FPGA-based Parallel Hardware Architecture for Real-Time Image Classification

Murad Qasaimeh, Assim Sagahyroon, and Tamer Shanableh

Abstract- **This paper proposes a parallel hardware architecture for real-time image classification based on Scale Invariant Feature Transform (SIFT), Bag of Features (BoF) and Support Vector Machine (SVM) algorithms. The proposed architecture exploits different forms of parallelism in these algorithms in order to accelerate their execution to achieve real-time performance. Different techniques have been used to parallelize the execution and reduce the hardware resource utilization of the computationally intensive steps in these algorithms. The architecture takes a 640×480 pixel image as an input and classifies it based on its content within 33ms. A prototype of the proposed architecture is implemented on an FPGA platform and evaluated using two benchmark datasets, namely; Caltech-256 and the Belgium Traffic Sign datasets. The architecture is able to detect up to 1270 SIFT features per frame with an increment of 380 extra features from the best recent implementation. We were able to speed up the feature extraction algorithm when compared to an equivalent software implementation by ×54 and for classification algorithm by ×6, while maintaining the difference in classification accuracy within 3%. The hardware resources utilized by our architecture were also less than those used by other existing solutions.**

Index Terms— **Image classification, hardware implementation, field-programmable gate array, Scale Invariant Feature Transform**

I. INTRODUCTION

Object detection and classification is the process of finding 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. Feature-based object classification is a common image classification method in computer vision. It typically uses one of the feature extraction algorithms to extract the image's important content. Then, it uses one of the classification algorithms to label images into one of the trained categories. Image classification has many potential applications including autonomous robots, intelligent traffic systems, computer human interactions, quality control in production lines, and biomedical image analysis.

Many algorithms have been proposed for feature extraction and classification in the last two decades. These algorithms involve a trade-off between the quality of the extracted features, the classification accuracy and the computational complexity of these algorithms. As an example, the unsophisticated FAST corner detector [1] executes in 13 ms for a 1024×768 image on a desktop machine, but the extracted features are not robust to changes in illumination, scale, or rotation. In contrast, a software implementation of computationally expensive algorithms like SIFT [2] and HOG [3], process the same image within 1920 ms. However, the extracted features are invariant to change in illumination, scale, viewpoint and rotation. The argument also applies to classification algorithms. A simple classification algorithm like Naive Bayes takes noticeably lower time to execute in comparison to a sophisticated algorithms like RBF-SVM. However, the classification accuracy achieved by SVM is much higher than that of Naive Bayes. So, implementing robust image classification system using complex algorithms like SIFT [2] and SVM [4] are computationally intensive and it is very hard to reach real-time performance (33ms).

To address the complexity problem, a number of simplified versions of the SIFT algorithm have been proposed, such as SURF [5] and the fast SIFT [6] algorithms. The SURF reduces the complexity of the SIFT algorithm by approximating the Laplacian of Gaussian (LoG) using filters in the orientation assignment and feature description steps. This reduces the computation time from 1036 ms for SIFT to 354 ms for SURF on a standard Linux PC (Pentium IV, 3GHz) [5]. Many SURF hardware implementations have been proposed to accelerate the algorithm [7, 8, 9]. Even thought, it reduces the computational time for extracting the local features, it produces fewer reliable features in comparison with the SIFT algorithm. Moreover, the hardware implementations require a considerable internal memory, almost four times the memory requirement of SIFT [10, 11].

Other researchers tried to accelerate the SIFT algorithm using a GPU platform [12, 13, 14]. The implementation in [12] achieved a speedup of 4-7× over an optimized CPU version when tested on a dataset of 320 × 280 pixel images. The power consumption of this implementation on the NVIDIA Tegra 250 development board was 3383 mW. In [14], another GPU-based implementation was proposed. It was able to detect SIFT features in images (640×480 pixels) within 58ms, which is around 20 frames per second. The GPU-based implementations can accelerate the SIFT algorithm to reach near real-time 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 not suitable for portable embedded systems with limited power. Other solutions tried to accelerate the SIFT algorithm using multi-core processors [15]. 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 algorithms using FPGA hardware architectures [16, 17, 18, 19, 20, 21]. However, the current implementations either work on small image resolution (320 × 240 pixels) like the work in [16, 17, 19, 21] or compute small number of SIFT features within real-time constraints. In [22], the implementation can detect up to 890 SIFT features per frame within the 33ms. Other implementations use large hardware resources [18, 19, 20] like 35,000-45,000 LUTs. Therefore, there is a strong need to propose a customized parallel hardware architecture that accelerates the execution of the image classification algorithms (SIFT, BoF, SVM) on VGA images to reach the real-time performance with low hardware resources utilization and high classification accuracy.

In this work, new techniques are used to accelerate the computationally intensive parts in the SIFT feature extraction and SVM classification algorithms and to reduce the hardware utilization. The main contributions of this paper are: (1) to the best of authors' knowledge, this is the first complete hardware architecture that implements SIFT, BoF and SVM algorithms on FPGA. (2) Reducing the hardware utilization in the SIFT's Gaussian scale space step by using the multiplierless multiple constant multiplication (MCM) with common subexpression elimination algorithm. (3) Our SIFT module architecture is able to detect and describe up to 1270 SIFT features in 33 ms with an increment of 380 features from the best recent implementation. (4) Introduce a new design for multi-ported memories to store the SIFT descriptor histogram values by reordering the SIFT's 128 values to utilize the fabricated block RAMs in the FPGA. (5) To the best of our knowledge, no other research has successfully used datasets with as much image variation as the ones in Caltech-256 and KUL Belgium traffic sign classification datasets to verify their hardware image classification systems.

The rest of this paper is organized as follows. Section II reviews the algorithms used in our architecture. Section III presents the software implementation of these algorithms. Section IV provides a detailed description of the proposed architecture and its various building blocks. Section V discusses the experimental results and architecture performance and Section VI concludes the paper.

II. REVIEW OF THE ALGORITHMS

In this work, we used Scale Invariant Feature Transform (SIFT) for feature extraction, Bag of Features (BoF) algorithm for image representation and Support Vector Machine (SVM) algorithm for classification. This section reviews these three algorithms.

A. Scale Invariant Feature Transform (SIFT)

SIFT is an algorithm used to extract and describe local features in images. It was proposed by Lowe [2] in 2004. 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. Descriptors generation is the second stage. It can be divided into two sub-tasks; dominant orientation assignment and 128-d descriptor generation.

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, σ_i) to build a Gaussian scale space as defined by Equations1 and 2. Where σ_i is the Gaussian filter scale, L(x, y, σ_i) is the filtered image and i is a scale index. The * in Equation 1 is 2-D convolution operation in x and y, and G(x, y, σ_i) is the Gaussian Kernel. In this work, we used six scales (σ_0, k× σ_0, $k^2\times\sigma_0$, $k^3\times\sigma_0$, $k^4\times\sigma_0$, $k^5\times\sigma_0$), where σ_0=1.6, and k= $\sqrt[3]{2}$. After computing the Gaussian filtered images, the next step is to compute the difference of Gaussian scale space (DoG) by subtracting each two consecutive images as defined in Equation 3.

$$L(x,y,\sigma_i) = G(x,y,\sigma_i) * I(x,y) \tag{1}$$

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

$$D(x,y,\sigma_i) = L(x,y,K\sigma_i) - L(x,y,\sigma_i) \tag{3}$$

2) Local Extrema Detection

In this step, the DoG images are scanned to find the candidate keypoints. Each pixel in the D(x, y, σi) image at location (x, y) is compared with its 3×3 neighbors in the same scale and the adjacent scales. If the pixel is local maxima or local minima out of the total 26 neighboring 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.

3) Keypoint Detection

The goal of this step is to eliminate the candidate keypoints that have low contrast or are 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 a poorly localized peak, a 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.

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

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

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 6 and 7.The gradient magnitude and gradient orientation are computed from (Δx)and (Δy) as given in Equations 8 and 9.

$$\Delta x = L(x+1,y) - L(x-1,y)/2 \tag{6}$$

$$\Delta y = L(x,y+1) - L(x,y-1)/2 \tag{7}$$

$$m(x,y) = \sqrt{\Delta x^2 + \Delta y^2} \tag{8}$$

$$\theta(x,y) = \tan^{-1}(\frac{\Delta y}{\Delta x}) \tag{9}$$

5) *Descriptor Generation*

To compute the SIFT descriptors, there are two main tasks: dominant orientation computation and descriptor generation. The dominant orientation is computed by building the gradient orientation histogram around a keypoint. The gradient orientations in a region are mapped into one of 36 bins, where each bin represents 10 degrees. At the end, the bin with the largest value or count will represents the dominant orientation. 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 4×4 squares and building the gradient histogram over 8 bins, where each bin covers 45 degrees. The gradient magnitudes are weighted by a Gaussian filter prior to building the histogram. Lastly, the 16 histograms with 8 bins are each represented in the SIFT descriptor. The SIFT descriptor has 4×4×8 values.

B. *Bag of Features (BoF)*

Bag of feature (BoF) is one of the popular image representation methods [23]. It is used to represent images as orderless collections of local features. 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. 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 are 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 descriptor into its nearest cluster center. The normalized histogram of the quantized features is the BoF model representation of that image.

C. *Support Vector Machine (SVM)*

SVM is a machine-learning algorithm proposed by Vapnik in 1990s [4]. The algorithm's idea is to find the decision hyperplane that defines decision boundaries between two 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, as given by Equations 10 to 13.

$$\text{Linear:} \quad K(x,z) = x \cdot z \tag{10}$$

$$\text{Polynomial:} \quad K(x,z) = \left((x \cdot z) + 1\right)^d,\ d > 0 \tag{11}$$

$$\text{RBF:} \quad K(x,z) = \exp(-\parallel x - z \parallel^2 / (2\sigma^2)) \tag{12}$$

$$\text{Sigmoid:} \quad K(x,z) = \tanh(K(x \cdot z) + \Phi) \tag{13}$$

SVM was originally designed as a binary classification algorithm, but it is possible to extend it to multiclass classification. The most common methods of multiclass SVM are the 'one against all' and the 'pairwise' classifiers methods. In this work, we used one against all multiclass SVM with RBF kernel.

III. Software Implementation

We used a software implementation of the SIFT-BoF-SVM algorithms for three reasons. First, for validation, we compare the results generated by our hardware implementation with the software implementation's results. Second, for performance measurement, we measure the processing time in software and compare it with the processing time in the hardware implementation. Third, to choose the best classification algorithm for our system by conducting comparison between six different classification algorithms.

In the software implementation, we used an open source SIFT library implemented in C by Hess [24]. For the SVM classification algorithm, we used LibSVM [25], an open source C++ library implemented by Chang et al. The validation and performance measurement parts are reported in the experimental result section. In this section, we elaborate why we choose RBF-SVM over the other classification algorithms.

Selecting the best classification algorithm for our system was an important step. So that, we perform an experiment to find out which classifition algorithm achieves higher accuracy within an accepted processing time. The algorithms tested are: K-nearest-neighbor (KNN), Naïve Bayes, Decision Tree, Adaboost, linear support vector machine (SVM), and Non-linear SVM. For this experiment, we used Caltech-256 [26] benchmark dataset. We used five classes: airplane, human face, motorbike, hours, and watch. Figure 1 shows example images from each category.

Figure 1: Five classes from the Caltech-256 dataset

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 datasets of sizes (50, 100, 200, 300, 400, and 500) and measuring its accuracy we obtained the results of Figure 2.

The RBF-SVM classifier outperforms other classifiers on all training sizes. In terms of accuracy, the RBF 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.

To assess how the processing time is effected by increasing the testing set size, we measured the processing time on datasets of sizes of 50,100,200,300,400 and 500. The results are shown in Figures 3. The KNN classifier has the highest processing time followed by the RBF SVM classifier and linear SVM. The Adaboost and naïve Bayes classifiers have lower processing time. In this work, we opted to use the RBF SVM classification algorithm due to its high classification accuracy. We then use techniques to accelerate the processing time of RBF-SVM algorithm using hardware implementation to reach real-time performance.

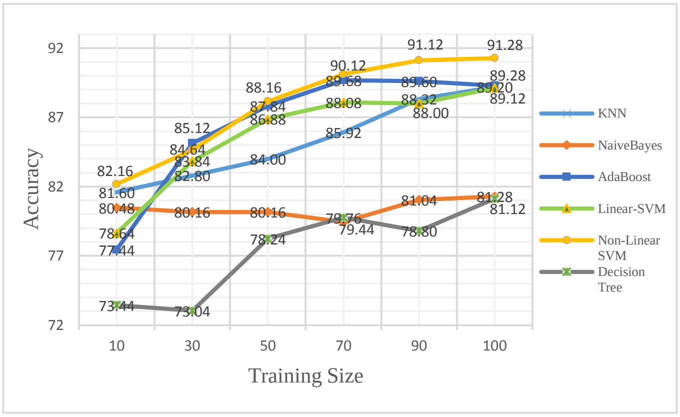


Figure 2: Comparison between six classification algorithms

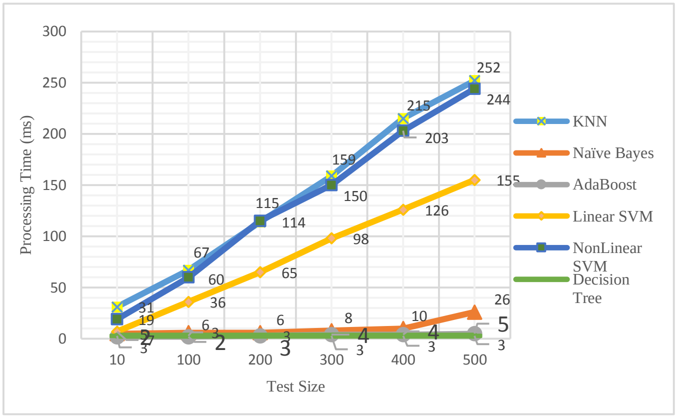


Figure 3: Testing processing time of six classification algorithms

IV. Hardware Architecture

The proposed pipelined architecture consists of three major modules: 1) SIFT modules 2) Bag of feature (BoF) module, and 3) a support vector machine (SVM) module. The SIFT block receives its input as a stream of pixels from the source image and performs the feature extraction operation. The BoF block converts the set of extracted SIFT features into a BoF vector. The SVM block takes the BoF vector and classifies it into one of the trained classes.

A. *SIFT Module implementation*

The high level block diagram of the proposed SIFT architecture is shown in Figure 4. 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 represent the SIFT feature descriptor.

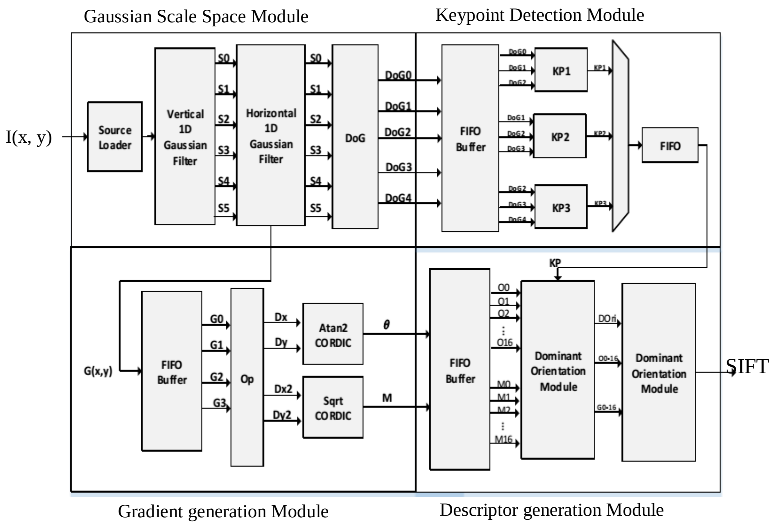

Gaussian Scale Space Module
Keypoint Detection Module
I(x, y)
Source Loader
Vertical 1D Gaussian Filter
Horizontal 1D Gaussian Filter
DoG
FIFO Buffer
KP1
KP2
KP3
FIFO
G(x,y)
Op
Atan2 CORDIC
Sqrt CORDIC
Dominant Orientation Module
SIFT
Gradient generation Module
Descriptor generation Module

Figure 4: The overall architecture of the proposed SIFT hardware

The primary task of GSS module is to compute the Gaussian-filtered images and the difference of the Gaussian images. The input to this module is a stream of pixels from the source image, while its output 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 module computes the gradient magnitude and orientation from the Gaussian-filtered image. The KDS module reads one stable keypoint at a time and computes the SIFT descriptor based on the gradient values. In the next subsection, the internal architecture of each module is presented.

1) *Implementation of GSS module.*

The Gaussian scale space (GSS) module computes the Gaussian filtered images and the difference of Gaussian images (DoG) from the input 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 and so on. The final step is to compute the DoG images by subtracting each two adjacent Gaussian filtered images in the same octave. In this work we used three octaves with six scales in each octave.

There are two approaches in the literature for generating the Gaussian scale space; each one has its advantages and disadvantages. First, cascade filtering approach used in [17, 16]. In this approach, the Gaussian-filtered image $G(x, y, \sigma_{i+1})$ is generated by convolving $G(x, y, \sigma_i)$ with a small kernel. This method reduces number of multipliers required in each Gaussian filters. However, it needs a large amount of memory to save the intermediate results between each two filters. The second approach, direct filtering approach used in [22, 18]. This method generates all Gaussian-filtered images directly from the source image using Gaussian kernel with different size. The size of these kernel is large, so this method needs large number of multipliers. But there is no need for memory to save any intermediate results.

In our GSS implementation, we modified the second approach to take the advantage of a low memory requirement. To reduce the hardware utilization by the large number of multipliers, we used the multiplierless multiple constant multiplication (MCM) with common subexpression elimination algorithm [27] to implement the Gaussian filters in the GSS module. Moreover, we used the separability and symmetry properties of the Gaussian filter to reduce hardware resources, by separating each Gaussian filter into two smaller 1D vertical and horizontal filters.

The high level architecture for the 1st octave in GSS module is shown in Figure 5. 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 element represented as 8 bits (greyscale values). The FIFO buffers are implemented using Xilinx LogiCORE™ IP FIFO Generator.

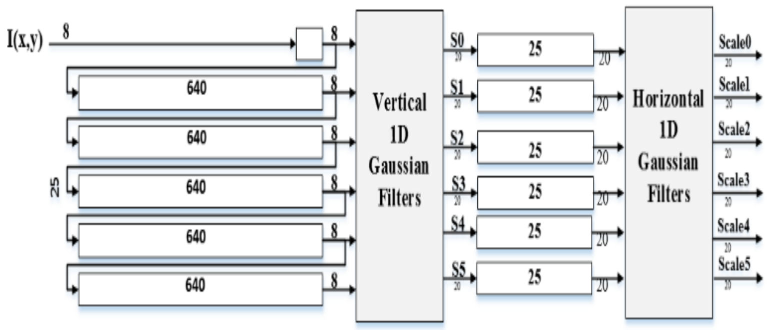


Figure 5: GSS's first octave module

The module has two states; loading and a computation. In the loading state, the stream of pixels is shifted into the buffer line one value to the right every clock cycle as shown in Figure 5. After 24× (640) +1 clock cycles the loading state ends and the computation state start. In the computation state, the first block (vertical 1D Gaussian filter) reads the left most 25 pixels from the buffer lines (pixels 0,1,2 ...24), and computes the first pixel of each Gaussian filter images (S0_1D, S1_1D … S5_1D). The results from the first block are buffered in 6 buffer lines with 25 elements each. After computing 25 valid values, the second block (horizontal 1-D Gaussian filter) reads each 25 values and computes the corresponding 1-D horizontal filtered pixels. In the next clock cycle, a new pixel from the source image shifted into the buffer lines and the next sliding window will be computed. This operation will be repeated until the whole source image is shifted into the GSS module.

The architecture for the vertical and horizontal 1-D Gaussian filter block is shown in Figure 6. The architecture contains 13 blocks to compute the multiplication of each pixel with 6 different constants from 6 different Gaussian filters. We used the symmetrical property of Gaussian filters to reduce the hardware utilization by adding the pixels before multiplying them with the corresponding filter constants. Figure 7 shows the constants of the six Gaussian filters (9×9, 11×11, 13×13, 15×15, 21×21, 25×25). All masks are extended to 25 values by appending zeros to the left and right. These constants are generated using Matlab then the multiplierless multiple constant multiplication with a common subexpression elimination algorithm is applied to find the optimized tree architecture of each block. Figures 8 and 9 show the tree implementations for block0 and block1, respectively.

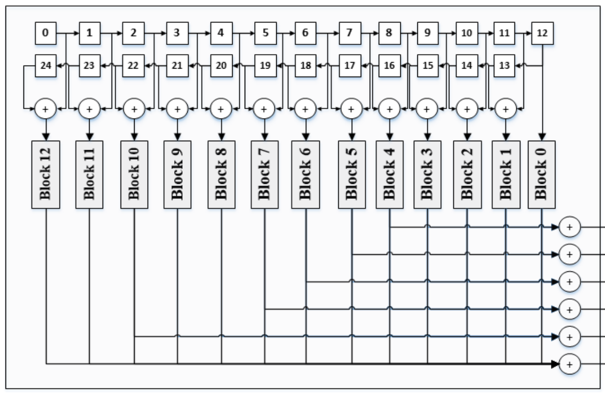


Figure 6: 1-D Gaussian filter block

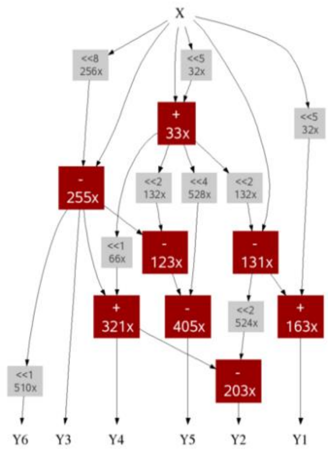


Figure 8: Block 0 in 1-D Gaussian filter block

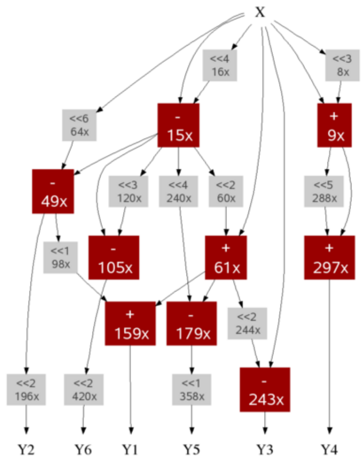


Figure 9: Block 1 in 1-D Gaussian Filter Block

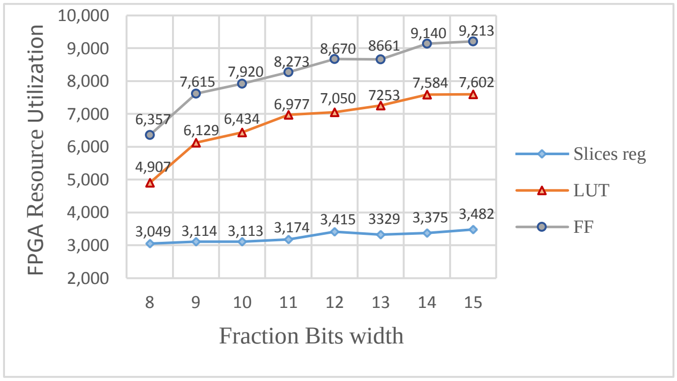
The idea of using the multiplierless multiple constant multiplication algorithm with common subexpression elimination is to represent the multiplication operation as a number of shifts, additions, and subtractions operations. The common subexpression elimination method is used to extract the common parts among these constants in order to minimize number of multiplications needed. Figures 8-9 show the architecture for Block0 and Block1. In Figure 9, the input X is the summation of two pixels in the 25 buffers labelled with indecies 11 and 13. The block's output is the result of multiplying X by the corresponding constant from the Gaussian filter mask in Figure 7. That is, Y1= X×0.205, Y2= X×0.175, Y3= X×0.145, Y4= X×0.119, Y5= X×0.096 and Y6= X×0.078. The other blocks are implemented in the same way.

The output of the GSS module is the DoG images. The DoG images are generated by subtracting each two adjacent Gaussian filtered images in the same octave. Each pixel in these images is represented by an integer part and fraction part. To find the optimal number of bits, an analysis is conducted to assess the effect of the number of bits on the accuracy of the result DoG image and hardware utilization. The DoG pixels are represented by 8 to 15 bits.

pixel #	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
Block #	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	0.01	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	0.02	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 7: 1D Gaussian Filters Values

Figure 10 shows how the hardware resources increase almost linearly with the increase in the number of bits. On the other hand, the error decreases with a sharp slope between 8 and 11 bits and then the improvement becomes negligible. These results indicate that the resources utilized by the proposed hardware module is proportional to the number of pixels. To reduce the FPGA resource utilization without significantly scarifying the accuracy, we found that the optimal number of pixels is 11.Therefore, the output pixels in the GSS module are represented with 20 bits; 11 bits for fractional part and 9 bits for integer part.



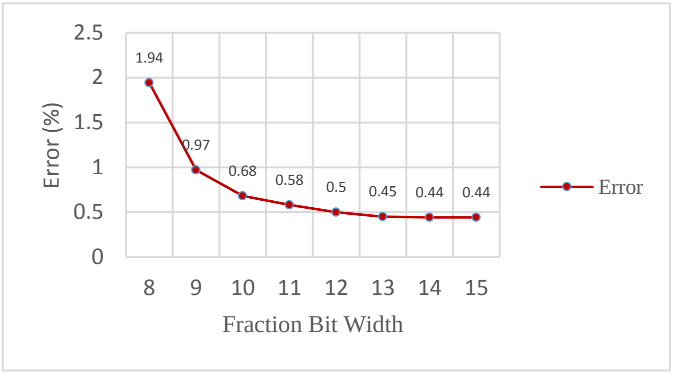


Figure 10: Effect of the number of bits on the accuracy of the GSS module

2) *Implementation of the Keypoint Detection Module*

The keypoint 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 in buffer lines and then detects keypoints by analyzing the DoG images. The module's output is the keypoint for the x and y positions.

In this module, we parallelized the process of checking for candidate keypoints 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 11. The module has two states: loading and checking. In the loading state, the module shifts the input pixels into the buffer lines. After (2×image_width+3) clock cycles, the buffer line's content becomes valid and the checking state begin.

In the checking states, each keypoint detector block compares the pixel in the center with its surrounding neighbors. If the current pixel is the maximum or the minimum out of the 27 pixels, then the pixel is a candidate keypoint. Most of the previous implementations skipped the local minima to reduce the hardware cost and to speed up the process. This simplification could overlook some important key points. Therefore, to reach the highest accuracy, we compute the local maxima and minima in this module.

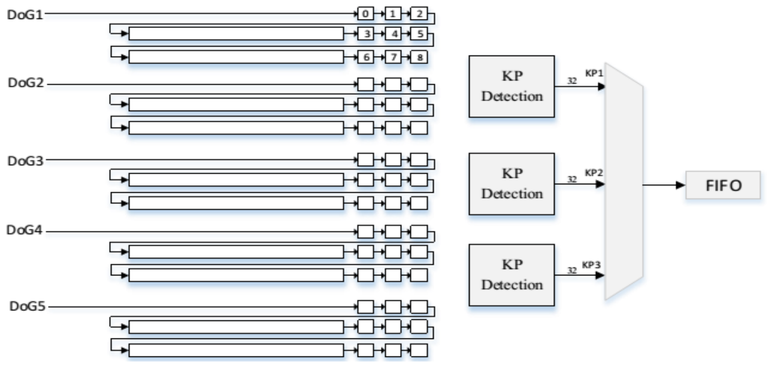


Figure 11: Keypoint detection module architecture

3) *Implementation of Gradient Magnitude and Orientation Module*

The gradient generation module is the third module in the SIFT architecture. It is used to compute gradient magnitude and orientation from the Gaussian filtered images. The module's architecture consists of two buffer lines, three adders, two multipliers and two CORDIC IP Cores. The two buffer lines are connected to three pixel elements as shown in Figure 12 to generate the values of G(x, y+1), G(x, y-1), G(x+1, y), and G(x-1, y) after the (2×image_width+3) clock cycles.

In this implementation, the square root, and the tan inverse functions are computed using two CORDIC Xilinx IP cores. 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 1 summarizes the input and output data format.

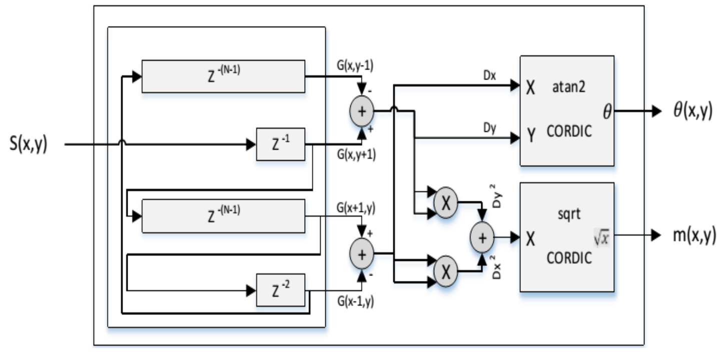


Figure 12: Gradient magnitude and orientation module architecture

Table 1: Data format in the gradient generation block

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

In the atan2 IP, we used the coarse rotation module to extend the range of CORDIC from the first quadrant (+Pi/4 to -Pi/4 Rad) 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.

4) *Dominant Orientation Module*

The dominant orientation module computes the principle orientation for each stable keypoint. The module's inputs are the key point's position (KP_in), and the gradient magnitude and orientation (Gradient_M, Gradient_O). 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 13 shows the module's inputs, outputs, and its internal structure.

The module architecture consists of 2× (17 MUXs (2×1), 17 buffer lines (640 value), 17 FIFO (17 elements), 17 MUXs (17×1), two position counters (X, Y) and the Dom_Ori Block). The module has five states: 1) initialization, 2) window_shifting, 3) searching, and 4) loop_back state as shown in Figure 14. Initially all buffer lines are empty, when the valid gradient magnitude and orientation generated by the GMO module become available, the module start shifting these values into the buffer lines.

After 640×17 clock cycles, the buffer lines become full and the window_shifting state starts. In this state, a position counter (X, Y) are used to keep track the current window position. Every clock cycle, a new pixel shifted into the buffer lines and the counter X incremented by 1, after every 640 clock cycles the counter y incremented by 1.

When a new keypoint found, the searching state begins. First, the Kp_x and Kp_y values updated by the new keypoint position. Second, every clock cycle the current position counter value (X, Y) is compared to the (Kp_x and Kp_y). If they are equal, then the loop_back state starts. In this state, the MUX selectors changed to (1) and a disable signal sent to the GSS module to stop reading from the input image. After 640 clk cycles, the 17×17 window around the keypoint will be shifted into the Dom_Ori module and the buffer lines values will stay the same.

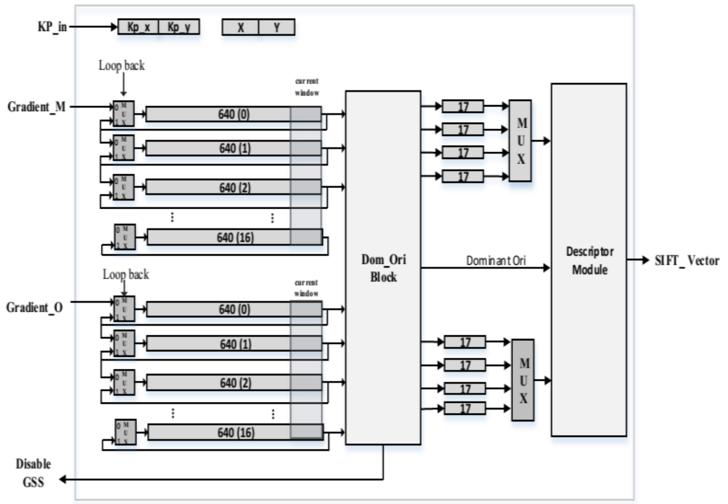


Figure 13: Dominant orientation generation module

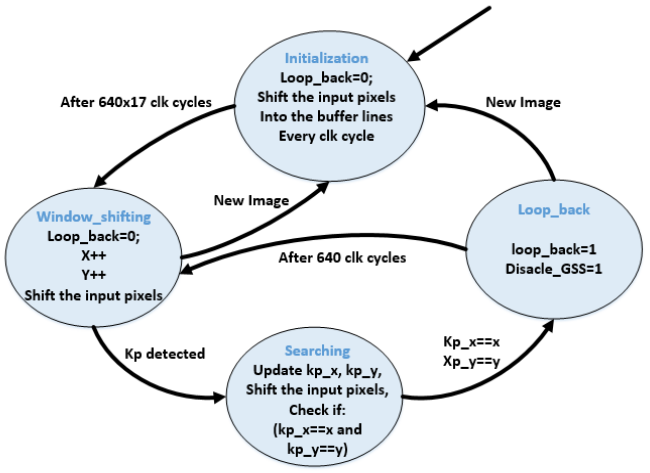


Figure 14: State machine diagram of dominant orientation module

The Dom_Ori module reads every clock cycle a 17 new gradient magnitude values, and 17 new gradient orientation values. 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. The task of the decoder is to find the correct bin out of the 36 bins. 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. 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 module finishes the current keypoint, it will read the next keypoint and repeat the whole process again.

5) *Descriptor Generation Module*

The SIFT descriptor generation process 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 a 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 15. It consists of four main blocks: rotation module, Gaussian weight generation module, and 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 rotates the gradients within the region around a keypoint relative to the principle orientation. It 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) based on Equations 14 to 16. Where SBP is a constant equal to 3 times the keypoint scale (SPB= 9.6), and NBO is number of orientation bins (NBO=8).

$$\mathrm{Xr} = (\cos(\mathrm{Og}) \times \mathrm{X} + \sin(\mathrm{Og}) \times \mathrm{Y})/\mathrm{SBP} \tag{14}$$

$$\mathrm{Yr} = (-\sin(\mathrm{Og}) \times \mathrm{X} + \cos(\mathrm{Og}) \times \mathrm{Y})/\mathrm{SBP} \tag{15}$$

$$\mathrm{Or} = \mathrm{NBO} \times (\mathrm{Og} - \mathrm{Op})/(2\pi) \tag{16}$$

The division operations in equations (14-16) are converted to multiplication to reduce the hardware utilization. The Sine and Cosine components are computed using the Xilinx LogiCORE™ IP CORDIC core. The input angles are expressed as a fixed-point numbers with (3.13) format. The outputs (Sine, Cosine) are expressed as a pair of fixed-point 2's complement numbers with format (2.14). The orientation generation 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 (16).

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 Trilinear interpolation module distributes the result of multiplying Wg×Mg into eight adjacent bins in the SIFT descriptor histogram based on the Xr, Yr, and Or values. The eight values with its corresponding eight addresses are then sent to the histogram memory.

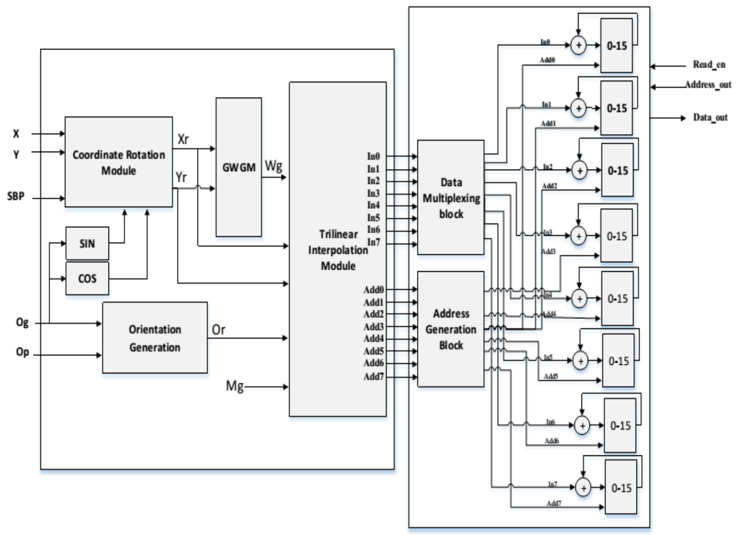
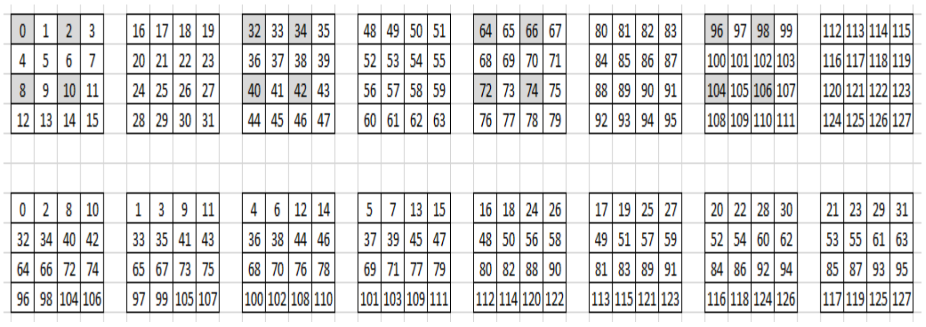


Figure 15: SIFT descriptor Module Architecture

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 16 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 16: Histogram Memory Implementation

B. *BoF Module implementation*

The bag of features module is used to convert the set of SIFT descriptors (k×128) extracted by SIFT module into a BoF vector of size N. The first step is to cluster the SIFT descriptors into N clusters using K-mean clustering algorithms and find the clusters centers. 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 centroids that minimize the sum of the squared Euclidean distances between the points and their nearest cluster centroids. We used Matlab code to implement the k-means clustering algorithm. Then, the cluster centroids are then loaded into the bag of feature module in the hardware using N FIFO buffers with a length of 128 elements. When the module's enable signal is activated, every clock cycle one element of the SIFT descriptor (SIFT[i]) is shifted into the module. Using N subtractors, this value is subtracted from the proper cluster centroid's element. The results are accumulated using N accumulators. After 128 clock cycles, the distances between the SIFT descriptor and the clusters centers are available in the accumulators. The min block searches the N accumulators to find the minimum value. The minimum value means that the distance between the input SIFT descriptor and that cluster centroid is the minimum. The final step is to increment the corresponding histogram bin by one.

C. *SVM Module Implementation*

In the SVM architecture, we implement 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 of the SVM classifier in hardware. The SVM model was generated using a software implementation though.

The input to the SVM classifier module is the vector generated by BoF module. The output is the class label after the classification process is completed. In this implementation, the training phase of SVM is computed offline. 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 in the hardware.

In the SVM testing phase, the new data vector (**x**) is classified according to the decision function of Equation 17. Where k (.,.) is the kernel function, N_{sv} is the number of support vectors generated in the training phase, αi is weights for each SV and b is a bias constant. While $\mathbf{x}_i$ and y_i represent the SV and its class label y_i= {-1, 1} respectively.

$$f(x) = sign(\sum_{i=1}^{Nsv} yi \times \alpha i \times k(xi, x) + b) \tag{17}$$

$$K(x, z) = \exp(-\| x - z \|^2 / (2\sigma^2)) \tag{18}$$

In the kernel function 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 18. 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 20 instead of Equation 19. 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.

$$\| x - z \| = \sqrt{(x1 - z1)^2 + (x2 - z2)^2 + \cdots + (xn - zn)^2} \tag{19}$$

$$\| x - z \|^2 = (x1 - z1)^2 + (x2 - z2)^2 + \cdots + (xn - zn)^2 \tag{20}$$

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 n clock cycles, 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 21, 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).

$$\text{Accumulator bit width} = \log_2(c \times (2^{20} - 1)) \tag{21}$$

After n clock cycles, the output of the accumulator is multiplied by $(-1/(2\sigma^2))$. For the implementation of the exponential function, we represent the exponential function as a summation of hyperbolic sine and hyperbolic cos of the x, as given in Equation 22. 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.

$$\exp(x) = Sinh(x) + Cosh(x) \tag{22}$$

The overall architecture for SVM module is depicted in Figure 17. We used FIFOs buffer lines to store the SVs and $yi \times \alpha i$ values. The value of these buffers come from the training phase carried out offline. In this architecture, we used 20 modules to compute the kernels function between the input vector **x** and 20 SVs.

D. *FPGA Implementation*

We developed a prototype of our architecture on the FPGA. We used ML509 evaluation platform which is equipped with Virtex 5 LX110T FPGA from Xilinx. The overall system consists of the SIFT processor core, BoF module, SVM classifier module and MicroBlaze soft-core processor. The hardware modules are implemented using Verilog hardware description language. Our architecture interfaces with the Microblaze processor via two FSL bus systems for I/O purposes. We used Xilinx ISE Design Suite, Xilinx Platform Studio (XPS), and Xilinx Software Development Kit (SDK) platforms to design and implement the architecture. The overall system architecture is illustrated in Figure 18.

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.

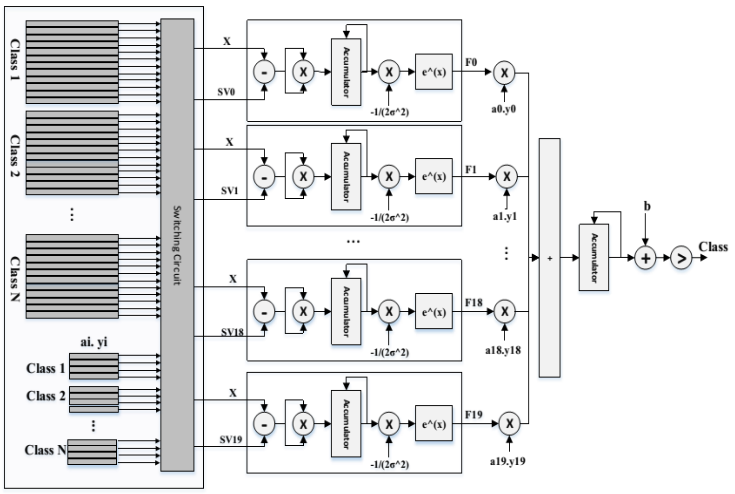


Figure 17: The overall architecture for SVM

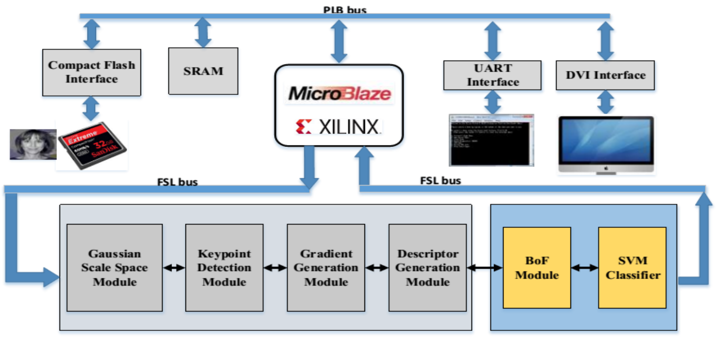


Figure 18: Block diagram of the FPGA prototype system

E. *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. Table 2 summarizes the hardware resources used to implement the whole object detection architecture.

Table 2: Hardware Utilization 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-Flip Flop pairs	10,981	43,841	25%
Number of Block RAM/FIFO	128	148	86%
Number of DSP48Es	64	64	100%

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. 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. Using the MCM algorithm in GSS step reduces the hardware utilization by a reduction gain of 2 when compared to previous works. Tables 3, 4 and 5, summarize the hardware utilization for each of the major modules of the design.

Table 3: Hardware Utilization 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 4: Hardware Utilization 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 5: Hardware Utilization for the SVM 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%

V. EXPERIMENTAL RESULT

The performance of the proposed hardware architecture can be measured in terms of three metrics: classification accuracy, speed up compared with the equivalent software and hardware implementations, and hardware resources 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 the highest accuracy within real-time constraints which is processing 30 images per second.

A. Modules' Accuracy

In the hardware implementation, we used fixed point numbers, while the software implementation 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, we computed the percentage error between the software and hardware values based on Equation 23.

$$\text{Error} = \frac{|\text{Software value} - \text{Hardware value}|}{\text{Software value}} \text{ x } 100\ \% \qquad (23)$$

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. The average errors in the first octave for its six scales are reported using a human face image. The errors in scales 0 to 5 were 1.760 %, 2.040 %, 2.930 %, 3.870 %, 4.310 % and 5.670 %, respectively.

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 of 99.3% in term of the keypoints position. The accuracy is computed by dividing the 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.

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 error in gradient magnitude was equal to 3.65% and in gradient orientation was equal to 1.527%. The error in Gaussian weight generation module was approximately 4.3%. For the SIFT feature generation module, the SIFT vector's element is represented by fixed point numbers with 20 bits in (9:11) format. The error between the software and hardware values is equal to 3.36%. We can achieved a higher accuracy in these modules by increasing the word lengths of related data, but we allow a relatively small errors to save hardware cost. We did an analysis to find the optimal bit width that gives us the best accuracy and the lower hardware cost.

B. *Classification Rate Evaluation*

In order to assess the classification rate of the proposed hardware implementation, we used two popular image classification datasets as mentioned previously; Caltech-256 dataset [26] and KUL Belgium Traffic Sign Classification dataset [28]. These datasets are considered a challenging datasets in image classification. So that, if our architecture achieved well in these two datasets that means that it will also do well on other datasets.

1) *Experiment 1: Caltech-256*

In the first experiment, we used ten different randomly selected subsets from the Caltech-256 dataset. Each subset consists of five different classes. The classes were: horse, face, motorbike, watch, airplane for the first subset. Blimp, bowling-pin, boxing-glove, brain, bulldozer for the second subset. Figures 19 shows examples from the first and second subsets.

The classification results in this section are averaged over the five classes of 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. In the training phase, for each subset, we used 100 images, 20 images per class, and in the testing phase, for each subset, we used another 100 images, 20 images from each class. Figure 20 shows the average confusion matrix using the ten subsets. We computed the average confusion matrix by averaging the values for all five classes combined.

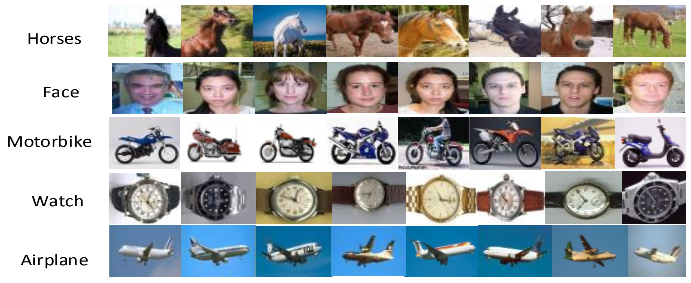


Figure 19: Example images from subset 1 - Caltech-256 dataset

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 20: Hardware and Software Average Confusion Matrix

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.

The classification accuracy in software implementation for subsets 1 to 10 was as follow (85%,69%,73%,79%,75%,74%,87%, 72%, 67% and 70%) and for hardware implementation (84%,69%, 71%, 77%,72%,70%,87%,71%,66% and 69%). 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 two values. For the LibSVM software implementation, the average classification rate for the 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 of the processed data.

The misclassification error may seems high even in the software implementation. This is due to the fact that the Caltech-256 dataset is considered a very challenging dataset. To date, the best classification accuracy achieved using this dataset is 46.6% using 30 sample from each category (256 categories) [29]. The goal of this experiment is to assess the accuracy difference between the software and our hardware implementation. A 3% accuracy difference between our architecture the equivalent software implementation is acceptable especially for such a challenge dataset.

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 21. 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 22 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. From these two experiments, the results show that the accuracy difference between our architecture and the equivalent software implementation will be within 3% using the same training and testing sets.

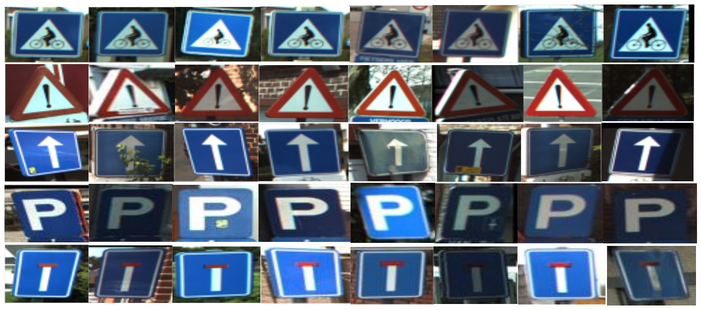


Figure 21: Example images from the KUL Belgium Traffic Sign Dataset

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 22: Hardware and Software Confusion Matrix

C. Processing Time

The processing time required to classify one frame is divided into two parts: 1) the SIFT feature extraction and description time, and 2) the SVM classification time. The processing time in each module is estimated by simulation the implementation in Verilog HDL and estimate the speed 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 24.

$$Time = \frac{\text{Number of clock cycles to finish the task}}{\text{Operationl frequency (Hz)}} \qquad (24)$$

1) SIFT Module Processing Time

The maximum operating frequency of the proposed design is 60.36 MHz, which is provided by the synthesis report. For the SIFT feature extraction module, it takes 640×480 clock cycles to scan the input image and detect the keypoints. It also takes for each keypoint (640+17×17+128) clock cycles to generate the SIFT descriptor. The processing time for the SIFT feature extraction and description module is estimated as shown in Equations 25 and 26. SIFT’s processing time for one frame (640x480) pixel. Where #KP represents number of SIFT descriptor detected in the image.

$$\text{Time} = \text{Detection Time} + \text{Descriptor Generation Time} \qquad (25)$$

$$\text{Time} = \frac{640 \times 480 + (640 + 17 \times 17 + 128) \times \#\,\text{KP}}{\text{Operationl frequency (Hz)}} \qquad (26)$$

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 (which is approximately 25.9 µ sec/ feature). 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 27.This is so because for real-time performance we need to achieve a rate of 30 frames per second.

$$33\text{ms} = \frac{(640 \times 480 + (640 + 17 \times 17 + 128) \times \#\,\text{KP})}{50\ MHz} \qquad (27)$$

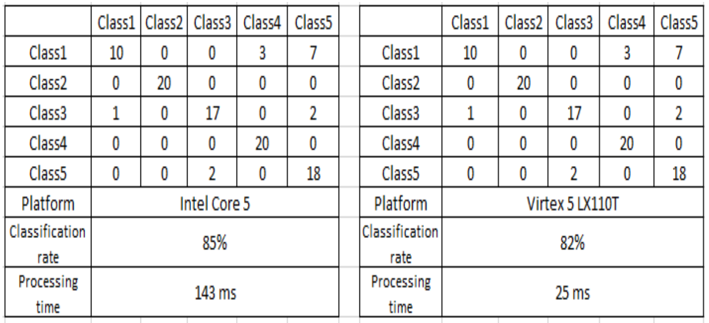
The maximum number of keypoints in each frame is thus 1270 keypoints per frames. Therefore, our architecture allows a VGA size image with about 0.4134% Keypoints to be processed at a speed of 30 frames per second. Due to parallelization technique proposed in the SIFT descriptor module, we can compute larger number of SIFT descriptor within the 33ms than the works proposed in [22] that can compute up 890 SIFT descriptor within the 33ms, and [16] that can compute up to 412 SIFT descriptors in real time performance. In our architecture, if the input image has more than 1270 SIFT feature, it will take more than 33ms to compute the SIFT features in the image. To compute the speed up achieved by our SIFT architecture, we compared the averaged processing time required to extract the SIFT features from 209-image (people) collection of the Caltech-256 data set and compare the time to the time reported in [24]. On average the processing time in [24] is 1.81sec per image, while our architecture extracts the SIFT features within 33ms per image, with a speedup equals to 54.8, as shown in equation (28).

$$\text{Speedup} = \frac{Harware\ processing\ time}{Software\ processing\ time} = \frac{1.81\ sec}{33\ msec} = 54.8 \qquad (28)$$

2) 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 $floor\left(\frac{N\_SV}{20}\right) \times \text{SV\_Dimentions} \times \#\text{Classes} \times \left(\frac{1}{\text{Maximum opertional frequency}}\right)$. In Caltech-256 dataset experiment, we used five classes with each class having 20 training images. 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 proposed architecture and the equivalent software implementation, we classified 100 test images into one of the five classes. Figure 23 summarizes the confusion matrices, the classification accuracy, and the processing time.



	Class1	Class2	Class3	Class4	Class5
Class1	10	0	0	3	7
Class2	0	20	0	0	0
Class3	1	0	17	0	2
Class4	0	0	0	20	0
Class5	0	0	2	0	18
Platform	Intel Core 5				
Classification rate	85%				
Processing time	143 ms				

	Class1	Class2	Class3	Class4	Class5
Class1	10	0	0	3	7
Class2	0	20	0	0	0
Class3	1	0	17	0	2
Class4	0	0	0	20	0
Class5	0	0	2	0	18
Platform	Virtex 5 LX110T				
Classification rate	82%				
Processing time	25 ms				

Figure 23: Software implementation results compared to our SVM architecture

We used a LibSVM library 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 the same 100 images with a speedup equals to 5.7, as shown in equation (29). The difference in classification accuracy in our implementation is 3% less than the software implementation.

$$\text{Speedup} = \frac{Harware\ processing\ time}{Software\ processing\ time} = \frac{143\ ms}{25\ ms} = 5.72 \tag{29}$$

D. 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 works in [16, 19, 17, 18, 20, 21] . Table 5 summarizes the performance and hardware utilization of each implementation. We also compared our SVM architecture with works in [30, 31, 32, 33]. Table 6 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+17×17+128) clock cycles to generate one descriptor. Hence, at 50MHz operational frequency, the time required to scan the input image and detect all keypoints is 6.14 ms and 25.9 µsec to generate each SFIT descriptor. While, in existing solutions it was around 10 ms to 30 ms to detect the SIFT keypoints as shown in Table 5. In [16], it took 80 µ sec to generate one SIFT feature and 11.7 ms in [17]. In term of keypoint ratio that can be processed in 33ms, our architecture has a 0.4134%, while the works in [16] and [17] have 0.2897% and 0.024%, respectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCI.2015.2424077, IEEE Transactions on Computational Imaging

Table 5: Comparing proposed SIFT results with existing soultions

	[30]	[31]	[32]	[33]	**Proposed**
Operating frequency	151 MHz	100 MHz	141MHz	30 MHz	50MHz
FPGA	Virtex4	Virtex-5	Virtex-5	Cyclone II	Virtex 5
# of LUTs	9,141	57,296	37,549	14,064	38,179
# of registers	11,589	23,220	37,067	-	9,646
# of DSP blocks	81	83	128	20	52
Dataset	Persian handwritten digits	Faces dataset	MNIST	COIL database	Caltech-256
Number of classes	3	-	-	4	5
Input dimensions	24	-	-	1024	500
# SVs	145	400	-	60	100
Classification accuracy	98.67%	77%	99.11%	96%	82%
Processing time	0.27 ms	40 frames/sec	Speedup 20x	2ms	0.25ms

Table 6: Comparing proposed SVM Results with existing solutions

	[16]	[17]	[18]	[19]	[20]	[21]	**Proposed**
Device	XC4VDX35	EP2S60F6	EP2C70F8	EP3C120F4	EP2C70F8	EP2S60F6	XC5VLX110T
LUT	18,195	43,366	35,889	43,563	32,592	16,832	16,138
Register	11,821	19,100	19,529	14,730	23,247	5,729	7,924
DSP blocks	56	64	97	45	258	8	53
BRAM (kbits)	2808	1350	256	2810	891	752	576
Resolution	320 × 256	320 × 240	640 × 480	320 × 240	640 × 480	320 × 240	640 × 480
Detector	10 ms	33 ms	31 ms	30 ms	31 ms	-	6.144 ms
Descriptor	80 µ sec/ F.	11.7 ms /F	-	-	-	-	25.9 µ sec/ F
Keypoint Ratio	0.2897 %	0.024 %	-	-	-	-	0.4134%
Accuracy	96.9%	95.47%	-	-	-	-	97.54%

The hardware resources utilized by our implementation was less than most of the existing solutions. Our architecture consumed 16,138 LUTs and 7,924 registers which is less than [17, 18, 19, 20] and almost the same as [16] and [21]. However, the image resolution in [16] and [21] is 320 ×240, while our architecture works on larger images with a resolution of 640 ×480 pixels.

Table 6 summarizes the performance of our SVM architecture against that reported in [30, 31, 32, 33] . In [31], the SVM architecture achieved an accuracy of 77% with 40 frame per second speed. It used large hardware resources to reach the 40 frames per second. In [32], the author implements a SVM classifier and assesses its performance using a simple dataset called MNIST [34]. They achieved a speedup of 20 compared to dual Opteron 2.2 GHz processor CPU. In [33], the dataset used was simple, the images are represented with 8 bits grey scale with a resolution of 32×32 pixels. The number of support vectors is also limited to 10 per binary classifier. Their architecture has 6 binary classifiers. The classification speed of the system was 2 ms. In [30], 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. The implementations [30, 31, 32, 33] either used a very simple dataset to achieve a high accuracy or consume more hardware resources to deal with large images or with more number of classes. Our implementation used a challenging dataset (Caltech-256) with 500 dimensional input vectors. The classification accuracy was equal to 82% within a processing time of 0.25 ms for each input image. We also worked on five classes while other implementations worked on three or four classes only.

In terms of hardware resources utilization, our SVM architecture consumed less resources than [31, 32]. It consumed only 38,179 LUTS and 9,646 Registers. Our implementation consumed more resources than [30] because the input vector dimensions in [30] is only 24, while in our work it is 500. Also the number of classes in [30] is restricted to 3. The architecture in [33] consumed less resources but the processing time is 10 times greater than our implementation.

VI. CONCLUSION

The paper proposed a hardware architecture capable of detecting SIFT features from images with a dimension of 640×480 pixels within 6.144 ms. It can also compute up to 1270 SIFT features for each image. The classification accuracy achieved by the proposed 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 3%. The speed up achieved in the feature extraction was ×54 and in the classification algorithm it was ×5.7 in comparison to software implementation. The proposed architecture FPGA resource utilization was moderate compared to existing hardware implementations. Hence, the implemented image classification architecture provides an embedded system solution that can be used for the detection and recognition of objects with real-time performance. In a future direction, and to achieve better performance and optimize power requirements, the ASIC implementation of the work discussed here can be pursued with some modifications to the FPGA design flow by including activities such as conversion of IP cores to Verilog Modules, performing post-layout synthesis and meeting design-for-test requirements.

REFERENCES

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

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

[3] D. Navneet and B. Triggs, "Histograms of oriented gradients for human detection," *Computer Vision and Pattern Recognition, 2005. IEEE Computer Society Conference on,* vol. 1, 2005.

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

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

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

[7] M. Pohl, M. Schaeferling and G. Kiefer, "An Efficient FPGA-based Hardware Framework for Natural Feature Extraction and Related Computer Vision Tasks," *24th International Conference on Field Programmable Logic and Applications (FPL),* 2014.

[8] A. Serackis and T. Sledevic, "SURF Algorithm Implementation on FPGA," *Biennial Baltic Electronics Conference,* 2012.

[9] J. e. Svab, "FPGA BASED SPEEDED UP ROBUST FEATURES," *Technologies for Practical Robot Applications,* 2009.

[10] w.-y. Lee and K.-j. Byun, "A hardware design of optimized ORB algorithm with reduced hardware cost," *Advanced Science and Technology Letters,* pp. 58-62, 2013.

[11] S. Panchal, "A Comparison of SIFT and SURF," *International Journal of Innovative Research in Computer and Communication Engineering,* vol. 1, no. 2, 2013.

[12] 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.

[13] C. Jiang, Z.-x. Geng, X.-f. Wei and C. Shen, "SIFT implementation based on GPU," *International Symposium on Photoelectronic Detection and*

Imaging, vol. 891304, 2013.

[14] S. Heymann, K. Müller, A. Smolic and B. Fröhlich, "SIFT Implementation and Optimization for General-Purpose GPU," *The 15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2007 in co-operation with EUROGRAPHICS,* 2007.

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

[16] 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 .

[17] 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.

[18] 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.

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

[20] 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.

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

[22] F.-C. Huang, S.-Y. Huang, J.-W. Ker and Y.-C. 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.

[23] C. Bray, G. Csurka, C. Dance, L. Fan and J. Willamowski , "Visual categorization with bags of keypoints," *Workshop on Statistical Learning in Computer Vision, ECCV,* pp. 1-22, 2004.

[24] R. Hess, "An open-source SIFT Library," *Proceedings of the international conference on Multimedia,* 2010.

[25] C.-J. Lin and C.-C. Chang , "LIBSVM : a library for support vector machines," *ACM Transactions on Intelligent Systems and Technology,* 2011.

[26] G. H. a. P. A. P. Griffin, "Caltech-256 Object Category Dataset," Technical Report 7694, California Institute, 2007.

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

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

[29] M. Sancho and D. G. Lowe, "Spatially local coding for object recognition," *Computer Vision–ACCV 2012. Springer Berlin Heidelberg,* 2013.

[30] 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.

[31] 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.

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

[33] 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.

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

Murad Qasaimeh received his B.S. degree in Computer Engineering from Jordan university of science and technology (JUST) in 2011, and his M.Sc. degree in Computer Engineering in 2014 from the American University of Sharjah (AUS), UAE. From 2014 to 2015, he was a research associate at Semiconductor Research Corporation (SRC) Khalifa University (KU). His research interests include image and video processing, parallel hardware architectures design, design hardware accelerators, FPGA/ASIC design, System on Chip (SoC) design, and real-time embedded systems design.

Assim Sagahyroon received the M.Sc. degree from Northwestern University, Evanston, IL, USA, and the Ph.D. degree from the University of Arizona, Tucson, AZ, USA. From 1993 to 1999, he has been a member of the Department of Computer Science and Engineering at Northern Arizona University, and then joined the Department of Math and Computer Science, California State University. In 2003, he joined the Department of Computer Science and Engineering at the American University of Sharjah where he is currently a Professor and Head of the Department. His research interests include power consumption and testing of VLSI designs, hardware description languages, FPGAs, computer architecture, and innovative applications of emerging technology.

Tamer Shanableh received his Ph.D. in Electronic Systems Engineering in 2002 from the University of Essex, UK. From 1998 to 2001, he was a senior research officer at the University of Essex, during which, he collaborated with BTexact on inventing video transcoders. He joined Motorola UK Research Labs in 2001. During his affiliation with Motorola, he contributed to establishing a new profile within the ISO/IEC MPEG-4 known as the Error Resilient Simple Scalable Profile. He joined the American University of Sharjah in 2002 and is currently an associate professor of computer science. Dr. Shanableh spent the summers of 2003, 2004, 2006, 2007 and 2008 as a visiting professor at Motorola multimedia Labs. He spent the spring semester of 2012 as a visiting academic at the Multimedia and Computer Vision and Lab at the School of Electronic Engineering and Computer Science, Queen Mary, University of London, London, U.K. His research interests include digital video processing and pattern recognition.