-04.

[9] R. Kern, "line_profiler and kernprof." `https://github.com/pyutils/line_profiler`. Accessed: 2020-06-04.