Study of Heterogeneous architectures for Embedded Systems using HOG SVM accelerator as a case study

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
December 2020

Dissertation Approval

The dissertation entitled

Study of Heterogeneous architectures for Embedded Systems using HOG SVM accelerator as a case study

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: 10th December, 2020

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 original 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

December 10, 2020

Acknowledgements

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 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 requirement is the need for high performance with minimum power dissipation and cost.In order to meet these demands, the industry is adopting more heterogeneous architectures that have reconfiguration capabilities. Unfortunately, this increases the complexity of the system and it overshadows the advantages of such architectures. This thesis address the 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 Support Vector Machine(SVM) ML algorithm to classify those objects.Here, HOG with SVM for car detection is used as a case study to look into increase in performance metric once incorporated as a Hardware-Software heterogeneous systems.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

In the present time, the demands for high performance computing is increasing every passing day. In order to increase the performance, the hardware industry initially resorted to increasing clock frequency of the single core processor, before it hit the power wall in the early 2000. However, the need for increasing hardware performance in accordance to the software demands is continually rising. One way to handle such performance requirements is through heterogeneous computing. It refers to the use of different types of processor in a computer system. It can also be referred to Hardware-Software Co-Design. There are 2 ways to implement an algorithm, first option is to 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.

1.2 Objective

The goal of the project is to implement the HOG SVM algorithm used for vehicle detection. 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, in order to enhance the performance.

1.3 Organization of the thesis

In the first part of the Thesis, the HOG SVM algorithm for car detection is implemented. For this, first , the algorithm is built in initially in python and from there , after training images we are able get the features for SVM 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 in 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. In the future work, 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

Background

2.1 Histogram of oriented gradients

The histogram of oriented gradients (HOG) is a feature descriptor used in computer vision and image processing for the purpose of object 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.

2.2 Object Detection using HOG SVM flow

HOG, or Histogram of Oriented Gradients, is a feature descriptor that is often used to extract features from image data. It is widely used in computer vision tasks for object detection.

The HOG descriptor focuses on the structure or the shape of an object. HOG is able to provide the edge direction as well. This is done by extracting the gradient and orientation (or you can say magnitude and direction) of the edges Additionally, these orientations are calculated in 'localized' portions. This means that the complete image is broken down into smaller regions and for each region, the gradients and orientation are calculated. We will discuss this in much more detail in the upcoming sections Finally the HOG would generate a Histogram for each of these regions separately. The histograms are created using the gradients and orientations of the pixel values, hence the name 'Histogram of Oriented Gradients'

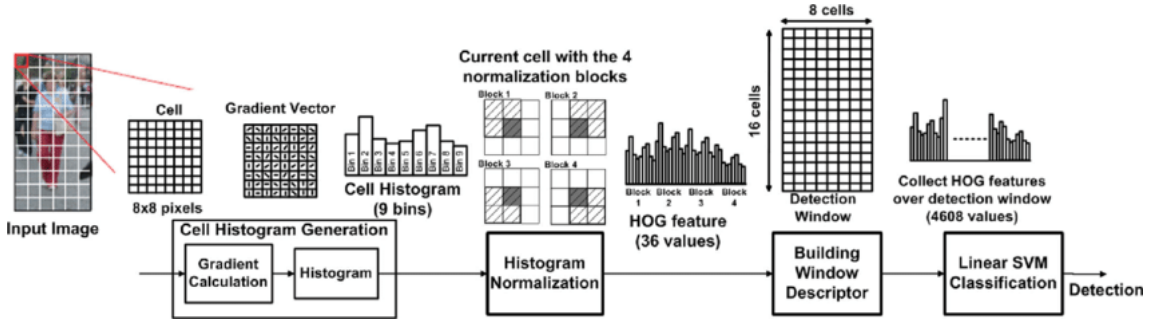

Figure 2.1: Object detection using HOG SVM algorithm

In this implementation , we are detecting a car, hence the sliding window used is 40x100 pixels with a stride of 10 pixels in the horizontal direction and 10 pixels in the vertical direction scans the entire image. 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

The detection algorithm is divided in the following steps:

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

2.2.1 Preprocessing

In this implementation , 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.

Gamma Normalization in HOG is actually Power Law Transformation

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

where s is output pixel, r is input pixel, c is constant and γ is exponent. Different devices used for image capture, display and printing use this power law transformation to correct image

intensity values and this process is known as gamma correction. In short, gamma normalization in HOG is same as gamma correction.

2.2.2 Gradient Vector Calculation

In the image processing, we want to know the direction of colors changing from one extreme to the other (i.e. black to white on a grayscale image). Therefore, we want to measure "gradient" on pixels of colors. The gradient on an image is discrete because each pixel is independent and cannot be further split.

The gradient vector is defined as a metric for every individual pixel, containing the pixel color changes in both x-axis and y-axis.

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:

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

There are two important attributes of an image gradient:
Magnitude is the L2-norm of the vector, g=$\sqrt{g_x^2 + g_y^2}$
Direction is the arctangent of the ratio between the partial derivatives,$\theta = arctan(g_y/g_x)$.

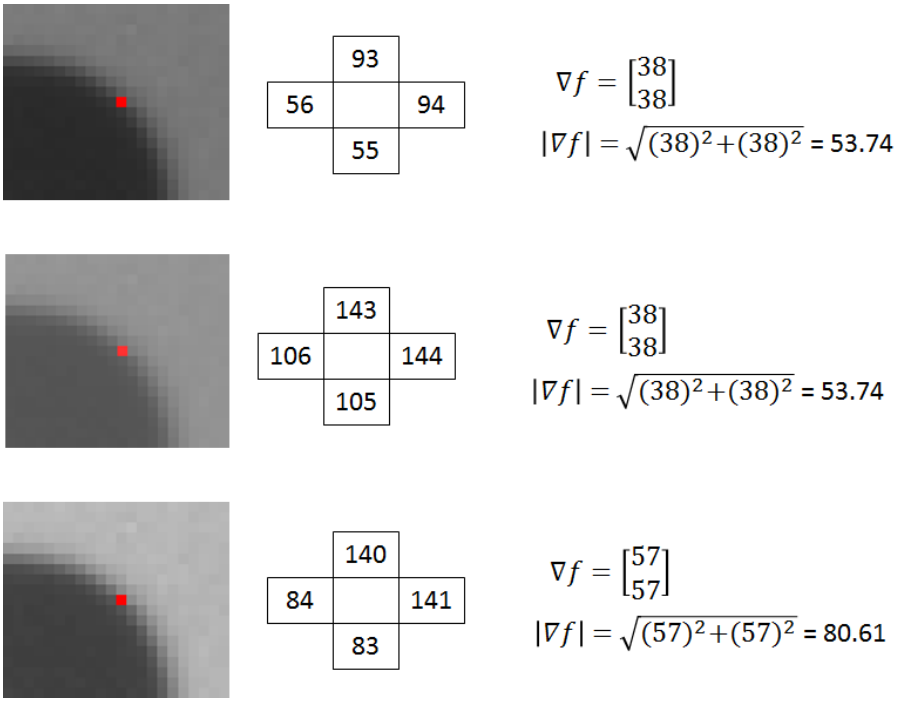

Figure 2.2: Gradient Vector Calculation

2.2.3 HOG feature Extraction

The sliding window is divided into 5x5 pixels forms a cell and 3x3 cells combine to form a block. Each cell of 5x5 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:
$\theta = arctan(g_y/g_x)$

Contribution of a pixel to a bin is done by adding the magnitude of gradient corresponding to this pixel to its assigned bin.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 40x100 pixel is composed of 6x18 blocks. Each block consists of 3x3 cells and a cell has a histogram of 9 discrete bins. Thus,

$$Totalno.of features = 6 * 18 * 3 * 3 * 9 = 8748$$

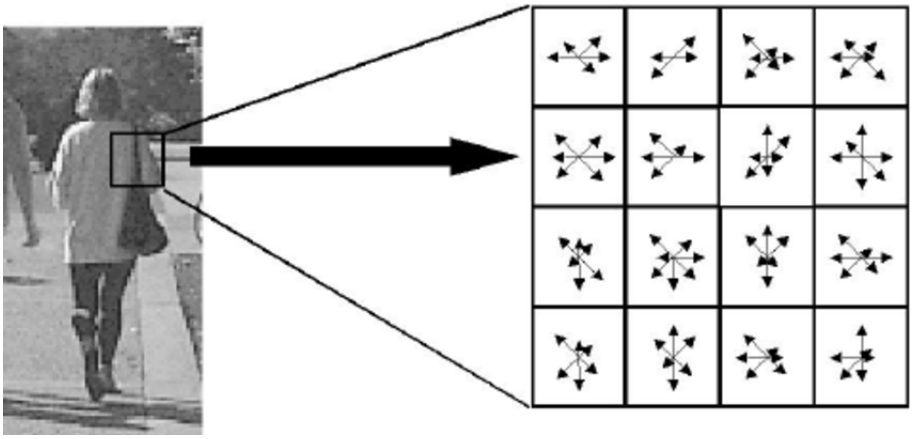

Figure 2.3: HOG Feature Extraction

2.2.4 Histogram Normalization

Here the orientation histograms are normalized over the blocks to which they belong. This normalization step introduces better invariance to illumination, shadowing, and edge contrast.

2.3 Support Vector Machine

A support vector machine (SVM) is a supervised machine learning model that uses classification algorithms for two-group classification problems. After giving an SVM model sets of labeled training data for each category, they're able to categorize new text.

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

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

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

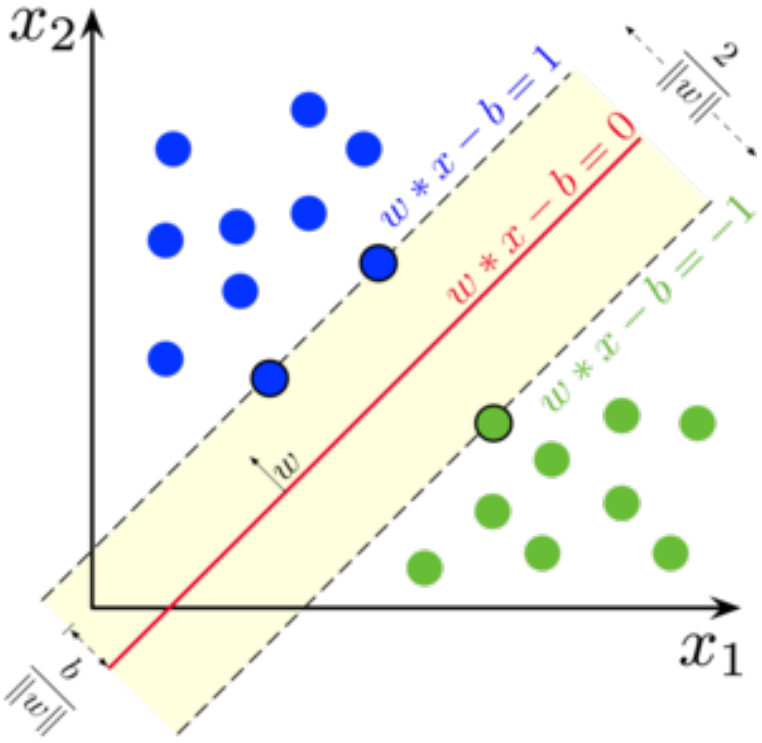

Figure 2.4: Linear SVM

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

$$y > 0; Detected$$

$$y <= 0; NotDetected$$

2.4 Non maximum suppression

History of Oriented Gradients(HOG) combined with Support Vector Machines(SVM) have been pretty successful for detecting objects in images but the problem with those algorithms is that

they detect multiple bounding boxes surrounding the objects in the image. Hence they are not applicable in our case that is detecting pedestrians on crowded roads. Here's where Non maximum suppression(NMS) comes to rescue to better refine the bounding boxes given by detectors.

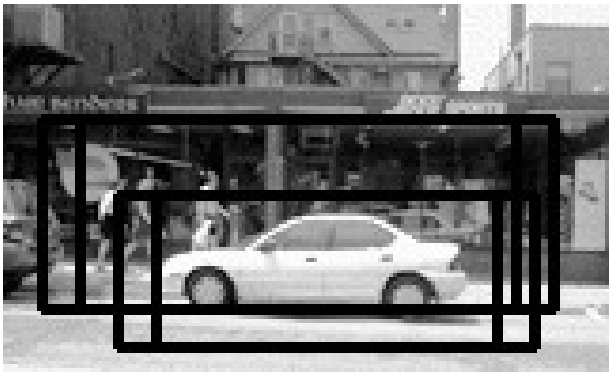

Figure 2.5: Multiple sliding windows that detected a car inside an image

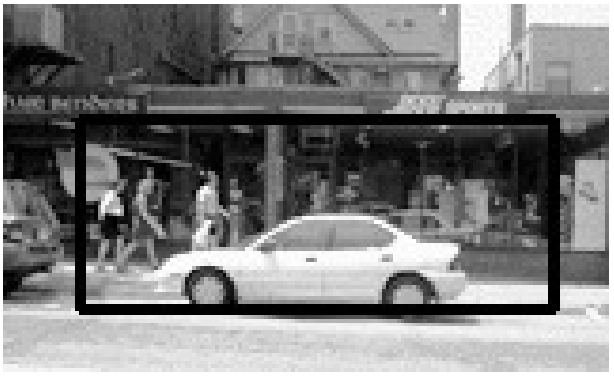

Figure 2.6: Filtered detected car sliding window using NMS

2.5 Linear interpolation of arctangent function

The orientation of the gradients is computed using 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 LUT-based approach of 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.

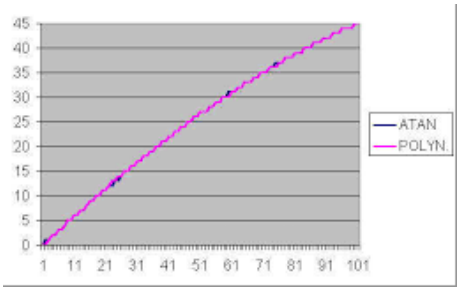

Figure 2.7: Linear Interpolation of arctangent function

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 method of approximating arctangent computation in heterogenous embedded applications.

Chapter 3

HOG Algorithm with SVM Classification Implementation

In this chapter, we shall discuss the implementation of the algorithm to extract the HOG features for a given input image. Then we shall use those image features as input to the SVM classifier to detect whether there is a car in that image. Here, a linear hyperplane is sufficient to classify the image and detect whether a car is present in the image, hence Linear SVM is implemented. Initially, the algorithm is implemented in Python to extract the necessary vectors needed in the hyperplane to compute the confidence value.

3.1 Python Implementation

I have created a single python script that can be used to test the code. The script will download the UIUC Image Database for Car Detection and train a classifier to detect cars in an image. The SVM model files will be stored in the directory, so that they can be reused later on.

To do the implementation five modules were done:

- **Feature Extraction** – This module is used to extract HOG features of the training images.
- **Train Classifier** – This module is used to train the classifier.
- **NMS** – This module performs Non Maxima Suppression.
- **Test Classifier** – This module is used to test the classifier using a test image.
- **Configuration file** – Imports the configuration variables from config file.

We know the equation of a hyperplane -

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

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

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

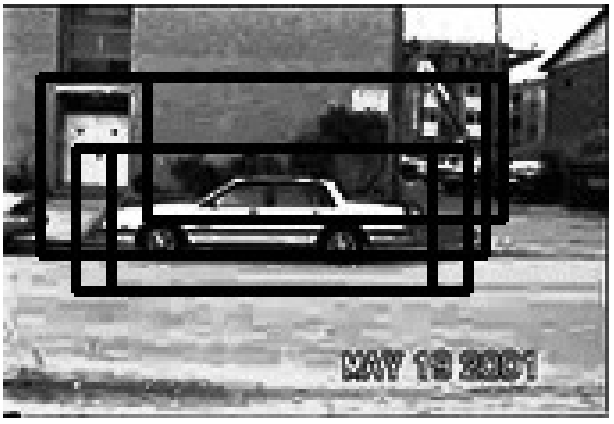

Figure 3.1: Car detection before applying NMS

Figure 3.2: Car detection after applying NMS

3.2 Design Implementation

After executing the 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 soft core Microblaze processor.

3.2.1 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* ← *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 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.

3.2.2 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 describes the arctangent algorithm necessary to get the orientation of the gradients by using linear interpolation. It is one of the fast method used to determine the phase in embedded applications.

3.3 Vivado Block Design

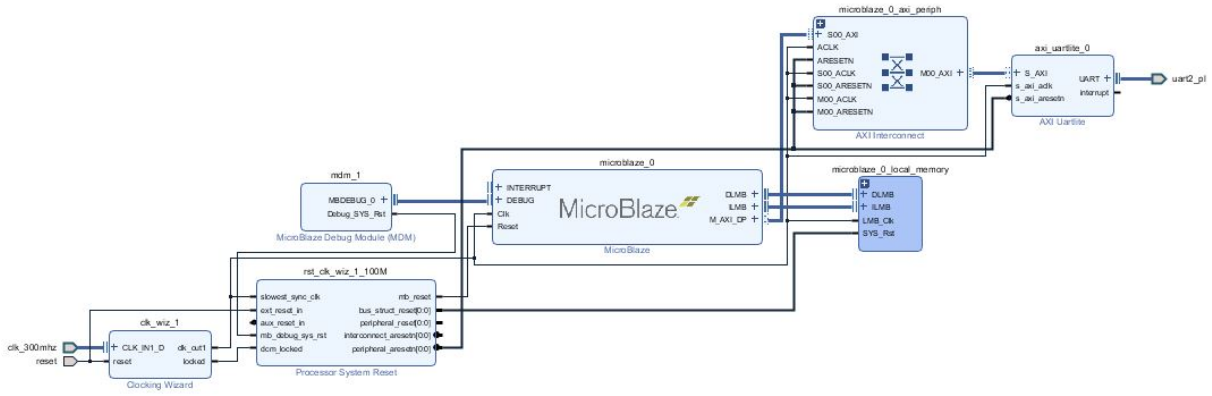

Figure 3.3: Block Design in Vivado

So, the entire algorithm is implemented in Vivado and integrated with Microblaze softcore processor. Here, the algorithm in 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.

3.4 Power Analysis in Vivado

Having designed the algorithm in python and then implementing the algorithm in Xilinx Vivado , integrating with soft core 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

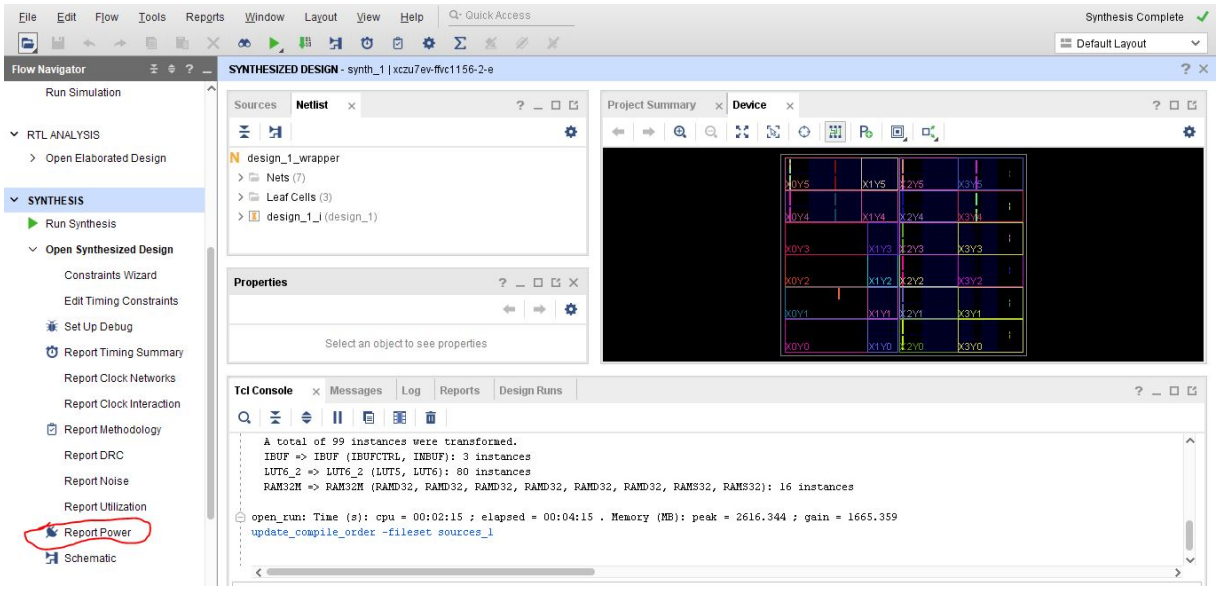

Figure 3.4: Synthesized Design in Vivado

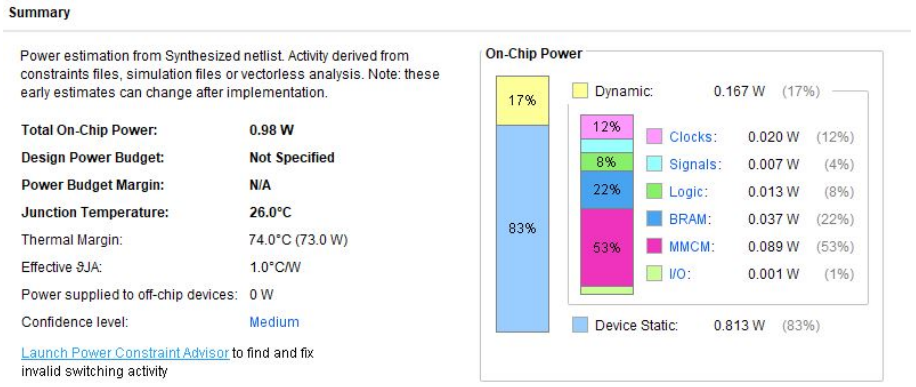

Figure 3.5: Power Analysis in Vivado

Chapter 4

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 about the HOG algorithm , how to extract features and then use those features in the SVM 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 SVM 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 Vivado Design suite. After that we create the block design in Vivado and run the Power Analysis.

Chapter 5

Conclusion and Future Work

Till now, we are able to write the entire algorithm in System C and run it on a Microblaze processor in Vivado. We were able to determine the power analysis in Vivado Suite. The system having HOG as IP is already been done by Neelam Sharma as her RnD project. So, the next step is to compute the power analysis of that system. Then, I would compare both the systems and draw baseline for the current heterogenous architecture.

Chapter 6

Appendix

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

6.1 HOG Feature extraction

```
from skimage.feature import local_binary_pattern
from skimage.feature import hog
from skimage.io import imread
import joblib
import argparse as ap
import glob
import os
from config import *

def feature_extractor():

    args = {}
    args["path"]='../TrainImages'
    args["descriptor"]="HOG"
    path = args["path"]
    cnt=0
    des_type = args["descriptor"]

    pos_feat = './features/pos'
    neg_feat = './features/neg'

    # If directories don't exist, create them
    if not os.path.isdir(pos_feat):
        os.makedirs(pos_feat)

    if not os.path.isdir(neg_feat):
        os.makedirs(neg_feat)
```

```
28
29
30      for i in os.listdir(path):
31
32       count=count+1
33        if os.path.isfile(os.path.join(path,i)) and 'pos' in i:
34
35          image_act_path=path+'/'+i
36          im = imread(image_act_path, as_gray=True)
37          if des_type == "HOG":
38              fd = hog(im, orientations, pixels_per_cell, cells_per_block, visualize=
                    visualize,transform_sqrt=transform_sqrt)
39          fd_name = os.path.split(i)[1].split(".")[0] + ".feat"
40          fd_path = os.path.join(pos_feat, fd_name)
41          joblib.dump(fd, fd_path)
42
43        elif os.path.isfile(os.path.join(path,i)) and 'neg' in i:
44          image_act_path=path+'/'+i
45          im = imread(image_act_path, as_gray=True)
46          if des_type == "HOG":
47              fd = hog(im, orientations, pixels_per_cell, cells_per_block, visualize=
                    visualize,transform_sqrt=transform_sqrt)
48          fd_name = os.path.split(i)[1].split(".")[0] + ".feat"
49          fd_path = os.path.join(neg_feat, fd_name)
50          joblib.dump(fd, fd_path)
51
52      print("Positive features saved in {}".format(pos_feat))
53
54      print("Negative features saved in {}".format(neg_feat))
55
56
57
58      print("Calculated features from training images")
59      print(count)
60
61  feature_extractor()
```

6.2 Training the Algorithm

```
1  from skimage.feature import local_binary_pattern
2  from sklearn.svm import LinearSVC
3  from sklearn.linear_model import LogisticRegression
4  import joblib
5  import argparse as ap
6  import glob
7  import os
8  from config import *
```

```
import numpy as np
from sklearn.svm import SVC

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()
```

6.3 Non Minimum Suppression

```
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)))
```

6.4 Testing the Algorithm

```
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')

```

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

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

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

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.

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

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

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

[5] https://medium.com/@mithi/vehicles-tracking-with-hog-and-linear-svm-c9f27eaf521a

[6] 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.