FPGA-BASED PARALLEL HARDWARE ARCHITECTURE FOR REAL-TIME OBJECT CLASSIFICATION

by
Murad Mohammad Qasaimeh

A Thesis Presented to the Faculty of the
American University of Sharjah
College of Engineering
in Partial Fulfilment
of the Requirements
for the Degree of

Master of Science in
Computer Engineering

Sharjah, United Arab Emirates

June 2014

Approval Signatures

We, the undersigned, approve the Master's Thesis of Murad Mohammad Qasaimeh.

Thesis Title: FPGA-based Parallel Hardware Architecture for Real-Time Object Classification.

Signature	**Date of Signature**
___________________________ Dr. Tamer Shanableh Associate Professor, Department of Computer Science and Engineering Thesis Advisor	_______________
___________________________ Dr. Assim Sagahyroon Professor, Department of Computer Science and Engineering Thesis Co-Advisor	_______________
___________________________ Dr.Tarik Ozkul Associate Professor, Department of Computer Science and Engineering Thesis Committee Member	_______________
___________________________ Dr. Khaled Assaleh Professor, Department of Electrical Engineering Thesis Committee Member	_______________
___________________________ Dr. Assim Sagahyroon Head, Department of Computer Science and Engineering	_______________
___________________________ Dr. Hany El-Kadi Associate Dean, College of Engineering	_______________
___________________________ Dr. Leland Blank Interim Dean, College of Engineering	_______________
___________________________ Dr. Khaled Assaleh Director of Graduate Studies	_______________

To my lovely family

for their endless love and support

Abstract

Object detection is one of the most important tasks in computer vision. It has multiple applications in many different fields such as face detection, video surveillance and traffic sign recognition. Most of these applications are associated with real-time performance constraints. However, the current implementations of object detection algorithms are computationally intensive and far from real-time performance. The problem is further aggravated in an embedded systems environment where most of these applications are deployed. The high computational complexity makes implementing an embedded object detection system with real-time performance a challenging task. Consequently, there is a strong need for dedicated hardware architectures capable of delivering high detection accuracy within an acceptable processing time given the available hardware resources. The presented work investigates the feasibility of implementing an object detection system on a Field Programmable Gate Array (FPGA) platform as a candidate solution for achieving real-time performance in embedded applications. A parallel hardware architecture that accelerates the execution of three algorithms is proposed. The algorithms are: Scale Invariant Feature Transform (SIFT) feature extraction, Bag of Features (BoF) and Support Vector Machine (SVM). The proposed architecture exploits different forms of parallelism inherent in the aforementioned algorithms to reach real-time constraints. A prototype of the proposed architecture is implemented on an FPGA platform and evaluated using two benchmark datasets. On average, the speedup achieved was ×55.06 times when compared with the feature extraction algorithm implemented in pure software. The speedup achieved in the classification algorithm was ×6.64 times. The difference in classification accuracy between our architecture and the software implementation was less than 3%. In comparison to existing hardware solutions, our proposed hardware architecture can detect an additional 380 SFIT features in real-time. Additionally, the hardware resources utilized by our architecture are less than those required by existing solutions.

Search Terms: Field Programmable Gate Array (FPGA), Scale Invariant Feature Transform (SIFT), Support Vector Machine (SVM), Object Recognition, Object Detection.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

BoF	Bag of Features
CLBs	Configurable Logic Blocks
DoG	Difference of Gaussian
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
GPU	Graphic Processing Unit
LUT	Lookup Table
ML	Machine Learning
SDK	Xilinx Software Development Kit
SIFT	Scale Invariant Feature Transform
SVM	Support Vector Machine
XPS	Xilinx Platform Studio

Chapter 1: Introduction

1.1 Overview

Object detection is the process of finding and identifying objects of a certain class, such as faces, cars, and buildings, in an image or a video frame. This task involves classifying the input image according to its visual content into a general class of similar objects. For example, the object detection system checks the input image if it contains a car or not. Over the few last decades, many approaches have been implemented to solve the object detection problem.

Object detection has many applications in different areas. In industry, it is used in quality control and inspection tasks to identify defective objects in production lines. Detection systems in automotive and driver assistance systems can be used to help the drivers to detect traffic signs and adjacent cars [1]. In surveillance and authentication systems, object detection is used to identify the face of suspicious people by using fixed cameras mounted in streets or at airports [2]. It is also used in autonomous robots, intelligent traffic systems, and computer human interactions, to name a few. Figure 1 shows some of object detection applications.

The state-of-the-art object detection algorithms for feature extraction and classification are often computationally expensive. Furthermore, these algorithms deal with a large amount of data that makes it impossible to reach real-time performance using software implementation only. Therefore, there is a strong need to propose a dedicated hardware architecture that performs the complex object detection algorithms in an efficient way in order to reach the real-time performance.

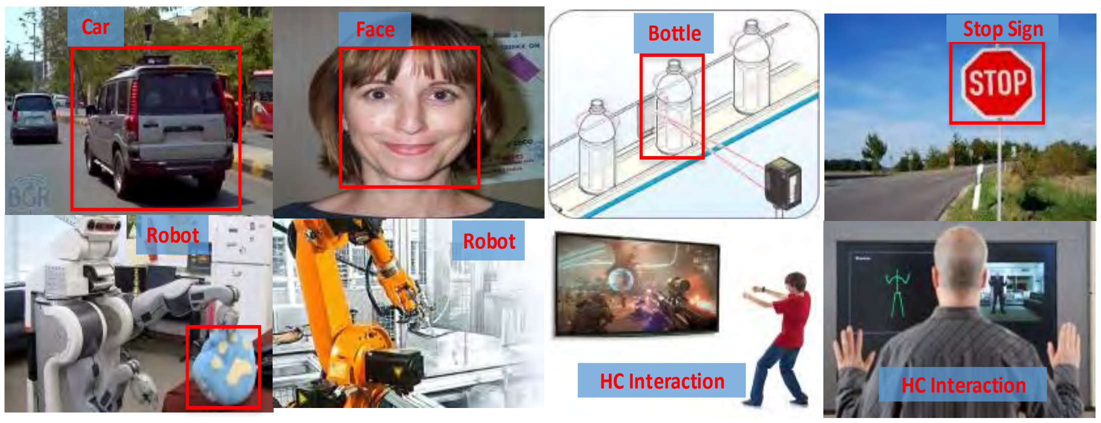


Figure 1: Object Detection Applications

Feature-based object detection is the most common object detection method in computer vision. It typically uses one of the feature extraction algorithms to extract the image's important content. It also uses one of the classification algorithms to build a classifier that can catalogue new unlabelled images into one of the trained categories.

Many features extraction algorithms have been suggested for use in the last decade. Each algorithm tried to improve the distinctiveness and robustness against possible image transformations. Canny and Sobel's edge detectors [3], Harris and features from accelerated segment test (FAST) corner detectors [4, 5] are local feature extraction algorithms that can be used in object recognition systems. However, the features extracted by these algorithms are highly sensitive to image transformation such as scale, rotate, translate, and shear.

The Scale-invariant feature transform (SIFT) algorithm proposed by Lowe [6, 7] is considered as one of the most robust image features extraction algorithms. The SIFT features are invariant to change in image scale, rotation, illumination, 3D camera viewpoint, and noise. It has been reliably employed in many applications in computer vision. In this work, we used the SIFT algorithm for feature extraction. For classification, we used multiclass support vector machine (SVM) algorithm.

To have a sense of the computational complexity needed by the feature extraction and classification algorithms in the object detection system, the SIFT algorithm was implemented on a PC with an Intel® i5 1.66 GHz processor with 8 GB RAM [8]. The authors indicated that it takes around 1.1 seconds to extract 514 features from an image of 320×240 pixels (the software codes are written by Hess [9]). This does not include the time required in the classification step that depends on the dataset size and number of classes. In [10], the authors used a Pentium 4 CPU 3.2 GHz PC to implement a recognition system with a video-based database size of 100 images (320×240 pixels). The recognition time was around 6 seconds, this being far from the desired real-time performance, specifically 30 frames per seconds (around 33 ms).

To reach real-time performance, some attempts were made to simplify these algorithms in order to reduce the computational complexity, but it was at the expense of recognition accuracy [11]. Others tried to parallelize the algorithm execution using multicore CPUs [12], or Graphic processing units (GPUs), or a field programmable gate array (FPGA) [8, 11, 13, 14].

By using one of these accelerators, more than one task can be executed in parallel so that the overall recognition time is reduced. However, the multi-core CPU and GPU implementations require excessive hardware resources all while consuming too much power thereby making them unsuitable for embedded systems. Nevertheless, Field-Programmable Gate Array (FPGA) integrated circuits provide a promising solution to implementing real-time object recognition systems using low resources and a power budget.

The FPGA implementation can execute the algorithm's operations in parallel by designing a circuit dedicated to each operation. The distributed memory blocks in the FPGA can be used as cache that can be accessed in parallel. Therefore, FPGA embedded platforms may exhibit high performance in executing image processing algorithms.

In this research work, we explore the possibility of achieving real-time performance for object recognition system using FPGA embedded platforms. The algorithms used in the object recognition are SIFT and Bag of Features for feature extraction and modelling, and SVM for classification. These are state-of art algorithms that have achieve high recognition accuracy but are still computationally intensive.

1.2 Problem Statement

Current state-of-the-art object detection algorithms have high computational complexity and require large memory resources. Therefore, the implementation of these algorithms in embedded systems has a very low frame rate which is far from the real-time performance. In real-time systems, it is desired to process 30 video frames per second. The resources and computational power restrictions inherited in the embedded platforms make implementing these algorithms in real-time a challenging task that needs to be addressed. Therefore, there is a need for customized parallel hardware architecture to accelerate the object detection algorithms within the computational resource available inside the embedded systems. FPGA provides a promising solution to solve this problem by designing parallel hardware architecture able to perform the feature extraction and classification algorithms in an efficient way in order to reach real or near real-time performance.

1.3 Thesis Contribution

The main contribution of this research work can be summarized as follows:

- Implement an FPGA-based hardware architecture to detect objects within real-time constraints. More specifically, the target is to process 30 video frames per second.
- Accelerate the computationally intensive feature extraction algorithm (SIFT) by exploiting the parallelism in the algorithm.
- Develop a new technique to build the DoG pyramid in the SIFT algorithm by using multiplierless multiple constant multiplication with common expression elimination algorithm to reduce the Hardware utilization.
- Develop a new method to implement multi-port memory in a SIFT descriptor.
- Design new architecture for computing the dominant key point orientation in the SIFT algorithm.
- Accelerate the support vector machine classifier by using concurrent data-path units to compute different parts of the kernel.
- Build extendable SVM classifier hardware architecture to implement one-against-all multiclass SVM algorithm.
- Analysis of the computational load of each step in the object detection system and to estimate the parallelism gain required to reach the real-time performance.

1.4 Thesis Outline

This thesis is outlined as follows:

Chapter 1 includes a brief description of embedded object detection systems and their performance limitations. It also includes the problem statement and thesis contribution.

Chapter 2 presents the theoretical background required for this thesis. It explains SIFT feature extraction algorithm, bag of features modelling, and the support vector machine algorithm. It also discusses FPGAs technology and its suitability to implement image processing algorithms.

In Chapter 3, we explore the literature that is relevant to our problem. We review the related work in accelerating the feature extracting algorithms and classification algorithm using FPGAs. We also summarize the results achieved so far and the limitations in each implementation.

In Chapter 4, we present the software implementation of SIFT, BoF, and SVM algorithms using Matlab. We also compare the classification accuracy of a number of well-known classifiers.

In Chapter 5, we describe our hardware architecture for SIFT, BoF, and SVM algorithms in details. We describe the internal architecture of each module and discuss how it works. The FPGA implementation of our prototype is also presented. The Microblaze based embedded system used for our architecture is explained

In Chapter 6, the experimental results are summarized. The accuracy of each module is compared to the accuracy of its software implementation. The hardware utilization and processing time are measured and reported. A comparison with previous solutions is also presented.

In Chapter 7, we conclude the work of this thesis and suggest future work.

Chapter 2: Background

This chapter presents some background for the research presented in this thesis. Section 2.1 reviews the scale invariant feature extraction algorithm, SIFT, used in our work. In Section 2.2, we discuss the concept of the bag of feature modeling. Section 2.3 presents the theory of the support vector machine (SVM) classification algorithm. Lastly, Section 2.4 introduces the field programmable gate array (FPGA) technology, and its efficiency in image-processing algorithms.

2.1 Scale Invariant Feature Transform (SIFT)

SIFT is an algorithm used to extract and describe local features in images. It was proposed by Lowe [6] in 2004 and was patented in the U.S. to University of British Columbia (UBC). The features extracted by the SIFT algorithm are invariant to change in scale, image rotation, different illumination, change in camera viewpoint, or addition of noise. It is considered as one of the most robust feature detection and description algorithm in computer vision. Therefore, SIFT is one of the most trusted and widely used feature extraction algorithm in the field. SIFT algorithm takes N × N pixels image as input and produces a set of distinctive features that represent the image content.

The SIFT algorithm process can be divided into two main stages: keypoint detection, and keypoint description. In the first stage, the image is scanned to search for distinctive and repeatable points called Keypoints. This stage consists of three sub-tasks: Gaussian scale space generation, local extrema detection, and keypoint detection. The second stage generates the keypoint descriptors that are divided into two sub-tasks: dominant orientation assignment and descriptor generation. SIFT's steps are shown in the Figure 2 and further details are available in [6].

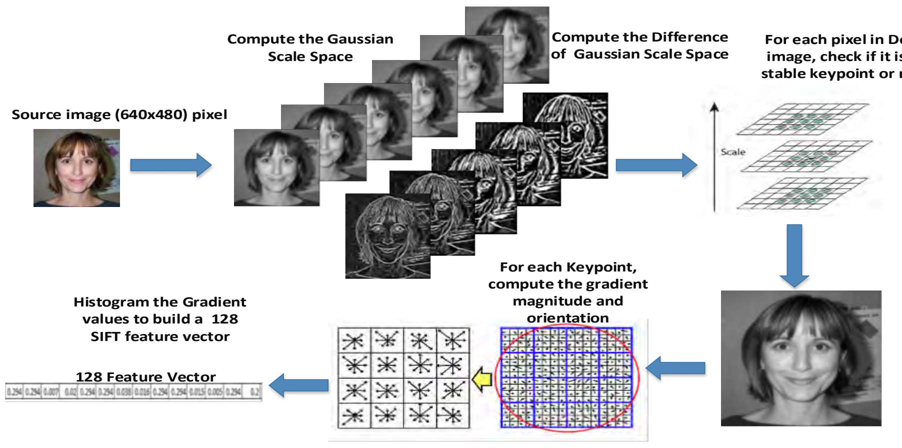

Source image (640x480) pixel
Compute the Gaussian Scale Space
Compute the Difference of Gaussian Scale Space
For each pixel in DoG image, check if it is a stable keypoint or not
Scale
For each Keypoint, compute the gradient magnitude and orientation
Histogram the Gradient values to build a 128 SIFT feature vector
128 Feature Vector

Figure 2: SIFT Feature Extraction Algorithm's Steps

2.1.1 Gaussian scale space generation. In the first step, the input image I(x, y) is convolved with a series of Gaussian filters G(x, y, σi) to build a Gaussian scale space as defined by equations (1-2). Where σ_i is the Gaussian filter scale, L(x, y, σ_i) is the Gaussian filtered image and i is a scale index. The * in equation (1) is 2-D convolution operation in x and y, and S is the number of scaled images to be generated, and G(x, y, σ) is the Gaussian Kernel.

$$L(x, y, \sigma i) = G(x, y, \sigma i) * I(x, y) \qquad (1)$$

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)} \qquad (2)$$

$$\textit{for } \text{i=0, 1 … S+2}$$

Table 1 summarizes the Gaussian filter's standard deviations for the first octave of Gaussian scale space. In this work, we used six scales (σ0 , k.σ0 , k^2. σ0, k^3. σ0, k^4. σ0, k^5. σ0), where σ0=1.6, and k= $\sqrt[3]{2}$.

The first set of the S+2 Gaussian filtered images is called the first octave of the Gaussian scale space. The second octave is derived by down-sampling the Ls image in the first octave into half size and repeating the same operations that were applied to the first octave. After computing the Gaussian filtered images, the next step is to build the difference of Gaussian scale space (DoG) by subtracting each two consecutive images in the same octave, as defined in equation (3).

$$D(x, y, \sigma) = L(x, y, K\sigma) - L(x, y, \sigma) \qquad (3)$$

Figure 3 shows the process to construct the DoG pyramid from the input image. In this figure there are two octaves, each one has a group of five Gaussian filter images and four DoG images.

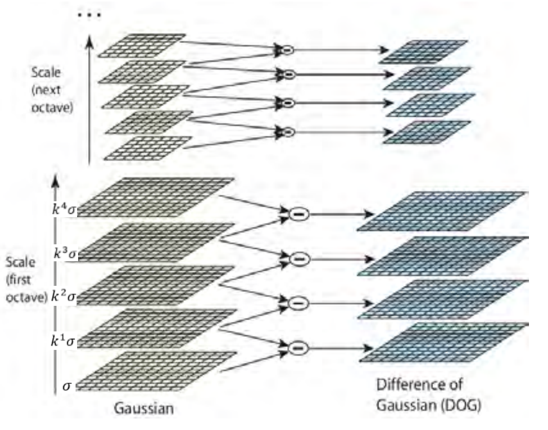


Figure 3: Gaussian Scale Space Pyramid *[6]*

Table 1: Gaussian filter's scales for the first octave

Filter Number	Filter Scale	Filter scale value
1	σ	1.6
2	$k^1\sigma$	$\sqrt[3]{2}^{1}1.6$
3	$k^2\sigma$	$\sqrt[3]{2}^{2}1.6$
4	$k^3\sigma$	$\sqrt[3]{2}^{3}1.6$
5	$k^4\sigma$	$\sqrt[3]{2}^{4}1.6$
6	$k^5\sigma$	$\sqrt[3]{2}^{5}1.6$

2.1.2 Local extrema detection. In this step, the DoG image is scanned to find the candidate key points. Each pixel in the D(x, y, σ) image at location (x, y) is compared with its 3×3 neighbours in the same scale and the adjacent scales. If the pixel is local maxima or local minima out of the total 26 neighbouring pixels, it will be considered as a candidate keypoint. This operation is performed for every pixel in the DoG images and what results is a list of keypoint candidates. Figure 4 shows the keypoint's 3×3 neighbours.

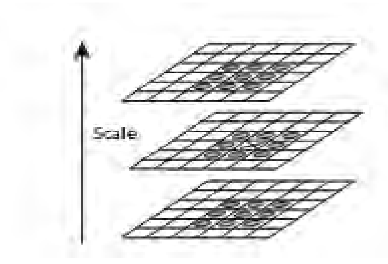


Figure 4: Keypoint's 3×3 neighbours *[6]*

2.1.3 Keypoint detection. The goal of this step is to eliminate the candidate key points that have low contrast or poorly localized along edges. To detect a low contrast keypoint, the value of the pixel is compared with a predefined threshold. If the value is less than the threshold, the keypoint will be rejected.

To find the poorly localized peak, the keypoint is tested using the inequality defined in equation (4). Where H is the Hessian matrix computed as defined in (5), and Tr (H) is the trace of H, Det(H) is the determinant of H, and r is a constant value. The Δs are computed as given in equations (6, 7, and 8). In this work, we used the threshold value equal to 3, and constant value for r equal to 10, which eliminates keypoints that have a ratio between the principal curvatures greater than 10.

$$\frac{\mathrm{Tr(H)}^2}{Det(H)} < \frac{(r+1)^2}{r} \tag{4}$$

$$\mathrm{H} = \begin{bmatrix} \Delta\mathrm{xx} & \Delta\mathrm{xy} \\ \Delta\mathrm{xy} & \Delta\mathrm{yy} \end{bmatrix} \tag{5}$$

$$\Delta\mathrm{xx} = \mathrm{D(x+1,y)} + \mathrm{D(x-1,y)} - 2D(x,y) \tag{6}$$

$$\Delta\mathrm{yy} = \mathrm{D(x,y+1)} + \mathrm{D(x,y-1)} - 2D(x,y) \tag{7}$$

$$\begin{aligned}\Delta\mathrm{xy} = (&\mathrm{D(x+1,y+1)} - \mathrm{D(x-1,y+1)} - \\ &\mathrm{D(x+1,y-1)} + \mathrm{D(x-1,y-1)})/4\end{aligned} \tag{8}$$

2.1.4 Orientation assignment. The gradient magnitude and orientation are computed for all pixels around the stable keypoints. The gradient is computed in both the horizontal and vertical direction as defined in equations (9) and (10).The gradient magnitude and gradient orientation are computed from (Δx)and (Δy) as given in equations (11) and (12).

$$\Delta x = L(x + 1, y) - L(x - 1, y)/2 \tag{9}$$

$$\Delta y = L(x, y + 1) - L(x, y - 1)/2 \tag{10}$$

$$m(x, y) = \sqrt{\Delta x^2 + \Delta y^2} \tag{11}$$

$$\theta(x, y) = \tan^{-1}(\frac{\Delta y}{\Delta x}) \tag{12}$$

2.1.5 Descriptor generation. To compute the SIFT descriptor, there are two main tasks: dominant orientation computation and descriptor generation. The dominant orientation is computed by building the gradient orientation histogram around the keypoint. The gradient orientations in the region are mapped into one of 36 bins, where each bin represents 10 degrees. In the end, the bin with the largest value or count represents the dominant orientation, as shown in Figure 5.

After computing a dominant orientation, the coordinates of the pixels around a keypoint are rotated relative to the dominant orientation. A SIFT descriptor is computed by dividing the region around the keypoint into 4x4 squares and building the gradient histogram over 8 bins, where each bin covers 45 degree. The gradient magnitudes are weighted by Gaussian filter before building the histogram. Lastly, the 16 histograms with 8 bins are each represented in the SIFT descriptor. The SIFT descriptor has 4x4x8 values. Figure 6 shows the first 2x2 blocks out of the 4x4 blocks around a keypoint.

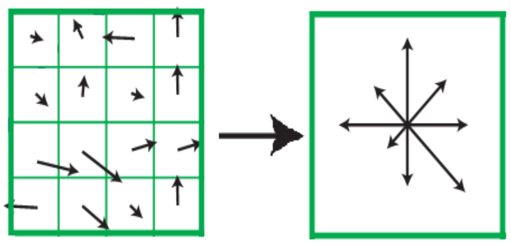


Figure 5: Dominant Orientation's Histogram [6]

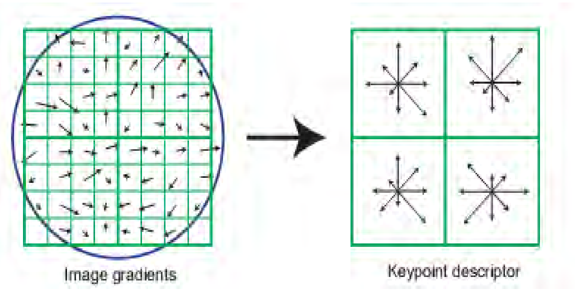


Figure 6: the first 2x2 blocks in SIFT descriptor [6]

2.2 Bag of Features Model

Bag of feature modelling (BoF) is one of the most popular representation methods used to simplify the representation of an image content. The idea is to quantize each of the extracted keypoints into one of the visual words (points in the feature space), and then to represent each image by a histogram of the visual words. Representing images using the BoF model has many benefits such as, simplifying the image representation and representing the images in an unified input format that is required for classification algorithms.

The process starts by clustering the SIFT descriptors in their feature space (128-dimensional space) in large numbers of clusters by using the K-means clustering algorithm. The centroids of these clusters then become the BoF visual words. The visual words represent a particular local pattern shared by the keypoints in that cluster. The next step is to assign each SFIT descriptor into its nearest cluster center. The normalized histogram of the quantized features is the BoF model representation of that image. Figure 7 shows the process of BoF modelling.

The BoF image representation procedure consists of two steps:

2.2.1 Building the vocabulary. In this step, SIFT descriptors are clustered using the k-means algorithm in order to find the best k clusters. The process of the k-means clustering an algorithm starts by choosing a set of centroid positions. Next, one must measure the distance between features and the centroid positions using one of the common distance measuring equations, such as Manhattan, Euclidean, or Mahalanobis.

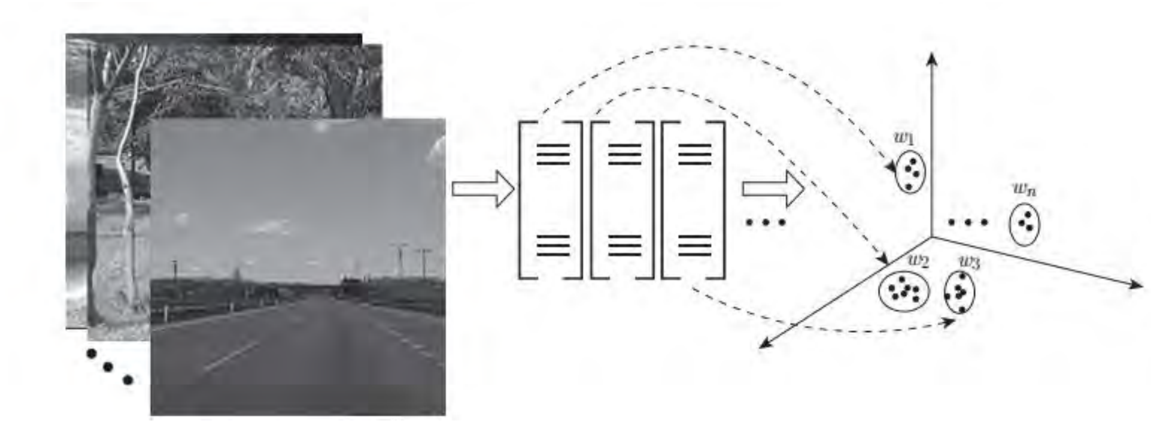


Figure 7: The process of Bag of Feature modelling

After the assignment, the centroids are shifted to the average location of all the keypoints assigned to them, and the assignments are redone. This procedure repeats until the assignments stop changing. The result will be k centroids representing our BoF visual words.

The number of the clusters (codebook size or k) depends on the type of the input data. If we choose it to be too small then each the bag of words vector will not represent all the keypoints extracted from its related image. If it was too large then there is quantization over fitting because of insufficient samples of the keypoints extracted from the training image.

2.2.2 Term generation. The goal of this step is to build a term vector, where the term's weight represents the frequency of word occurrence in the image. The computed weight for term n in the term vector w, denoted by w_n is defined using equation (13).

$$wn = \sum_{i=1}^{N} \sum_{j=1}^{Mi} \frac{1}{2^{i-1}} \, dist(j, n) \qquad (13)$$

where:

- Mi: is the number of features in the image.
- dist(j,n): measures the distance between feature j and term n.
- N: Number of neighbours used in the soft weighting strategy; this is used when the features are close to two terms and almost the same distance apart.

The output of the bag of features model is a term vectors that represent the image. Figure 8 shows the whole process for three class images with a code size equal to four (k=4).

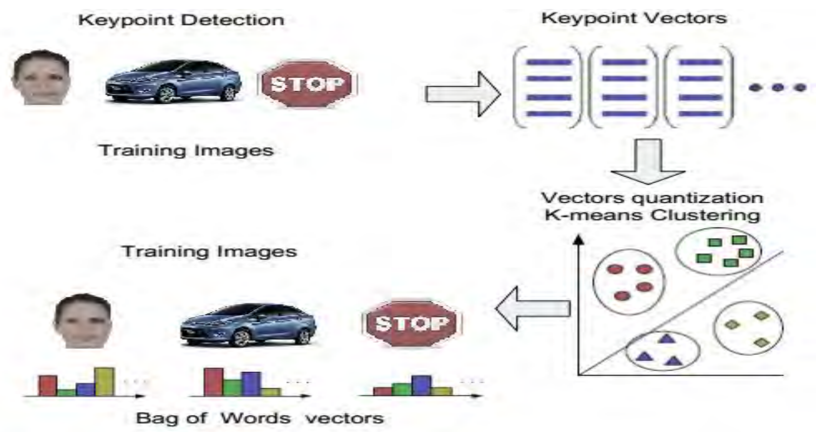


Figure 8: BoF modelling for three class images with code size equal 4

2.3 Support Vector Machine (SVM)

The literature shows that many classification algorithms have been developed. These include naïve Bayes classifier [15], k-nearest neighbours [16], Adaboost, and decision trees. Each of these algorithms has a good classification rate. However, the support vector machine (SVM) classifier shows the highest classification rates and outperforms other algorithms in many computer vision applications.

SVM is a machine-learning algorithm proposed by Vapnik in 1990s. The algorithm's idea is to find the decision hyperplane that defines decision boundaries between different classes. Given a set of training examples, each one is labelled to one of two categories, D= {(x, y)| x→ data sample, y→ class label}. The SVM algorithm finds the optimal hyperplane that splits the training data into two categories. Any new examples can be classified based on the location relative to that hyperplane.

SVMs utilize a technique called kernel trick that represents the data in a higher dimensional space than the original feature space. This mapping makes it easier to find a separation hyperplane in non-linearly separable data. The most common kernels are Linear, Polynomial, Radial Basis Function (RBF), and Sigmoid kernels:

Linear: $$K(x,z) = x \cdot z \qquad (14)$$

Polynomial: $$K(x,z) = \left((x \cdot z) + 1\right)^{d}\,,\ d > 0 \qquad (15)$$

RBF: $$K(x,z) = \exp(-\| x - z \|^{2}/(2\sigma^{2}) \qquad (16)$$

Sigmoid: $$K(x,z) = \tanh(K(x \cdot z) + \Phi) \qquad (17)$$

SVM was originally designed as a binary classification algorithm, but it is possible to extend it to multiclass classification. The most common methods of multi-class SVM are one against all and the pairwise classifiers method:

- One-against-all method: for n classes, we need n binary SVMs, so that in the ith classifier the objective is to separate class (i) from the rest of the classes. The ith classifier is trained with all of the examples in the ith class with positive labels, and all other with negative labels.
- Pairwise classification method: for n classes, we need $n(n$-1)/2 binary SVM classifiers. There is one classifier for each pair of classes. Each classifier gives the input vector {0} if input is class (i_1) or {1} if input is class (i_2). The class with the most votes will be assigned to the input vector.

The goal of the training step in SVM is to find a separation hyperplane that provides a maximum margin between two different classes. For example, given *l* training sample $\{x_i, y_i\}$, i=1….*l*, where $x_i \in R^n$, and a class label y= {-1, 1}. The goal is to find a hyperplane in R^n expressed in the equation that separates the data into two classes:

$$\mathbf{w} \cdot x + b = 0 \tag{18}$$

Where **w** is a vector orthogonal to the hyperplane, and b is a constant. To find the hyperplane that maximizes the distance to the closest data points (also known as Support Vectors), the following Quadratic Programming Problem (QP) is solved:

$$\text{Minimize: } W(\alpha) = -\sum_{i=1}^{l} \alpha i + \left(\frac{1}{2}\right) \sum_{i=1}^{l} \sum_{j=1}^{l} yi \cdot yj \cdot \alpha i \cdot \alpha j(xi \cdot xj) \tag{19}$$

$$\text{Subject to: } \sum_{i=1}^{l} yi \cdot \alpha i = 0 \qquad 0 \le \alpha i \le C \tag{20}$$

Where α is the vector of *l* Lagrange multipliers to be determined, and C is a constant. In this work, the training phase of the SVM is implemented in the MATLAB environment using the LibSVM library. Upon system training, the α and y vectors computed in Matlab, are stored and used in the testing phase implemented in hardware as described in Chapter 5.

In this work, we used the nonlinear SVM with the Gaussian kernel (RBF) decision function:

$$d(x) = \sum_{i=1}^{SV} \alpha i \cdot yi \cdot e^{-\gamma \|x - xi\|^2} + b = 0 \tag{21}$$

Where SV is the support vectors, γ equal to $1/2\sigma^2$ and b is bias constant. In this work, we implement the decision function in hardware. First, we find the optimal value for γ, then using the LibSVM library training code, the αi and yi vectors are extracted. The αi and yi vectors are used in hardware to compute the final class.

2.4 Field Programmable Gate Array (FPGA)

The Field-Programmable Gate Array (FPGA) is a semiconductor device designed to be configured after manufacturing. It consists of a matrix of configurable logic blocks (CLBs) of different types, including general logic, memory, and multiplier blocks, surrounded by a programmable routing fabric that connects the CLBs, as shown in Figure 9. The array is surrounded by input/output blocks to connect the chip to the outside world.

When the FPGA is programmed to implement customized digital circuits, each of the CLBs is configured to perform a simple unique logic function. The CLBs implement the logic function by using a lookup table-based mapping technique. The routing fabric is used to connect the CLBs together in order to build the complete digital circuit. The IO blocks make the connections from FPGA's logic matrix to the outside.

FPGAs have shown high performance in image processing applications. Although FPGAs have low operational frequency compared to a Graphic Processing Unit (GPU) and a general-purpose Central Processing Unit (CPU), they do have high performance in image processing applications for couple of reasons. First, FPGAs can efficiently expose high parallelism in image processing algorithms at many different levels, like bit and instruction levels. Second, they have a large number of internal memory banks that can be accessed in parallel which makes it suitable for processing algorithms like image processing. Third, most of operations in image processing algorithms are 8 bits operations [17, 18] that can be implemented efficiently in FPGAs. All of these factors make the FPGA a promising solution to implement high performance embedded image processing algorithms.

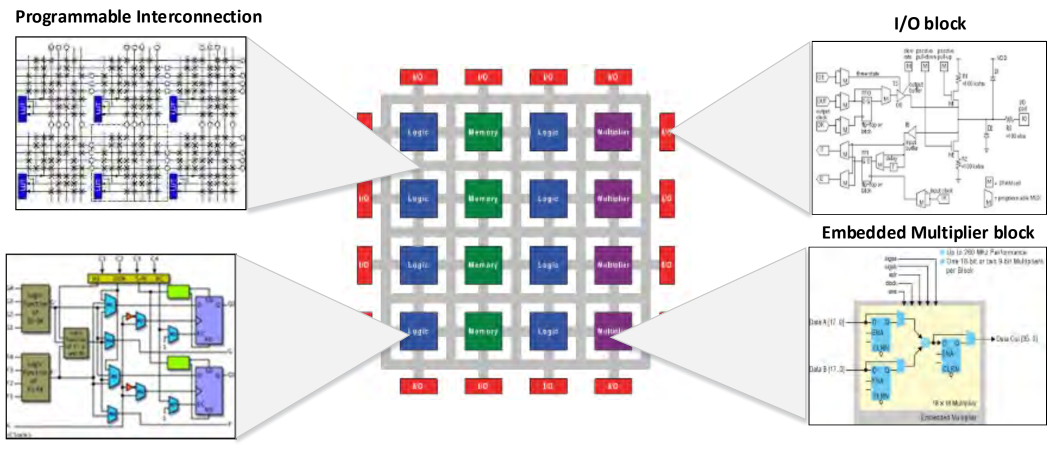


Figure 9: FPGA internal Architecture

Image processing algorithms have high inherent parallelism, and the data width of its operations is usually less than 16 bits. FPGA can execute these operations in parallel by designing a dedicated circuit for each operation and then by executing them concurrently. The distributed memory blocks in the form of the RAM blocks can be used as temporary storage bank that can be accessed in parallel. These features make it possible to implement a parallel hardware architecture for image processing applications with real-time performance.

In this work, we used the Verilog hardware description language to design our hardware architecture. We used a soft-core CPU called Microblaze to stream the data into the hardware architecture and control the dataflow. The Xilinx XUPV5 LX110T evaluation platform equipped with the Virtex-5 FPGA is used in this work. We also used Xilinx ISE Design Suite, Xilinx Platform Studio (XPS), and Xilinx Software Development Kit (SDK) platforms to design and implement the architecture.

Chapter 3: Literature Review

This chapter surveys recent published research in the field of accelerating object detection algorithms using hardware implementation on FPGA and GPU platforms. In Section 3.1, we summarize the existing solutions intended to accelerate the SIFT feature extraction algorithm using hardware implementation. Section 3.2 presents the work done to implement the SVM classifier using hardware architecture.

3.1 Hardware Implementations of Features Extraction algorithm

Since Lowe [6] proposed the SIFT algorithm in 2004, it has been one of the most complete and robust feature extraction algorithm in computer vision. However, it has a high computational complexity and memory requirements. These limitations make reaching real-time performance using a pure software implementation very difficult.

To solve this problem, a number of simplified versions of the SIFT algorithm have been proposed, such as the SURF [19] and the fast SIFT [20] algorithms. These algorithms tried to reduce the computational complexity of the SIFT algorithm by sacrificing the accuracy of the extracted features. Some of these solutions reached near real-time performance, but the quality of accuracy features was not good enough for many real-life applications.

Others tried to accelerate the SIFT algorithm using a GPU platform. The authors in [21] proposed a heterogeneous dataflow scheme to accelerate the SIFT algorithm using a GPU in a mobile device. They achieved a speedup of ×4-7 over an optimized CPU version and a ×6.4 speedup over other GPU implementations. In [22], a GPU-based SIFT implementation for image matching was proposed. These implementations might have reached near real-time frame rate performance, but they require an excessive amount of hardware resources and they consume too much power compared to other hardware platforms. This makes the GPU implementations of the SIFT algorithm not suitable for portable embedded systems with limited power.

Other solutions tried to accelerate the SIFT algorithm by fully utilizing the computing power of available multi-core processors. The results showed that the performance achieved by a multi-core CPU is almost the same as the implementation on GPUs.

Other attempts have been made to accelerate the SIFT algorithm using FPGA hardware architectures. In [8], a parallel hardware architecture for SIFT features extraction was proposed. The extracted features are used in the simultaneous localization and mapping (SLAM) problem. The architecture was a stand-alone architecture where it was able to read the input image directly from CMOS sensor. It also provides the results for on-chip applications or is accessible via an Ethernet connection. Furthermore, it provides some flexibility to customize the feature descriptors based on the application. The system architecture is composed of three hardware blocks that run in parallel (DoG, OriMag, and KP). The DoG block receives a stream of pixels from CMOS sensor and performs a Gaussian filter and difference of Gaussian operations. The results are then sent to both the OriMag and Kp blocks. The Kp block detects the stable keypoints, while the OriMag computes the gradient magnitude and orientation. The NIOS II processor generates the descriptor for each keypoint generated by the KP block based on the gradient magnitude and orientation produced by the OriMag block. The system was able to detect features from 320×240 pixel images at a rate of up to 30 frames per second. However, the system did not generate the SIFT descriptor in real-time because it was implemented in the soft core processor NIOS II. In this implementation, it took around 11.7 ms to generate one SIFT descriptor which is far from the real-time performance.

Other FPGA-based SIFT acceleration architecture [14] was proposed to resolve the power consumption and hardware resource usage problems. The proposed method divides the input image into a number of regions of interests (ROI) and processes each one individually as opposed to the whole image as the original algorithm. Moreover, the architecture used the implementation of two 1-D Gaussian filters instead of one 2-D Gaussian filters in order to reuse intermediate results efficiently. Moreover, to increase the overall throughput the architecture used a pipelining of three stages: ROI reading, Gaussian filtering, and key-point extraction. The Gaussian filtering module consists of 60 parallel systolic-array architecture used to distribute the workload. The keypoint extraction module consists of three keypoints extraction cores. The implementation can generate the SIFT descriptor vectors for images of size 640×480 pixels at a rate of 56 frames per second. However, the author claims that ROI-method will have a low accuracy degradation. Also, there are no details about the descriptor vector generation stage.

The SIFT algorithm was optimized by [11] to obtain a high-speed feature detector with a low hardware resource usage. The optimized algorithm uses only four scales with two octaves in the DoG stage. It also reduces the dimension of the SIFT feature descriptor from 128 values into 72 values. The system consists of two modules: the SIFT feature detection hardware core module and the SIFT feature generation software module. It has three pipelined stages: Read Data, Gaussian Smooth, and Feature Detect/GradOrien components that all operate in parallel. The parallel structure of Gaussian smooth component consists of seven smooth units in order to process 7x12 pixels arrived at from the Read Data component. The Feature Detect component has a high parallel structure, where it was able to complete 28 pairs of comparison within one clock cycle. The architecture was able to detect the SIFT features of an image of 640x480 pixels within 31 milliseconds. However, this optimising largely reduce the accuracy of the SIFT descriptor.

Zhong et al. [23] implemented the FPGA/DSP design for a SIFT feature extraction. The SIFT feature detection stage is implemented using FPGA with a parallel architecture to reduce the overall detection time, while the feature description stage is implemented using a high-performance fixed-point DSP chip. The system contains two parallel copies of the OriMag module and the stable keypoint detection module. The first copy is dedicated to octave 0, while the second copy is shared by octave 1 and octave 2. These modules communicate with the DSP processor through a high-performance processor interface (HPINF) module by sending the stable keypoints to generate the features description vectors. The Gaussian filter module used two 1-D kernels to replace the traditional 2D convolution in order to reduce the resource utilization. Their system performs the SIFT features detection at a speed of 100 frames/sec on images of size 320×256 pixels, and it takes about 80 µsec per feature in the description stage. However, this system depends on a DSP processor to generate the feature descriptors. In addition, it works with small image resolutions to be able to store the whole image in the chip.

The authors in [13] implemented a parallel hardware architecture with a three-stage pipeline SIFT accelerator. The architecture consists of two hardware components, one for key point detection, and the other for feature descriptor generation. They make the first component act as a main processor that reads the source image and detects the keypoints. For each stable keypoint, it invokes the second component (coprocessor) to start the feature descriptor generation process. Moreover, they propose a buffer scheme that reduces memory requirements by 50%. The module was able to detect the SIFT features within 3.4 ms for images of VGA resolution (640×480 pixels). The overall SIFT processing time, including the feature descriptor generation, is kept within 33ms when the number of feature points to be extracted is fewer than 890.

The authors in [24] build the SIFT accelerator to extract features from images of size (320 × 240) pixels. The accelerator reads the input image pixels stream, and then feeds it into two 1D Gaussian filters. In such an implementation, the intermediate values are stored using a buffering line technique. The results of the DoG operations are sent to gradient orientation and magnitude generation blocks. It is also sent to the keypoint detection and the stability checking blocks. The system is implemented in the Altera Cyclone III FPGA and achieved 30 frames per second, however, the hardware utilization for this architecture was high. Table 2 summarizes the SIFT accelerators' parameters, performance and hardware utilization of the implementations discussed in the literature.

Table 2: SIFT System's paramters, performance and hardware requirements

	[14]	[11]	[8]	[23]	[13]	[24]
Resolution	640 × 480	640×48	320 ×240	320×256	640×480	320 × 240
Frame rate(fps)	56	32	30	100	30	30
Op-Freq(MHz)	50	100	50	106	100	100
FPGA	Cyclone	Virtex-5	Stratix II	Virtex-4	-	Cyclone
# of LUTs	32,592	35,889	43,366	18,195	13,200	43,563
# of registers	23,247	19,529	19,100	11,821	-	14,730
# of DSP blocks	258	97	64	56	-	45
Memory (Mbits)	0.67	-	-	2.7	5.729	2.81

3.2 Hardware Implementation of Classification Algorithm

This section surveys the research work intended to accelerate the SVM algorithm using the FPGA hardware implementations. For each architecture, the kernel computation hardware implementations and other hardware modules are explained. The architectures' performance results and limitations are also discussed.

In [25], an FPGA-based nonlinear SVM architecture is implemented. The architecture exploited the potential parallelism in the matrix-vector computations. The support vectors are stored in the FPGA's internal memory and the test images are streamed from the external RAM. The support vectors and new instances are connected to a parallel processing unit called the classifier hypertile. The hypertile computes the kernel operations based on the input Support Vector SV and the input feature vector. The results are then added in parallel using an adder tree. The result of the decision function is streamed out of the system in order to be used in the next stage. The implementation results present a speed up factor of 2-3 compared to the CPU implementation and 7 times compared to a GPU implementation. However, the work includes only the two-class SVM algorithm implementation.

The authors in [26] implemented a parallel array architecture for the SVM-based object detection system. The architecture is based on an array of processing elements that compute the SVM feed-forward phase for an input image. The system consists of three main regions: the memory region that is comprised of a chain of memory units where the training data is stored. The vector processing region that is responsible for the vector processing, and the scalar region that processes the results produced from the vector operation. It also has the FSM control unit to synchronize the array operations. The system is implemented using the Virtex-5 FPGA and is evaluated using three applications: face detection, pedestrian, and car side-view detection. The result shows a high detection accuracy (76-78 percent) and frame rate (40-122fps) for these applications.

The FPGA simulation of nonlinear SVMs is addressed by [27] using a Graphical simulator and system generator. The training phase is performed offline in a computer by Matlab, and the extracted parameters are used in implementing the decision function in the hardware. The system used CORDIC blocks to implement the exponential functions in the Gaussian kernels. Parallel adders and multipliers are used to find the ($\alpha i \times yi$) values in the decision function. A dataset of handwritten digits of three different classes (1, 4, and 8) are used for training and testing the proposed SVM architecture. A total of 800 samples are used from each class, where 600 samples have been used for training and 200 samples for testing. Each sample is a feature vector of 24 dimensions. They used the Xilinx Virtex4-xc4vsx35 device for implementing the design. The simulation results showed that the classification error rate was around 1.33%. The time to complete the testing phase was equal to 0.27 ms at a 151.286 MHz operational frequency.

The authors in [28] implemented an FPGA coprocessor architecture for SVM classifiers. This architecture is based on clusters of vector processing elements (VPEs) operating in a SIMD mode, with each cluster serviced by a separate off-chip bank of DDR2 memory. They used clusters of processing elements operating in single-instruction multiple data mode to take advantage of large amounts of data parallelism in the SVM algorithm. To increase the parallelism they reduced the precision in SVM kernel arithmetic. The system was built using an off-the-shelf PCI-based FPGA card with a Xilinx Virtex 5 FPGA and 1GB DDR2 memory. The results show that the speed up in the training stage reached around 9 billion multiply-accumulates per second and at a classification of 14 GMACs.

The authors in [10, 29] implemented a hand gesture recognition system by extracting the SIFT features from the input images and using multiclass SVM classifiers to recognize the hand gestures. They used 100 training images with 320×240 pixels resolution to represent four hand gestures in different conditions. By increasing the training images for the same class they increased the robustness of the clustering and SVM classifier model. The results achieved using a Pentium 4 CPU 3.2 GHz PC took about 6.994 seconds to recognize one hand gesture image with high classification accuracy above 90%. However, 6 seconds is considered far away from the real-time performance, in addition to that, the input image resolution is also low. Table 3 summarizes the SVM systems parameters and its performance results.

Table 3: SVM systems Parameters and performance results

	[25]	[27]	[26]	[28]	[29]
Operating frequency	250MHz	202.840 MHz	100 MHz	141 MHz	6.994sec
FPGA	Stratix III	Virtex4	Virtex-5	Virtex 5	PC
# of LUTs	~50%	9,141	57,296	37,549	-
# of registers	~50%	11,589	23,220	37,067	-
# of DSP blocks	~50%	81	83	128	-
Classification accuracy	98.96%	98.67%	78%	-	96.23%

Chapter 4: Software Implementation

In this chapter, the software implementation of the object detection's algorithms is discussed. Section 4.1 presents the software implementation of the SIFT algorithm. In Section 4.2, performance comparison between six of the most popular classification algorithms in terms of classification accuracy and processing time is presented.

4.1 Feature Extraction Algorithm

To extract the SIFT features from a set of images, we used a VLFeat library which is a MATLAB/C implementation of the SIFT detector and descriptor written by [30]. The tool reads the images and extracts the SIFT keypoints locations and SIFT descriptors.

4.2 Classification Algorithms

Selecting the best classification algorithm for our problem was an important step. The goal of this experiment was to find out which classifier achieved higher accuracy within an accepted processing time. This section presents and compares six classification algorithms. The algorithms tested are: K-nearest-neighbour (KNN), Naïve Bayes, Decision Tree, Adaboost, linear support vector machine (SVM), and Non-linear SVM.

For this experiment, we used Caltech-256 [31] benchmark dataset. It is an image dataset created at the California Institute of technology in 2007, a successor to Caltech-101. It is a set of 256 categories containing a total of 30607 images. In this work, we used five classes: airplane, human face, motorbike, hours, and watch. Figure 10 shows example images from each category.

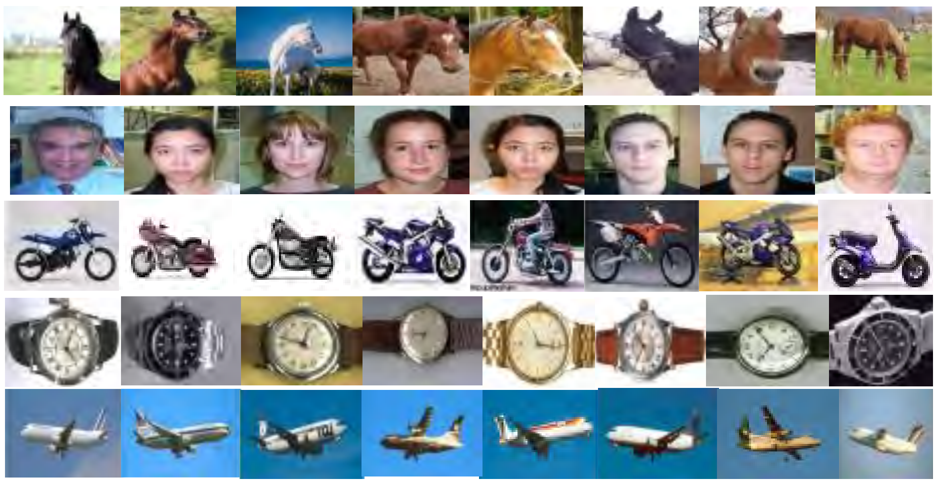


Figure 10: Five Classes from Caltech-256 Dataset

The experiment consists of three stages: data preparation, classifiers training, classifiers testing and performance evolution. In the first step, we read the training images and represent them as BoF histograms. To do this, the SIFT features are extracted from the images using SIFT Matlab code. Then, the k-means clustering algorithm is used to find the BoF codewords (cluster centre). In this experiment, we used 500 codewords. For each image, the SIFT features are quantized to the nearest cluster centre, and based on a number of features in each centre the BoF histogram is built.

In the training step, the images' histograms are used to train the classifiers. We used different sizes for the training set; we used a dataset of size 50, 100, 200, 300, 400, and 500 to see the effect of changing the training size on the classification accuracy. The overall processes are shown in Figure 11.

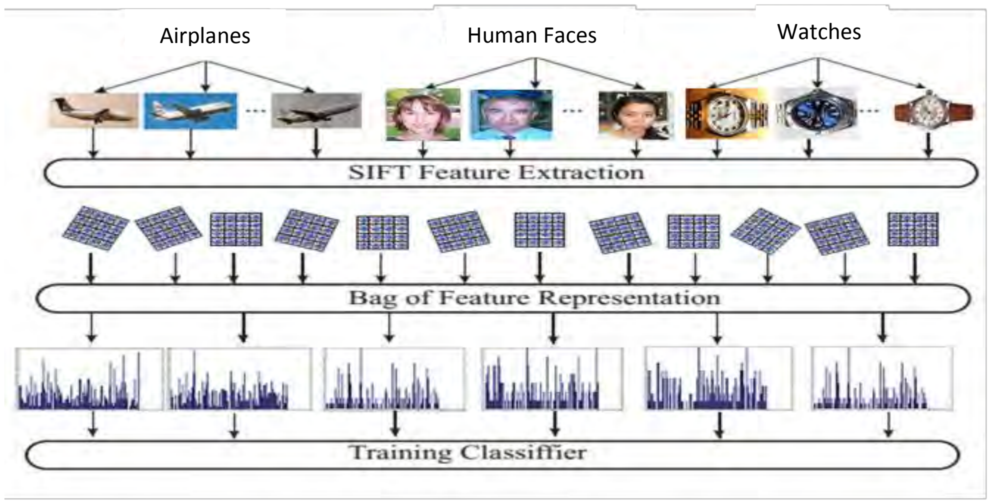


Figure 11: The Process of Training Stage

In this testing step, we used 100 images from each class summing up to 500 images in total. We represent these images as BoF histograms and apply them to each classifier. Then, the classification accuracy of each classifier is reported. The testing process is shown in Figure 12.

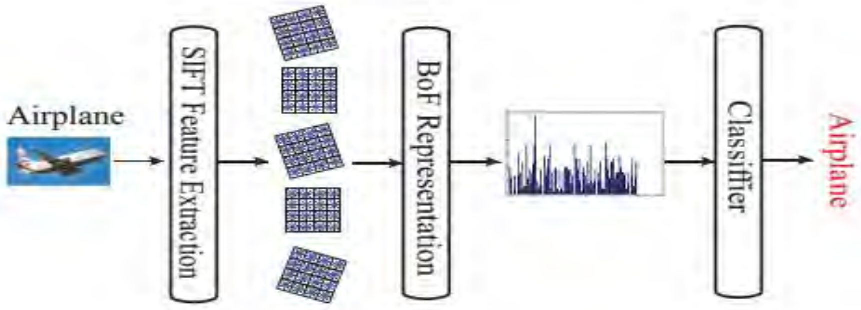


Figure 12: Testing phase's Steps

For evaluation purposes, we used precision, recall, True Negative Rate (TNR), and accuracy as performance metrics. The precision is defined as the fraction of retrieved instances that are relevant. The recall is the fraction of relevant instances that are retrieved. The TNR measures the proportion of negatives that are correctly identified. The accuracy is the proportion of true results. Equations (22-25) summarize the precision, recall, TNR, and accuracy, respectively.

$$\text{Precision}_{\text{class A}} = \text{TP}_{\text{class A}} / (\text{TP}_{\text{class A}} + \text{FP}_{\text{class A}}) \tag{22}$$

$$\text{Recall}_{\text{class A}} = \text{TP}_{\text{class A}} / (\text{TP}_{\text{class A}} + \text{FN}_{\text{class A}}) \tag{23}$$

$$\text{TNR}_{\text{class A}} = \text{TN}_{\text{class A}} / (\text{FP}_{\text{class A}} + \text{TN}_{\text{class A}}) \tag{24}$$

$$\text{Accuracy}_{\text{class A}} = (\text{TP}_{\text{class A}} + \text{TN}_{\text{class A}}) / (\text{P} + \text{N}) \tag{25}$$

To find out which classification algorithm has the best accuracy, we build the learning curve for each classifier. The learning curve shows how the classifier performance is affected by increasing the size of the training set. By training the classifiers on a dataset of size (50, 100, 200, 300, 400, and 500) and measuring its accuracy we obtained the results as seen in Figure 13. We can see that the RBF-SVM classifier outperforms other classifiers on all training sizes. In terms of accuracy, the RFB SVM is followed by the Adaboost algorithms, which is then followed by the linear SVM. The naive Bayes and decision tree classifiers have poor classification rates.

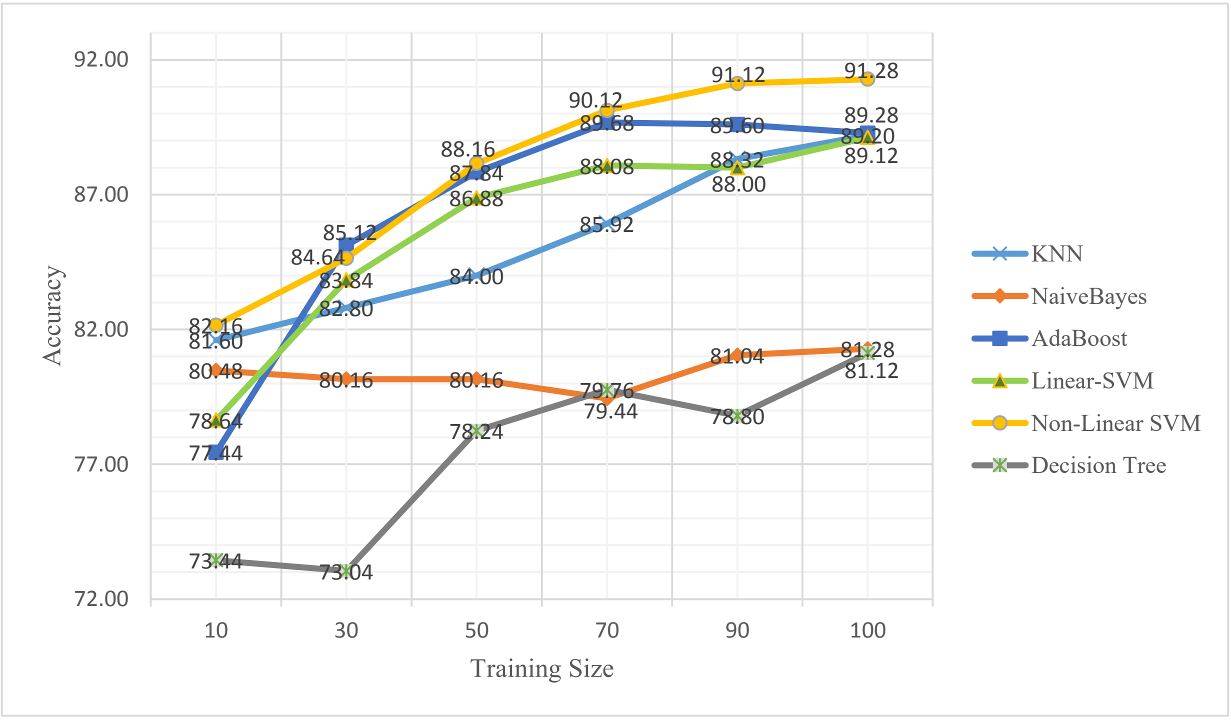


Figure 13: Comparison between six classification algorithms using the Caltech-256 dataset

A comparison based on the average precision, average recall and F_1-measure is shown in Figure 14. The RBF SVM achieved the highest Precision, Recall, and F_1-measurement over the other classifiers. High recall means that the algorithm returned most of the relevant results, while high precision means that the algorithm returned substantially more relevant than irrelevant results. The F_1-measure is an accuracy measure based on a weighted average of the precision and recall. Where an F_1 score reaches its best value at 1 and worst score at 0.

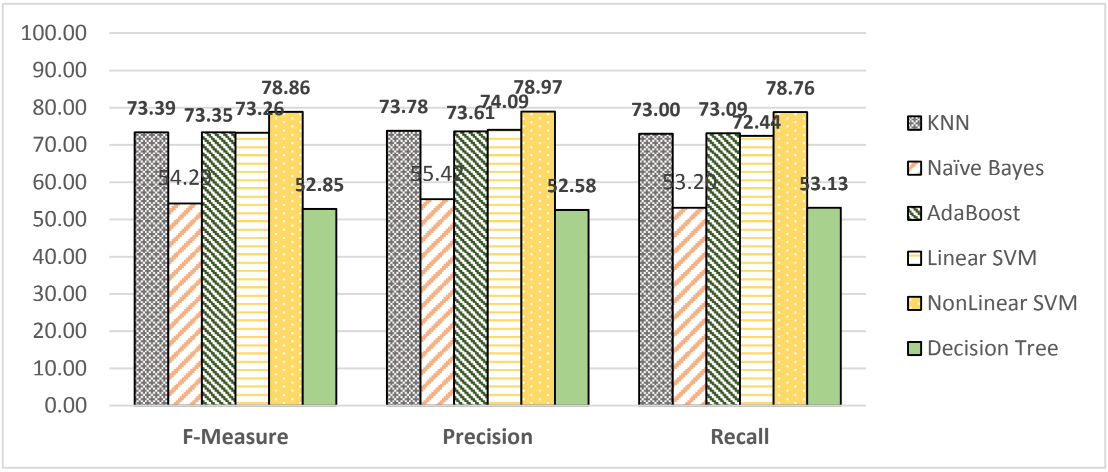


Figure 14: Comparison based on precision, recall and F_1-measure

For a comparison between the algorithms based on the processing time, we executed the algorithms on a PC with Intel Core i5 with 8 GB RAM. We used the trained classifiers to classify datasets of sizes equal to 10, 100, 200, 300, 400, and 500. Figure 15 shows that the KNN classifier has the highest processing time followed by the RBF-SVM classifier.

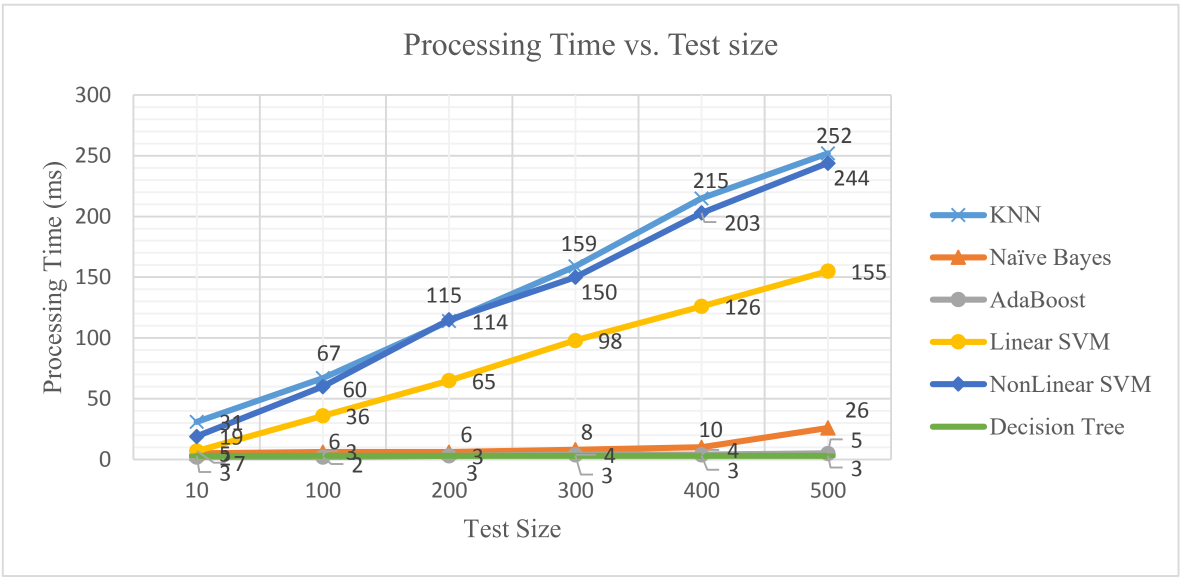


Figure 15: Classification processing time of six different classifiers using the Caltech-256 dataset.

To see how the processing time is effected by increasing the training set size, we measured the processing time on datasets of sizes of 50,100,200,300,400 and 500. The results are shown in Figure 16. Again, the KNN classifier has the highest processing time followed by the RBF SVM classifier.

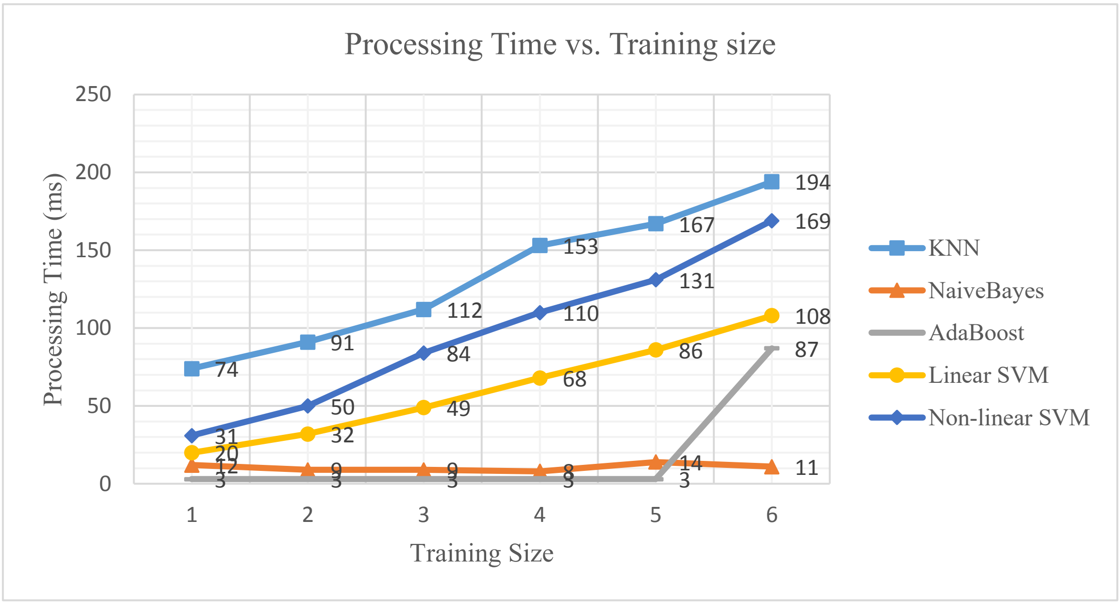


Figure 16: Training processing time of six different classifiers using the Caltech-256 dataset.

In conclusion, the RBF SVM classification algorithm has the highest classification accuracy. However its processing time is also high. In this work, we used the RBF SVM classifier whilst accelerating its operations in FPGA to reach real-time performance as introduced in the next chapter.

Chapter 5: Proposed Hardware Implementation

This chapter presents the proposed hardware architecture for our image recognition system. The chapter is divided into three sections. Section 5.1 describes the optimized hardware architecture for SIFT algorithm. In Section 5.2, we present the hardware architecture for BoF module. In Section 5.3, we presents the parallel hardware implementation of the SVM algorithm.

5.1 SIFT Module Implementation

The overall architecture of the proposed SIFT hardware is shown in Figure 17. It consists of four main modules, namely, Gaussian scale space generation (GSS), keypoint detection (KPD), gradient magnitude and orientation generation (GMO), and keypoint description module (KDS). The first two modules represent the SIFT feature detector, while the last two modules represent the SIFT feature descriptor.

The primary task of GSS module is to build the difference of the Gaussian scale space module. The input to this module is a stream of pixels from the source image with one pixel occurring every clock cycle. The output of GSS module is sent to the KPD module and GMO module at the same time. The KPD module computes the stable keypoints and stores them in a FIFO buffer, while the GMO computes the gradient magnitude and orientation. The KDS module reads one keypoint from the FIFO at a time and computes the SIFT descriptor based on the gradient values in the region around the keypoint.

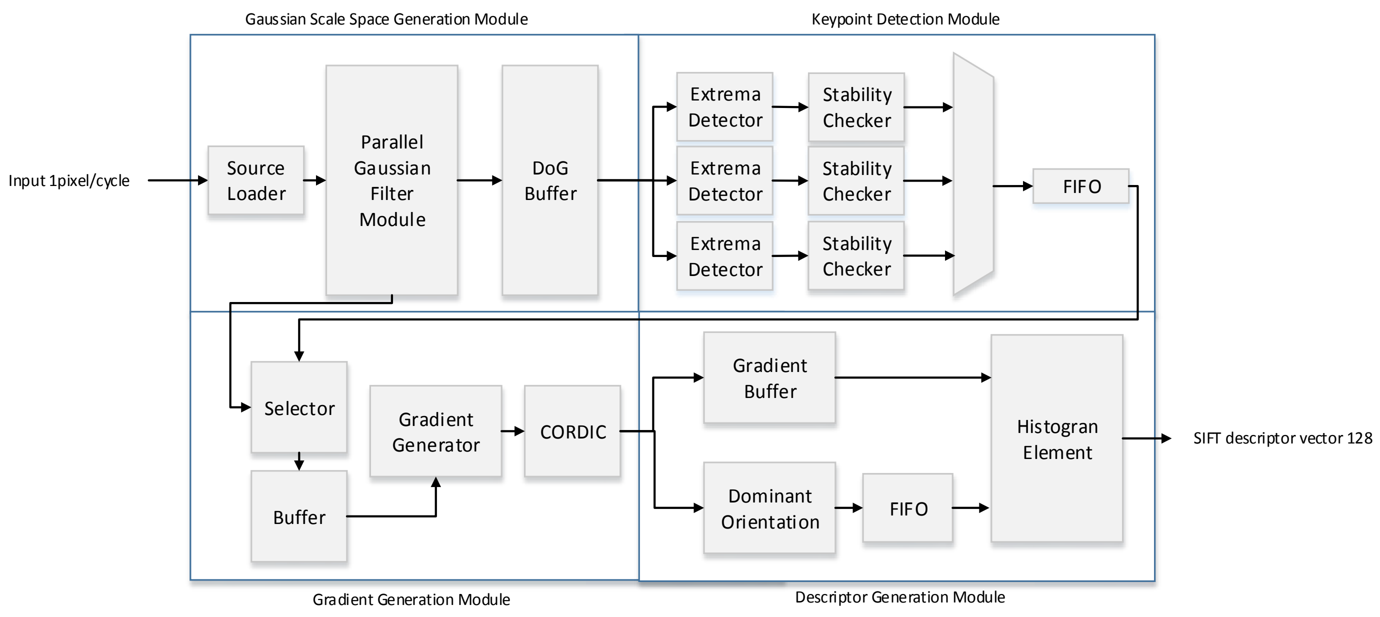


Figure 17: The Overall Architecture of the Proposed SIFT hardware

5.1.1 Implementation of GSS module. The GSS module computes the Gaussian filtered images and the difference of Gaussian pyramid (DoG) from the source image. First, it convolves the input image with a series of Gaussian filters to build the first octave. Then, it reduces the image size in half and repeats the same operations in order to compute the second octave's images. The final step is to compute the DoG images by subtracting each two adjacent Gaussian filtered images in the same octave.

There are two approaches in literature to generating the Gaussian pyramid; each one has its advantages and disadvantages. First, using a cascade filtering approach [8] where the Gaussian filtered images are computed from the input image by recursive convolution. This method reduces the number of multipliers required in each Gaussian filters, which in turn reduces the hardware resources. However, this method needs a large amount of memory to save the intermediate results. The second approach [32] generates the Gaussian filtered images using Gaussian filters with large kernel sizes. There is no need for memory to save the intermediate results but the number of multipliers in this method is large.

In our implementation, we modified the second approach to take advantage of a low memory requirement. We reduced the number of multipliers by using multiplierless multiple constant multiplication (MCM) with common subexpression elimination algorithm. Moreover, we used the separability property of the Gaussian filter to reduce hardware, by separating each Gaussian filter into two smaller 1D vertical and horizontal filters.

Table 4 summarizes the Gaussian filter scales for the first octave and the filter mask size. The mask size is often taken as five times the standard deviation of each Gaussian filter. In this work, we used three octaves with six scales in each octave.

Table 4: First octave filters scales and mask sizes

Filter #	Standard Deviation	Standard Deviation	Mask size
1	σ	1.6	9×9
2	$k^1\sigma$	$\sqrt[3]{2}^1 1.6$	11×11
3	$k^2\sigma$	$\sqrt[3]{2}^2 1.6$	13×13
4	$k^3\sigma$	$\sqrt[3]{2}^3 1.6$	15×15
5	$k^4\sigma$	$\sqrt[3]{2}^4 1.6$	21×21
6	$k^5\sigma$	$\sqrt[3]{2}^5 1.6$	25×25

The high level architecture for the 1st octave in GSS module is shown in Figure 18. The architecture consists of buffer lines to store the input pixels and two 1-D Gaussian filter blocks. The buffer lines consists of 25 FIFO buffer lines (the largest mask size). Each buffer line consists of 640 (image width) elements with each one being 8 bits; this is because the input image is represented in greyscale (0-255).

The octave architecture has two states: initialization and a computing state. In the initialization state, the input pixels is shifted into the buffer line one value to the right every clock cycle, as shown in Figure 18. After 640 clock cycles, the first buffer line becomes full. In the next clock cycle the first line output become the second line input. The shifting operation is done to reach the first window in the left most of the source image of size 25×25. After 24× (640) +1 clock cycles the initialization state ends and the computing state start.

In the computing state, the first block (vertical 1D Gaussian filter) reads the left most at 25 pixels from the buffer lines and it computes the first pixel of each Gaussian filter images. In this case, it will generate six pixels from six Gaussian filtered images every clock cycle. The computed result is represented using fixed point format with 9 bits for the integer part, and 11 bits for the fraction part.

The results from the first block are buffered in 6 buffer lines with 25 elements each. After saving 25 valid values in the buffers, the second block (horizontal 1-D Gaussian filter) reads each 25 value and computes the corresponding 1-D horizontal filtered pixels. The result is represented using 20 bit fixed point format with 9 bits for the integer part and 11 bits for the fraction part. In the next clock cycle, the new pixel shifted into the buffer lines and the second window was computed. This operation was repeated until the whole source image shifted into the GSS module.

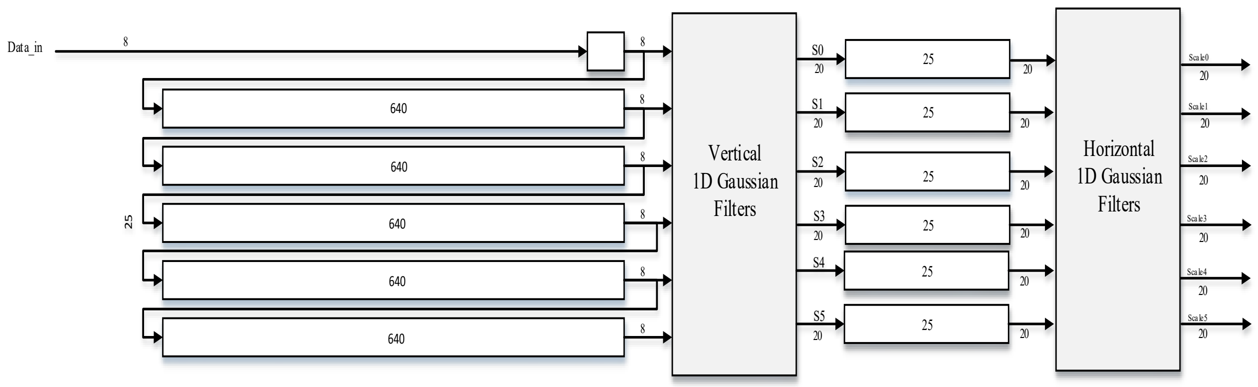


Figure 18: GSS's First Octave Module

The architecture used to implement the vertical and horizontal 1-D Gaussian filter block is shown in Figure 19. In this architecture, we used the symmetrical property of Gaussian filters to reduce the hardware utilization by adding the pixels before multiplying them with the corresponding filter constant. The architecture contains 13 blocks to compute the multiplication of each pixel with 6 different constants from 6 different Gaussian filters. Figure 20 shows the constants of each Gaussian filters.

Block 0 computes the multiplication of pixel 12 from 25 pixels with the six constants from six filters {0.249, 0.198, 0.157, 0.125, 0.099, and 0.08}. The block 1 computes the multiplication of (pixel 11+ pixel13) from 25 pixels with the six constants from six filters {0.205, 0.175, 0.145, 0.119, 0.096, and 0.078} and so on. Each block generates six values y1, y2…, y6. The summation of y1's generated from blocks 0-12 equal to the final 1-D Gaussian filter pixel.

The filter's masks are extended to 25 values by appending zeros. The mask's constants are generated using Matlab. Next, a multiplierless multiple constant multiplication with a common subexpression elimination algorithm is used to optimize the architecture of each block.

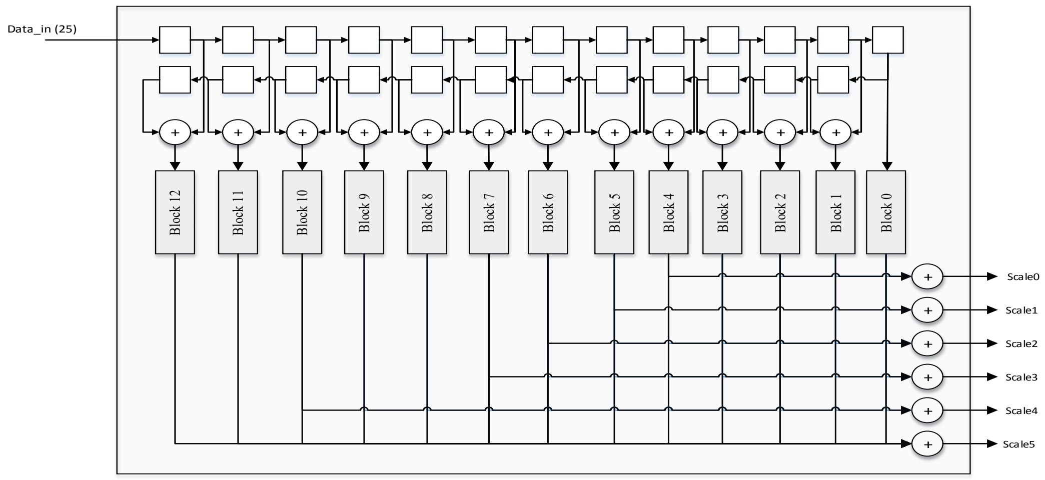
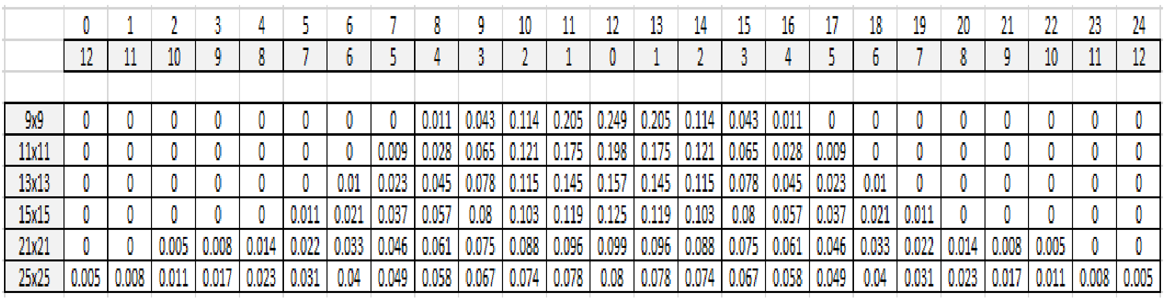


Figure 19: 1-D Gaussian Filter Block



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	12	11	10	9	8	7	6	5	4	3	2	1	0	1	2	3	4	5	6	7	8	9	10	11	12
9x9	0	0	0	0	0	0	0	0	0.011	0.043	0.114	0.205	0.249	0.205	0.114	0.043	0.011	0	0	0	0	0	0	0	0
11x11	0	0	0	0	0	0	0	0.009	0.028	0.065	0.121	0.175	0.198	0.175	0.121	0.065	0.028	0.009	0	0	0	0	0	0	0
13x13	0	0	0	0	0	0	0.01	0.023	0.045	0.078	0.115	0.145	0.157	0.145	0.115	0.078	0.045	0.023	0.01	0	0	0	0	0	0
15x15	0	0	0	0	0	0.011	0.021	0.037	0.057	0.08	0.103	0.119	0.125	0.119	0.103	0.08	0.057	0.037	0.021	0.011	0	0	0	0	0
21x21	0	0	0.005	0.008	0.014	0.022	0.033	0.046	0.061	0.075	0.088	0.096	0.099	0.096	0.088	0.075	0.061	0.046	0.033	0.022	0.014	0.008	0.005	0	0
25x25	0.005	0.008	0.011	0.017	0.023	0.031	0.04	0.049	0.058	0.067	0.074	0.078	0.08	0.078	0.074	0.067	0.058	0.049	0.04	0.031	0.023	0.017	0.011	0.008	0.005

Figure 20: 1D Gaussian Filters Values

The idea of using the multiplierless multiple constant multiplication algorithm [33] with common subexpression elimination is to represent the multiplication operation as a number of shifts, additions, and subtractions operations. The common subexpression elimination is to try to extract common parts among the constants in order to minimize the number of additions in finding the final result.

To find the common subexpression between the coefficients, first we represent coefficients using the canonical sign digit (CSD) representation. In the CSD code of a number, each bit is set to 0, 1, or –1. Then search for the common pattern is carried within the matrix. These patterns represent the common Subexpression.

Figures 21-33 show the internal tree organization for blocks 0 to 12. Each block generates six outputs y_1, y_2…, y_6. If the figure doesn't show any y-value this means it is equal to zero. The red (dark) squares represent the common sub expressions.

In block 10, the input X is the summation of (pixel_2+ pixel_22) from 25 buffers represented as 9 bits. The block's output y_1= (pixel_2+ pixel_22) ×0.011 and y_2= (pixel_2+ pixel_22) ×0.005. The input X is stored in the least significant 9 bits of a 20 bit output meaning we divide the input by (2^{11} =2048). For y_1, we shift the input five times to the right (X×32) and three times to the right (X ×8). The addition of (X+ X ×8) equals to (X ×9). The final step is to compute (X ×32 - X ×9) which equals (X × 23). As a result, y_1= X × (23 / 2^{11}) = X× (23/2048) = X × 0.0112. The same idea applies to the other numbers.

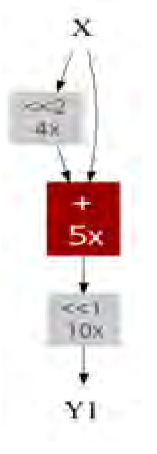


Figure 21: Block 12 in 1-D Gaussian Filter Block

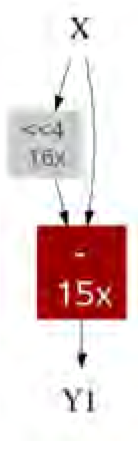


Figure 22: Block 11 in 1-D Gaussian Filter Block

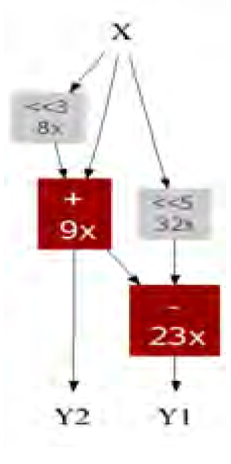


Figure 23: Block 10 in 1-D Gaussian Filter Block

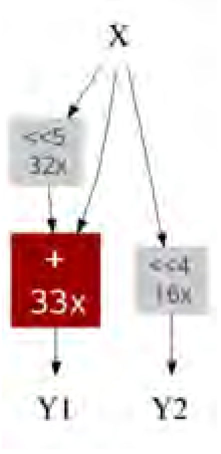


Figure 24: Block 9 in 1-D Gaussian Filter Block

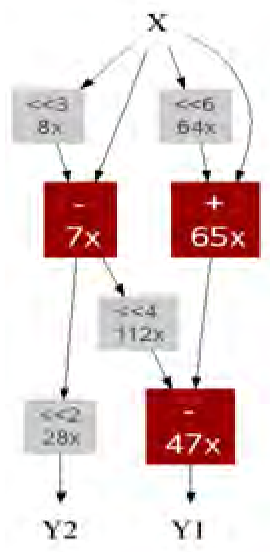


Figure 25: Block 8 in 1-D Gaussian Filter Block

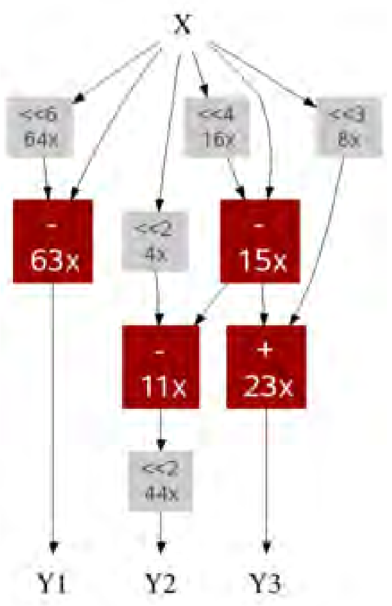


Figure 26: Block 7 in 1-D Gaussian Filter Block

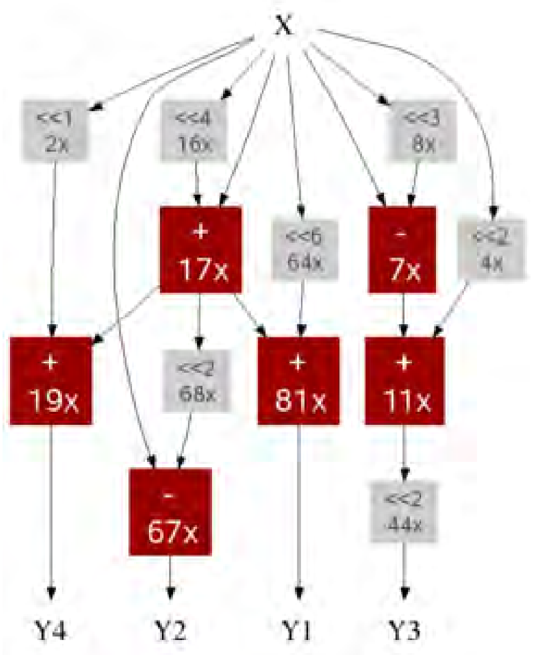


Figure 27: Block 6 in 1-D Gaussian Filter Block

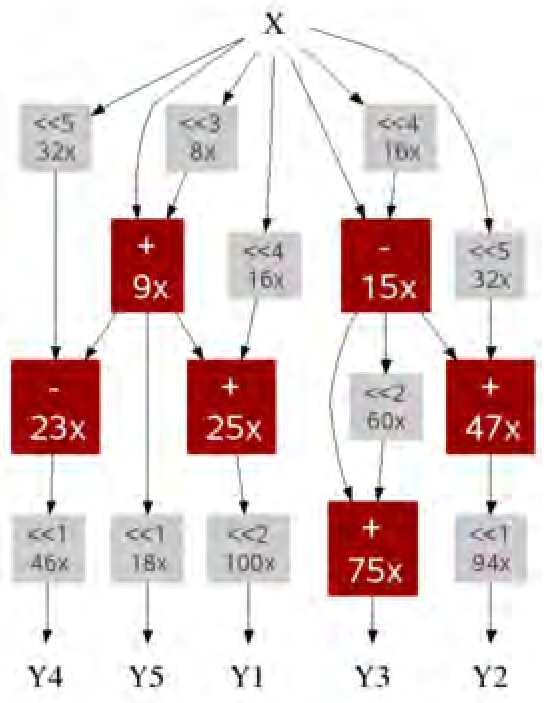


Figure 28: Block 5 in 1-D Gaussian Filter Block

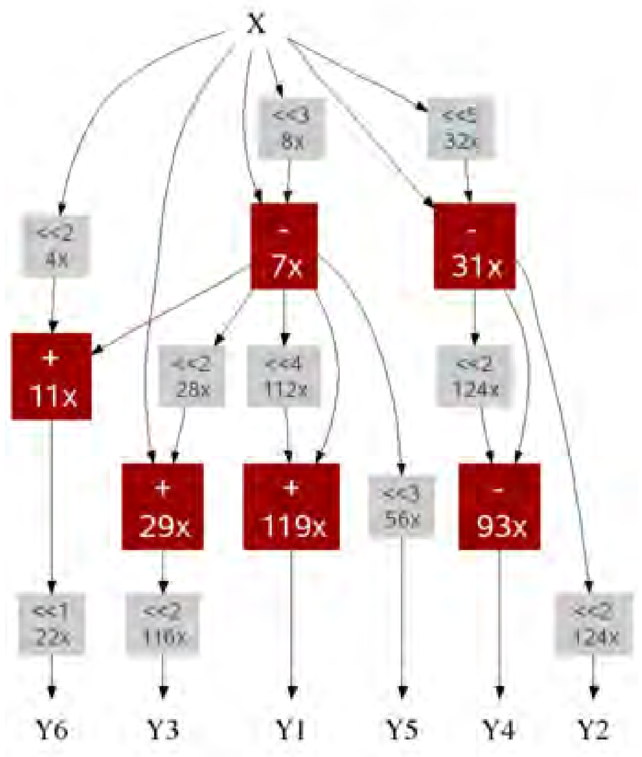


Figure 29: Block 4 in 1-D Gaussian Filter Block

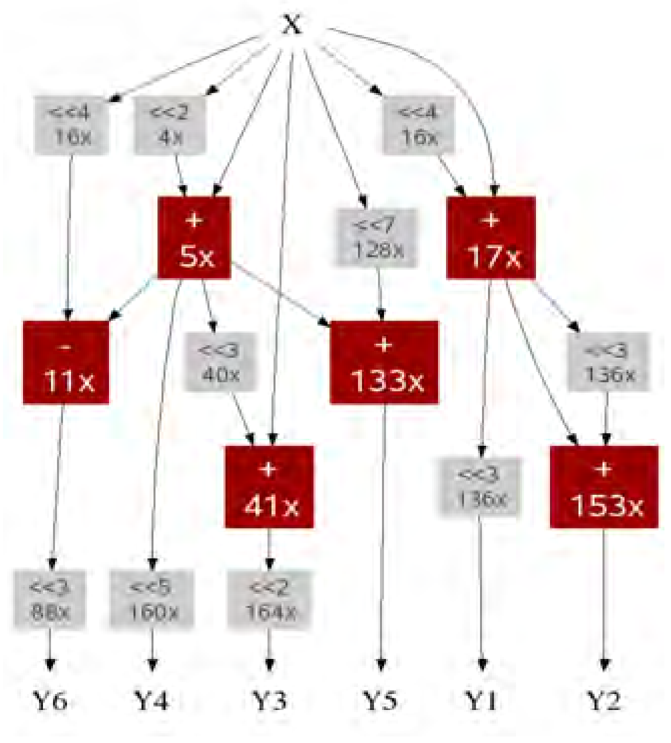


Figure 30: Block 3 in 1-D
Gaussian Filter Block

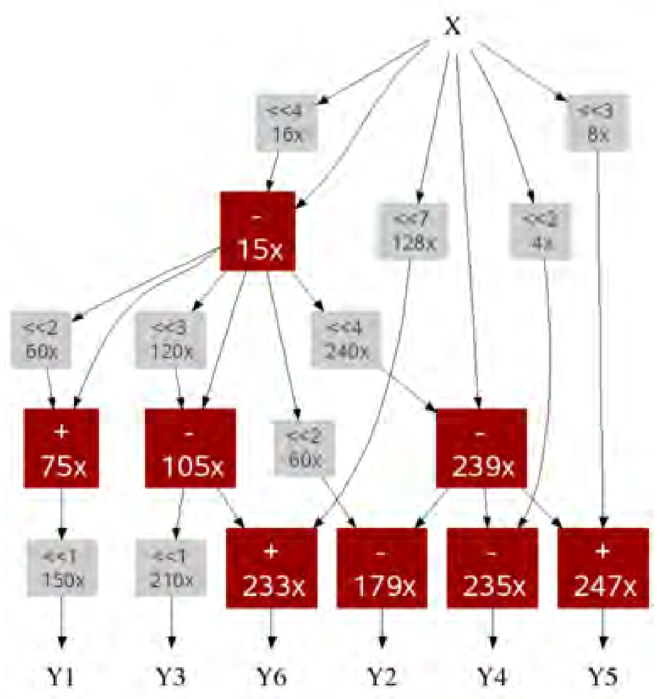


Figure 31: Block 2 in 1-D
Gaussian Filter Block

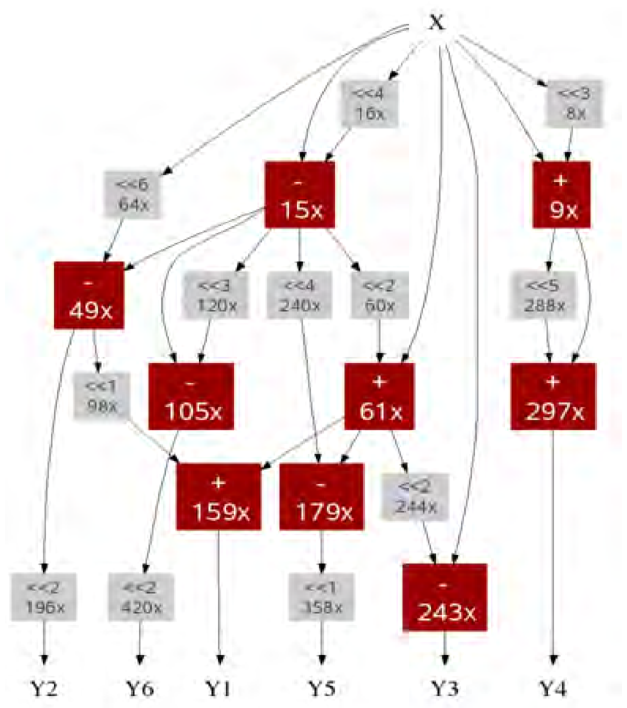


Figure 32: Block 1 in 1-D
Gaussian Filter Block

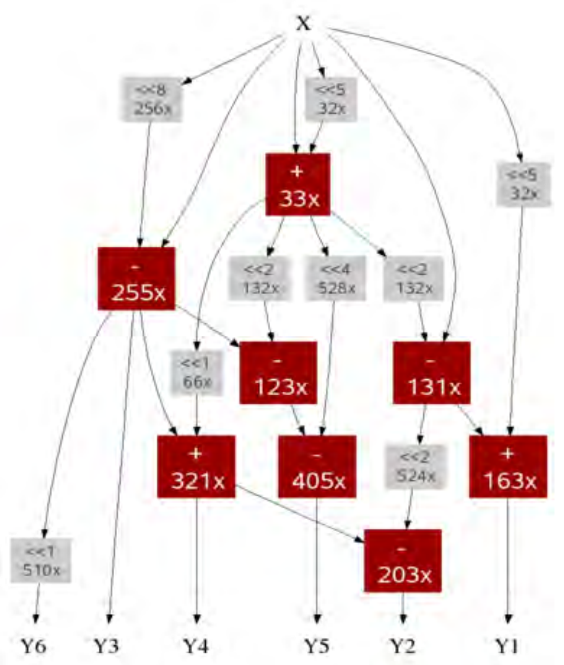


Figure 33: Block 0 in 1-D
Gaussian Filter Block

5.1.2 Implementation of the Keypoint Detection Module. The key point detection (KPD) module receives its input from the DoG module. The input is a stream of Gaussian filtered images one pixel every clock cycle. The module shifts and stores the pixels into a buffer lines and then detects keypoints by analyzing the DoG images. The module's output is the key point for the x and y positions.

In this module, we parallelized the process of checking for candidate key points by using three KP detector blocks working together. Each block analyzes three DoG images and checks if the current pixel in the second image is a candidate keypoint, as shown in Figure 34. The module has two states: initialization and a checking state. In the initialization, the module shifts the input pixels into the buffer lines. After (2×image_width+3) clock cycles, the buffer lines content becomes valid.

In the checking states, each keypoint detector block compares the pixel in the center of the second window with its eight surrounding neighbours in the same DoG image, and the nine surrounding neighbours in upper DoG image, and the nine surrounding neighbours in lower DoG image. If the current pixel is the maximum or the minimum out of the 27 values, then the pixel is a candidate keypoint.

Once the feature candidates have been located, we should eliminate the low contrast and strong edge response points. To eliminate the low contrast key points, we compare the absolute value DoG image pixel to a constant value. In our implementation, we used a constant threshold equal to three. The strong edge response points can be detected by computing the principle curvature across the edge and in the perpendicular direction using the Hessian matrix.

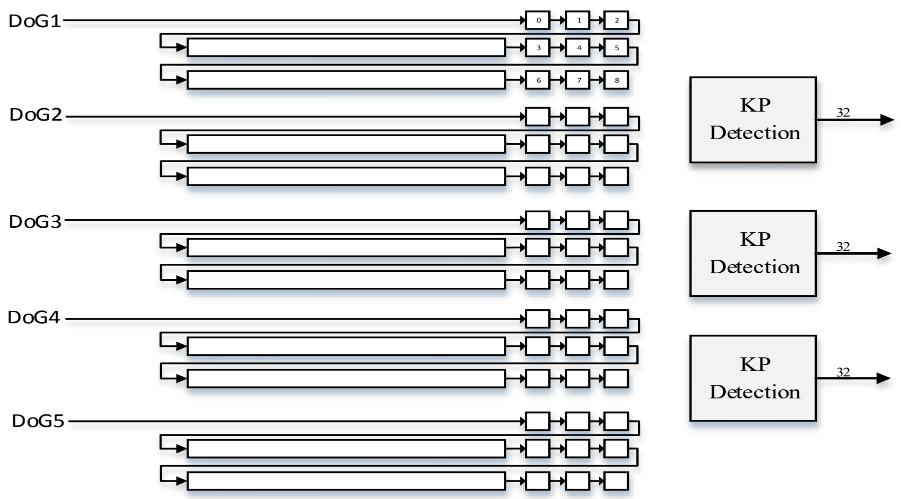


Figure 34: Keypoint Detection Module Architecture

5.1.3 Implementation of Gradient Magnitude and Orientation Module. The gradient orientation and magnitude generation module is the third module in our architecture. It is used to compute gradient values from the Gaussian filtered images. The module's input come from the Gaussian scale space module as a stream of pixels, while the output is the gradient magnitude and orientation values.

The module works in two states, initialization and computing stages. In the first stage, the module shifts the input pixels into buffering lines. The buffering lines are connected to three pixel buffer elements as shown in Figure 35. After the (2×image_width+3) cycles, the values at G(x, y+1), G(x, y-1), G(x+1, y), and G(x-1, y) become valid and contain the pixel at position (x, y+1), (x, y-1), (x+1, y), and (x-1, y), respectively, as shown in Figure 35.

The outputs of the buffer lines are connected to two subtractors in order to compute the difference Δx and Δy. Two multipliers are used to compute the $(\Delta x)^2$ and$(\Delta y)^2$, and then one is added to compute the summation$(\Delta x)^2 + (\Delta y)^2$. The square root and tan inverse functions are computed using Xilinx IP cores CORDIC blocks. The input is the atan2 module (Δx) and (Δy), while the output is the phase angle, as given by equation (26). The gradient magnitude is computed using the square root of $(\Delta x)^2 + (\Delta y)^2$ as given by equation (27).

$$m(x,y) = \sqrt[2]{\left(G(x+1,y) - G(x-1,y)\right)^2 + (G(x,y+1) - G(x,y-1))^2} \tag{26}$$

$$\theta(x,y) = \tan^{-1}\left(\frac{G(x,y+1) - G(x,y-1)}{G(x+1,y) - G(x-1,y)}\right) \tag{27}$$

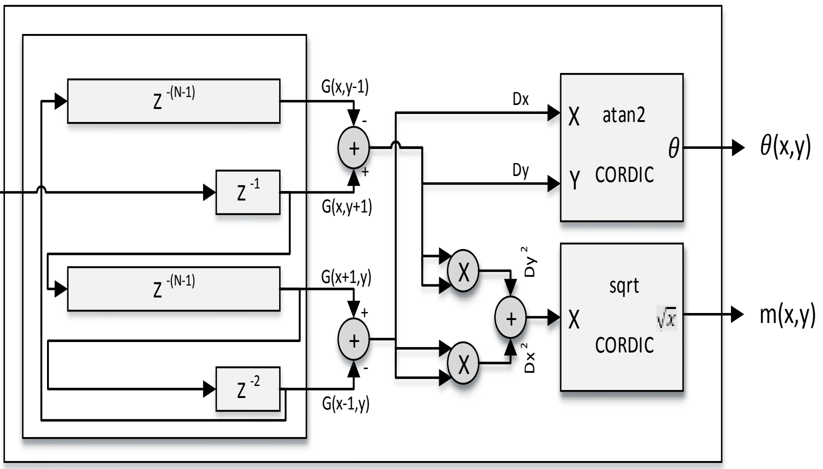


Figure 35: Gradient Magnitude and Orientation Module Architecture

In this implementation, the square root, and the atan2 functions are computed using two CORDIC Xilinx IP cores. First, the atan2 IP core computes the tan inverse by rotating the input vector (X, Y) using the CORDIC algorithm until the Y component becomes zero. The output is the phase angle that is equal to atan2(Y/X). Second, the square root of the IP core takes the $(\Delta x)^2 + (\Delta y)^2$ as an input and generates the correct square root value using the CORDIC algorithm.

To make these IP cores work correctly, we modify the input and output data format to match with the IP core inputs and outputs. The Input format for the atan2 IP core is a fixed-point number with a 2 bits integer and 14 bits fraction (2.14) for X_atan2 and Y_atan2. The output is an angle with the format 3.13 and range from (-3.14 to 3.14). The input of the sqrt IP core is an unsigned integer with 20 bits width with a 9.11 fixed-point number, and the output is an unsigned integer with a 20 bit width. Table 5 summarizes the input and output data format. The gradient orientation (θ) represented in fixed-point numbers with 3 bits for integer part and 13 bits for fraction part. The gradient magnitude (m) represented in fixed-point numbers with 9 bits for the integer part and 11 bits for the fraction part.

In the atan2 IP, we used the optional coarse rotation module to extend the range of CORDIC from the first quadrant (+Pi/4 to - Pi/4 Radians) to the full circle. We also used the parallel architectural configuration with single-cycle data throughput instead of the word serial architectural configuration for small area configuration.

In this implementation, the output becomes valid after (2×image_width+3) clock cycles from presenting the input to the module. We used a valid output signal to indicate when the output becomes valid. The gradient magnitude and orientation generated by this module goes to the dominant orientation generation module.

Table 5 Gradient Generation block (Data Format)

Port	Data Type	Bit length	Format
θ	Output	16	3.13
m	Output	20	9.11
G	Input	20	9.11
X_atan2	Wire	16	2.14
Y_atan2	Wire	16	2.14
X_sqrt	Wire	20	20.0

5.1.4 Dominant Orientation Generation Module. The dominant orientation generation module computes the principle orientation for each stable keypoint. The module's inputs are the key point's position generated by the KP detection module, and the gradient magnitude and orientation come from the GMO module. The dominant orientation module reads one keypoint at a time and computes the principle orientation from (17×17) pixels around the keypoint. The module also extracts the gradient magnitude and orientation values in the region around the key point and sends them to the keypoint descriptor module. Figure 36 shows the module's inputs, outputs, and its internal structure.

The enable signal in the domination orientation module comes from the Gradient magnitude and orientation generation module. The module shifts the gradient magnitude and orientation values into a FIFO buffer lines as shown in Figure 36. When a keypoint is detected, the KP detection module pushes and stores it to a FIFO buffer. The dominant orientation module reads the keypoint and starts the search phase. This phase includes waiting until reaching the valid window around the keypoint. This process is carried out by comparing the current window index with the last key point index. When both values become the same this means that the current window is the valid window.

When the DOM module finds the valid window, it sends a disabling signal to the DoG module to stop reading from the source image. The module then starts the loop back process by switching the line of the buffer's input from the new value source to the first value in line buffer. Using this technique, the state of the module will return to the original state after 640 clock cycles.

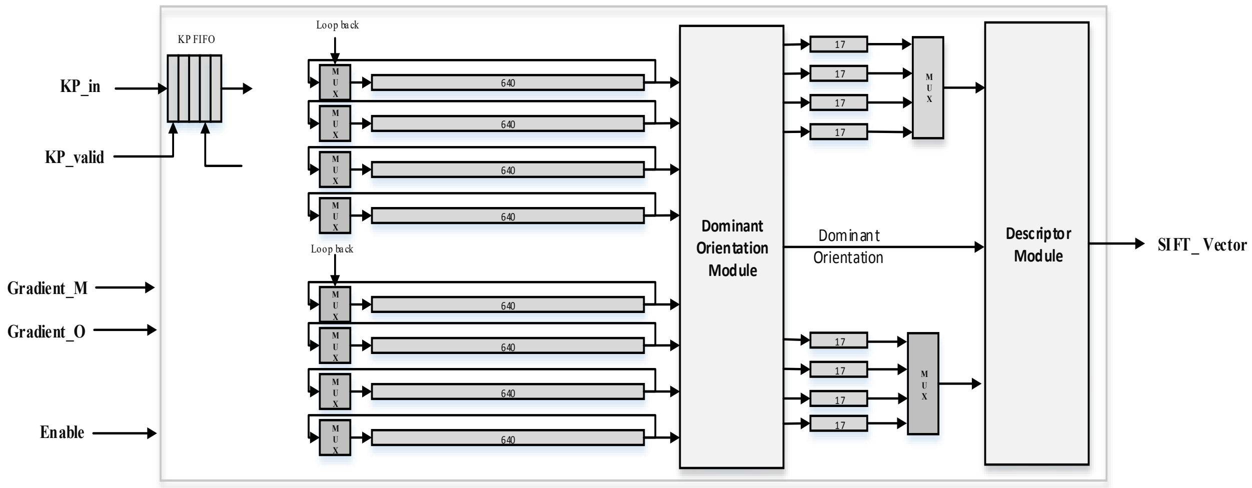


Figure 36: Dominant Orientation Generation Module

The dominant orientation generation module reads 17 new gradient magnitude values, and 17 new gradient orientation values occur every clock cycle. The data's index in our implementation is (-8, 8), which is the relative location to the keypoint position. Each gradient orientation value is connected to a 10 degree decoder module as shown in Figure 37. The task of the decoder is to find the correct bin out of the 36 bins. The input angle is compared with a lookup table to find the right bin. Table 27 in appendix B summarizes the angle range for each bin in the 10 degree decoder module. Each gradient magnitude is multiplied with the corresponding Gaussian weight using 17 multipliers. The output is connected to a switching circuit that updates the correct histogram's values.

The switching circuit builds a gradient histogram by using the gradient orientation of each pixel as an address to the histogram memory, and the gradient magnitude as new data to be added to the memory location. At the rising edge of the clock, the memory block will update its value by adding the current value to the newly applied input.

After 17 clock cycles, the histogram will include data from the whole window. The max block finds the bin with the largest value in the histogram and writes it to the output. When the DOM module finishes the current keypoint, it will read the next keypoint and repeat the whole process again. If there is no new keypoint, it will keep shifting the data into the block until a new keypoint is detected.

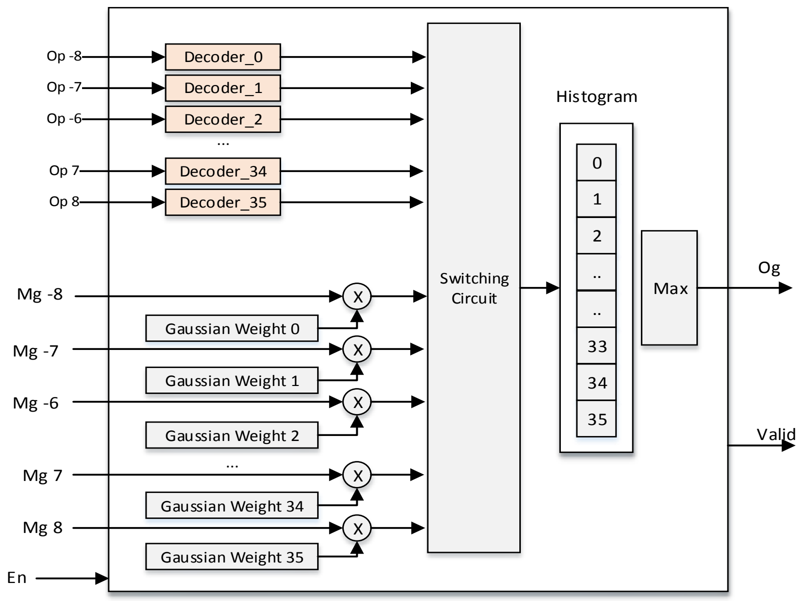


Figure 37: Architecture of Dominant Orientation Generation Module

5.1.5 Descriptor Generation Module. The process of generating the SIFT descriptor involves four tasks: coordinate rotation, Gaussian weight generation, trilinear interpolation, and normalization. The first three tasks are repeated N times, where N is the number of pixels in the window around the keypoint. After N iterations, the normalization step is performed to obtain the 128 elements SIFT descriptor vector.

The block diagram of the SIFT descriptor module is shown in Figure 38. It consists of four main blocks: rotation module, Gaussian weight generation module, trilinear interpolation module and normalization blocks. The module's inputs are the pixel coordinates (X, Y), the gradient orientation (Op), the gradient magnitude (Mg), and the keypoint dominant orientation (Og). The module's output is the SIFT vector with 128 elements.

The rotation module generates the rotated coordinate (Xr, Yr) and rotated orientation (Or) from the pixel coordinates (X, Y) and the gradient orientation (Op). The Gaussian weight generation module takes the Xr, Yr as an input and generates the appropriate Gaussian weight (Wg) using a lookup table technique. The trilinear interpolation module distributes the result of multiplying the gradient magnitude with the Gaussian weight (mg x wg) into 8 bins in its histogram bins based on the rotated coordinate (Xr, Yr) and the rotated orientation (Or). The normalization is used to normalize the output vector as defined in the SIFT algorithm.

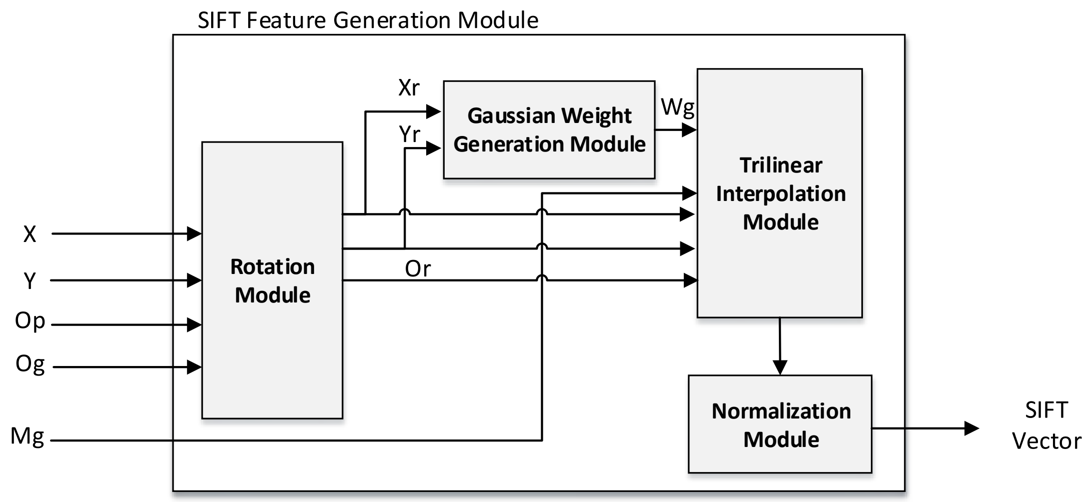

SIFT Feature Generation Module
Xr
Yr
Gaussian Weight Generation Module
Wg
Trilinear Interpolation Module
X
Y
Op
Og
Mg
Rotation Module
Or
Normalization Module
SIFT Vector

Figure 38: SIFT Descriptor Module Architecture

5.1.5.1 Rotation module. The rotation module rotates the gradients within the region around the keypoint relative to the principle orientation. The module takes the pixel's gradient orientation (Op) and dominant orientation (Og) as an inputs, and generates the rotated coordinates (Xr, Yr) and the rotated pixel gradient orientation (Or) as shown in Figure 39.

The rotation module consists of four main blocks: coordinate rotation, Sine, Cosine, and orientation rotation blocks. The rotated coordinates Xr and Yr are computed based on equations (28-29), where SBP is a constant equaling to 3 times the keypoint scale (SPB= 9.6), and NBO is number of orientation bins (NBO=8). The division operations in equation (1-2) are converted to multiplication to reduce the hardware utilization.

$$\mathrm{Xr} = (\cos(\mathrm{Og}) \times \mathrm{X} + \sin(\mathrm{Og}) \times \mathrm{Y})/\mathrm{SBP} \qquad (28)$$
$$\mathrm{Yr} = (-\sin(\mathrm{Og}) \times \mathrm{X} + \cos(\mathrm{Og}) \times \mathrm{Y})/\mathrm{SBP} \qquad (29)$$
$$\mathrm{Or} = \mathrm{NBO} \times (\mathrm{Og} - \mathrm{Op})/(2\pi) \qquad (30)$$

The Sine and Cosine components in equations (28-29) are computed using the Xilinx LogiCORE™ IP CORDIC core. The input angle is expressed as a fixed-point 2's complement number with format (3.13) and it has a range from –π to π. The outputs (Sine, Cosine) are expressed as a pair of fixed-point 2's complement numbers with format (2.14) with range from -1 to 1. Table 6 summarizes the module port's bit width and data format.

The orientation rotation block consists of one comparator, one subtractor, and one adder. First, the gradient orientation (Op) is subtracted from the (Og). If the result is less than zero, a (π=3.1416= 16'b0110010010001000) is added to the result to get the positive equivalent of the result with range (0-2π). Finally the output is multiplied by a constant (1/2π) to generate the rotated angle (Or), as given by equation (30).

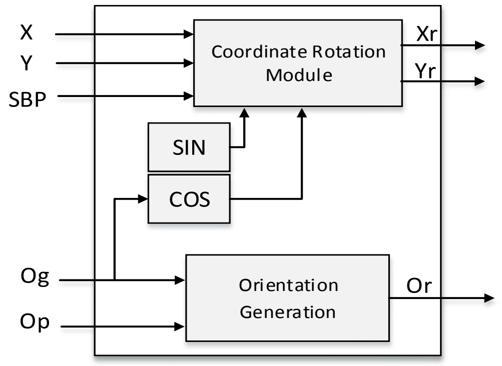


Figure 39: Rotation Module Architecture

Table 6: Rotation Module Data Format

	Data Type	Data width	Data Format
X	Input	16	5.11
Y	Input	16	5.11
Op	Input	16	3.13
Og	Input	16	3.13
Xr	Output	16	5.11
Yr	Output	16	5.11
Or	Output	16	4.12
Clk	Input	1	1

5.1.5.2 Gaussian weight generation module. The Gaussian weight generation module takes the rotated coordinates of Xr and Yr as input and generates the proper Gaussian weight value. The module consists of a 16 element lookup table (LUTs), and one multiplier. The lookup table stores the 1-D Gaussian filter coefficients. The final output (Wg) is determined by multiplying the two values obtained from the lookup table based on Xr, and Yr. The output is represented in fixed point 2's complement with 8 bits for the integer part and 11 for the fractional part.

5.1.5.3 Trilinear interpolation. The Trilinear interpolation module distributes the result of multiplying Wg×Mg into eight adjacent bins in the SIFT descriptor histogram. The module's inputs are the rotated coordinate Xr, Yr, and Or, the gradient magnitude (Mg), and the Gaussian weight (Wg). The output is eight values with its corresponding eight address that represent the trilinear interpolation result as shown in equations 31-38.

For each new input, eight elements out of the 128 elements in the SIFT histogram will be updated. The address for these elements is (x1, y1, z1), (x1, y1, z2),…, (x2, y2, z2) as shown in equations 31-38. The (x1, y1, z1) represents the left most bin, and (x2, y2, z2) represents the right most bin. The (W) value represents the (Wg×Mg).

Equations 31-38 show how the value is distributed into eight adjacent bins.

$$\mathrm{h(x1,y1,z1) = h(x1,y1,z1) + w \times (1 - \frac{x - x1}{bx})(1 - \frac{y - y1}{by})(1 - \frac{z - z1}{bz})} \tag{31}$$

$$\mathrm{h(x1,y1,z2) = h(x1,y1,z2) + w \times \left(1 - \frac{x - x1}{bx}\right)\left(1 - \frac{y - y1}{by}\right)\left(\frac{z - z1}{bz}\right)} \tag{32}$$

$$\mathrm{h(x1,y2,z1) = h(x1,y2,z1) + w \times \left(1 - \frac{x - x1}{bx}\right)\left(\frac{y - y1}{by}\right)\left(1 - \frac{z - z1}{bz}\right)} \tag{33}$$

$$\mathrm{h(x1,y2,z2) = h(x1,y2,z2) + w \times \left(1 - \frac{x - x1}{bx}\right)\left(\frac{y - y1}{by}\right)\left(\frac{z - z1}{bz}\right)} \tag{34}$$

$$\mathrm{h(x2,y1,z1) = h(x2,y1,z1) + w \times \left(\frac{x - x1}{bx}\right)\left(1 - \frac{y - y1}{by}\right)\left(1 - \frac{z - z1}{bz}\right)} \tag{35}$$

$$\mathrm{h(x2,y1,z2) = h(x2,y1,z2) + w \times \left(\frac{x - x1}{bx}\right)\left(1 - \frac{y - y1}{by}\right)\left(\frac{z - z1}{bz}\right)} \tag{36}$$

$$\mathrm{h(x2,y2,z1) = h(x2,y2,z1) + w \times \left(\frac{x - x1}{bx}\right)\left(\frac{y - y1}{by}\right)\left(1 - \frac{z - z1}{bz}\right)} \tag{37}$$

$$h(x2,y2,z2) = h(x2,y2,z2) + w \times \left(\frac{x - x1}{bx}\right)\left(\frac{y - y1}{by}\right)\left(\frac{z - z1}{bz}\right) \tag{38}$$

Table 7 summarizes the data format for the trilinear interpolation module inputs and outputs. Figure 40 shows the overall architecture for the SIFT descriptor generation module. After NxN clock cycles, the values in the buffers represent the SIFT 128 elements. A total of 128 clock cycles are required to read the SIFT vector and send it to the next module, namely the Bag of Feature (BoF) module. When the Read_en signal is active, the Data_out data bus equals to the buffer value defined by the Address_out signal.

Table 7: Triliniear Interpolation Module Data Format

Value	Data Type	Data width	Format (Integer.Fraction)
Mg	Input	20	9.11
Wg	Input	20	9.11
Xr	Input	16	5.11
Yr	Input	16	5.11
Or	Input	16	2.14
In 0-7	Output	20	9.11
Add0-7	Output	16	3.13

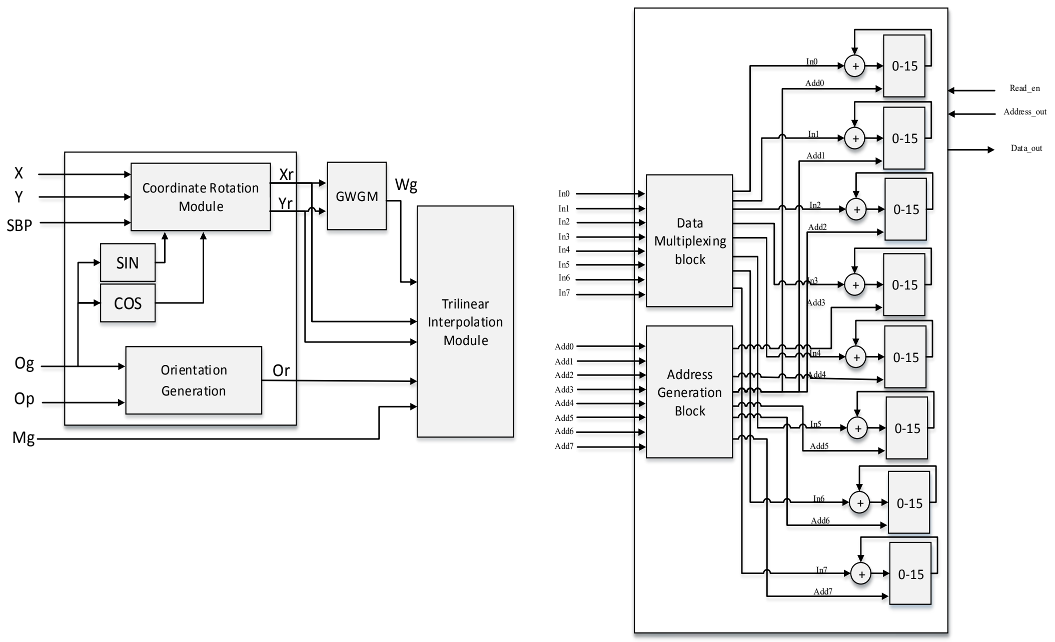


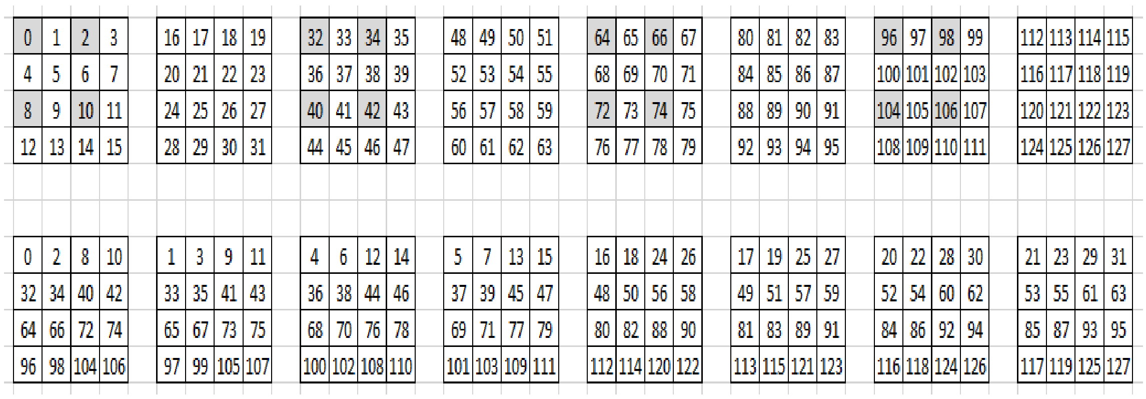
Figure 40: SIFT descriptor Module Architecture

5.1.5.4 Histogram memory implementation. Implementing multi-ported memories in FPGA is a challenging task, because the FPGA block RAMs include in the fabric typically have only two ports. In the Trilinear interpolation module, we had to update the eight data element every clock cycle. Therefore, the memory in the SIFT descriptor module should have an eight data input with eight address lines. In our architecture, we want to implement a histogram memory so that it should provide multi-ports for input and output.

To solve this problem, we reordered the SIFT's 128 values into 8 block RAMs with 16 elements each, where the elements in each block won't be accessed at the same time. Therefore, we can implement each block with one FPGA BRAM. The top 8 blocks in the first line in Figure 41 represent the normal distribution of SIFT vector element in memory, while the lower set represents our implementation.

In the upper set, the grey elements (0,2,8,10,32,34,40,42,64,66,72,74,96,98,104,104) will never be accessed at the same time. Therefore, we reordered these elements and put them in one block memory as shown in the lower set. The same process was applied to the other seven memory blocks.

This memory unit is interfaced with the trilinear interpolation module. The output from the trilinear interpolation module is eight address lines, eight data elements. Before updating the elements, we had to translate the input address to match our implementation. Therefore, we designed a circuit (address converter) that converts the input address into two parts. The first part represents the block number (0-7) and the second part represents the address inside the block (0- 15).



0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67	80	81	82	83	96	97	98	99	112	113	114	115
4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71	84	85	86	87	100	101	102	103	116	117	118	119
8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75	88	89	90	91	104	105	106	107	120	121	122	123
12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79	92	93	94	95	108	109	110	111	124	125	126	127

0	2	8	10	1	3	9	11	4	6	12	14	5	7	13	15	16	18	24	26	17	19	25	27	20	22	28	30	21	23	29	31
32	34	40	42	33	35	41	43	36	38	44	46	37	39	45	47	48	50	56	58	49	51	57	59	52	54	60	62	53	55	61	63
64	66	72	74	65	67	73	75	68	70	76	78	69	71	77	79	80	82	88	90	81	83	89	91	84	86	92	94	85	87	93	95
96	98	104	106	97	99	105	107	100	102	108	110	101	103	109	111	112	114	120	122	113	115	121	123	116	118	124	126	117	119	125	127

Figure 41: Histogram Memory Implementation

5.2 Bag of Feature Hardware Implementation

The bag of feature module is used to represent the image as a one vector of SIFT features. It converts the image from a set of SIFT descriptors (k×128) into a vector of size (N). The first step is to cluster the SIFT descriptors into N clusters using K-mean clustering algorithms. The second step is to quantize each SIFT descriptor into the nearest cluster center.

In our implementation, the SIFT features are extracted from a number of images from each class. These descriptors are used to find the best (N) cluster centres that minimize the sum of the squared Euclidean distance between the points and their nearest cluster centre. We used Matlab code to implement the k-means clustering algorithm. Then, the cluster centres are loaded to the bag of feature module in the hardware using (N) FIFO buffers with a length of 128 elements.

When the enable signal is activated, every clock cycle then one element of the SIFT descriptor (SIFT[i]) is shifted inside the module. Using N subtractors, this value is subtracted from the proper cluster centre's element. The results are accumulated using N accumulators.

After 128 clock cycles the max block searches the N accumulators to find the minimum value. The minimum value means that the distance between the input SIFT vector and that cluster centre is the minimum. The final step is to increment the histogram element by one. Figure 42 shows the BoF hardware architecture.

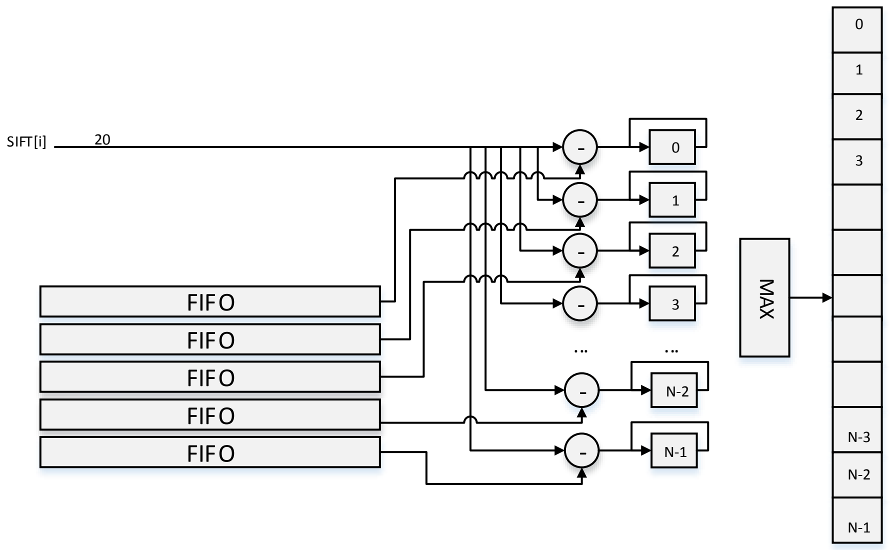


Figure 42: Bag of Feature Module

5.3 Support Vector Machine Hardware Implementation

In this section, the proposed hardware architecture for SVM classifier is presented. The architecture implements the one-against-all multi-class SVM classifier with RBF kernel. The architecture exploits the parallelism in computing the RBF kernel in the decision function to reach real-time performance. We implement the decision function in SVM classifier, while we use off-line training phase using software running on a PC.

The input to the SVM classifier module is an image represented as a vector generated by the BoF module. The output is the class label after the classification process is completed. In this implementation, the training phase of SVM is off-line. The LibSVM Matlab code is used to solve the dual Lagrange problem in the SVM training phase. The output of the training phase is a set of support vectors that build the boundary hyperplane and the set of weights (α). The extracted parameters are used to implement the testing phase of the SVM on the hardware.

In the SVM testing phase, the new data vector (x) is classified according to the decision function in equation (39). Where k (.,.) is the kernel function, Nsv is the number of support vectors generated in the training phase, αi is weights for each SV and b is a bias constant. While xi and yi represent the SV and the SV's class label. yi= {-1,1}.

$$f(x) = sign(\sum_{i=1}^{Nsv} yi \ \times \alpha i \times k(xi, x) + b) \tag{39}$$

$$K(x, z) = \exp(-\| x - z \|^2 / (2\sigma^2)) \tag{40}$$

The architecture for the kernel function is shown in Figure 43. In this module, the input image and the SV values are shifted into the module one value every clock cycle. The output represents the Kernel value defined by equation (40).

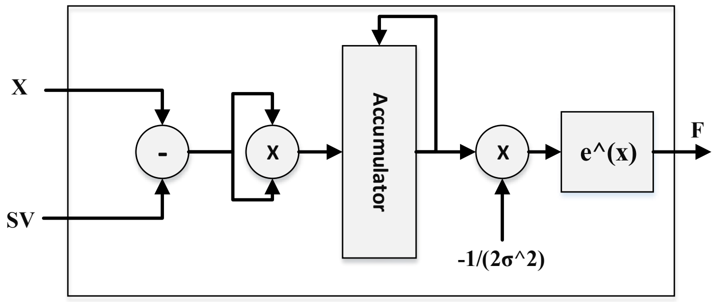


Figure 43: RBF Kernel Function Architecture

The RBF kernel module consists of one subtractor, two multipliers, one accumulator and one module to compute the exponential function. To compute the norm value in the kernel function$\| x - z \|^2$, we simplify the computation by using equation (42) instead of equation (41).

In this case we don't have to compute the square root. We can compute the summation of $(xi - zi)^2$, then multiply it by itself using one multiplier.

$$\| x - z \| = \sqrt{(x1 - z1)^2 + (x2 - z2)^2 + \cdots + (xn - zn)^2} \qquad (41)$$
$$\| x - z \|^2 = (x1 - z1)^2 + (x2 - z2)^2 + \cdots + (xn - zn)^2 \qquad (42)$$

In the kernel module, a new value from the X and SV is shifted into the module each clock cycle; the subtractor computes the difference and the multiplier computes the square value of the difference. After the n clock cycle, the value of $\| x - z \|^2$ becomes valid and the exponential block computes the exponential value and sends it to the output port where n is the length of SV and X.

The accumulator size in the kernel function was chosen based on equation (43), which depends on the data width (20 bits) and the size of the input (c). In our case (c) equal to the SV and X length (c=500).

$$\text{Accumulator bit width} = \log_2(c \times (2^{20} - 1)) \qquad (43)$$

After n clock cycle, the output of the accumulator is multiplied by $(-1/(2\sigma^2))$. For implementation of exponential function, we represent the exponential function as a summation of hyperbolic sine and hyperbolic cos of the x, as given in equation (44).

$$\exp(x) = Sinh(x) + Cosh(x) \qquad (44)$$

The Xilinx IP Core CORDIC block is used to produce the sinh and cosh of the input. The input range for CORDIC block is from -pi/4 to pi/4.

The data format for SVs and the input X is 20 bits with 1 bit for the integer part and 19 bits for the fractional part. The output of the second multiplier represented as 20 bits with 3 bits for the integer and 17 bits for the fractional part. The exponential module's output was 20 bits with 2 bits for the integer and 18 bits for the fraction.

The overall architecture for SVM module is depicted in Figure 44. We used FIFOs buffer lines to store the SVs and $yi \times \alpha i$ values. The value of these buffers comes from the training phase carried out on the PC. In this architecture, we used 20 modules to compute the kernels function between the input vector X and 20 SVs.

When a new input vector enters the SVM module, the hardware architecture computes the SVM decision function and this is based on the sign of the highest bit in the last accumulator the class label will be assigned, as shown in Figure 44.

In the SVM architecture, the processing time required to classify one image is computed by equation (45):

$$\text{Time} = floor\left(\frac{N\_SV}{20}\right) \times \text{SV\_dimension} \times \#\text{Classes} \times \left(\frac{1}{\text{opertional frequency}}\right) \quad (45)$$

For example, for a 5 class problem with the SV dimension equal to 100 and 400 SVs we will need to 20×100×5 = 10000 clock cycles to find the class of the input image.

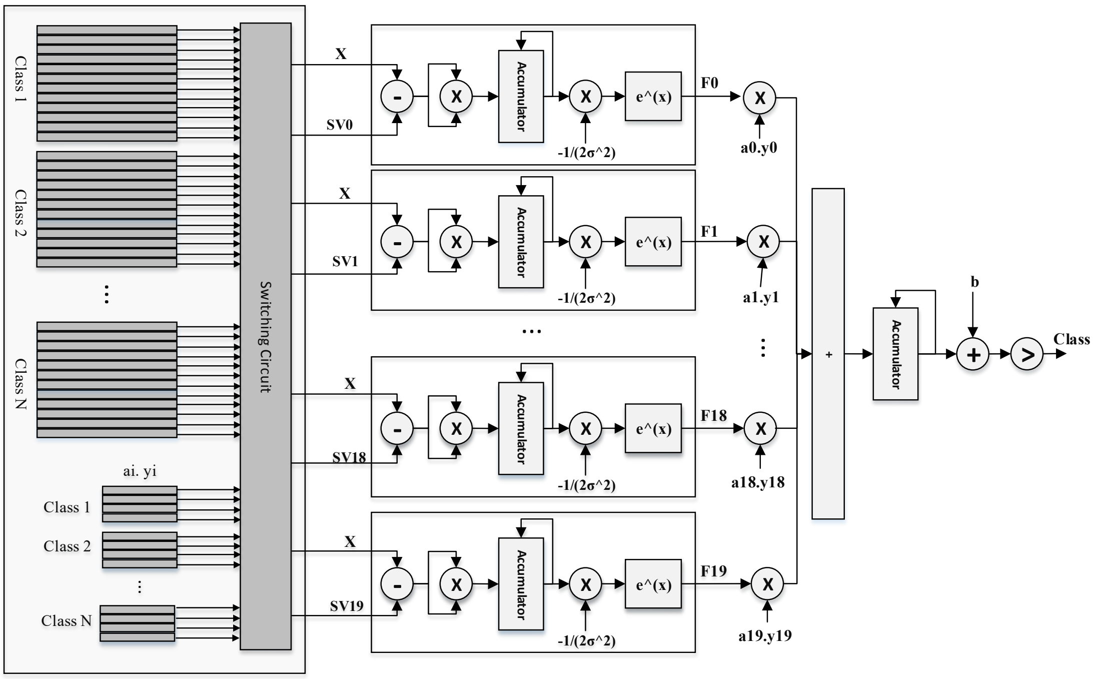


Figure 44: The Overall Architecture for SVM

5.4 FPGA Implementation

In this chapter, we developed a prototype of our architecture on the FPGA. We used ML505 evaluation platform which is equipped with Virtex 5 LX110T FPGA. It also has an external SRAM with a capacity of 8 MB, DVI output and compact flash card reader, which makes it suitable for our implementation. The overall system consists of the SIFT processor core, BoF module, SVM classifier core and MicroBlaze soft-core processor. Our prototype is interfaced with the Microblaze processor via two FSL bus systems for I/O purposes. The overall system is illustrated in Figure 45.

5.4.1 I/O System Based on Microblaze. Microblaze processor handles tasks such as data transferring between I/O peripherals and hardware modules, as well as controlling the data flow. Initially, the input image frames is stored in the compact flash card (acting as the image acquisition source). The Microblaze loads the frame to the external SRAM before the SIFT extraction phase. Microblaze reads one pixel data from the SRAM. After that the processor will read one pixel at a time and send it to the SIFT core via Fast-Simples-Link (FSL) bus interface. When the final hardware module's valid signal becomes high, the microblaze processor will read the results and store them in a predifiened array in the SRAM.

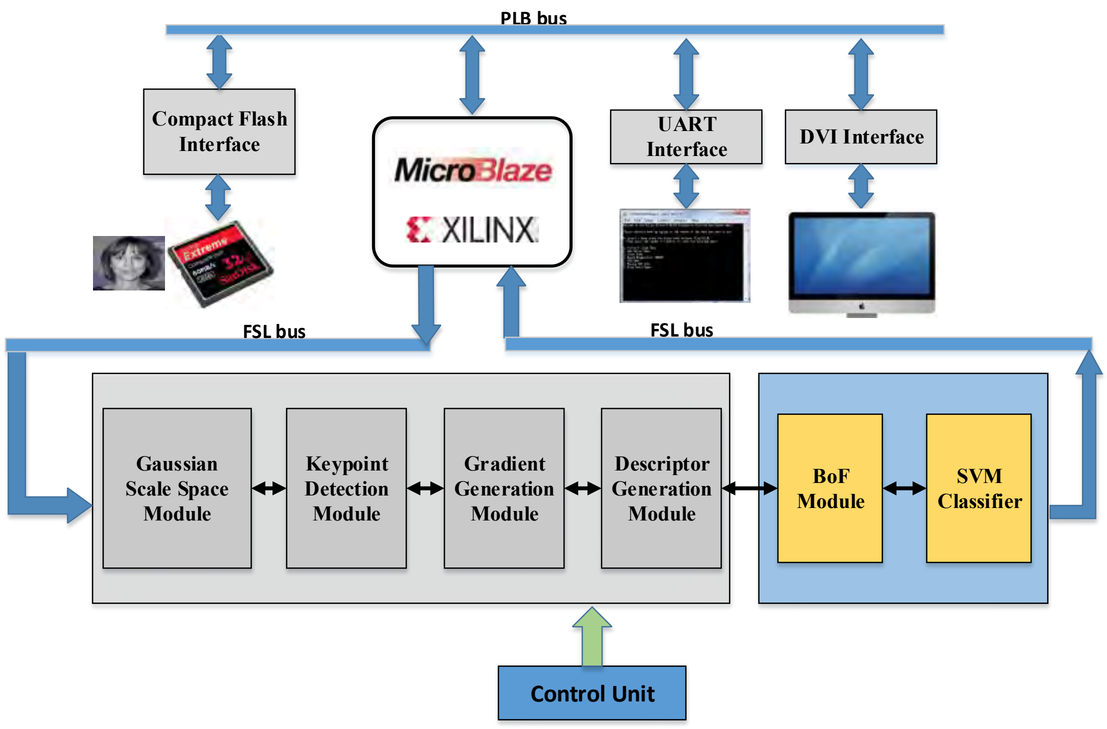


Figure 45: Block Diagram of the FPGA Prototype System

In our implementation, we used Fast Simplex Link (FSL) connections to interface our customized hardware architecture with the MicroBlaze processor. Figure 46 shows in details how our architecture is connected into the MicroBlaze FSL interfaces For the FSL0 connection, the MicroBlaze is a Master on the FSL bus and our architecture is the Slave. Thus, the MicroBlaze controls the data flow on the FSL0 bus. For the FSL1 bus, it is vice versa, our architecture is the Master and the MicroBlaze is the Slave.

Each FSL bus has the following signals: FSL_Clk, FSL_rst, FSL_M_Data, FSL_M_Control, FSL_M_Write and FSL_M_Full in the master side and FSL_S_Data, FSL_S_Control, FSL_S_Read and FSL_S_Exist in the slave side. These signals make the FSL unidirectional point-to-point communication bus. Table 8 summarizes the different FSL bus signals.

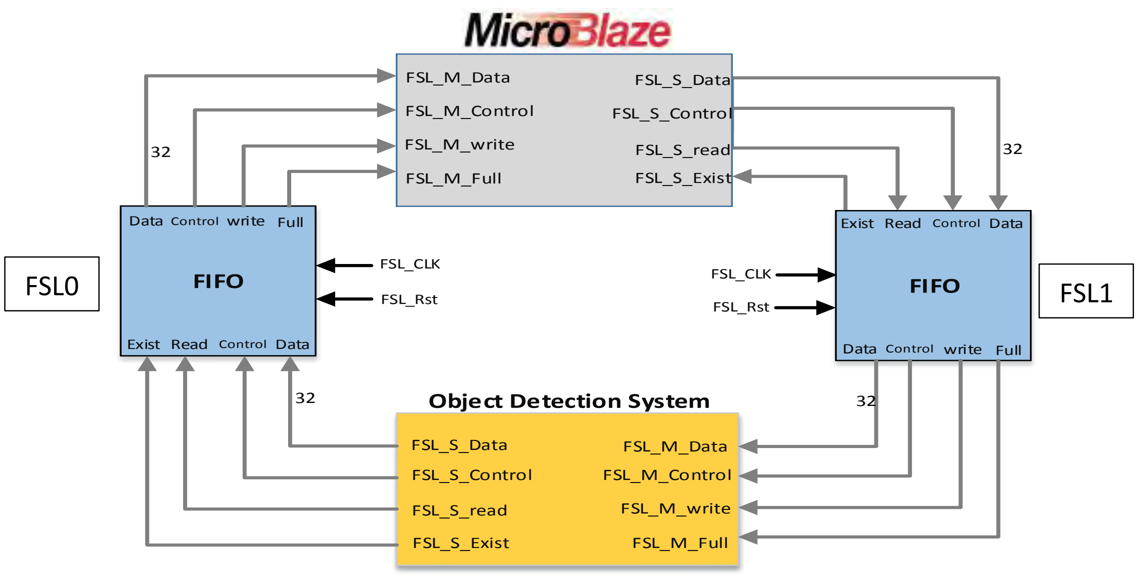


Figure 46: FSL Connections between MicroBlaze and our Architecture

Table 8: FSL Bus Signals

Signal Name	I/O	width	Description
FSL_Clk	I	1	Input clock to the FSL bus
FSL_rst	I	1	External system reset
FSL_M_Data	I	32	Data input to the master side
FSL_M_Control	I	1	Single bit control
FSL_M_Write	I	1	Controls the write enable signal
FSL_M_Full	O	1	Indicates that the FIFO is full
FSL_S_Data	O	32	Data output bus Slave side
FSL_S_Control	O	1	Single bit control
FSL_S_Read	I	1	Controls the read acknowledge signal
FSL_S_Exist	O	1	Indicates that FIFO contains valid data

In the Microblaze C code, we used the non-blocking read and write FSL functions (nputfsl (input, input_slot_id) and ngetfsl (value, output_slot_id)) to send and receive data to/from the FSL bus FIFO. We also used the (fsl_isinvalid () and fsl_iserror()) functions to check the errors and invalid flags after each operation.

The customized hardware architecture is implemented using Verilog hardware description language. We used Xilinx ISE environment, XPS and SDK to edit, implement and verify the functionality of the design. The whole system is prototyped using the ML505 evaluation platform. Figure 47 shows the ML505 evolution platform.

Figure 47: ML505 Evaluation Board [53]

Chapter 6: Experimental Results

In this chapter, we present the experimental results for the object detection architecture proposed in Chapter 5. The performance can be measured in terms of three metrics: classification accuracy, speed up compared to a CPU implementation, and hardware utilization. There is a trade-off between these metrics where higher accuracy requires more hardware resources and processing time. In this work, the priority was to achieve higher accuracy within real-time constraints which is processing 30 frames per second with an image resolution of (640×480).

6.1 Modules' Accuracy

In the hardware implementation, we used fixed point numbers, while the SIFT software used floating point numbers. This can lead to losing some accuracy when implementing the algorithms in hardware. To assess the accuracy achieved by our implementation, the error is approximated based on equation (46):

$$Error(\%) = \frac{|Software\ value - Hardware\ value|}{Software\ value}\ x\ 100\ \% \qquad (46)$$

The percentage error is computed by subtracting the software value from the value generated by the proposed hardware, then divided by the software value and multiplied by 100.

Figure 48 shows how we compute the error between the software's results and the hardware results. Matlab code generates the result of each module and saves it in files. The output of the modules is also saved to files. The errors are computed by comparing the contents of these files. In the next subsection, the accuracy of each hardware module in the proposed architecture is presented.

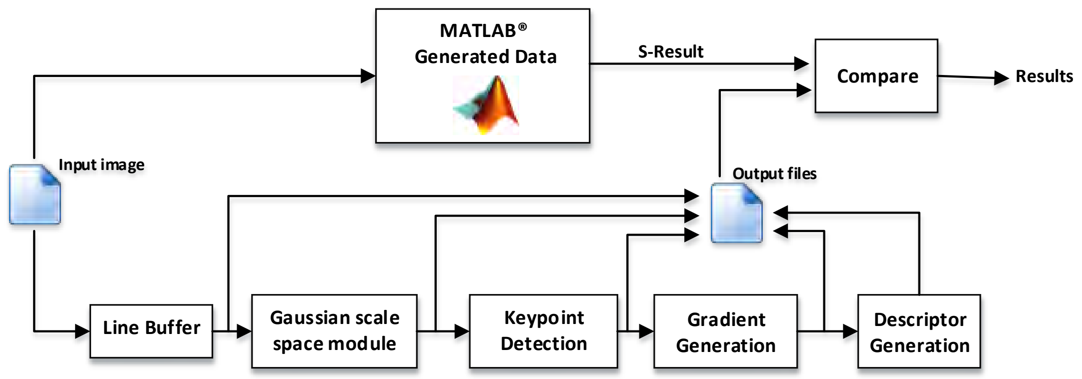


Figure 48: Modules' Accuracy Calculation Technique

6.1.1 Gaussian Scale Space Generation Module Accuracy. In the GSS hardware module, the input image's pixels are represented using 8 bits integer numbers while the pixels of Gaussian filtered images are represented as 9:11 fixed point numbers, with 9 bits for the integer part and 11 bits for the fractional part. To compute the error we used the human face image shown in Figure 49. Using Matlab code, the Gaussian filtered images in the first octave are computed and saved. These images are shown in the upper set of Figure 49. The results of the GSS hardware module are saved and converted from hex to fixed point numbers. These images are shown in the lower set of Figure 49.

For each pixel in the image, the average errors in the first octave for its six scales are reported using this human face image. The errors in scale0, scale1, scale2, scale3, scale4 and scale5 were 1.760 %, 2.040 %, 2.930 %, 3.870 %, 4.310 % and 5.670 %, respectively.

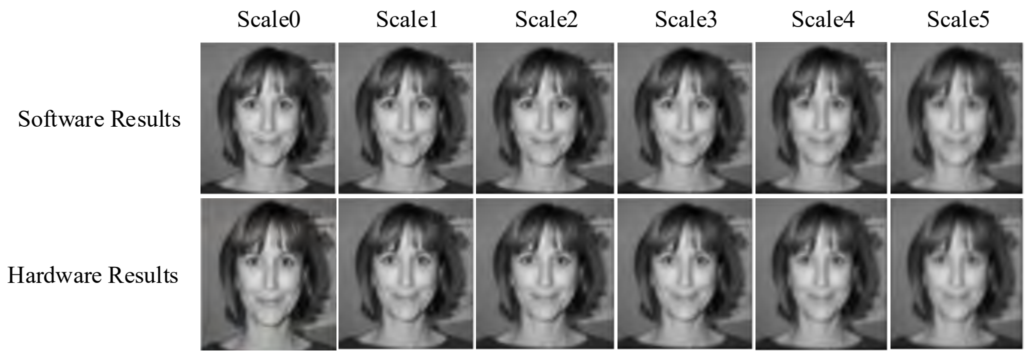


Figure 49: Filtering difference between Software Implementation and GSS Module

6.1.2 Keypoint Detection Module Accuracy. The keypoint detection module's accuracy is measured by comparing the number of detected keypoints in both software and hardware modules. In our implementation, the number of keypoints in hardware was the same as that in the software implementation. Hence, there was no over-detection or under-detection in the keypoint detection module. Our implementation achieved an accuracy equals to 99.93%. The accuracy is computed by dividing number of true matches over the total number of keypoints. The true matches are the keypoints with distance less than 5 pixels from the original location detected in software.

6.1.3 Gradient Magnitude and Orientation Module Accuracy. The gradient magnitude values are represented using 20 bits fixed point numbers with 9:11 integer and fraction bits. The gradient orientation value represented by 16 bits with 3:13 integer and fraction bits. The accuracy of the proposed hardware implementation was computed by comparing the software generated values against the hardware generated values.

The error in gradient magnitude results was equal to 3.65% and in gradient orientation results it was equal to 1.527%. These errors can be reduced by increasing the width of fraction bits. However, these errors are acceptable because it does not affect the quality of the extracted SIFT features.

6.1.4 Rotation Module Accuracy. To measure the rotation module accuracy, we generate the rotated coordinates and orientations (X_r, Y_r, and O_r) using Matlab. Table 25 in appendix B shows the first 100 results obtained for one keypoint. It shows the errors in Xr, Yr, and Or. The average errors for one keypoint in Xr, Yr and Or were 0.257%, 0.257% and 0.252% respectively.

6.1.5 Gaussian Weight Generation Module Accuracy. The Gaussian weight generation module represents the Gaussian weight using 20 bits in a (9:11) format. The error between the values of the software and the proposed hardware is approximately 4.3%.

6.1.6 SIFT Descriptor Generation Module Accuracy. For the SIFT feature generation module, the SIFT element is represented by fixed point numbers with 20 bits in (9:11) format. Table 26 in Appendix B shows one SIFT features extracted from the human face image using software and hardware implementations.

The SIFT feature generated by the software consists of 128 values distributed as $8\times4\times4$ and is shown in the left side of Table 26. The SIFT feature generated by the hardware is shown in the right side of Table 26. The error between the values is computed and is equal to 3.36%.

6.2 Classification Rate Evaluation

In order to assess the classification rate of the proposed hardware implementation, we used two popular image classification datasets: Caltech-256 dataset [31] and KUL Belgium Traffic Sign Classification dataset [34]. These datasets contain a challenging set of object categories. Each category contains a set of images with large variation. These datasets were used to measure the classification rate.

6.2.1 Experiment 1: Caltech-256. In the first experiment, we used ten different subsets from the Caltech-256 dataset. Each subset consists of five different classes. Figures 50 and 51 show examples from the first and the second subset. The classes were faces, airplanes, horses, motorbikes, and watches. Appendix D shows some examples from each subset.

The training phase of the SVM was carried out using the LibSVM Matlab code. The extracted parameters (α, SV and y) were used in the proposed hardware architecture as an input. For training, we used 100 images, 20 images per class, and for testing we used another 100 images, 20 images from each class.

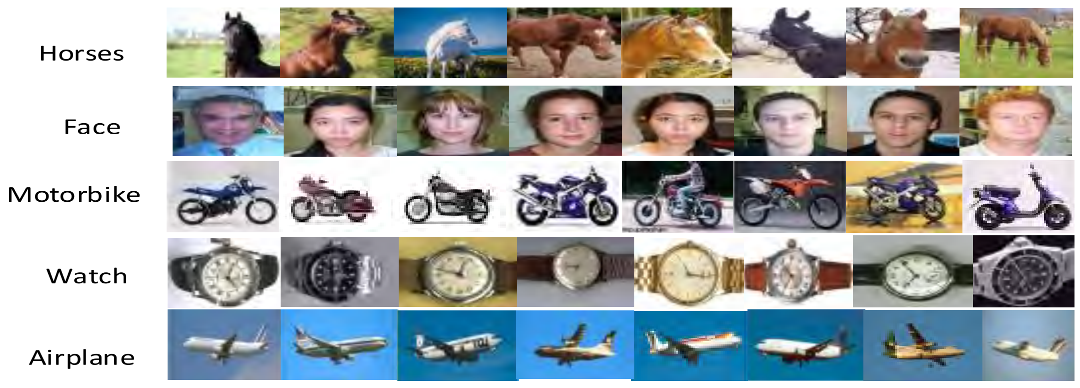


Figure 50: Example images from subset 1 -Caltech-256 dataset

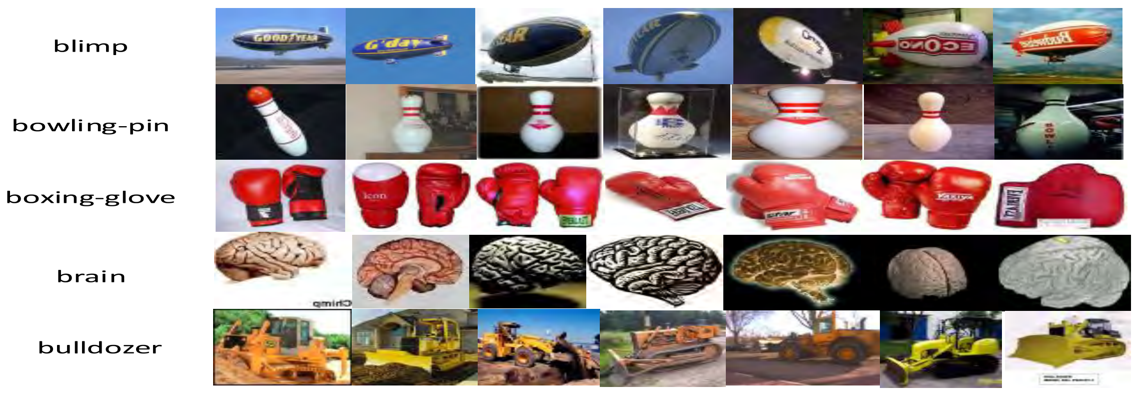


Figure 51: Example image from Subset 2 -Caltech-256 dataset

Table 9 summarizes the classification accuracy in software and hardware implementations for each subset. The highest accuracy achieved was (87%) in subset 7 and the lowest accuracy was (66%) in subset 10. Other subsets have an accuracy between these values.

Table 9: Classification accuracy for ten subsets from Caltech-256

Set #	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6	Set 7	Set 8	Set 9	Set10
Software Accuracy	85%	69%	73%	79%	75%	74%	87%	72%	67%	70%
Hardware Accuracy	84%	69%	71%	77%	72%	70%	87%	71%	66%	69%

For the LibSVM software implementation, the average classification rate for ten subsets was (75.1%) with a standard deviation of (6.7). While our hardware implementation achieved an average classification rate of (73.6%) with a standard deviation of (6.9). The difference in the classification accuracy is due to the fact that the LibSVM software implementation uses floating point numbers while our proposed system uses fixed point numbers which could reduce the accuracy.

Figure 52 shows the average confusion matrix using the ten subsets. We computed the average confusion matrix by averaging the values for all classes combined. The average confusion matrix represents the average accuracy for classifiers obtained through ten different subsets. The experiment is implemented using both software and hardware. The diagonal elements represent the correctly classified images, while other elements represent the misclassified images.

Software Result

	Class 1	Class 2	Class 3	Class 4	Class 5
Class 1	14	1.3	0.9	1.6	2.2
Class 2	1	15.5	0.7	1.1	1.7
Class 3	1.7	0.8	14	1.5	1.8
Class 4	0.7	0.5	0.5	17.4	0.9
Class 5	1.2	1	1.1	0.8	16

Hardware Result

	Class 1	Class 2	Class 3	Class 4	Class 5
Class 1	14	1.3	1	1.3	2.4
Class 2	1	15.4	0.5	1.4	1.7
Class 3	1.7	0.8	13.7	1.5	2.1
Class 4	0.7	0.8	0.5	16.4	1.6
Class 5	1.4	1	0.8	0.8	16

Figure 52: Hardware and Software Average Confusion Matrix

6.2.2 Experiment 2: KUL Belgium Traffic Sign.

In the second experiment, we used five classes from the KUL Belgium Traffic Sign Classification dataset; some examples from each class are shown in Figure 53. In the SVM training phase we used 100 images, 20 images from each class. In the testing phase we used different 100 images, again, 20 images from each class.

For the LibSVM software implementation, the classification rate was (80%), while our hardware implementation achieved a classification rate equal to (78%). Figure 54 shows the confusion matrices. Again, the difference in the classification is due to the fact that the LibSVM software implementation uses floating point numbers while our proposed system uses fixed point numbers which could reduce the accuracy.

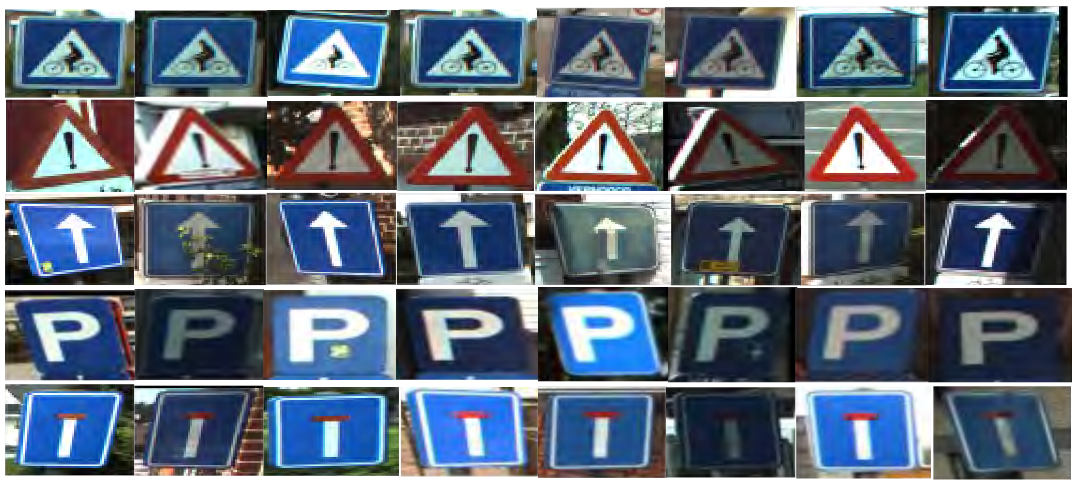
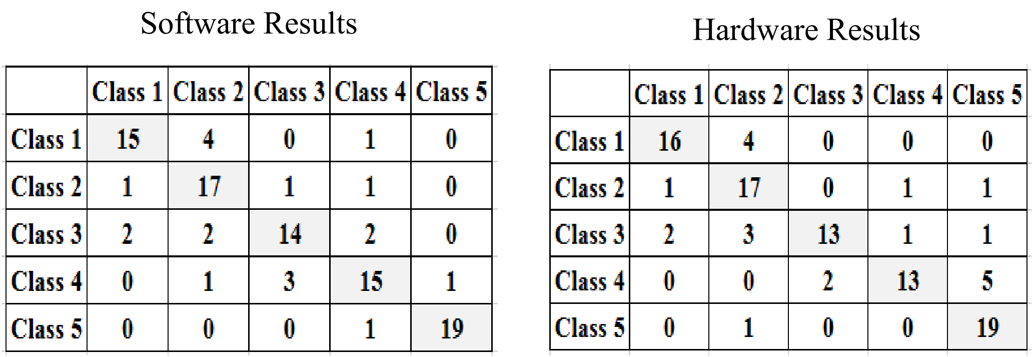


Figure 53: Example images from the KUL Belgium Traffic Sign Dataset


Software Results

	Class 1	Class 2	Class 3	Class 4	Class 5
Class 1	15	4	0	1	0
Class 2	1	17	1	1	0
Class 3	2	2	14	2	0
Class 4	0	1	3	15	1
Class 5	0	0	0	1	19

Hardware Results

	Class 1	Class 2	Class 3	Class 4	Class 5
Class 1	16	4	0	0	0
Class 2	1	17	0	1	1
Class 3	2	3	13	1	1
Class 4	0	0	2	13	5
Class 5	0	1	0	0	19

Figure 54: Hardware and Software Confusion Matrix

6.2.3 SVM Kernel Parameter Selection. In the SVM implementation, we used the RBF kernel function which is given in equation (47). Choosing the appropriate value for C and gamma in RBF kernel is important to achieve a high classification rate. Low C makes the decision surface smooth, while a high C aims at classifying all training examples correctly. The gamma parameter defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'.

$$K(x,z) = \exp(-\| x - z \|^2/(2\sigma^2)) \qquad (47)$$

To find the appropriate values for C and gamma, we compute classification accuracy as a function of sigma. Where gamma (γ) =1/2σ^2.

For the first experiment using the Caltech-256 dataset, the best accuracy was 85%. This is achieved with sigma value of 0.075 as shown in Figure 55.

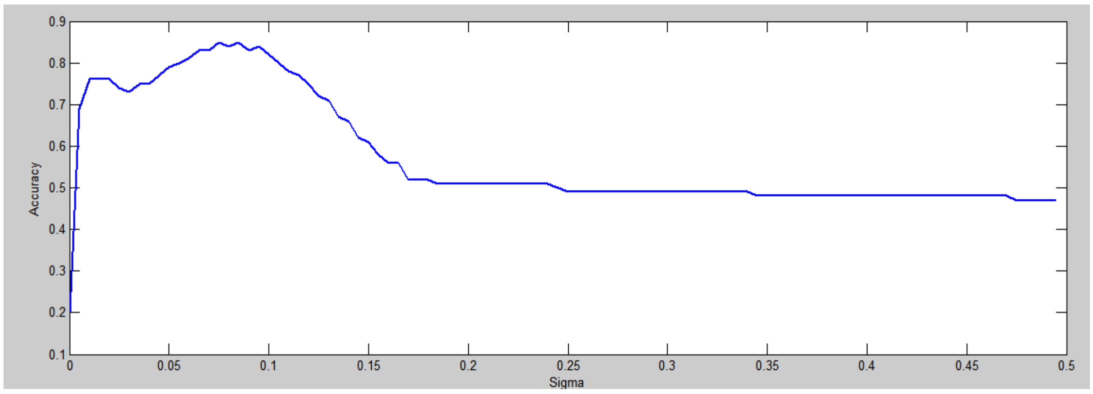


Figure 55: Determining the value of SVM RBF sigma using the Caltech-256 dataset

When the same experiment is repeated using the KUL Belgium Traffic Sign dataset, the best accuracy achieved is (80%) with a sigma of 0.06 as shown in Figure 56.

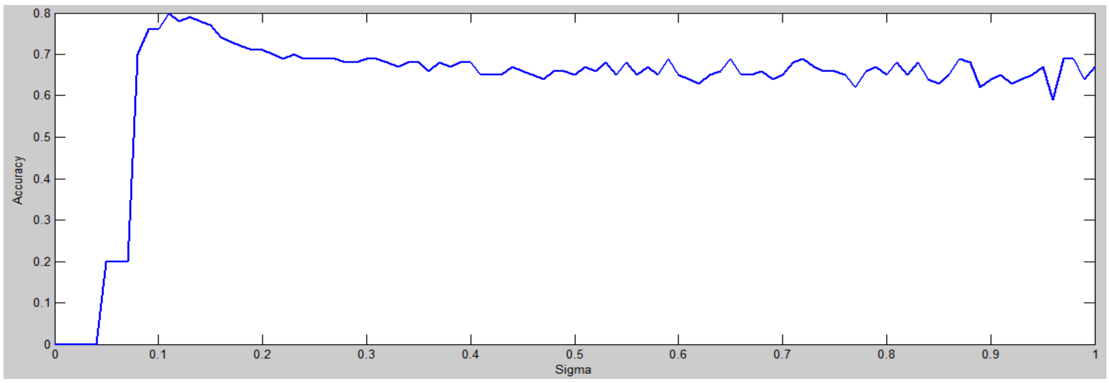


Figure 56: Determining the value of SVM RBF sigma using the KUL dataset

6.2.4 Fraction Part Bit Width and Accuracy.

To assess the effect of increasing the bit width of the fraction part of the data on the accuracy and hardware utilization, the DoG pixels are represented by bit widths ranging from 8 to 15. Figure 57 shows how the hardware resources increase linearly with the bit width. On the other hand, Figure 58 shows that the error decreases with a sharp slope between 8 and 11 bits and then improvement becomes negligible.

These results shows that the resource utilized by the proposed hardware module is directly proportionate to the pixel width. To reduce the FPGA resource utilization without significantly scarifying the accuracy, we found that the optimal pixel width is 11 bits. Therefore, the output pixels in the GSS module are represented with 20 bits. Where 11 bits for fractional part and 9 bits for integer part.

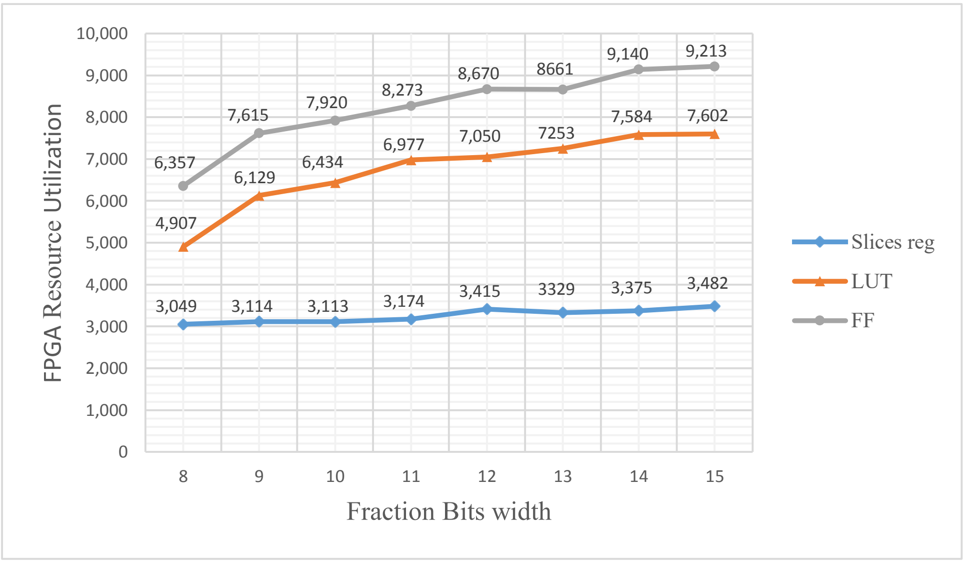


Figure 57: Bit width vs. hardware utilization

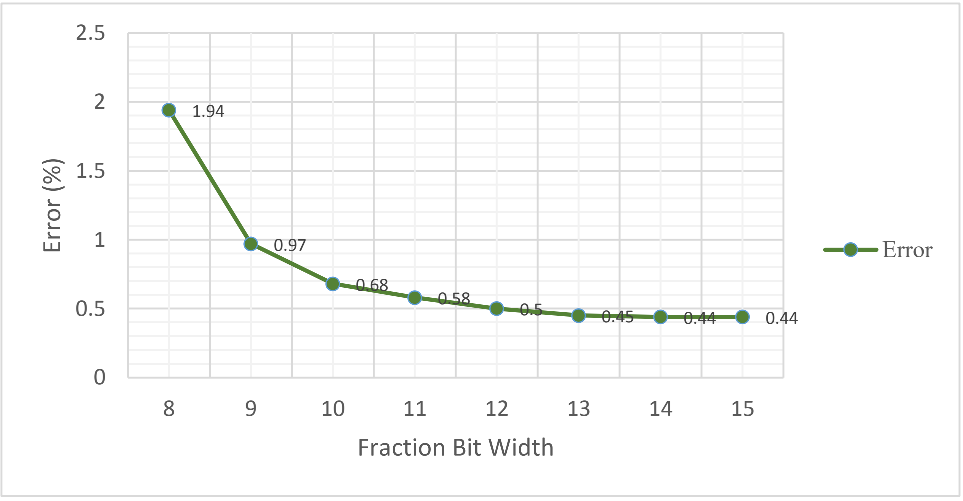


Figure 58: Bit width in the proposed hardware implementation vs. error

6.3 Processing Time

The processing time for each module in our architecture is estimated based on the number of clock cycles required to complete each task and the operational frequency for that module. The processing time is estimated based on equation (48).

$$Processing\ Time(\text{sec}) = \frac{Number\ of\ clock\ cycles\ to\ finish\ the\ task}{Operationl\ frequency\ (Hz)} \qquad (48)$$

In the next section, we will measure the processing time for the three modules in our architecture: the SIFT module, the BoF module, and the SVM module.

6.3.1 SIFT Module Processing Time. The maximum operating frequency of the proposed design is 60.369MHz, which is provided by the synthesis report. For the independent block modules, the maximum frequencies are as follows:

a. 110.373MHz for the Gaussian scale space module.
b. 84.338MHz for the keypoint detection module.
c. 90.651MHz for the orientation assignment module.
d. 130.293MHz for the descriptor generation module.
e. 121.393MHz for the SVM classifier module.

For the SIFT feature extraction module, it takes 640×480 clock cycles to scan the input image and detect the keypoints. For each keypoint, it takes (640+17×17+128) clock cycles to generate the SIFT descriptor so that the processing time for the SIFT feature extraction and description module is estimated as shown in equations (49) and (50).

SIFT's processing time for one frame (640x480) pixel:

$$\text{Processing Time} = \text{Keypoint Detection Time} + \text{Descriptor Generation Time} \qquad (49)$$

$$\text{Processing Time(sec)} = \frac{(640 \times 480 + (640 + 17 \times 17 + 128) \times \#\ \text{KP})}{\text{Operationl frequency (Hz)}} \qquad (50)$$

Where #KP represents number of SIFT descriptor detected in the input image.

At the operation frequency of 50MHz, the processing time of SIFT detector for a VGA frame (640× 480) is about 640×480 /50 MHz = 6.144 ms. The SIFT descriptor's processing time is proportional to the number of detected keypoints. In our architecture, it takes (640+17×17+128) clock cycles to generate one descriptor.

To compute the maximum number of keypoints that can be extracted from each frame, we computed the maximum number of keypoints that could be extracted within 33ms as given by equation (51). This is so because for real-time performance we had to achieve 30 frames per second.

$$33\text{ms} = \frac{(640 \times 480 + \ (640 + 17 \times 17 + 128) \ \times \#\ \text{KP})}{50\ MHz} \tag{51}$$

The maximum number of keypoints in each frame is 1270 keypoints per frames. Therefore, our architecture allows a VGA size image with about 0.4134 % Keypoints to be processed at a speed of 30 frames per second.

To compare the performance of our architecture with the PC-based performance, we begin by computing the SIFT descriptors for one frame using Matlab. We then compare the processing time with our proposed hardware architecture. Figure 59 shows the comparison graph of processing times for the four modules in the SIFT architecture.

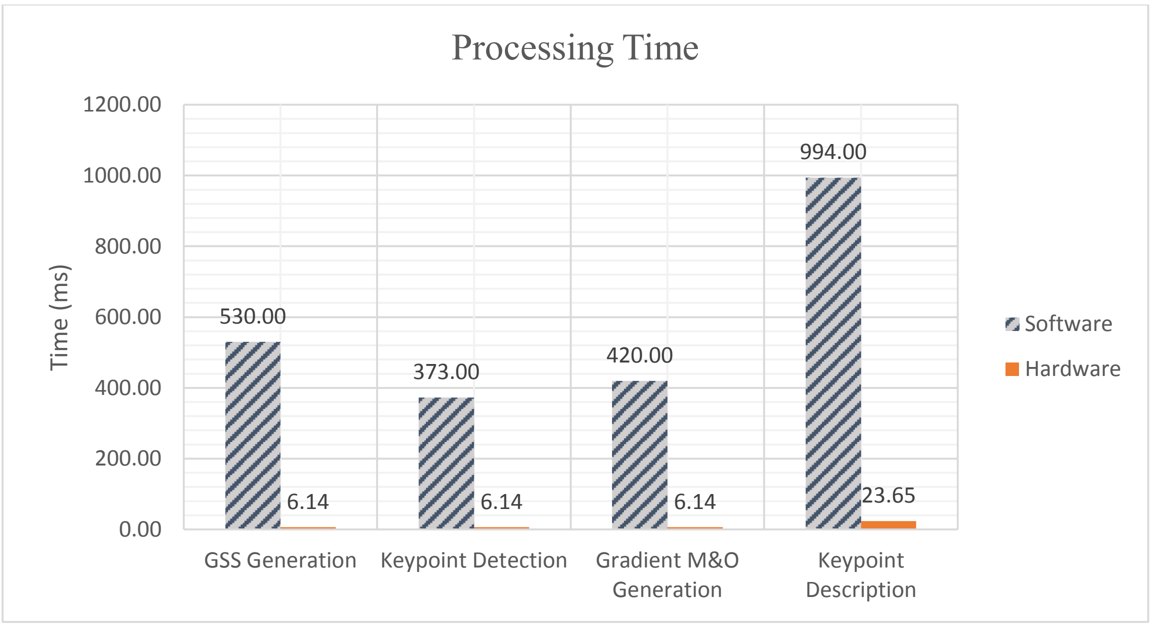


Figure 59: SIFT's Module Processing Time

Table 10 summarizes the processing time for both the software and hardware implementations. It also shows the speedup achieved by each module. The Gaussian scale space module achieved the highest speedup of 86.26, while the keypoint description module achieved the lowest speed up at 42.03. On average the speedup achieved by our architectural hardware is ×55.06 times.

Table 10: Speed up of SIFT's Module Processing Time using the proposed hardware

	Software Time(ms)	Hardware Time (ms)	Speedup
GSS Generation	530.00	6.14	86.26
Keypoint Detection	373.00	6.14	60.71
Gradient M&O Generation	420.00	6.14	68.36
Keypoint Description	994.00	23.65	42.03
Total	2317.00	42.08	55.06

To compare the performance between our architecture and the software implementation using multiple frames, we measured the processing time for extracting the SIFT features for a range of images from 1 to 6. Figure 60 shows the time required to extract the SIFT features. The processing time in the software implementation increases sharply when number of images increases. In the hardware implementation, the increase in processing time is not as sharp as in the software implementation.

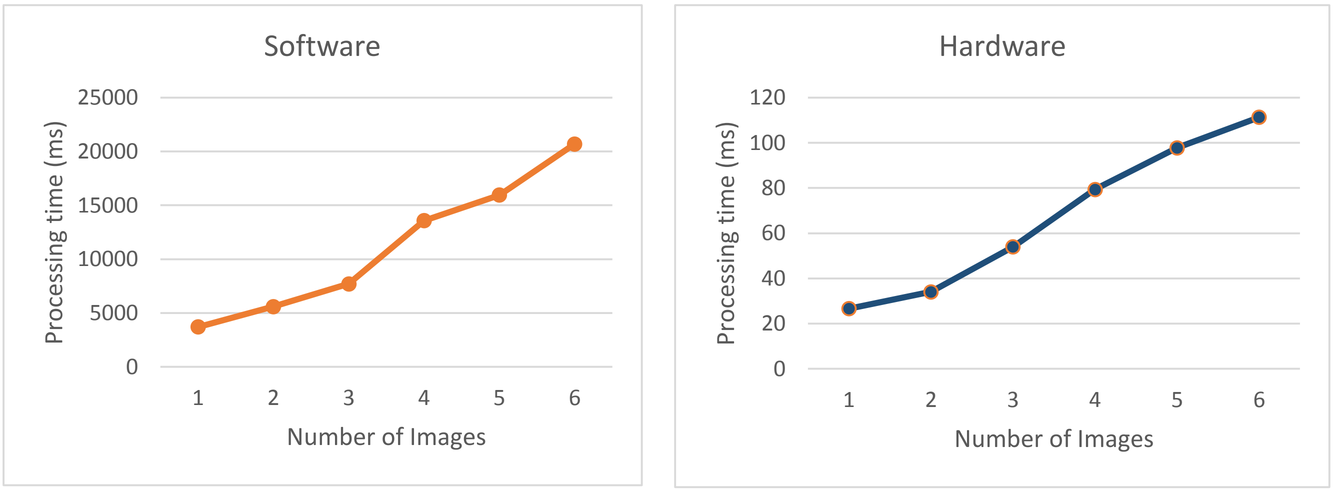


Figure 60: Processing Time for extracting SIFT features from multiple frames

Table 11 shows the processing time results for the software implementation compared to our architecture. It shows the total number of keypoints in each case. We can see that our architecture made a drastic improvement in computing the SIFT features especially when we dealing with multiple frames.

Table 11: Number of keypoints and SIFT processing time results for a range of images

Number of images	1	2	3	4	5	6
Number of Keypoints	971	1323	1789	3370	3864	4977
Software Time (ms)	3711.0	5602.0	7710.0	13580.0	15961.0	20697.0
Hardware Time (ms)	26.67	34.11	53.96	79.39	97.83	111.36

It is clear that the software implementation does not scale up. Whereas in the proposed hardware solution real-time processing is achieved.

6.3.2 BoF Module Processing Time. The BoF module builds the image histogram while the SIFT features are being generated. The processing time for the BoF module is not computed from the overall processing time in our architecture as it is performed in parallel and does not add any extra processing time.

6.3.3 SVM Module Processing Time. The processing time of the SVM classifier is linearly dependent on the number of support vectors and the dimensionality of the feature vectors. As the number of support vectors increases, the processing time increases. Likewise, if the number of features in the input vector increases the processing time increases linearly.

In our SVM architecture, the processing time required to classify one image equals to $floor\left(\frac{N\_SV}{20}\right) \times \text{SV\_Dimentions} \times \text{\#Classes} \times \left(\frac{1}{\text{Maximum opertional frequency}}\right)$.

As an example using the Caltech-256 dataset, we used five classes with each class having 20 images for training. Each image is represented as a vector with 500 elements so that the time required to classify one image is equal to $floor\left(\frac{100}{20}\right) \times 500 \times 5 \times \left(\frac{1}{50\ MHz}\right) = 2.5\ \times 10^{-4} sec.$

To compare the processing time between the software and our proposed architecture, we classified 100 test images into one of the five classes. Table 12 summarizes the confusion matrices, the classification accuracy, and the processing time.

Using Matlab running on an Intel i5 processor with 8 GB RAM, it took 166 ms to classify 100 images, while in our implementation it took only 25 ms to classify these same 100 images. The classification accuracy in our implementation is 3% less than the software implementation.

Table 12: Software implementation results compared to our SVM architecture

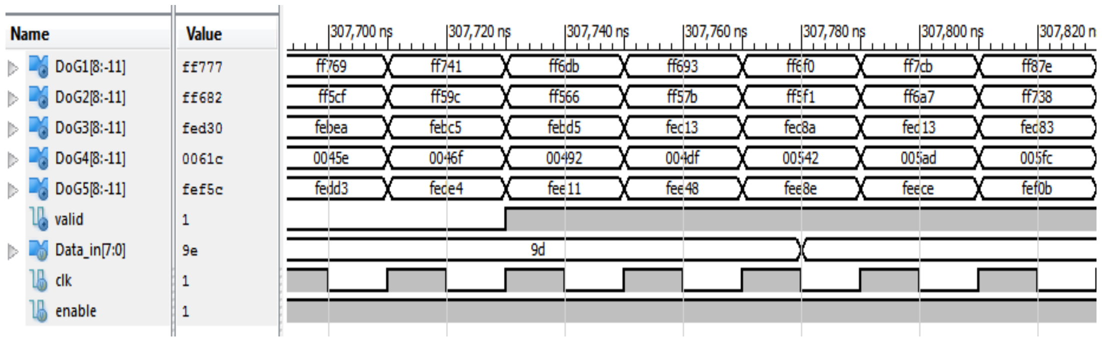
	Class1	Class2	Class3	Class4	Class5		Class1	Class2	Class3	Class4	Class5
Class1	10	0	0	3	7		10	0	1	1	8
Class2	0	20	0	0	0		0	20	0	0	0
Class3	1	0	17	0	2		1	0	15	0	4
Class4	0	0	0	20	0		0	0	0	17	3
Class5	0	0	2	0	18		0	0	0	0	20
Platform	Intel Core i5						Virtex 5 LX110T				
Classification rate	85 %						82 %				
Processing time	166 ms						25 ms				

6.4 Simulation Results:

To debug and verify the design of each module in our proposed architecture, we used the Xilinx ISim simulator. The ISim is a simulation tool integrated into the Xilinx ISE that allows its designers to perform functional and timing simulations for their designs.

In this section, the behaviour of the Gaussian scale space generation module, the keypoint detection module the gradient generation module, the SIFT feature generation module and the SVM module are verified. The source images are read and converted into Hex format and saved to external files using Matlab. In the test bench, an external file is read and in each clock cycle, one pixel is fed into the hardware module. When the valid signal is activated, the test bench starts reading the results generated from the module and saves it back as a text file which is tested after the simulation is finished.

Figure 61 shows the simulation result of Gaussian scale space module. The module has three input ports: Data_in, clk, and enable signal, as well as, six output ports: DoG1, DoG2, DoG3, DoG4, DoG5, and valid. The test bench reads the source image from the external file and at the rising edge of clock, a new pixel is entered into the Gaussian scale module. When the valid signal becomes high (1), the output of DoG is valid.

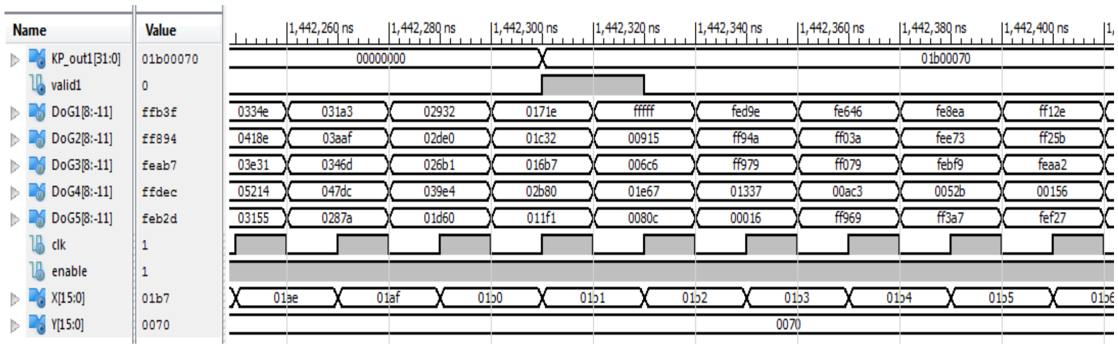


Name
Value
DoG1[8:-11] ff777
DoG2[8:-11] ff682
DoG3[8:-11] fed30
DoG4[8:-11] 0061c
DoG5[8:-11] fef5c
valid 1
Data_in[7:0] 9e
clk 1
enable 1
307,700 ns
307,720 ns
307,740 ns
307,760 ns
307,780 ns
307,800 ns
307,820 n
ff769 ff741 ff6db ff693 ff6f0 ff7cb ff87e
ff5cf ff59c ff566 ff57b ff5f1 ff6a7 ff738
febea febc5 febd5 fec13 fec8a fed13 fed83
0045e 0046f 00492 004df 00542 005ad 005fc
fedd3 fece4 fee11 fee48 fee8e feece fef0b
9d

Figure 61: Gaussian Scale Space Module Simulation Result

In this simulation, we used a clock with a 20 ns clock period. As shown in the figure, at (307,720) ns, the valid signal becomes high and the first pixel in the DoG1, DoG2, DoG3, D0G4, and DoG5 images is (0xFF6DB), (0xFF566), (0xFEBD5), (0x00492) and (0xFEE11), respectively. When the valid signal becomes valid, the test bench starts saving the output in an external file. After the simulation is finished, input and output external files are compared for validation.

The simulation results of the keypoint detection module is shown in Figure 62. The module has seven input ports: clk, enable, DoG1, DoG2, DoG3, DoG4, and DoG5. The module also has two output ports: KP_out and valid_out. The DoG1, DoG2, DoG3, DoG4, DoG5 data come from the DoG module, and the enable signal connected to the valid signal in DoG module. When the keypoint detection module finds a stable keypoint, the valid signal becomes high and the KP_out displays the keypoint position. The first 16 bits of KP_out contains the key point’s X-position and last 16 bits contains the key point’s Y-position, as shown in Figure 62 at (1,442,300) ns.



Name
Value
KP_out1[31:0] 01b00070
valid1 0
DoG1[8:-11] ffb3f
DoG2[8:-11] ff894
DoG3[8:-11] feab7
DoG4[8:-11] ffdec
DoG5[8:-11] feb2d
clk 1
enable 1
X[15:0] 01b7
Y[15:0] 0070
1,442,260 ns
1,442,280 ns
1,442,300 ns
1,442,320 ns
1,442,340 ns
1,442,360 ns
1,442,380 ns
1,442,400 ns
00000000 01b00070
0334e 031a3 02932 0171e fffff fed9e fe646 fe8ea ff12e
0418e 03aaf 02de0 01c32 00915 ff94a ff03a fee73 ff25b
03e31 0346d 026b1 016b7 006c6 ff979 ff079 febf9 feaa2
05214 047dc 039e4 02b80 01e67 01337 00ac3 0052b 00156
03155 0287a 01d60 011f1 0080c 00016 ff969 ff3a7 fef27
01ae 01af 01b0 01b1 01b2 01b3 01b4 01b5 01b6
0070

Figure 62: Keypoint Detection Module Simulation Result

The simulation results of the gradient generation module is shown in Figure 63. This module has three input ports: clk, enable, and Data_in, as well as, three output ports: valid, G_mag, and G_ori. The Data_in is the third image of the first octave in the DoG module, and the enable is connected to the valid signal in the DoG module.

As shown in Figure 63, at (26,020) ns, the valid is high. The first valid gradient magnitude is (0x000D9) and the gradient orientation (0xF277). We used a fixed-point number with 3 bits for the integer and 13 bits for the fraction so that we could represent the gradient orientation. Therefore, the (0xF277) equals to (-0.4230). For gradient magnitude we used 9 bits for the integer and the 11 bits for fraction. Therefore, the (0x000D9) equals to (0.1060).

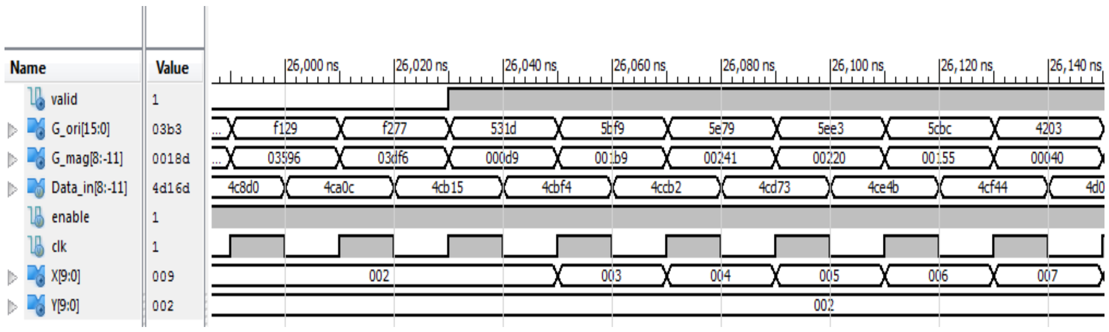


Figure 63: Gradient Generation Module Simulation Result

The simulation result of dominant orientation module is shown in Figure 64. The module has five input ports: keypoint location, gradient magnitude, gradient orientation, clk and enable. It also has two output ports: dominant orientation and disable DoG module signal.

The dominant orientation module reads a keypoint at location (0x011d0021). The keypoint x-position equals to (0x011d) and the y-position equals to (0x0021). But the current window is (Xp=0x0116) and (Yp=0x0017), so that the dominant orientation module will wait until the current window becomes equal to the keypoint position. In other words: Xp=key point's x-position. & Yp=key point's y-position.

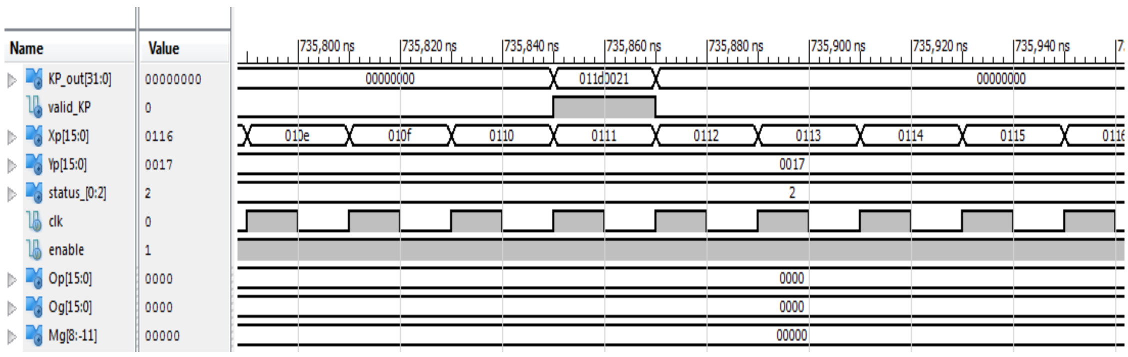


Figure 64: Dominant Orientation HW Module Simulation Results

When the current window becomes equal to the keypoint's position, the state of the dominant orientation module changes from (2) to (1), as shown in Figure 65. At (864,050) ns, the state changed to (1) and the Disable_DoG signal became high (logic 1) to stop reading from the source image. State (1) lasted to 640 clock cycles. In state (1) the buffer lines shifted to the right to compute the principle orientation from the (17×17) window around the keypoint.

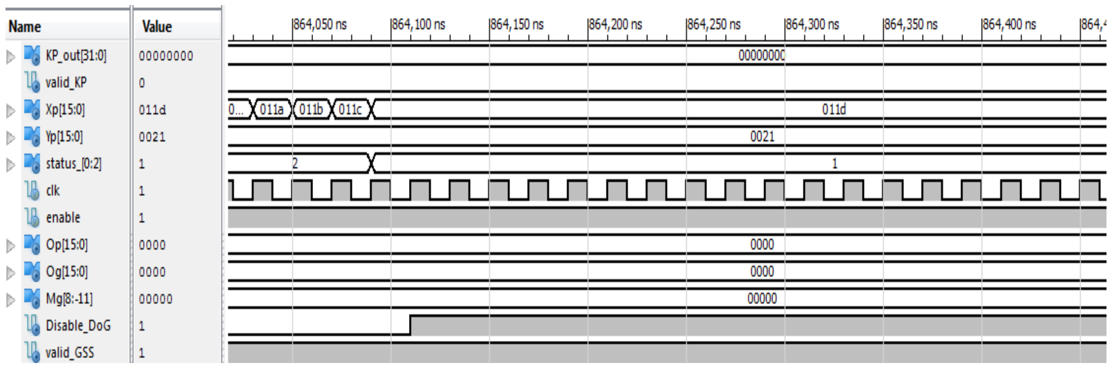


Figure 65: Dominant Orientation Module Simulation Result 2

After 640 clock cycles, the principle orientation value became valid (Og =6) and the Disable_DoG returned to (0), as shown in Figure 66.

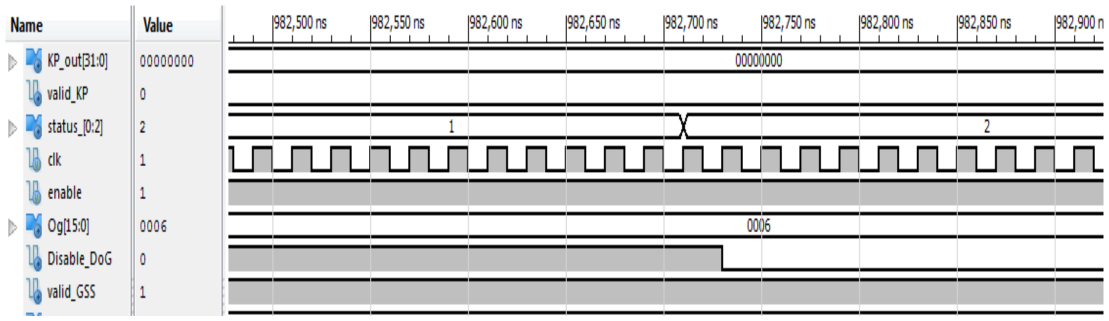


Figure 66: Dominant Orientation Module Simulation Result 3

The SIFT descriptor module simulation results are shown in Figure 67. At (8,050) ns, the SIFT descriptor vector values became valid. The next module read the SIFT vector enabling it to read the _en signal. The 128 values of SIFT vector is read from the element output port.

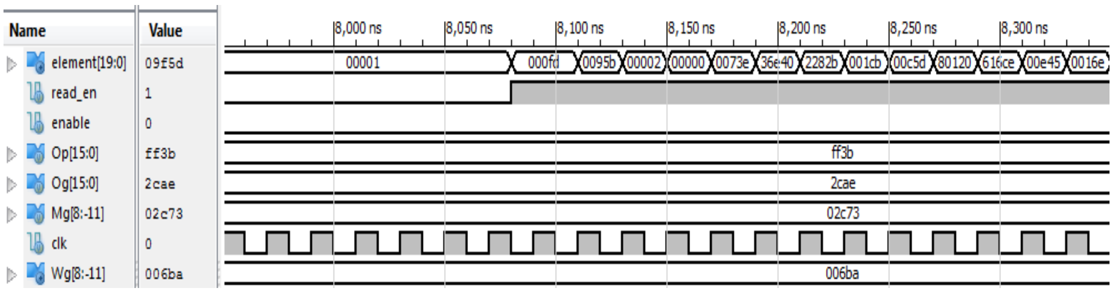


Figure 67: SIFT Descriptor Module Simulation Result

The simulation result of SVM engine is shown in Figure 68. The input to the module is new data (X) and the support vector (SV). Each one has 500 elements. The module starts working when the enable signal becomes high. The Sub represents the difference between X and SV. The accumulator accumulates the square value of the difference between X and SV.

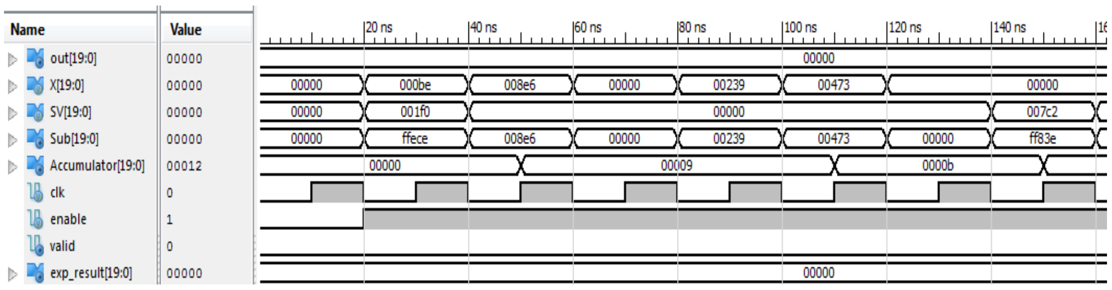


Figure 68: SVM Engine Simulation Result 1

After 500 clock cycles, the accumulator contains the addition of the square root of difference between X and SV. The output value in Figure 69 represents the exponential of the accumulator value multiplied by the gamma constant, as defined by the RBF SVM equation.

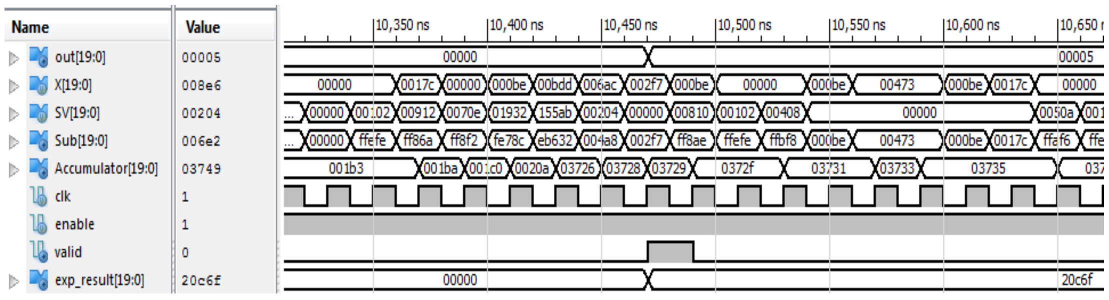
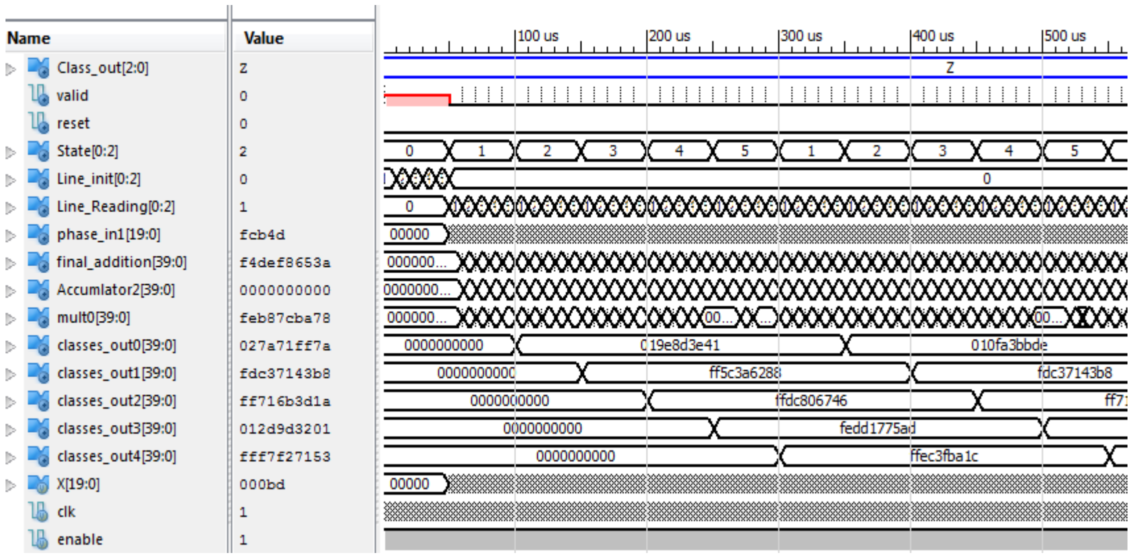


Figure 69: SVM Engine Simulation Result 2

Figure 70 shows the simulation results of the SVM module. In this simulation, the number of classes was five. And each image is represented with a feature vector composed of 500 elements. In the initialization phase, the SVs and α×y values are shifted into the FIFO buffer lines. The input image X is shifted into the module one value every clock cycle. In this implementation, we used five SVM modules to implement five one-against all multi-class SVM.

For the first image: class_out0 = 0x 019E8D3E41, class_out1= 0xFF5C3A6288 class_out0 = 0x FFDC806746, class_out1= 0xFEDD1775AD class_out0 = 0x FFEC3FBa1C. In this case, class_out0 was positive meaning that the first image belongs to class A. The class_out1 value was negative meaning the first image does not belong to class B. The same is applied for classes C, D, and E.



Name	Value
Class_out[2:0]	z
valid	0
reset	0
State[0:2]	2
Line_init[0:2]	0
Line_Reading[0:2]	1
phase_in1[19:0]	fcb4d
final_addition[39:0]	f4def8653a
Accumlator2[39:0]	0000000000
mult0[39:0]	feb87cba78
classes_out0[39:0]	027a71ff7a
classes_out1[39:0]	fdc37143b8
classes_out2[39:0]	ff716b3d1a
classes_out3[39:0]	012d9d3201
classes_out4[39:0]	fff7f27153
X[19:0]	000bd
clk	1
enable	1

Figure 70: SVM HW Module Simulation Result

6.5 Hardware Utilization

This section presents the hardware utilization for each module in the proposed architecture. These modules are SIFT, BoF, and SVM. The main hardware resources in the FPGA are slice registers, slice LUTs, LUT flip flop pairs, DSP blocks, and memory. The results are reported from the synthesis reports generated by Xilinx ISE environment.

Tables 13, 14 and 15 summarize the hardware resources used to implement the Gaussian scale space module, keypoint detection module, gradient magnitude, and orientation module, respectively.

Table 13: Utilized Hardware for the Gaussian Scale Space Module

Device Utilization Summary			
Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	3,415	69,120	4%
Number of Slice LUTs	7,155	69,120	10%
Number of LUT Flip Flop pairs used	8,215		
Number of Block RAM/FIFO	15	148	10%

Table 14: Utilized Hardware for the Keypoint Detection Module

Device Utilization Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	1,900	69,120	2%
Number of Slice LUTs	4,911	69,120	7%
Number of fully used LUT-FF pairs	665		
Number of Block RAM/FIFO	10	148	6%

Table 15: Utilized Hardware for the Gradient Magnitude and Orientation Module

Device Utilization Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	654	69,120	0%
Number of Slice LUTs	709	69,120	1%
Number of fully used LUT-FF pairs	273	1090	25%
Number of Block RAM/FIFO	2	148	1%
Number of DSP48Es	2	64	3%

Tables 16, 17 and 18 summarize the hardware resources used to implement the dominant orientation module, SIFT descriptor module, and the whole architecture of the SIFT algorithm, respectively.

Table 16: Utilized Hardware for the Dominant Orientation Module

Device Utilization Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	2,449	69,120	3%
Number of Slice LUTs	2,937	69,120	4%
Number of fully used LUT-FF pairs	1,226	4,160	29%
Number of Block RAM/FIFO	27	148	18%
Number of BUFG/BUFGCTRLs	1	32	3%

Table 17: Utilized Hardware for the SIFT Descriptor Module

Device Utilization Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	410	69,120	0%
Number of Slice LUTs	2,152	69,120	3%
Number of fully used LUT-FF pairs	229	2,333	9%
Number of Block RAM/FIFO	2	148	1%
Number of DSP48Es	23	64	35%

Table 18: Utilized Hardware for the SIFT Module

Device Utilization Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	7,924	69,120	11%
Number of Slice LUTs	16,138	69,120	23%
Number of LUT Flip Flop pairs used	4,097	65,284	6%
Number of Block RAM/FIFO	68	148	45%
Number of DSP48Es	53	64	82%

Table 19, 20 and 21 summarize the hardware resources used to implement the bag of feature module, one support vector engine module, and the architecture of SVM classification module, respectively.

Table 19: Utilized Hardware for the Bag of Feature Module

Device Utilization Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	6,200	69,120	9%
Number of Slice LUTs	5,504	69,120	7%
Number of fully used LUT-FF pairs	2,900	4,402	65%
Number of Block RAM/FIFO	50	148	16%
Number of BUFG/BUFGCTRLs	2	32	6%

Table 20: Utilized Hardware for the Support Vector Engine

Device Utilization Summary			
Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	117	69,129	1%
Number of Slice LUTs	211	69,129	0%
Number of BUFG/BUFGCTRLs	1	32	3%
Number of DSP48Es	2	64	3%

Table 21: Utilized Hardware for the Support Vector Machine Module

Device Utilization Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	9,646	69,120	13%
Number of Slice LUTs	38,179	69,120	55%
Number of fully used LUT-FF pairs	3,984	43,841	9%
Number of Block RAM/FIFO	60	148	40%
Number of BUFG/BUFGCTRLs	1	32	3%
Number of DSP48Es	52	64	81%

Table 22 summarize the hardware resources used to implement the whole object detection architecture. Our architecture fits in 78% of LUTs and 25% of slice registers in Virtex-5 XC5VLX110T FPGA. It consumed 86% of Block RAM memory because the implemented application required saving a lot of temporary results inside the FPGA chip.

Table 22: Utilized Hardware for the Object Detection Architecture

Device Utilization Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	23,770	69,120	34%
Number of Slice LUTs	59,821	69,120	86%
Number of fully used LUT-FF pairs	10,981	43,841	25%
Number of Block RAM/FIFO	128	148	86%
Number of DSP48Es	64	64	100%

In the SIFT module, the GSS and dominant orientation generation modules consumed most of the hardware recourses. The GSS used 15 blocks of RAM/FIFO to implement the buffer lines that save the input image while the orientation generation module used 27 blocks of RAM/FIFO to save 25 lines of gradient magnitude and 25 lines of gradient orientation.

The BoF module used 50 blocks of RAM/FIFO to save the cluster centres in addition to 60 blocks of RAN/FIFO used by SVM modules to save the SVs and $\alpha \times y$ values. These blocks can be removed to reduce the hardware utilization by using external memory to save these values instead of saving them inside the chip. However, this will increase the processing time.

6.6 Comparison with existing solutions

In this section, a comparison between our implementation and existing solutions is presented. First, we compared our SIFT feature extraction architecture with the work of [8, 11, 13, 14, 23, and 24]. Table 23 summarizes the performance and hardware utilization of each implementation. We also compared the SVM architecture with [26, 27, 28, 29, and 30]. Table 24 summarizes the performance of each SVM architecture.

In our architecture, it takes 480×640 clock cycles to detect all keypoints in the input image. It also takes $(640+17\times17+128)$ clock cycles to generate one descriptor. So that, at 50MHz operational frequency, the time required to scan the input image and detect all keypoints is 6.144 ms and 25.98 μsec to generate each SFIT descriptor. While, in existing solutions it was around 10ms to 30 ms to detect the SIFT keypoints as shown in Table 23. In [23], it needs 80 μ sec to generate one SIFT feature and 11.3 ms in [11].

In terms of hardware utilization, our architecture is considered a lightweight architecture. The LUTs, registers, DSP blocks and BRAM resources for each implementation is summarized in Table 23. Our architecture utilizes less resources than [8, 11, 14, 23, and 24]. It also utilizes almost the same as [13] but our input image resolution is 640×480 while that in [13] is 320×240. In terms of accuracy, we achieved higher accuracy than [23] and [11] which is 97.5%. The work in [8, 13, 14, and 24] did not report the SIFT descriptor accuracy.

Table 23: Comparing proposed SIFT results with existing soultions

	[23]	[11]	[8]	[14]	[24]	[13]	Proposed
Device	XC4VDX35	EP2S60F6	EP2C70F8	EP2C70F8	EP3C120F4	EP2S60F6	XC5VLX110T
LUT	18195	43,366	35,889	32,592	43,563	16,832	16,138
Register	11821	19,100	19529	23,247	14,730	5,729	7,924
DSP blocks	56	64	97	258	45	8	53
BRAM (kbits)	2808	1350	256	891	2810	752	576
Resolution	320 x 256	320 x 240	640 x480	640 x 480	320 x240	320x240	640 x480
Detector	10 ms	33 ms	31 ms	31 ms	30 ms	-	6.144 ms
Descriptor	80 µ sec/ F.	11.7 ms /F	-	-	-		25.9 µ sec/ F
Accuracy	96.90%	95.47%	-	-	-	-	97.535 %

The hardware resources utilized by our implementation was less than most of the existing solutions. Our architecture consumed 16,138 LUTs and 5,729 registers which is less than [8, 11, 14, and 24] and almost the same as [13 and 23]. However, the image resolution in [13 and 23] is 320 ×240, while our architecture works on larger images with a resolution of 640 ×480.

Table 24 summarizes the performance of our SVM architecture against that reported in [26, 27, 28, and 29]. In [27], the SVM architecture achieved an accuracy of 77% with 40 frame per second speed. The architecture is tested using a dataset of faces. In [29], the author implements a SVM classifier and assesses its performance using the MNIST dataset [35]. They achieved a speedup of 20 compared to dual Opteron 2.2 GHz processor CPU.

In [30] the dataset used is so simple, the images are represented with 8 bits grey scale with a resolution of 32×32 pixels. The number of support vectors is limited to 10 per binary classifier. Their architecture has 6 binary classifiers. The classification speed of the system was 2ms. In [28], the authors used 3 classes of Persian handwritten digits to assess their architecture. They achieved a very high accuracy rate and small hardware utilization, but their input vector dimension is limited to only 24.

Table 24: Comparing proposed SVM Results with existing soultions

	[26]	**[29]**	[27]	**[43]**	**Proposed**
Operating frequency	100 MHz	141MHz	151 MHz	30 MHz	50MHz
FPGA	Virtex-5	Virtex-5	Virtex4	Cyclone II	Virtex 5
# of LUTs	57,296	37,549	9,141	14,064	38,179
# of registers	23,220	37067	11,589	-	9,646
# of DSP blocks	83	128	81	20	52
Dataset	Faces dataset [36]	MNIST	Persian handwritten digits	COIL database	Caltech-256
Number of classes	-	-	3	4	5
Input dimensions	-	-	24	1024	500
# SVs	400	-	145	60	100
Classification accuracy	77%	99.11%	98.67%	96%	82%
Processing time	40 frames/sec	Speedup 20x	0.27 ms	2ms	0.25ms

The implementation in [28, 43] used a very simple dataset to achieve a high accuracy. While the implementation in [27] used large hardware resources to reach the processing of 40 frames per second. Finally in [28], the authors used only 24 dimensions for the input vector to reach the 0.27ms. Our implementation used a challenging dataset with 500 dimensional input vectors. All other reviewed solutions used simple datasets. Our architecture accuracy was equal to 82% within a processing time of 0.25 ms for each input image.

In terms of hardware resources utilization, our SVM architecture consumed less resources than [26 and 29]. It consumed only 38,179 LUTS and 9,646 Registers. Our implementation consumed more resources than [27] because the input vector dimensions in [27] is only 24, while in our work it is 500. Also the number of classes in [27] is restricted to 3. The architecture in [43] consumed less resources but the processing time is 10 times greater than our implementation.

Chapter 7: Conclusion and Future Work

Object detection is an important task in computer vision. This thesis addressed the problem of accelerating the object detection to reach real-time performance with a high level of accuracy using a lightweight hardware architecture. In this work, an FPGA-based parallel hardware was implemented to accelerate the computationally intensive algorithms to achieve real-time performance in an embedded system environment. A parallel hardware architecture which implements Scale Invariant Feature Transform (SIFT), Bag of Features (BoF), and Multiclass Support Vector Machine (SVM) was presented.

Experimental results show that the proposed hardware architecture can detect SIFT features from images with a dimension of 640×480 within 6.144 ms. It can also compute up to 1270 SIFT features in each image. The classification accuracy achieved by the architecture on benchmark datasets for five different classes was 85% for Caltech-256, and 78% for KUL Belgium traffic sign dataset. The difference in classification accuracy between the proposed architecture and the software implementation was less than 0.03 %. The speed up achieved in the feature extraction was ×55.06 and ×6.64 in the classification algorithm when compared with a pure software implementation. The architecture FPGA resource utilization was lightweight compared to existing implementations. Hence, the proposed object detection architecture can be used as an embedded system solution to detection and recognize objects in real-time performance.

Although the results presented have demonstrated the effectiveness of our architecture to solve performance problem in the object detection, it could be further improved in a number of ways. First, modify the architecture to read a video stream directly from a camera. Second, further optimization to the SVM architecture are required to further reduce the hardware utilization and increase the number of detected categories. Third, design a high-speed PCB with FPGA device to make the proposed architecture available to many portable applications. Lastly, deploy the implementation in real life sceneries and applications for further enhancement.

References:

[1] H. Xiaoguang, Z. Xinyan, L. Deren and L. Hui, "Traffic Sign Recognition using scale invariant feature transform and SVM," *A special joint symposium of ISPRS Technical Commission IV & AutoCarto,* 2010.

[2] V. Atienza-Vanacloig and J. Rosell-Ortega , "Feature sets for people and luggage recognition in airport surveillance under real-time constraints," *VISIGRAPP08,* pp. 662-665, 2008.

[3] J. Canny, "A Computational Approach to Edge Detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on* , vol. 8, no. 6, pp. 679 - 698 , 1986 .

[4] H. Chris and S. Mike, "A combined corner and edge detector," *Proceedings of the 4th Alvey Vision Conference,* p. 147–151, 1988.

[5] E. Gülch and W. Förstner, "A Fast Operator for Detection and Precise Location of Distinct Points, Corners and Centres of Circular Features," *Proc. ISPRS intercommission conference on fast processing of photogrammetric data.,* pp. 281-305, 1987.

[6] D. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision,* vol. 60, no. 2, pp. 91-110, 2004.

[7] D. Lowe, "Object recognition from local scale-invariant features," *International Conference of Computer Vision,* vol. 2, pp. 1150 - 1157, 1999.

[8] V. Bonato, E. Marques and G. Constantinides, "A Parallel Hardware Architecture for Scale and Rotation Invariant Feature Detection," *IEEE Transactions on Circuits and Systems for Video Technology* , vol. 18, no. 12, pp. 1703-1712, 2008.

[9] R. Hess, "SIFT Feature Detector (Source Code)," [Online]. Available: http://web.engr.oregonstate.edu.

[10] N. Georganas and N. Dardas, "Real-Time Hand Gesture Detection and Recognition Using Bag-of-Features and Support Vector Machine Techniques," *IEEE Transactions on Instrumentation and Measurement,* vol. 60, no. 11, 2011.

[11] W. Feng, D. Zhao, Z. Jiang, Y. Zhu, H. Feng and L. Yao, "An Architecture of Optimised SIFT Feature Detection for an FPGA Implementation of an Image Matcher," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on* , Sydney, NSW , 2009.

[12] Q. Zhang, Y. Chen, Y. Zhang and Y. Xu, "SIFT Implementation and Optimization for Multi-Core Systems," Intel Corporation , 2008.

[13] F. Huang, S. Huang, J. Ker and Y. Chen, "High-Performance SIFT Hardware Accelerator for Real-Time Image Feature Extraction," *IEEE Transactions on Circuits and Systems for Video Technology* , vol. 22, no. 3, 2012.

[14] K. Mizuno, H. Noguchi, G. He, Y. Terachi, T. Kamino, H. Kawaguchi and M. Yoshimoto, "Fast and Low-Memory-Bandwidth Architecture of SIFT Descriptor Generation with Scalability on Speed and Accuracy for VGA Video," in *International Conference on Field Programmable Logic and Applications*, Milano, 2010.

[15] K. Murphy, "Naive Bayes classifiers," [Online]. Available: www.cs.ubc.ca/~murphyk/Teaching/CS340-Fall06/reading/NB.pdf.

[16] S. Delany and P. Cunningham, "k-Nearest Neighbour Classifiers," *Technical Report UCD-CSI-2007-4, Dublin: Artificial Intelligence Group,* 2007.

[17] T. Saegusa, T. Maruyama and Y. Yamaguchi, "How fast is an FPGA in image processing?," *Field Programmable Logic and Applications,* pp. 77 - 82, 2008.

[18] S. Sirowy and A. Forin, "Where's the Beef? Why FPGAs Are So Fast," MSR-TR-2008-130, Redmond, 2008.

[19] H. Bay, T. Tuytelaars and L. Van Gool, "SURF: Speeded Up Robust Features," *Computer Vision,* vol. 3951, pp. 404-417, 2006.

[20] M. Grabner, H. Grabner and H. Bischof, "Fast Approximated SIFT," *Computer Vision* , vol. 3851, pp. 918-927 , 2006.

[21] B. Rister, G. Wang, M. Wu and J. R. Cavallaro, "A Fast and Efficient SIFT Detector Using the Mobile GPU," in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Vancouver, BC , 2013.

[22] C. Jiang, Z.Geng, X. Wei and C. Shen, "SIFT implementation based on GPU," *International Symposium on Photoelectronic Detection and Imaging,* vol. 891304, 2013.

[23] S. Zhong, J. Wang and L. Yan, "A real-time embedded architecture for SIFT," *Journal of Systems Architecture: the EUROMICRO Journal,* vol. 59, no. 1, pp. 16-29 , 2013 .

[24] H. Borhanifar and V. Naeim, "High Speed Object Recognition Based on SIFT Algorithm," *International Conference on Image, Vision and Computing,* 2012.

[25] M. Papadonikolakis and C.-S. Bouganis, "A Novel FPGA-based SVM Classifier," in *International Conference on Field-Programmable Technology (FPT)* , 2010.

[26] C. Kyrkou and T. Theocharides, "A Parallel Hardware Architecture for Real-Time Object Detection with Support Vector Machines," *IEEE Transactions on Computers,* vol. 61, 2012.

[27] D. Mahmoodi, A. Soleimani, H. Khosravi and M. Taghizadeh, "FPGA Simulation of Linear and Nonlinear Support Vector Machine," *Journal of Software Engineering and Applications,* vol. 4, pp. 320-328, 2011.

[28] S. Cadambi, I. Durdanovic, V. Jakkula and M. Sankaradass, "A Massively Parallel FPGA-Based Coprocessor for Support Vector Machines," in *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium*, Napa, CA , 2009.

[29] N. Dardas, Q. Chen and N. D. Georganas, "Hand Gesture Recognition Using Bag-of-Features and Multi-Class Support Vector Machine," in *Haptic Audio-Visual Environments and Games (HAVE), 2010 IEEE International Symposium on*, Phoenix, AZ , 2010.

[30] F. Vedaldi, "VLFeat: An open and portable library of computer vision algorithms," in *Proceedings of the international conference on Multimedia*, 2010.

[31] G.Griffin, "Caltech-256 Object Category Dataset," Technical Report 7694, California Institute, 2007.

[32] E. Lee, "A novel hardware design for SIFT generation with reduced mempry requirement," *Journal of Semiconductor Techonlogy and Science,* vol. 13, no. 2, 2013.

[33] Y. Markus, "Multiplierless multiple constant multiplication," *ACM Trans. Algorithms,* vol. 3, no. 1549-6325, p. 11, 2007.

[34] R. Timofte, "KUL Belgium traffic signs and classification benchmark datasets," [Online]. Available: http://www.vision.ee.ethz.ch/~timofter/. [Accessed 2014].

[35] Y.LeCun, "The MNIST database of handwritten digits," 1988. [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed 2014].

[36] "CBCL Face Database #1: " Jan 2010. [Online]. Available: http://cbcl.mit.edu/software-datasets/FaceData2.html. [Accessed June 2014].

[37] A. Nikitakis, S. Papaioannou and I. Papaefstathiou, "A novel low-power embedded object recognition system working at multi-frames per second," *ACM Transactions on Embedded Computing Systems,* vol. 12, no. 1, p. 20, 2013.

[38] U. Michael Schaeferling, "Object Recognition and Pose Estimation on Embedded Hardware: SURF-Based System Designs Accelerated by FPGA Logic," *International Journal of Reconfigurable Computing,* 2012.

[39] M. Zheng, Z. Song, K. Xu and H. Liu, "Parallelization and Optimization of SIFT Feature Extraction on Cluster System," *International Journal of Computer and Information Engineering,* 2012.

[40] K. Mikolajczyk and C. Schmid, "A Performance Evaluation of Local Descriptors," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 27, no. 10, 2005.

[41] J. Sivic and A. Zisserman, "Video Google: Efficient Visual Search of Videos," *In Toward Category-Level Object Recognition,* vol. 4170, pp. 127-144, 2006.

[42] G. Csurka, C. R. Dance and L. Fan, "Visual Categorization with Bags of Keypoints," *ECCV International Workshop on Statistical Learning in Computer Vision,* 2004.

[43] J. Kim, B. Kim and S. Savarese, "Comparing Image Classification Methods: K-Nearest-Neighbor and Support-Vector-Machines," *American conference on Applied Mathematics,* pp. 133-138, 2012.

[44] C. Corinn and V. Vapnik, "Support-Vector Networks," *Machine Learning,* vol. 20, no. 3, pp. 273-297, 1995.

[45] M. Ciletti, Advanced Digital Design with the Verilog HDL, New Jersey: Prentice Hall PTR, 2003.

[46] M. Ruiz-Llata, G. Guarnizo and M. Yébenes-Calvino , "FPGA Implementation of a Support Vector Machine for Classification and Regression," *Neural Networks (IJCNN), The 2010 International Joint Conference,* pp. 1-5, 2010.

[47] B.Schunck, R. Kasturi and R. Jain, "Object Recognition," in *Machine Vision*, McGraw-Hill, Inc, 1995, pp. 459- 491.

[48] K. Mikolajczyk, "A performance evaluation of local descriptors," *Pattern Analysis and Machine Intelligence, IEEE Transactions on,* vol. 27, no. 10, pp. 1615 - 1630, 2005.

[49] J. Qiu, Y. Lu, T. Huang and T. Ikenaga "A FPGA-Based Real-Time Hardware Accelerator for Orientation Calculation Part in SIFT," in *Intelligent Information Hiding and Multimedia Signal Processing*, Kyoto , 2009.

[50] J. Weston and C. Watkins, "Multi-class support vector machines," in *Proceedings of ESANN99*, Brussels, Belgium, 1999.

[51] M. Papadonikolaki and C.Bouganis, "A Scalable FPGA Architecture for Non-Linear SVM Training," in *ICECE Technology, 2008. FPT 2008. International Conference*, Taipei , 2008.

[52] J. Zhang, M. Marszałek, S. Lazebnik and C. Schmid, "Local Features and Kernels for Classification of Texture and Object Categories: A Comprehensive Study," in *Proc. IEEE Conf. Computer Vision and Pattern*, New York, 2006.

[53] Xilinx, "Xilinx University Program XUPV5-LX110T Development System," 2014. [Online]. Available: http://www.xilinx.com/univ/xupv5-lx110t.htm. [Accessed 2014].

Appendix A: Verilog Modules Header

- **Module** Octave_1(Data_in,Scale1,Scale2,Scale3,Scale4,Scale5,Scale6,clk,valid);
- **Module** DoG_octave1(Data_in,DoG1,DoG2,DoG3,DoG4,DoG5,clk,valid);
- **Module** MBlock0(X,Y1,Y2,Y3,Y4,Y5,Y6);
- **Module** MBlock1(X,Y1,Y2,Y3,Y4,Y5,Y6);
- **Module** MBlock2(X,Y1,Y2,Y3,Y4,Y5,Y6);
- **Module** KP_detection (DoG1, DoG2, DoG3, DoG4, DoG5, KP_out1, KP_out2, KP_out3, valid_out1,valid_out2,valid_out3, clk, enable);
- **Module** OriMag (Data_in, G_mag, G_ori, enable, clk, rst, Valid, X, Y);
- **Module** Domienant_Orientation(Gradient_M, Gradient_O, KP_in, KP_ready, enable, clk, Rst, valid ,Op,Mg ,Og,X ,Y);
- **Module** SIFT_descriptor(X, Y, Op, Og, Mg, element, address, read_en, enable, clk);
- **Module** Rotation_Module(X, Y,Op,Og,Xr,Yr,Or);
- **Module** Gaussian_weight_Generation(Xr,Yr,Wg,clk);
- **Module** Trilinear_IM (Xr, Yr, Or, Mg, Wg, clk, Element, address, read_en, a0, a1, a2, a3,a4,a5,a6,a7,r1,r2,r3,r4,r5,r6,r7,r8, enable,value);
- **Module** EGU(in0, in1, in2, in3, in4, in5, in6, in7, a0, a1, a2, a3, a4, a5, a6, a7, clk, enable, element, address_out, read_en);
- **Module** Angle_conv(Angle_in,Angle_out);
- **Module** Address_block(address_in,address_out,module_out);
- **Module** Memory(data_out, address1, data_in1, write_enable, clk);
- **Module** Decoder_10_deg(Angle_in, Bin_out, clk);
- **Module** Gaussian_WG(X, Y, Wg, clk);
- **Module** SVM_module (X, clk,enable, Class_out, valid, Data_in, a_y);
- **Module** SVM_engine(X, SV, clk, enable, out, valid ,rst);
- **Module** Exp_function (phase_in,x_out ,y_out, rdy, clk, nd);

Appendix B

Table 25: Rotation module results

Software Results			Hardware Results					
Xr	Yr	Or	Xr	Yr	Or	Error_Xr(%)	Error_Yr(%)	Error_Or(%)
-0.9654	0.676	0.6947	-0.96436	0.674805	0.69458	0.10819673	0.176821	0.0172624
-0.9473	0.5734	1.8119	-0.94629	0.572266	1.811768	0.10671778	0.197833	0.00730845
-0.9292	0.4708	2.2243	-0.92822	0.469727	2.224121	0.1051812	0.228003	0.00804326
-0.9111	0.3682	2.389	-0.91016	0.367188	2.388672	0.10358358	0.274986	0.01373483
-0.893	0.2656	2.5381	-0.89209	0.265137	2.537842	0.10192119	0.174428	0.01017309
-0.8749	0.163	2.9235	-0.87402	0.162598	2.92334	0.10019002	0.246837	0.00547824
-0.8569	0.0605	0.4647	-0.85596	0.060059	0.4646	0.1100442	0.729597	0.02160332
-0.8388	-0.0421	1.4559	-0.83789	-0.04248	1.455811	0.10841381	0.903726	0.00614418
-0.8207	-0.1447	1.7207	-0.81982	-0.14453	1.720703	0.1067115	0.116621	0.00018161
-0.8026	-0.2473	1.893	-0.80176	-0.24707	1.892822	0.10493241	0.092878	0.00938903
-0.7845	-0.3499	2.1456	-0.78369	-0.34961	2.145264	0.10307122	0.083059	0.01567525
-0.7664	-0.4525	2.6568	-0.76563	-0.45215	2.656494	0.10112213	0.077693	0.01151232
-0.7483	-0.555	3.483	-0.74756	-0.55469	3.48291	0.09907875	0.056306	0.00257949
-0.7302	-0.6576	0.2479	-0.72949	-0.65674	0.248047	0.09693406	0.13104	0.05924768
-0.7121	-0.7602	0.9119	-0.71143	-0.75928	0.911865	0.09468035	0.12137	0.00381244
-0.6941	-0.8628	1.4589	-0.69336	-0.86182	1.458984	0.10670292	0.114	0.00578347
-0.676	-0.9654	1.7756	-0.67529	-0.96436	1.775391	0.10459042	0.108197	0.01179179
-0.8628	0.6941	7.9029	-0.86182	0.692871	7.902588	0.1140002	0.17705	0.0039493
-0.8447	0.5915	0.6422	-0.84375	0.590332	0.64209	0.11246596	0.197459	0.01715295
-0.8266	0.4889	1.5761	-0.82568	0.487793	1.576172	0.11086454	0.226433	0.00456031
-0.8085	0.3863	1.9819	-0.80762	0.385254	1.981689	0.1091914	0.270798	0.01062349
-0.7904	0.2837	2.1984	-0.78955	0.283203	2.198242	0.10744164	0.175141	0.00717852
-0.7724	0.1811	2.5372	-0.77148	0.180664	2.536865	0.11854285	0.240716	0.01319429
-0.7543	0.0785	0.2886	-0.75342	0.078125	0.288574	0.11693375	0.477707	0.00893321
-0.7362	-0.024	1.5091	-0.73535	-0.02441	1.509033	0.11524552	1.72526	0.00442627
-0.7181	-0.1266	1.7818	-0.71729	-0.12695	1.781494	0.11347218	0.27893	0.01716575
-0.7	-0.2292	1.9872	-0.69922	-0.229	1.987061	0.11160714	0.085556	0.00701757
-0.6819	-0.3318	2.3135	-0.68115	-0.33154	2.313477	0.10964309	0.077466	0.00101308
-0.6638	-0.4344	2.9211	-0.66309	-0.43408	2.920898	0.10757193	0.073197	0.00690023
-0.6457	-0.537	3.6189	-0.64502	-0.53662	3.618652	0.10538466	0.07056	0.00684341
-0.6276	-0.6395	0.0839	-0.62695	-0.63867	0.08374	0.10307122	0.129496	0.19042387
-0.6096	-0.7421	0.5285	-0.60889	-0.74121	0.52832	0.11700808	0.119804	0.03399953
-0.5915	-0.8447	1.0985	-0.59082	-0.84375	1.098389	0.11490913	0.112466	0.01013456
-0.5734	-0.9473	1.5434	-0.57275	-0.94629	1.543213	0.11267767	0.106718	0.01212319
-0.7602	0.7121	3.7099	-0.75928	0.710938	3.709717	0.1213702	0.16325	0.00493822
-0.7421	0.6096	7.987	-0.74121	0.608398	7.986572	0.1198036	0.197107	0.00535538
-0.724	0.507	0.7276	-0.72314	0.505859	0.727539	0.11815867	0.224975	0.00837514

Table 26: SIFT generator Module results vs. Software Implementation

Software Generated SIFT vector			
0.1294	0.9186	1.5588	0.1856
1.1808	109.81	256.33	20.126
0.0019	69.107	195.13	21.735
0	0.2296	1.8075	0.603
0.4214	2.209	2.4242	0.088
3.9134	23.204	24.665	0.3758
0.6189	16.584	55.242	2.0747
0.2614	2.7491	2.6879	0.1054
0.3474	5.7239	6.5752	0.0381
3.3534	25.063	18.027	0.0471
4.7971	27.064	7.0275	0
1.1919	7.4562	0.7461	0
0.0178	2.1195	2.4277	0.0553
8.917	58.558	12.431	0.2374
12.847	84.39	8.0842	0
0.0157	0.3126	0.1719	0
0.0203	0.4536	0.5079	0.1222
11.031	101.09	13.668	0.5061
11.244	114.55	11.275	0
0	0.0045	0.0079	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0.0453	0.0104
0	6.8117	17.924	3.1778
0	5.0237	9.1345	1.8359
0	0.0007	0.051	0.0408

Hardware Generated SIFT vector			
0.1235	0.9053	1.5454	0.1787
1.1694	109.78	255.86	19.92
0.001	69.021	194.85	21.511
0	0.2241	1.7837	0.5854
0.4121	2.1855	2.3994	0.0835
3.8945	23.189	24.639	0.3618
0.5278	16.181	55.137	2.0356
0.1841	2.3755	2.6431	0.0967
0.3423	5.6807	6.5283	0.0371
3.3384	25.057	18.008	0.0454
4.4136	25.417	7.0078	0
0.8613	5.9438	0.7256	0
0.0156	2.0981	2.4053	0.0532
8.8569	58.604	12.403	0.2344
12.757	84.328	8.0659	0
0.0127	0.3022	0.1675	0
0.0171	0.4487	0.5029	0.1187
10.931	101.08	13.631	0.5
11.135	114.36	11.245	0
0	0.0039	0.0078	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0.0449	0.0098
0	6.8223	17.963	3.1528
0	5.0254	9.146	1.8193
0	0	0.0493	0.0391

Table 27: Decoder 36 bins: (10°/ bin) angles

Angle1			Angle2			Bin
Degree	Radian	Hex(3.13)	Degree	Radian	Hex(3.13)	
0	0	'0000'	10	0.174533	'0596'	0
10	0.174533	'0596'	20	0.349066	'0B2C'	1
20	0.349066	'0B2C'	30	0.523599	'10C1'	2
30	0.523599	'10C1'	40	0.698132	'1657'	3
40	0.698132	'1657'	50	0.872665	'1BED'	4
50	0.872665	'1BED'	60	1.047198	'2183'	5
60	1.047198	'2183'	70	1.22173	'2718'	6
70	1.22173	'2718'	80	1.396263	'2CAE'	7
80	1.396263	'2CAE'	90	1.570796	'3244'	8
90	1.570796	'3244'	100	1.745329	'37DA'	9
100	1.745329	'37DA'	110	1.919862	'3D70'	10
110	1.919862	'3D70'	120	2.094395	'4305'	11
120	2.094395	'4305'	130	2.268928	'489B'	12
130	2.268928	'489B'	140	2.443461	'4E31'	13
140	2.443461	'4E31'	150	2.617994	'53C7'	14
150	2.617994	'53C7'	160	2.792527	'595C'	15
160	2.792527	'595C'	170	2.96706	'5EF2'	16
170	2.96706	'5EF2'	180	3.141593	'6488'	17
180	3.141593	'6488'	190	3.316126	'A10E'	18
190	3.316126	'A10E'	200	3.490659	'A6A4'	19
200	3.490659	'A6A4'	210	3.665191	'AC39'	20
210	3.665191	'AC39'	220	3.839724	'B1CF'	21
220	3.839724	'B1CF'	230	4.014257	'B765'	22
230	4.014257	'B765'	240	4.18879	'BCFB'	23
240	4.18879	'BCFB'	250	4.363323	'C290'	24
250	4.363323	'C290'	260	4.537856	'C826'	25
260	4.537856	'C826'	270	4.712389	'CDBC'	26
270	4.712389	'CDBC'	280	4.886922	'D352'	27
280	4.886922	'D352'	290	5.061455	'D8E8'	28
290	5.061455	'D8E8'	300	5.235988	'DE7D'	29
300	5.235988	'DE7D'	310	5.410521	'E413'	30
310	5.410521	'E413'	320	5.585054	'E9A9'	31
320	5.585054	'E9A9'	330	5.759587	'EF3F'	32
330	5.759587	'EF3F'	340	5.934119	'F4D4'	33
340	5.934119	'F4D4'	350	6.108652	'FA6A'	34
350	6.108652	'FA6A'	360	6.2832	'0000'	35

Appendix C: KNN Classifier

K-NN Classifier	Caltech 256 Database		
Codewords=100	K=5	Training size =50	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	66.33	47.79	90.93	79.20
Horse	13.86	43.75	81.41	79.00
Motorbikes	17.59	46.34	80.61	77.80
Faces	80.41	27.66	91.28	55.40
Watch	4.17	44.44	81.26	80.60
Testing Time (sec) =	0.074 s	Total Accuracy (%)	74.40	

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	65	0	2	31	0	33
Faces	36	14	3	46	2	87
Motorbikes	21	2	19	64	2	89
Horse	10	6	2	78	1	19
Watch	4	10	15	63	4	92
False negative	71	18	22	204	5	
True negative	331	381	370	199	399	

K-NN Classifier	Caltech 256 Database		
Codewords=100	K=5	Training size = 100	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	81.05	67.54	95.34	89.00
Horse	62.75	58.18	90.26	83.20
Motorbikes	12.84	56.00	80.00	78.80
Faces	68.32	62.73	91.79	85.40
Watch	75.27	49.65	93.59	81.20
Testing Time (sec) =	0.091 s	Total Accuracy (%)	83.52	

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	77	2	5	8	3	18
Faces	8	64	3	11	16	38
Motorbikes	22	23	14	13	37	95
Horse	3	13	1	69	15	32
Watch	4	8	2	9	70	23
False negative	37	46	11	41	71	
True negative	368	352	380	358	336	

K-NN Classifier	Caltech 256 Database		
Codewords=100	K=5	Training size = 200	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	85.71	72.00	97.00	92.00
Horse	64.58	60.78	91.46	85.20
Motorbikes	62.50	61.90	90.13	84.20
Faces	58.56	80.25	89.02	87.60
Watch	69.52	65.18	91.75	85.80
Testing Time (sec) =	0.112 s	Total Accuracy (%)	86.96	

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	72	5	4	1	2	12
Faces	5	62	10	7	12	34
Motorbikes	9	20	65	5	5	39
Horse	5	8	13	65	20	46
Watch	9	7	13	3	73	32
False negative	28	40	40	16	39	
True negative	388	364	356	373	356	

K-NN Classifier	Caltech 256 Database		
Codewords=100	K=5	Training size = 300	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	82.14	75.82	96.33	92.60
Horse	55.24	66.67	88.62	84.80
Motorbikes	56.44	67.06	89.42	85.80
Faces	68.00	64.15	91.88	86.00
Watch	80.00	66.67	94.02	86.80
Testing Time (sec)=	0.153 s	Total Accuracy (%)	87.20	

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	69	2	4	7	2	15
Faces	5	58	14	15	13	47
Motorbikes	9	11	57	7	17	44
Horse	4	11	5	68	12	32
Watch	4	5	4	9	88	22
False negative	22	29	27	38	44	
True negative	394	366	372	362	346	

K-NN Classifier	Caltech 256 Database		
Codewords=100	K=5	Training size = 400	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	86.46	78.30	96.70	92.80
Horse	78.13	66.37	94.57	88.20
Motorbikes	63.46	68.75	90.59	86.40
Faces	61.22	70.59	90.84	87.40
Watch	78.30	83.00	94.25	92.00
Testing Time (sec) =	0.167 s	Total Accuracy (%)	89.36	

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	83	1	1	9	2	13
Faces	4	75	5	2	10	21
Motorbikes	12	14	66	8	4	38
Horse	3	18	16	60	1	38
Watch	4	5	8	6	83	23
False negative	23	38	30	25	17	
True negative	381	366	366	377	377	

K-NN Classifier	Caltech 256 Database		
Codewords=100	K=5	Training size = 500	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	87.00	80.56	96.68	93.20
Faces	72.04	75.28	93.67	90.40
Motorbikes	66.67	73.47	91.04	87.60
Horse	78.00	72.22	94.39	89.60
Watch	68.69	70.10	92.31	88.00
Testing Time (sec) =	0.194 s	Total Accuracy (%)	89.76	

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	87	0	2	7	4	13
Faces	4	67	7	2	13	26
Motorbikes	11	11	72	8	6	36
Horse	2	5	9	78	6	22
Watch	4	6	8	13	68	31
False negative	21	22	26	30	29	
True negative	379	385	366	370	372	

- <u>Naïve Bayes classifier</u>

Naive Bayes Classifier	Caltech 256 Database	
Codewords=100		Training size = 50

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	26.53	45.61	83.75	79.40
Horse	46.53	49.47	86.67	79.60
Motorbikes	65.74	34.98	87.54	66.20
Faces	49.48	52.17	87.99	81.40
Watch	27.08	49.06	84.34	80.60
Testing Time =	0.012 s	Total Accuracy		77.44

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	26	17	32	12	11	72
Faces	8	47	21	17	8	54
Motorbikes	18	9	71	7	3	37
Horse	2	16	26	48	5	49
Watch	3	6	53	8	26	70
False negative	31	48	132	44	27	
True negative	371	351	260	359	377	

Naive Bayes Classifier	Caltech 256 Database	
Codewords=100		Training size = 100

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	36.84	74.47	86.75	85.60
Horse	44.12	68.18	86.87	84.40
Motorbikes	14.68	53.33	80.21	78.60
Faces	45.54	58.23	86.94	82.40
Watch	93.55	31.29	97.30	60.60
Testing Time =	0.009 s	Total Accuracy		78.32

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	35	8	1	10	41	60
Faces	6	45	2	13	36	57
Motorbikes	3	7	16	6	77	93
Horse	1	6	11	46	37	55
Watch	2	0	0	4	87	6
False negative	12	21	14	33	191	
True negative	393	377	377	366	216	

Naive Bayes Classifier	Caltech 256 Database	
Codewords=100		Training size = 200

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	29.76	51.02	86.92	83.40
Horse	33.33	61.54	85.71	83.20
Motorbikes	73.08	30.04	88.66	59.00
Faces	43.24	55.17	84.75	79.60
Watch	14.29	25.42	79.59	73.20
Testing Time =	0.009 s	Total Accuracy		75.68

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	25	5	31	9	14	59
Faces	8	32	25	16	15	64
Motorbikes	7	10	76	8	3	28
Horse	8	5	38	48	12	63
Watch	1	0	83	6	15	90
False negative	24	20	177	39	44	
True negative	392	384	219	350	351	

Naive Bayes Classifier	Caltech 256 Database	
Codewords=100		Training size = 300

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	36.90	53.45	88.01	84.00
Horse	40.95	55.84	85.34	80.80
Motorbikes	9.90	22.73	80.04	75.00
Faces	39.00	55.71	85.81	81.60
Watch	80.00	35.06	91.16	63.00
Testing Time =	0.008 s	Total Accuracy		76.88

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	31	10	13	3	27	53
Faces	9	43	8	7	38	62
Motorbikes	2	16	10	9	64	91
Horse	10	5	12	39	34	61
Watch	6	3	1	12	88	22
False negative	27	34	34	31	163	
True negative	389	361	365	369	227	

Naive Bayes Classifier	Caltech 256 Database	
Codewords=100		Training size = 400

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	30.21	47.54	84.74	80.20
Horse	40.63	45.88	86.27	79.40
Motorbikes	7.69	50.00	80.17	79.20
Faces	33.67	64.71	85.52	83.40
Watch	90.57	33.45	95.31	59.80
Testing Time =	0.014 s	Total Accuracy		76.40

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	29	8	1	8	50	67
Faces	20	39	1	1	35	57
Motorbikes	8	14	8	7	67	96
Horse	2	18	6	33	39	65
Watch	2	6	0	2	96	10
False negative	32	46	8	18	191	
True negative	372	358	388	384	203	

Naive Bayes Classifier	Caltech 256 Database	
Codewords=100		Training size = 500

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	39.00	54.93	85.78	81.40
Horse	31.18	55.77	85.71	82.60
Motorbikes	11.11	57.14	79.96	79.00
Faces	45.00	61.64	87.12	83.40
Watch	91.92	32.16	96.31	60.00
Testing Time =	0.011 s	Total Accuracy		77.28

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	39	4	1	10	46	61
Faces	17	29	4	8	35	64
Motorbikes	10	10	12	6	70	96
Horse	5	6	3	45	41	55
Watch	0	3	1	4	91	8
False negative	32	23	9	28	192	
True negative	368	384	383	372	209	

- <u>Ada boost classifier:</u>

Ada Boost Classifier		Caltech 256 Database		
Codewords=100			Training size = 50	
	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	60.20	54.63	90.05	82.40
Horse	39.60	64.52	86.07	83.40
Motorbikes	34.26	49.33	83.29	78.20
Faces	81.44	47.59	94.61	79.00
Watch	44.79	48.31	87.10	80.20
Testing Time =		0.003 s	Total Accuracy	80.64

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	59	4	8	13	14	39
Faces	9	40	10	32	10	61
Motorbikes	23	8	37	20	20	71
Horse	2	7	7	79	2	18
Watch	15	3	13	22	43	53
False negative	49	22	38	87	46	
True negative	353	377	354	316	358	

Ada Boost Classifier		Caltech 256 Database		
Codewords=100			Training size = 100	
	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	73.68	64.81	93.62	87.40
Horse	58.82	67.42	89.78	85.80
Motorbikes	29.36	60.38	82.77	80.40
Faces	71.29	60.00	92.37	84.60
Watch	58.06	41.54	89.46	77.00
Testing Time =		0.003 s	Total Accuracy	83.04

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	70	1	2	9	13	25
Faces	9	60	5	10	18	42
Motorbikes	14	17	32	14	32	77
Horse	9	5	2	72	13	29
Watch	6	6	12	15	54	39
False negative	38	29	21	48	76	
True negative	367	369	370	351	331	

Ada Boost Classifier		Caltech 256 Database		
Codewords=100			Training size = 200	
	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	86.90	79.35	97.30	94.00
Horse	69.79	68.37	92.79	88.00
Motorbikes	54.81	57.00	88.25	82.00
Faces	69.37	73.33	91.39	87.60
Watch	61.90	61.90	89.87	84.00
Testing Time =		0.003 s	Total Accuracy	87.12

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	73	3	3	4	1	11
Faces	3	67	10	4	12	29
Motorbikes	3	13	57	15	16	47
Horse	6	5	12	77	11	34
Watch	7	10	18	5	65	40
False negative	19	31	43	28	40	
True negative	397	373	353	361	355	

Ada Boost Classifier		Caltech 256 Database		
Codewords=100			Training size = 300	
	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	78.57	80.49	95.69	93.20
Horse	60.95	74.42	90.10	87.40
Motorbikes	65.35	55.93	90.84	82.60
Faces	73.00	61.86	92.93	85.60
Watch	60.00	68.75	89.11	85.20
Testing Time =		0.003 s	Total Accuracy	86.80

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	66	1	7	9	1	18
Faces	8	64	10	11	12	41
Motorbikes	4	6	66	13	12	35
Horse	2	6	14	73	5	27
Watch	2	9	21	12	66	44
False negative	16	22	52	45	30	
True negative	400	373	347	355	360	

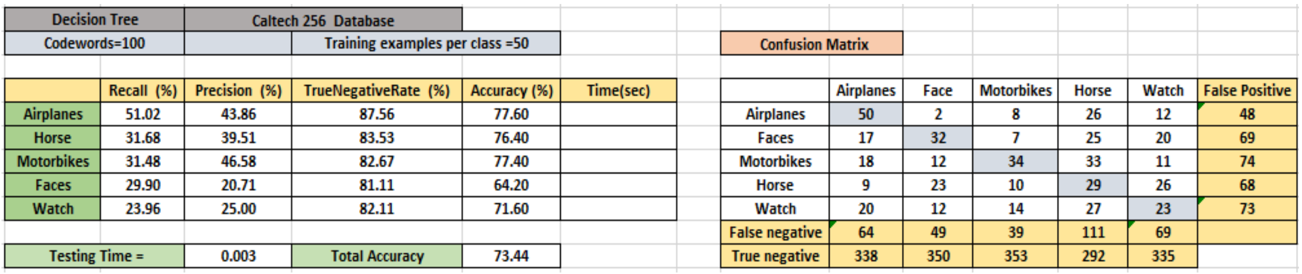
Ada Boost Classifier		Caltech 256 Database		
Codewords=100			Training size = 400	
	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	76.04	85.88	94.46	93.00
Horse	80.21	70.64	95.14	89.80
Motorbikes	61.54	69.57	90.20	86.40
Faces	75.51	67.89	93.86	88.20
Watch	69.81	70.48	91.90	87.40
Testing Time =		0.003 s	Total Accuracy	88.96

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	73	5	3	10	5	23
Faces	2	77	3	5	9	19
Motorbikes	5	8	64	13	14	40
Horse	3	10	8	74	3	24
Watch	2	9	14	7	74	32
False negative	12	32	28	35	31	
True negative	392	372	368	367	363	

Ada Boost Classifier		Caltech 256 Database		
Codewords=100			Training size = 500	
	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	88.00	83.02	96.95	94.00
Horse	76.34	75.53	94.58	91.00
Motorbikes	63.89	71.88	90.35	86.80
Faces	78.00	77.23	94.49	91.00
Watch	71.72	68.93	92.95	88.00
Testing Time =		0.087 s	Total Accuracy	90.16

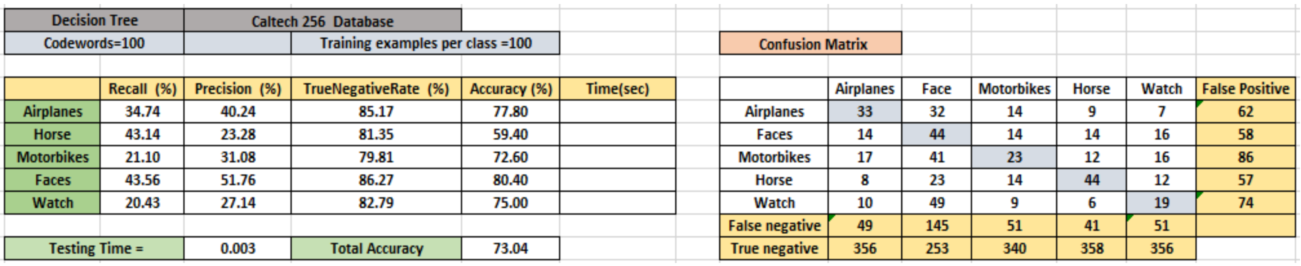
Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	88	2	0	6	4	12
Faces	3	71	6	2	11	22
Motorbikes	6	9	69	10	14	39
Horse	4	6	9	78	3	22
Watch	5	6	12	5	71	28
False negative	18	23	27	23	32	
True negative	382	384	365	377	369	

- Decision Tree classifier:



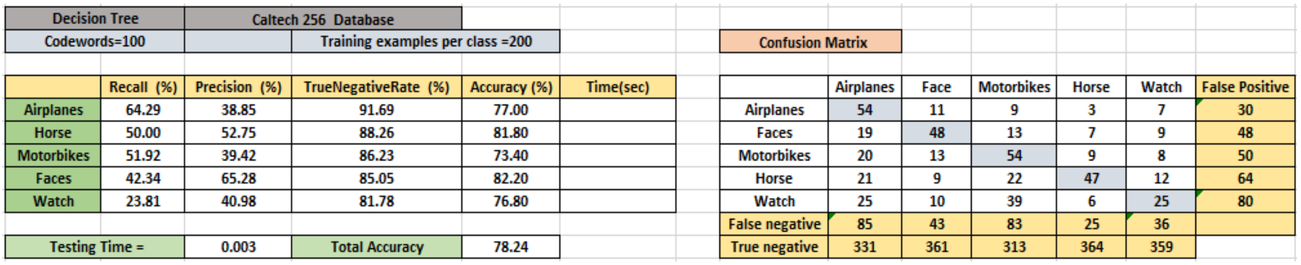
Decision Tree	Caltech 256 Database			
Codewords=100		Training examples per class =50		
	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)	Time(sec)
Airplanes	51.02	43.86	87.56	77.60	
Horse	31.68	39.51	83.53	76.40	
Motorbikes	31.48	46.58	82.67	77.40	
Faces	29.90	20.71	81.11	64.20	
Watch	23.96	25.00	82.11	71.60	
Testing Time =	0.003	Total Accuracy	73.44		

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	50	2	8	26	12	48
Faces	17	32	7	25	20	69
Motorbikes	18	12	34	33	11	74
Horse	9	23	10	29	26	68
Watch	20	12	14	27	23	73
False negative	64	49	39	111	69	
True negative	338	350	353	292	335	



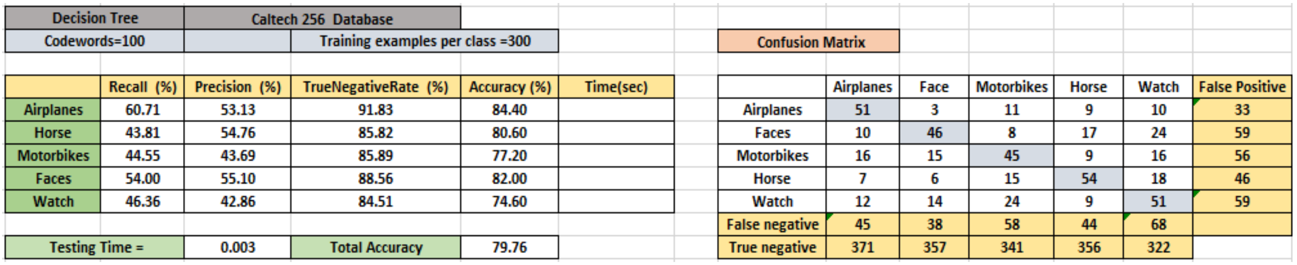
Decision Tree	Caltech 256 Database				
Codewords=100		Training examples per class =100			
	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)	Time(sec)
Airplanes	34.74	40.24	85.17	77.80	
Horse	43.14	23.28	81.35	59.40	
Motorbikes	21.10	31.08	79.81	72.60	
Faces	43.56	51.76	86.27	80.40	
Watch	20.43	27.14	82.79	75.00	
Testing Time =	0.003	Total Accuracy	73.04		

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	33	32	14	9	7	62
Faces	14	44	14	14	16	58
Motorbikes	17	41	23	12	16	86
Horse	8	23	14	44	12	57
Watch	10	49	9	6	19	74
False negative	49	145	51	41	51	
True negative	356	253	340	358	356	



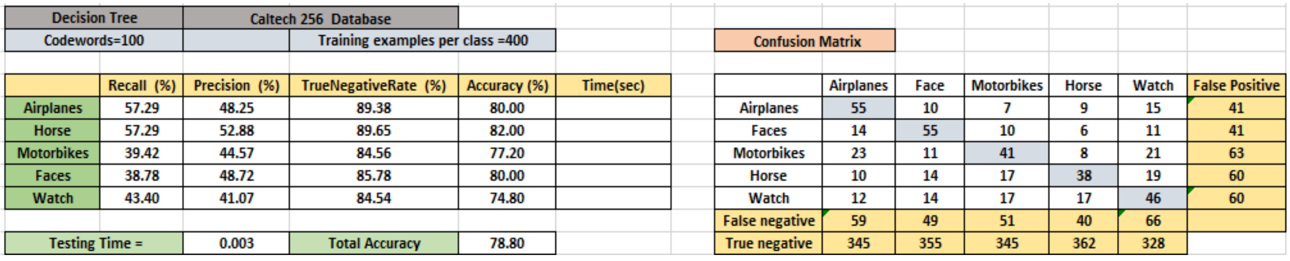
Decision Tree	Caltech 256 Database				
Codewords=100		Training examples per class =200			
	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)	Time(sec)
Airplanes	64.29	38.85	91.69	77.00	
Horse	50.00	52.75	88.26	81.80	
Motorbikes	51.92	39.42	86.23	73.40	
Faces	42.34	65.28	85.05	82.20	
Watch	23.81	40.98	81.78	76.80	
Testing Time =	0.003	Total Accuracy	78.24		

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	54	11	9	3	7	30
Faces	19	48	13	7	9	48
Motorbikes	20	13	54	9	8	50
Horse	21	9	22	47	12	64
Watch	25	10	39	6	25	80
False negative	85	43	83	25	36	
True negative	331	361	313	364	359	



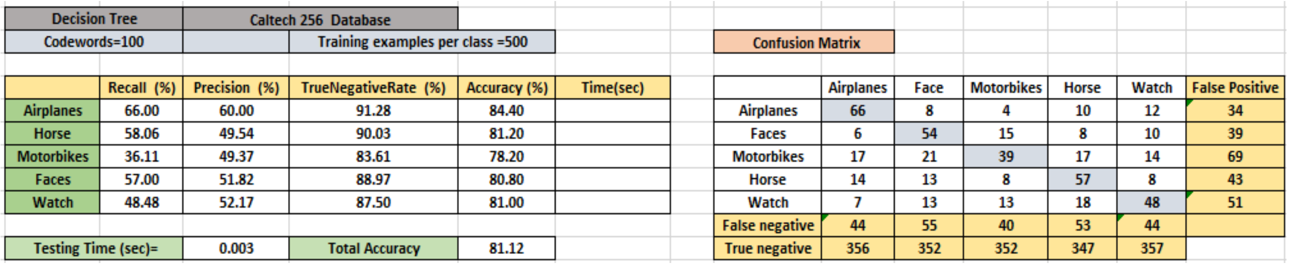
Decision Tree	Caltech 256 Database				
Codewords=100		Training examples per class =300			
	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)	Time(sec)
Airplanes	60.71	53.13	91.83	84.40	
Horse	43.81	54.76	85.82	80.60	
Motorbikes	44.55	43.69	85.89	77.20	
Faces	54.00	55.10	88.56	82.00	
Watch	46.36	42.86	84.51	74.60	
Testing Time =	0.003	Total Accuracy	79.76		

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	51	3	11	9	10	33
Faces	10	46	8	17	24	59
Motorbikes	16	15	45	9	16	56
Horse	7	6	15	54	18	46
Watch	12	14	24	9	51	59
False negative	45	38	58	44	68	
True negative	371	357	341	356	322	



Decision Tree	Caltech 256 Database				
Codewords=100		Training examples per class =400			
	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)	Time(sec)
Airplanes	57.29	48.25	89.38	80.00	
Horse	57.29	52.88	89.65	82.00	
Motorbikes	39.42	44.57	84.56	77.20	
Faces	38.78	48.72	85.78	80.00	
Watch	43.40	41.07	84.54	74.80	
Testing Time =	0.003	Total Accuracy	78.80		

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	55	10	7	9	15	41
Faces	14	55	10	6	11	41
Motorbikes	23	11	41	8	21	63
Horse	10	14	17	38	19	60
Watch	12	14	17	17	46	60
False negative	59	49	51	40	66	
True negative	345	355	345	362	328	



Decision Tree	Caltech 256 Database				
Codewords=100		Training examples per class =500			
	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)	Time(sec)
Airplanes	66.00	60.00	91.28	84.40	
Horse	58.06	49.54	90.03	81.20	
Motorbikes	36.11	49.37	83.61	78.20	
Faces	57.00	51.82	88.97	80.80	
Watch	48.48	52.17	87.50	81.00	
Testing Time (sec)=	0.003	Total Accuracy	81.12		

Confusion Matrix						
	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	66	8	4	10	12	34
Faces	6	54	15	8	10	39
Motorbikes	17	21	39	17	14	69
Horse	14	13	8	57	8	43
Watch	7	13	13	18	48	51
False negative	44	55	40	53	44	
True negative	356	352	352	347	357	

- Linear Support vector machine (SVM):

Linear SVM Classifier	Caltech 256 Database		
Codewords=100		Training size = 50	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	63.27	73.81	91.35	88.40
Horse	13.86	45.16	81.45	79.20
Motorbikes	27.78	68.18	82.89	81.60
Faces	81.44	34.50	93.36	66.40
Watch	50.00	42.86	87.63	77.60
Testing Time =	0.020 s		Total Accuracy	78.64

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	62	3	2	19	12	36
Faces	5	14	3	60	19	87
Motorbikes	7	6	30	38	27	78
Horse	3	5	4	79	6	18
Watch	7	3	5	33	48	48
False negative	22	17	14	150	64	
True negative	380	382	378	253	340	

Linear SVM Classifier	Caltech 256 Database		
Codewords=100		Training size = 100	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	83.16	66.39	95.80	88.80
Horse	61.76	65.63	90.35	85.60
Motorbikes	3.67	80.00	78.79	78.80
Faces	58.42	70.24	89.90	86.60
Watch	89.25	42.35	96.71	75.40
Testing Time =	0.032 s		Total Accuracy	83.04

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	79	3	0	1	12	16
Faces	6	63	1	13	19	39
Motorbikes	17	11	4	10	67	105
Horse	12	15	0	59	15	42
Watch	5	4	0	1	83	10
False negative	40	33	1	25	113	
True negative	365	365	390	374	294	

Linear SVM Classifier	Caltech 256 Database		
Codewords=100		Training size = 200	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	85.71	73.47	97.01	92.40
Horse	54.17	64.20	89.50	85.40
Motorbikes	65.38	49.28	90.06	78.80
Faces	60.36	72.04	89.19	86.00
Watch	50.48	58.89	87.32	82.20
Testing Time =	0.049 s		Total Accuracy	84.96

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	72	4	4	1	3	12
Faces	7	52	13	12	12	44
Motorbikes	6	8	68	10	12	36
Horse	9	9	16	67	10	44
Watch	4	8	37	3	53	52
False negative	26	29	70	26	37	
True negative	390	375	326	363	358	

Linear SVM Classifier	Caltech 256 Database		
Codewords=100		Training size = 300	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	85.71	81.82	97.09	94.40
Horse	53.33	80.00	88.60	87.40
Motorbikes	62.38	56.25	90.21	82.60
Faces	68.00	63.55	91.86	85.80
Watch	65.45	58.54	89.92	82.20
Testing Time =	0.068 s		Total Accuracy	86.48

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	72	2	5	1	4	12
Faces	3	56	10	15	21	49
Motorbikes	4	1	63	13	20	38
Horse	6	7	13	68	6	32
Watch	3	4	21	10	72	38
False negative	16	14	49	39	51	
True negative	400	381	350	361	339	

Linear SVM Classifier	Caltech 256 Database		
Codewords=100		Training size = 400	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	86.46	88.30	96.80	95.20
Horse	72.92	66.67	93.42	87.80
Motorbikes	66.35	62.16	91.00	84.60
Faces	61.22	82.19	91.10	89.80
Watch	69.81	63.25	91.64	85.00
Testing Time =	0.086 s		Total Accuracy	88.48

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	83	2	4	3	4	13
Faces	2	70	5	3	16	26
Motorbikes	4	7	69	5	19	35
Horse	4	14	16	60	4	38
Watch	1	12	17	2	74	32
False negative	11	35	42	13	43	
True negative	393	369	354	389	351	

Linear SVM Classifier	Caltech 256 Database		
Codewords=100		Training size = 500	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	88.00	94.62	97.05	96.60
Horse	62.37	69.88	91.61	88.00
Motorbikes	72.22	67.83	92.21	86.60
Faces	76.00	76.77	94.01	90.60
Watch	69.70	62.73	92.31	85.80
Testing Time =	0.108 s		Total Accuracy	89.52

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	88	0	2	6	4	12
Faces	2	58	7	5	21	35
Motorbikes	2	9	78	8	11	30
Horse	1	10	8	76	5	24
Watch	0	6	20	4	69	30
False negative	5	25	37	23	41	
True negative	395	382	355	377	360	

- <u>Non-linear Support vector machine (SVM):</u>

Non-Linear SVM (RBF)	Caltech 256 Database		
Codewords=100		Training size = 50	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	81.63	60.15	95.10	85.80
Horse	39.60	36.04	84.32	73.60
Motorbikes	6.48	53.85	79.26	78.60
Faces	71.13	39.43	91.38	73.20
Watch	37.50	52.94	86.11	81.60
Testing Time =	0.031 s	Total Accuracy	78.56	

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	80	14	0	3	1	18
Faces	20	40	2	31	8	61
Motorbikes	24	16	7	41	20	101
Horse	5	20	0	69	3	28
Watch	4	21	4	31	36	60
False negative	53	71	6	106	32	
True negative	349	328	386	297	372	

Non-Linear SVM (RBF)	Caltech 256 Database		
Codewords=100		Training size = 100	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	82.11	82.11	95.80	93.20
Horse	66.67	59.65	91.19	84.00
Motorbikes	23.85	81.25	82.26	82.20
Faces	79.21	55.17	94.08	82.80
Watch	69.89	57.02	92.75	84.60
Testing Time =	0.050 s	Total Accuracy	85.36	

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	78	3	0	11	3	17
Faces	0	68	3	23	8	34
Motorbikes	11	29	26	14	29	83
Horse	3	8	1	80	9	21
Watch	3	6	2	17	65	28
False negative	17	46	6	65	49	
True negative	388	352	385	334	358	

Non-Linear SVM (RBF)	Caltech 256 Database		
Codewords=100		Training size = 200	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	91.67	82.80	98.28	95.40
Horse	65.63	68.48	91.91	87.60
Motorbikes	59.62	65.26	89.63	85.00
Faces	85.59	70.37	95.62	88.80
Watch	60.00	74.12	89.88	87.20
Testing Time =	0.084 s	Total Accuracy	88.80	

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	77	3	1	3	0	7
Faces	1	63	4	16	12	33
Motorbikes	9	11	62	14	8	42
Horse	2	4	8	95	2	16
Watch	4	11	20	7	63	42
False negative	16	29	33	40	22	
True negative	400	375	363	349	373	

Non-Linear SVM (RBF)	Caltech 256 Database		
Codewords=100		Training size = 300	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	89.29	93.75	97.86	97.20
Horse	67.62	78.02	91.69	89.20
Motorbikes	70.30	71.72	92.52	88.40
Faces	85.00	67.46	95.99	88.80
Watch	70.91	75.00	91.92	88.40
Testing Time =	0.110 s	Total Accuracy	90.40	

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	75	2	3	4	0	9
Faces	0	71	10	15	9	34
Motorbikes	1	9	71	9	11	30
Horse	2	3	4	85	6	15
Watch	2	6	11	13	78	32
False negative	5	20	28	41	26	
True negative	411	375	371	359	364	

Non-Linear SVM (RBF)	Caltech 256 Database		
Codewords=100		Training size = 400	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	86.46	89.25	96.81	95.40
Horse	85.42	62.12	96.20	87.20
Motorbikes	65.38	77.27	91.26	88.80
Faces	68.37	79.76	92.55	90.40
Watch	78.30	80.58	94.21	91.40
Testing Time =	0.131 s	Total Accuracy	90.64	

Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	83	5	0	6	2	13
Faces	0	82	6	1	7	14
Motorbikes	6	15	68	5	10	36
Horse	3	21	6	67	1	31
Watch	1	9	8	5	83	23
False negative	10	50	20	17	20	
True negative	394	354	376	385	374	

Non-Linear SVM (RBF)	Caltech 256 Database		
Codewords=100		Training size = 500	

	Recall (%)	Precision (%)	TrueNegativeRate (%)	Accuracy (%)
Airplanes	91.00	93.81	97.77	97.00
Horse	83.87	78.00	96.25	92.60
Motorbikes	72.22	82.11	92.59	90.60
Faces	81.00	76.42	95.18	91.20
Watch	79.80	77.45	94.97	91.40
Testing Time =	0.169 s	Total Accuracy	92.56	

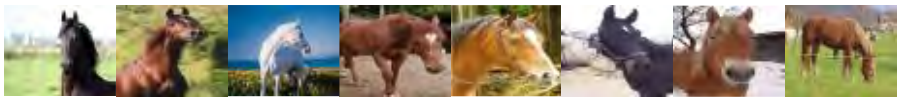
Confusion Matrix

	Airplanes	Face	Motorbikes	Horse	Watch	False Positive
Airplanes	91	1	0	6	2	9
Faces	1	78	4	3	7	15
Motorbikes	3	8	78	11	8	30
Horse	1	9	3	81	6	19
Watch	1	4	10	5	79	20
False negative	6	22	17	25	23	
True negative	394	385	375	375	378	

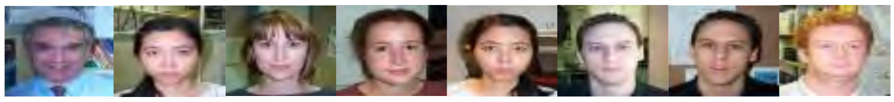
Appendix D: Ten 5-classes Subsets from Calthech-256

Subset #1:

Horses

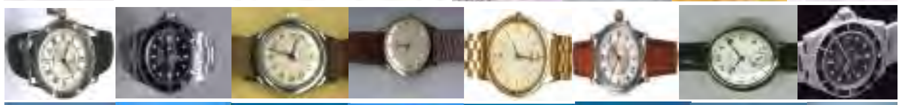


Face

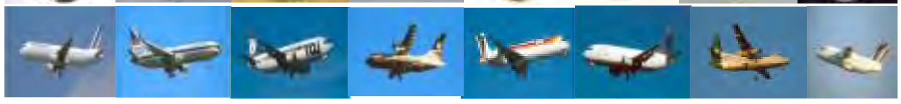


Motorbike

Watch



Airplane



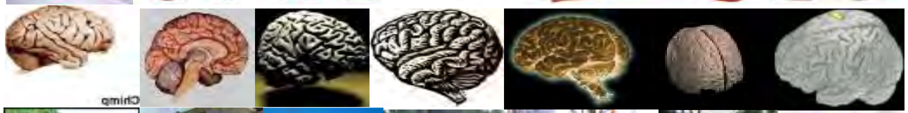
Subset #2:

blimp

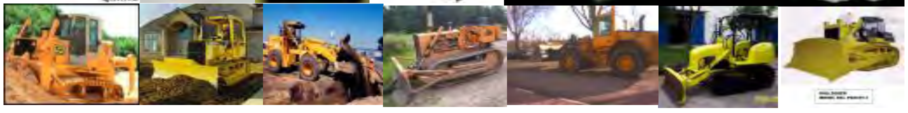
bowling-pin

boxing-glove

brain



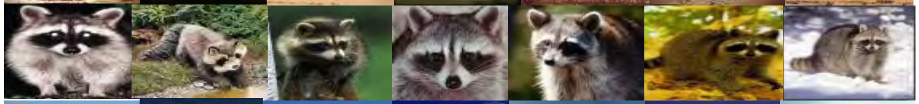
bulldozer



Subset #3:

pyramid

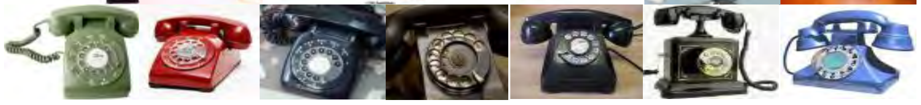
raccoon



radio-telescope

revolver-101

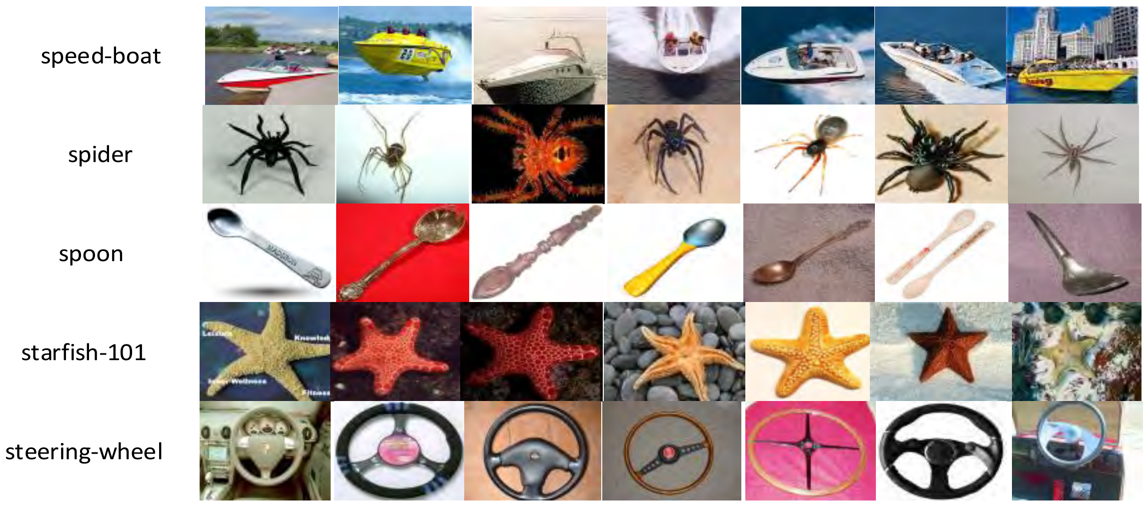
rotary-phone



Subset #4:

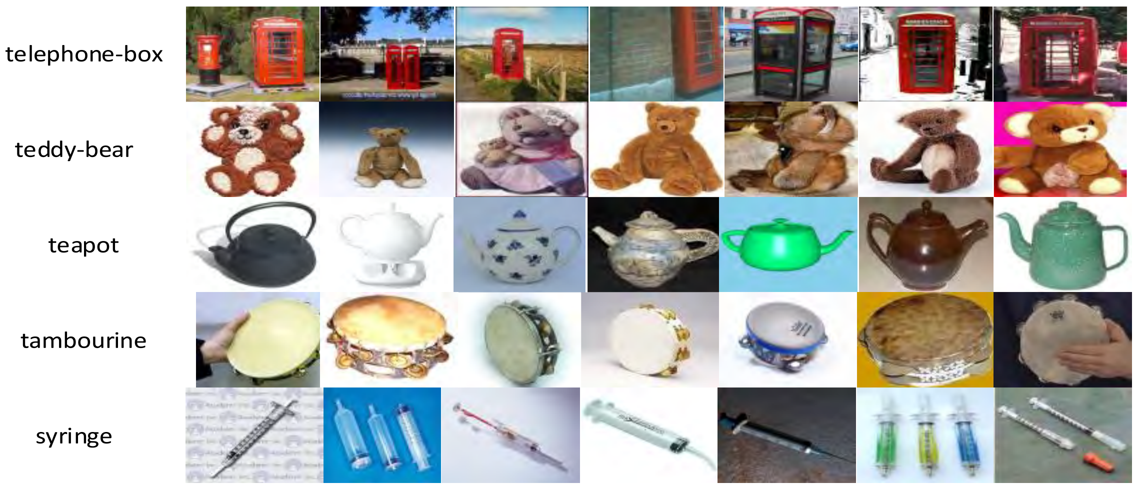
Subset #5:

Subset #6:



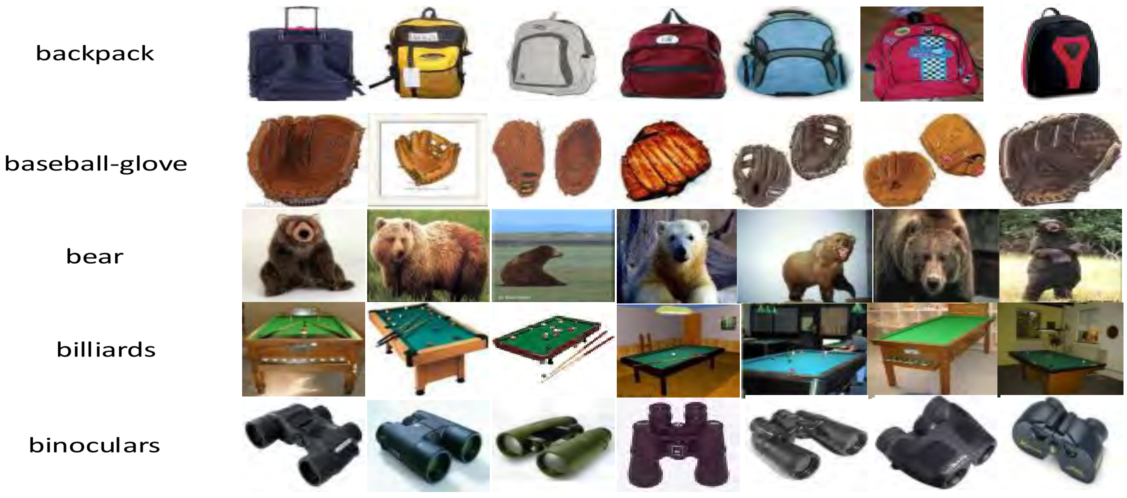
Subset #7:

Subset #8:



Subset #9:

Subset #10:



Vita

Murad Mohammad Qasaimeh was born on March 28, 1988, in Irbid, Jordan. He was educated and graduated from Irbid Secondary School, Irbid, in 2006. After that, he attended Jordan University of Science and Technology in Irbid, Jordan, from which he graduated in 2011 and obtained his Bachelor of Science in Computer Engineering with an excellent rating. In Spring 2012, Mr.Qasaimeh began his Master's program in Computer Engineering at the American University of Sharjah. His research interest includes real-time embedded systems, reconfigurable computing, and computer vision systems.