Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2011

MSc THESIS

Real-Time Object Identification using SURF Key-Points

Vikram Shivanna

Abstract

CE-MS-2011-35

The thesis here addresses the topic of image features and how they can be used in object identification. Two state of the art algorithms Scale Invariant Feature Transform (SIFT) and Speeded-Up Robust Features (SURF) are studied and their qualities are measured and based on the results of these tests the best algorithm, SURF, is chosen for building a real-time object identification application. The application is expected to run on an ARM Cortex-A8 based embedded processor platform known as i.MX515EVK. Being a computation intensive algorithm and due to limited hardware resources several optimization strategies were applied on the algorithm to bring up the speed, namely, Algorithmic Optimizations, Implementation Optimization and Application Optimization. Special emphasis is given to the SIMD unit of the Cortex-A8 core known as NEON, in fact the major contributor in bringing up the speed of the algorithm is due to extensive usage of NEON. Taking the most effective version of SURF algorithm implementation a real-time Euro currency notes identification application is built. Experiments are conducted to show how it is feasible for the application to be resilient to changing scale, illumination, blur and orientation conditions and still identify currency notes from image frames at a rate of 3.5 - 4 frames per second.

Faculty of Electrical Engineering, Mathematics and Computer Science

Real-Time Object Identification using SURF Key-Points

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Vikram Shivanna
born in Hindupur, India

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Real-Time Object Identification using SURF Key-Points

by Vikram Shivanna

Abstract

The thesis here addresses the topic of image features and how they can be used in object identification. Two state of the art algorithms Scale Invariant Feature Transform (SIFT) and Speeded-Up Robust Features (SURF) are studied and their qualities are measured and based on the results of these tests the best algorithm, SURF, is chosen for building a real-time object identification application. The application is expected to run on an ARM Cortex-A8 based embedded processor platform known as i.MX515EVK. Being a computation intensive algorithm and due to limited hardware resources several optimization strategies were applied on the algorithm to bring up the speed, namely, Algorithmic Optimizations, Implementation Optimization and Application Optimization. Special emphasis is given to the SIMD unit of the Cortex-A8 core known as NEON, in fact the major contributor in bringing up the speed of the algorithm is due to extensive usage of NEON. Taking the most effective version of SURF algorithm implementation a real-time Euro currency notes identification application is built. Experiments are conducted to show how it is feasible for the application to be resilient to changing scale, illumination, blur and orientation conditions and still identify currency notes from image frames at a rate of 3.5 - 4 frames per second.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2011-35

Committee Members :

Advisor:	dr.ir. S.D. Cotofana, CE, TU Delft
Advisor:	ir. Joost Mans, Philips Applied Technologies, Eindhoven
Chairperson:	dr.ir. S.D. Cotofana, CE, TU Delft
Member:	dr.ir. A.J. van Genderen, CE, TU Delft
Member:	dr.ir. René van Leuken, CAS, TU Delft

I would like to dedicate this work to my parents and friends.

Contents

List of Figures

List of Tables

Acknowledgements

I express my gratitude to Joost Mans from Philips Applied Technologies and Sorin Cotofana from CE group of EEMCS department at TU Delft, for accepting to guide me during this project work. Their timely and valuable suggestions helped me do better work and complete the project work and the thesis writing in time.

Next, I thank Pieter-Jan Kuyten at Philips Applied Technologies with whom I used to have extensive discussions about the project work in particular and in general about the exciting Computer Vision Systems.

I thank Hong Liu, Ying Zang, Wilco Boeije and Wilco Thissen, my other colleagues at office, for giving their valuable suggestions whenever I approached them.

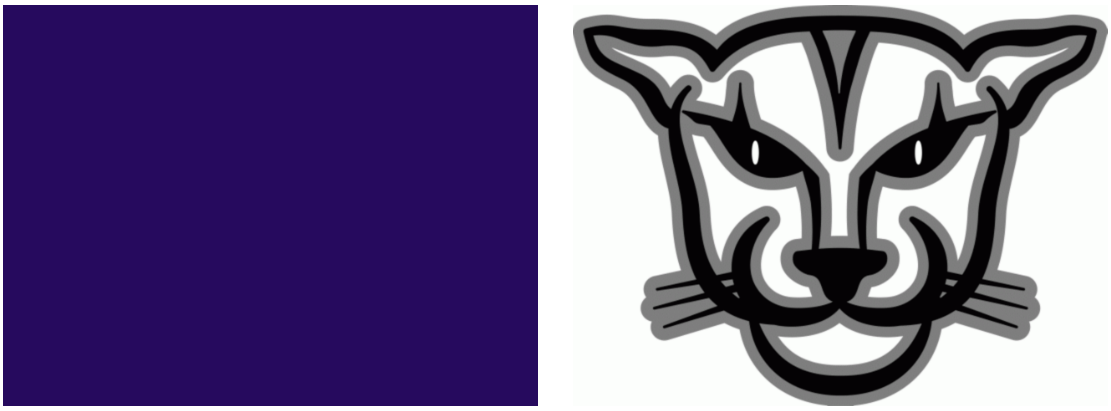
Lastly, I thank Philips Applied Technologies for offering this internship and financially support me during the work.

Vikram Shivanna
Delft, The Netherlands
February 6, 2011

1 Introduction

Extraction of information from data such as text, audio, images, video has been studied for several decades by now. In the field of computer vision, image features play an important role and they are as crucial as keywords or frequencies in text and audio data, respectively.

Extensive studies have been conducted over several decades now, to learn how the human brain gets stimulated by variations in types of objects [6] [7]. From a glance at Figure 1.1(a), our brain perceives that the image is uniform, while on the other hand when we look at Figure 1.1(b) our brain becomes sensitive to the variations in color, such variations in color stimulate the human brain differently which makes us recognize them. So, based on how humans perceive visual features, image features may be seen as variations in pixel values across an image. In digital imaging a pixel is the smallest unit of picture.

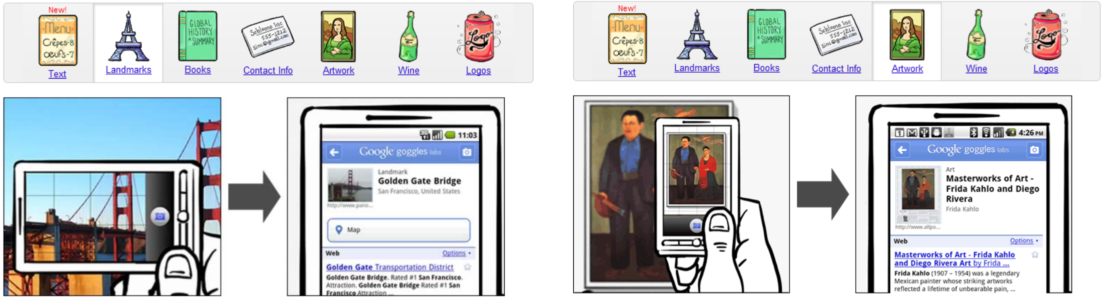


(a) Image with no variations in pixel information. *(b) Image with variations in pixel information.*

Figure 1.1: *Example images depicting features to naked eye.*

Feature extraction from images forms a crucial branch in computer vision research. Image features are the information content obtained from a large chunk of pixel data. These days, Digital cameras and other hand-held devices such as mobile phones with cameras are expected to have several functionalities such as - face recognition, text recognition, corner detection, gesture recognition, image stitching used in panorama view generation etc. To achieve such functionalities the image features are used extensively.

To understand how the features from images are utilized, let us consider *Google Goggles* software application as an example. *Google Goggles* [8] is an Android Operating System(OS) based application dedicated to on Web information searching; it requires a


New!
Text
Landmarks
Books
Contact Info
Artwork
Wine
Logos
Google goggles labs
Landmark
Golden Gate Bridge
San Francisco, United States
Map
Web
Options
Golden Gate Transportation District
Art
Masterworks of Art - Frida Kahlo and Diego Rivera
Frida Kahlo
Masterworks of Art - Frida Kahlo and Diego Rivera Art by Frida ...

(a) Google Goggles used to get information about a landmark.

(b) Google Goggles used to get information about an artwork.

Figure 1.2: *Information about Google Goggles application.*

mobile phone with a camera and an internet connection. Until now, the only options for web searching is by typing Key-words or to an extent by speaking words, but with *Google Goggles* application, one can do a search on the internet using images. Refer to Figure 1.2 for simple demonstrations showing how *Google Goggles* can be used.

An application which makes use of image features is ***Image Stitching***, which takes into account the common overlapping views from multiple images and builds a final image with larger dimensions, compared to each of the images that were used to build it. *SharpStitch* [9] is such an application, which is used for stitching two images. The Figure 1.3(c) a stitched image, is the output from *SharpStitch* and it was obtained by stitching two images 1.3(a) and 1.3(b) having overlapping views. Image stitching is fast becoming a standard feature in digital cameras [10] [11].

Some other applications which also make use of image features are:

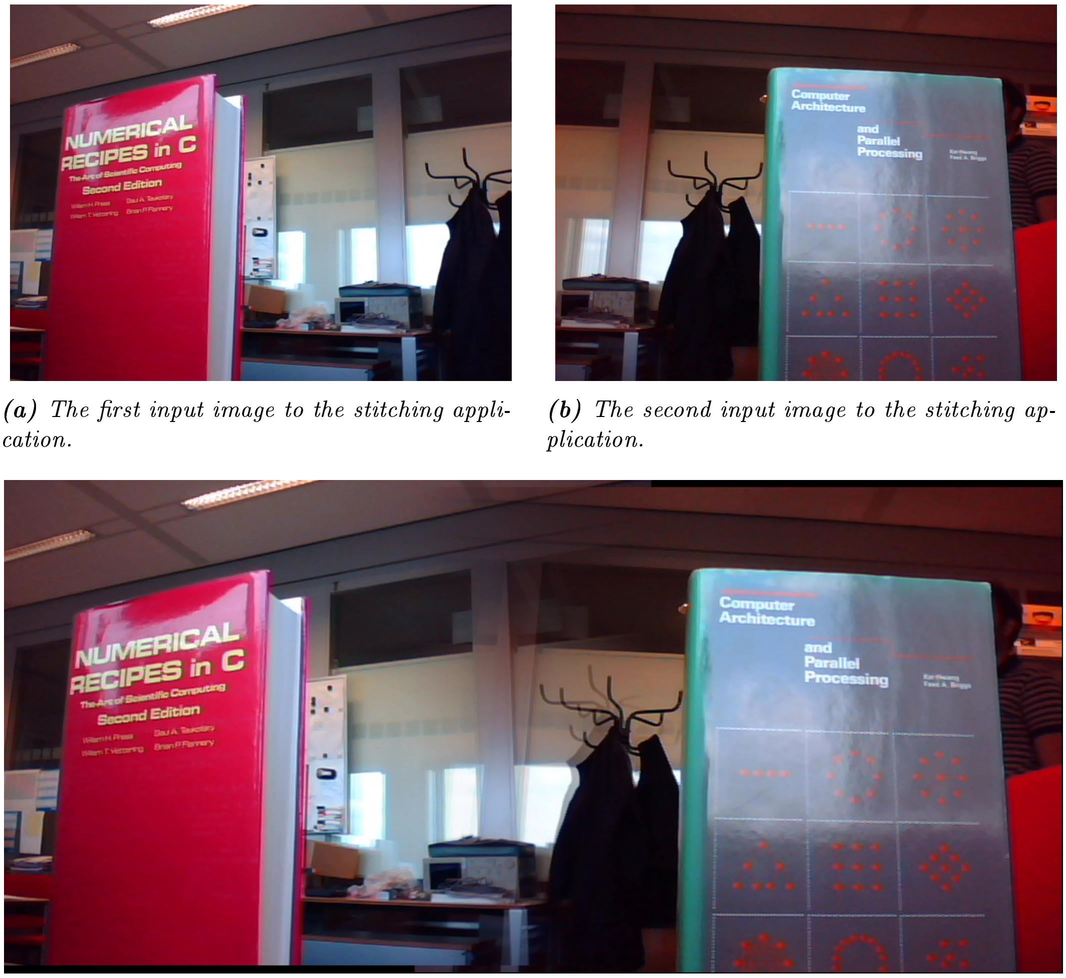
- Motion Detection [12],
- Stereo Vision based 3D modelling [13],
- Video Tracking [14],
- Gesture Recognition [15].

There are several feature extraction algorithms, the most common ones can be classified into the following groups, this classification is made mainly based on the nature of the features they extract.

- Blobs or Interest Points or Key-Points:

  Interest points (see Figure 1.4(a)) are pixels that capture significant local features of an image, and they are usually located around corners and edges of images[16]. Feature extraction algorithms which are based on this principle are:

  - Scale-Invariant Feature Transform (SIFT) [18]
  - Speeded Up Robust Features (SURF) [19]



(a) The first input image to the stitching application.

(b) The second input image to the stitching application.

(c) The final stitched output image obtained from the stitching application.

Figure 1.3: *SharpStitch Image Stitching Application*

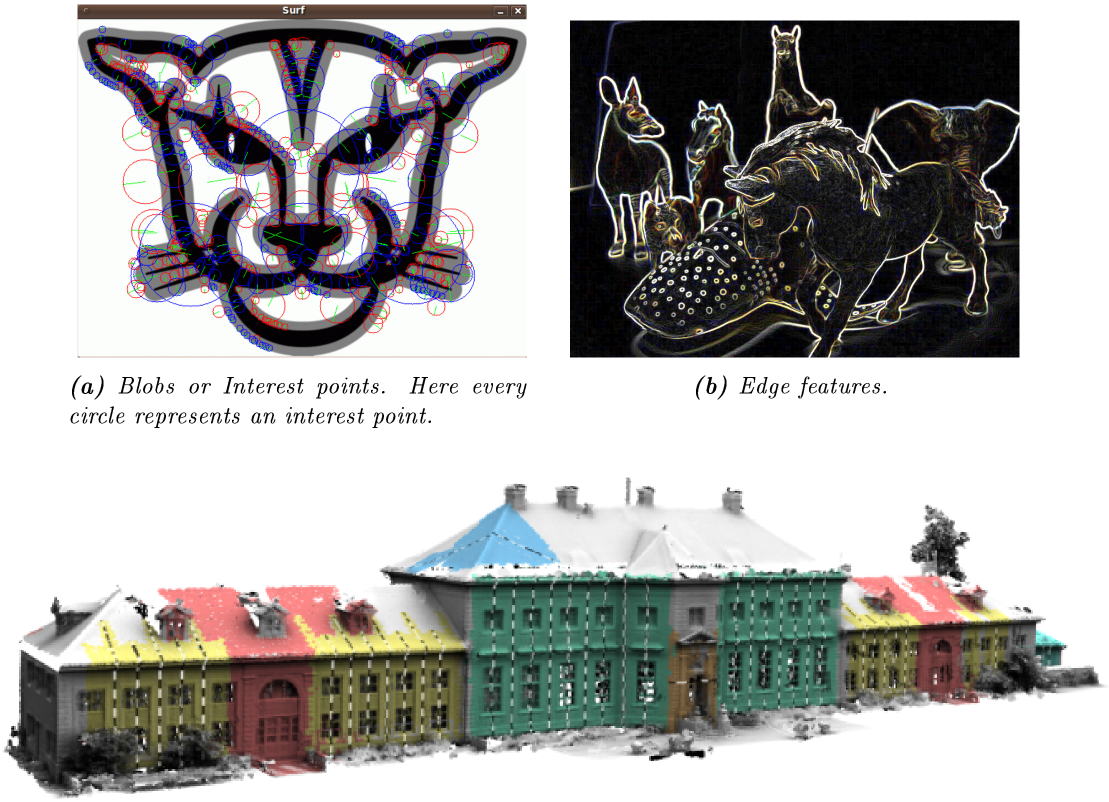
- Edges:

  Edges (see figure 1.4(b)) are image features which are sharp discontinuities in pixel values. Feature extraction algorithms which are based on this principle are:

  - Laplacian of Gaussian (LoG) [20]
  - Canny Edge Detector [21]
  - Harris Corner detector [22]

- Lines:

  While edges and general curves are suitable for describing the contours of natural objects, the man-made world is full of straight lines as one can observe in 1.4(c).



(a) Blobs or Interest points. Here every circle represents an interest point.

(b) Edge features.

(c) Symmetry Detection using line features, borrowed from [17].

Figure 1.4: *Different types of image features*

Detecting and matching these lines can be useful in a variety of applications, including architectural modelling, pose estimation in urban environments, and the analysis of printed document layouts [23]. Feature extraction algorithms which are based on this principle are:

– Image editing in the contour domain [24].

– Sinha et al's - Interactive 3D Architectural Modelling from Unordered Photo Collections [25].

Choosing a particular feature extraction algorithm depends on the nature of the problem which needs to be solved. For example, in this project work, a **real-time object identification** application needs to be built, which is targeted for an Embedded Platform with a camera. The problems unique to such a system, which works in real-time, are the constant changes in **scale**, **blur**, **orientation**, **illumination** and **viewpoint**. Such changes can be called as *disturbing conditions* and compared to edges or line features, which are not flexible to such changes. Key-Points are better at handling the changing conditions because they are expressed by what are known as *descriptions*, which describe a point and its surroundings. Key-Points generated by algorithms such as SIFT and SURF, have descriptions which are expressed in 64 floating point values. The Key-Point feature extraction algorithms are flexible or invariant to large changes to most of the disturbing conditions mentioned above, as such, even under changing conditions they produce stable Key-Points. The changing conditions, stable Key-Points and how they can be used are well demonstrated in Figure 1.5.

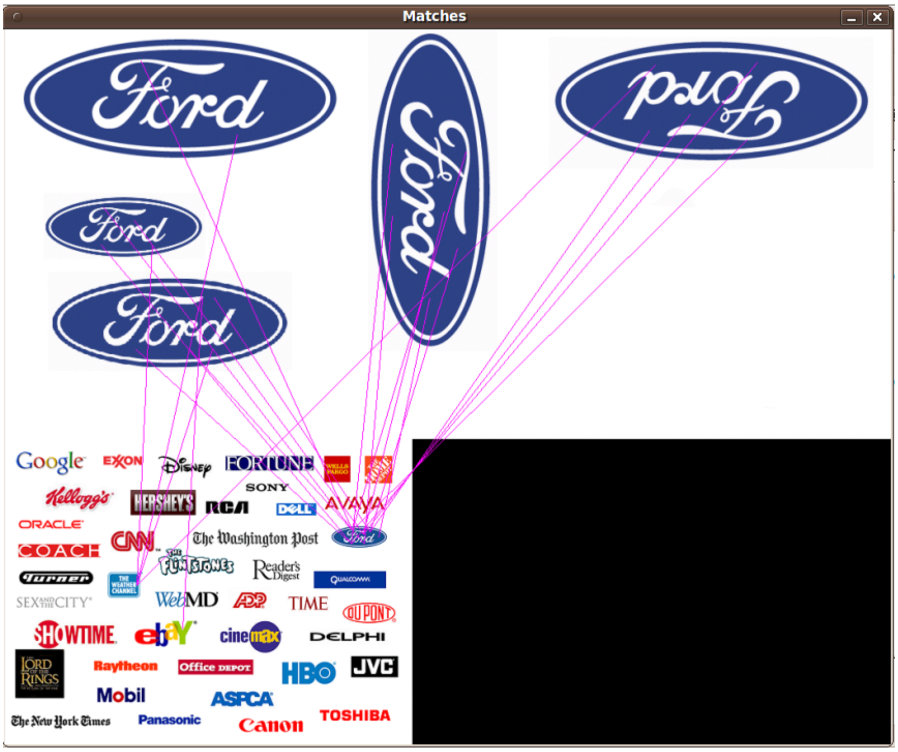


Figure 1.5: *The top part of this figure shows a logo with disturbances such as Scaling and Orientation. The top part of the image and bottom part of the image were given to the SIFT feature extraction and matching algorithm. The lines(pink) represent the matches found by the algorithm.*

The Figure 1.5 was a resultant image obtained from a SIFT feature extraction and matching implementation, where the top part of the image has a logo with several disturbances, such as:

- Scaling - There are 75% and 50% scaled variants of the logo.
- Orientation - There are variants which are at 90°and 180°orientation.

And, the bottom part of the image is the given image in which the logo has to be searched for.

Both images are given to a SIFT feature extraction and matching algorithm. The pink lines are the results which represent the matches obtained and out of 23 matches found 5 are false.

As shown in the above example, by identifying the nature of the problem which needs to be solved, one can choose the kind of feature extraction algorithm that best suites the problem.

And as always, another important feature which has to be kept in mind while choosing an algorithm for a particular application is its **Processing Time**.

In the case of *Goggle Goggles* application, the object identification is carried out on a remote system which might be very powerful with large computing resources and uninterrupted energy resources, in which case, a high precision algorithm could be used. On the other hand, in applications such as real-time object identification, where the user expects to get instantaneous response from the system, **a good frame rate becomes a necessity** and achieving it becomes a formidable challenge especially when it is intended to run in an environment with fewer resources and **limited energy resources**.

To corroborate this, from the tests conducted during the study it was found that to extract around 160 Key-Points, the SURF algorithm implementation running on a Linux/Pentium platform, i.e, Intel Pentium 4 CPU @ 2.80GHz with 1.49GB RAM, it would take **500ms**, in other words with this processing time, it could handle **2 frames per second**. For the same algorithm, to extract the same number of Key-Points, but running on a Linux/ARM platform, i.e., ARM Cortex-A8 CPU @ 800Mhz with 512MB RAM, the platform on which the real-time object identification application is intended to run, took **1280ms** or in other words, with this processing time, it could only handle **0.78 frames per second**.

Therefore, in the purview of such complex and time consuming algorithms on one hand, and low computing resources on the other hand, the two classic Embedded Real-Time System constraints, achieving a good frame rate indeed becomes a formidable challenge. Solving this problem by making use of techniques such as Algorithmic Optimizations, Algorithm Implementation Optimizations in the context of the hardware architecture, and Algorithm Usage Optimizations, forms the central research problem of this project.

1.1 Research Goals

The platform chosen for this project is Freescale's i.MX515EVK, some of the important features [1] of this platform, which are relevant to this project are listed below:

- CPU: Freescale's **i.MX515** at a clock speed of 800 MHz.
- Memory: 512 MB DDR-RAM.
- Ubuntu 9.10 OS
- *For the entire list of peripherals refer to Appendix B: i.MX515EVK Details 7.2.*

The primary research goal of this project is to determine how, a **fast** and yet **robust real-time object identification** software application can be built on the i.MX515EVK

[1] For the functional block diagram of the i.MX515EVK refer to Appendix B: i.MX515EVK Details 7.2.

embedded platform. Three of the most desired characteristics expected from any Embedded Software Application are its high speed of operation, less memory utilization and less energy consumption. Once the application is operational on the embedded platform, its speed of operation, its memory and energy utilizations are also measured.

As a first step in this direction, a proper type feature extraction algorithm has to be chosen. To this end the following are required:

- A comparative study is conducted to understand the principle and nature of two feature extraction algorithms, namely, SIFT and SURF.
- Based on this study, a decision is made on which algorithm is best suited to be used in building a real-time object identification software application.

After choosing a particular algorithm,

- The Embedded Platform and the available resources on it have to be studied.
- The objective here is to achieve a real time object identification, so based on the knowledge of the algorithm and the hardware platform, the algorithm can be optimized for speed. To achieve this, the resources of the processor have to be put to best use.
- Once a most effective version of the feature extraction algorithm is ready. The following steps are carried out:
  - A software application is built, which incorporates the optimized feature extraction algorithm and a Key-Points matching algorithm. Which can be used in identifying objects at real-time.
  - Once this application is operational, its speed of operation, its memory and energy utilizations are measured.

1.2 Thesis Contributions

The main contributions of this thesis can be summarized as follows:

- **Comparative Study:** As explained in the introduction of this chapter, from a given image, several kinds of image features can be extracted, but it is Key-Points which are the best kind of features which enables object identification even under image distortions. The two popular Key-Points extraction algorithms are SIFT and SURF, both have their own properties, in order to choose an algorithm, by keeping the possible image disturbances such as scaling, orientation, blurring, illumination and viewpoint changes are parameters, both SIFT and SURF are put to test to find out how well they can handle them. The two input images given to both the algorithms are (a) a benchmark image which is constant and (b) an image applied with some deformation. The metric chosen for these tests is repeatability. Repeatability is a ratio which indicates how well the Key-Points from two images match. From these tests it was found that while SIFT can handle Scaling, Orientation and

Viewpoint changes better than SURF; SURF on the other hand can handle the Blurring and Illumination conditions better than SIFT.

Apart from the disturbing conditions as the parameters on which the performance of both the algorithms are measured, another important characteristic expected from both algorithms is their processing time, which is a very important characteristic needed in an algorithm to achieve the goal of this project i.e., a real-time object identification. In this case SURF is a clear winner. While, the average time needed by SIFT to compute features from 33 images was **1.4s**, the average time needed by SURF to compute almost same number of Key-Points from same set of images was only **0.26s**.

Based on the results from these tests, the SURF algorithm is chosen over SIFT, the most influential result which favoured this choice was the less processing time.

- **Algorithm Optimizations:** The algorithmic optimization was the first step taken in reducing the computation time. In this project a statistical study was conducted and the algorithmic optimization was made based on the results of that study. During the study it was found out that, out of a given number of Key-Points computed by SURF algorithm, more than 80% of them are extracted from the first two Octaves alone. Upon studying the algorithm, it was found that, the large filter sizes and higher stepping values, accelerate the movement of box filters towards the end of the image. Which implies that, fewer computations are made in these higher Octaves, which in turn result in extracting fewer Key-Points from them. Therefore, the algorithm was modified to restrict from computing the Key-Points from Higher Octaves. By cutting down these computations, nearly **80 - 280ms** of processing time was improved for images with 640 × 480 pixels dimension and **20 - 100 ms** of processing time was improved for images with 320 × 240 pixels dimension.

- **Implementation Optimizations:** The second step in optimization for speed was achieved by making use of the Cortex-A8's SIMD unit known as NEON.

  The SURF method of computing Key-Points takes significantly lesser amount of time when compared to SIFT. The reduction in processing time can be attributed to the nature of filters used in building the Scale-Space. In SURF, Box filters are used to build the Scale-Space. The majority of the time spent in SURF algorithm is in computing the Box Filter responses, from the profiler report generated it was found that the total number of box filters used in SURF Algorithm were **3,383,284** [2] which accounts to nearly 70% of SURF algorithm's time. Improvement in processing time can be achieved only by somehow reducing the amount of time spent in box filter response computations.

  Box filter responses can be computed very efficiently (with only 3 additions) by making use of Integral Images. By taking advantage of the integral image and by making use of the interleaving instructions of NEON innovatively the number of box filters needed in Scale-Space building was brought down by **4 times** i.e., from

[2] For 640 × 480 size image.

2,658,304 to only 845,821. This optimization was the most significant contributor in improving the processing time needed by the algorithm.

In this project the box filter response computations were carried out innovatively by making use of NEON. This implementation brought down the number of box-filters used in the algorithm by 4 times, which is the most significant contributor in improving the processing time needed by the algorithm.

The next function which could be vectorized using NEON was Integral Image computation. Unlike box filter computation which could be completely vectorized, due to the datatype changes in Integral Image NEON could not be used to completely vectorize it. As a result of which an intermediate step was used to make the datatypes homogeneous and then the data was given to NEON. Although this intermediate step costs more cycles because it is accessing image data serially there was still a significant speed up was achieved due to NEON, $105.3ms \rightarrow 81.63ms$. As the image data was being fetched one-by-one serially, block transfer of data was carried out by using NEON, a concept known as data pre-fetching, this optimization brought further optimization to Integral Image computation, $81.63ms \rightarrow 66.86ms$.

Parts of other functions such as descriptor computation and matrix multiplication were also optimized in the similar way, but no significant improvement in processing time were observed.

By this step of Vectorization of some of the key time consuming functions, the computation time needed to extract Key-Points for images of 320×240 dimensions was brought down to 300 - 350ms, a gain of 41 - 46%, which translates to ~3fps.

- **A Real-Time Object Identification application:** Once an optimum version of SURF is implemented, it was used to build a Real-Time Euro currency notes Identification application. The objective of this application is to continuously capture image frames from a camera and identify the 7 Euro currency notes irrespective of the face in which they are presented in front of the camera. Key-Points from the $7 \times 2 = 14$ sides of images are extracted on a one-by-one fashion and they are stored within the application. When Key-Points from input image frames are extracted they are given to a matching algorithm which carries out a matching process between the freshly extracted Key-Points with the Key-Points within the application.

    - **Matching Algorithm:** The matching algorithm is based on Root of Sum of Squares (RSS) which gives a cumulative difference value between descriptions of two Key-Points, the Key-Points having the least difference in their descriptions are considered as a matching Key-Point pair.

      Two of the new functions $COLOR \rightarrow GRAYSCALE$ conversion and **Key-Points matching algorithms** introduced in the application were also vectorized using NEON. While implementing the matching algorithm it was found that, crucial part of that algorithm can be vectorized. The NEON vectorized matching algorithm takes nearly 56% lesser computation time to complete matching of 71 Key-Points with a collection of 501 Key-Points.

- **Progressive Matching:** An innovative idea which is called as progressive matching was used for the first time in this project. While studying the SURF algorithm it was learnt that the Key-Points extracted by SURF in each Octave are unique in nature, the motive behind progressive matching arises from this study. The idea here is, when the Key-Points extracted from an Octave are provided to matching algorithm and if it is able to extract enough number of matching Key-Point pairs to decide that the two input images match or not, further Key-Point computations are not needed any longer and processing of next frame can be started instantly. During the literature study conducted, it was found that no where in any of the existing object-identification applications is this method being used. The contribution of this approach was a significant amount of reduction in computation time when considerable number of image frames were matched by the end of first Octave. In one of the trials it was found that the total time taken for processing 100 frames was reduced by nearly 29% due to progressive matching.

By the end of progressive matching the average frame rate of the real-time object identification application was around 3.5 - 4 frames per second.

1.3 Thesis Organization

The reminder of this thesis is organised as follows:

Chapter 2 starts with a brief explanation on how SIFT and SURF, two Key-Points extraction algorithms work. Taking important parameters, such as - scale, blur, orientation, illumination, viewpoint and processing time into consideration, comparative tests are subsequently presented to asses how well the two algorithms perform on various inputs and conditions. Lastly, based on the outcome of the comparative study, a choice is made between the two considered algorithms.

Chapter 3 covers the four steps in extracting the Key-Points using the SURF algorithm and it also introduces to how and why the first optimization, the algorithmic optimization was carried out.

Chapter 4 covers briefly the details and the features of the i.MX515EVK and the tools used for implementing the chosen algorithm. At the core of the i.MX515 processor is an implementation of the ARM Cortex-A8. The Cortex-A8 has an SIMD co-processor known as NEON, special attention is paid on NEON to see how well it can be utilised for optimizing the algorithm.

Chapter 5 covers the details as to how the chosen Key-Points extraction algorithm is used in building a Real-Time object identification application. It also addresses a new algorithm known as matching algorithm, which makes a decision if the two sets of Key-Points provided to it match or not, if they match, it indicates that the two images from which they were extracted are same. It then introduces to a new method with which SURF Key-Points can be used for matching, which is called as Progressive matching in this project. It also covers the details of Camera and OpenCV - library of programming functions for real-time computer vision. Lastly, performance measurements, such as -

how well the application can handle optical deformities, its processing time, its memory requirements and also its energy requirements.

Chapter 6 covers conclusions of the over all work done in this project work and also recommendations for future work.

Key-Point Feature Extraction Algorithms 2

This chapter covers the first important research goal of the project - a comparative study between the SIFT and SURF algorithms.

Before going straight into the comparison of these two algorithms, it is important to know certain details about Key-Point Feature Extraction algorithms in general, **S**cale **I**nvariant **F**eature **T**ransform (SIFT) and **S**peeded-**U**p **R**obust **F**eatures(SURF) algorithms in particular, which is covered in the following section.

2.1 Key-Point Feature Extraction Algorithms

In any image, the edges and corners are the most likely locations where Key-Points may be found. Hence, as a first step in Key-Point detection, it is important to identify the edges and corners in a given image. To achieve this task, the Key-Point feature extraction algorithms are used as **edge or corner detectors**.

2.1.1 Edge or Corner Detectors

The Edge or Corner detectors, detect the regions of rapid intensity changes in pixel information. The **L**aplacian **of G**aussian filter (LoG or Marr Filter) is one of the very early and common edge detectors. The LoG filter works in two steps: **(a)** the image is blurred using a Gaussian filter, which removes the noise contents in the image and **(b)** A Laplacian Transformation is then applied to the Gaussian blurred image (a second order derivative) to highlight the edges and corners. The intermediate blurring step is very important here because the Laplacian operator is sensitive to noise, so the blurring action smooths out the noise content in the image.

'Scale invariance' is one of the most important characteristic of Key-Point features. *Scale invariance is a feature of objects or laws that do not change if length scales (or energy scales) are multiplied by a common factor* [26]. In order to get scale invariant Key-Points, a Scale-Space needs to be built. To understand Scale-Space, refer to Figure 2.1, which is a pictorial representation of how a Scale-Space is built and looks. In the figure, for the input image at Scale 1 the LoG filter is applied at four increasing blurring values, which forms the first Octave (Octave 1). Once again at Scale 2 the LoG is applied at four new increasing blurring values and this forms the second Octave (Octave 2). Similarly, the third Octave (Octave 3). An important point to note here is that, the filter blurring values are not only increasing within an Octave, but they also increase across the Octaves. The entire collection of such responses is known as **Scale-Space**.

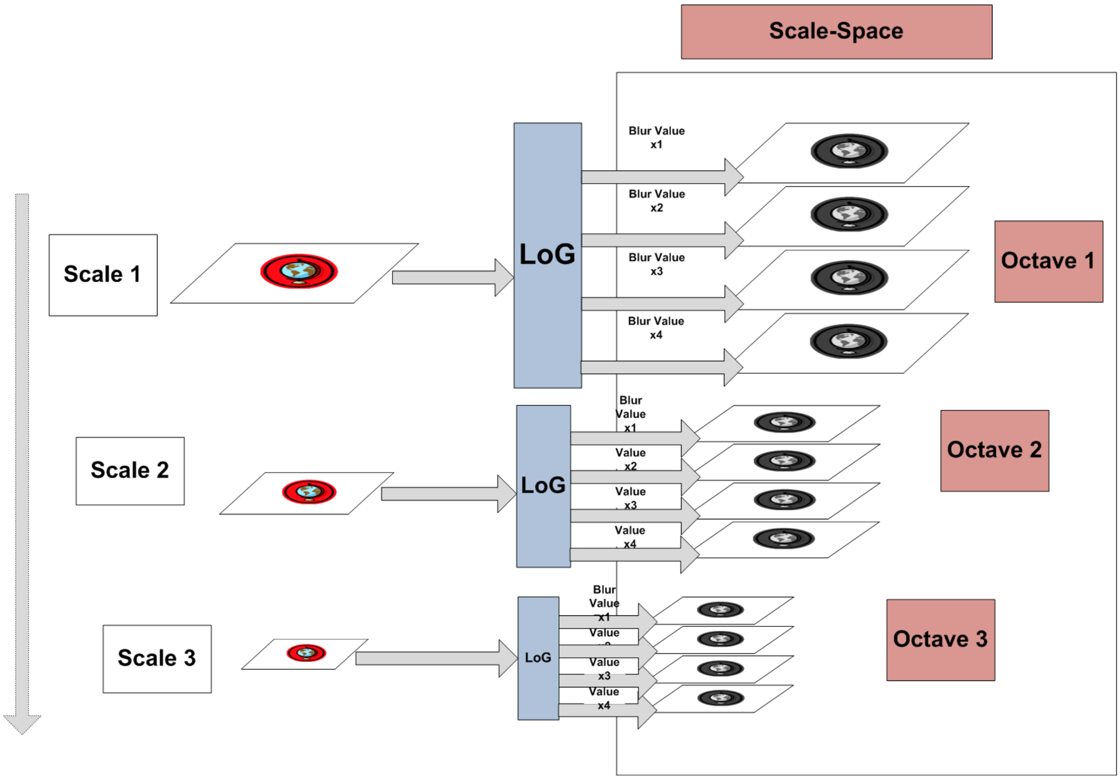


Figure 2.1: *A representation of the Scale-Space obtained by applying the LoG filter at different scales of the input image.*

The series of response-layers as seen in the Scale-Space, consist of highlighted edges and corners extracted from the input image. In the [27], a thorough study was conducted, which concludes that, better Key-Points are generated upon using the normalized LoG filter than by other methods of edge detection such as - Gradient or Hessian or Harris Corner function.

However, despite being one of the best methods for edge detection, one major drawback in using LoG is that it outweighs its exceptional characteristic of high **computation time**.

2.1.2 Scale Invariant Feature Transform or SIFT

David Lowe, from the University of British Columbia, proposed the SIFT algorithm [18]. In his novel approach, he shows that a **D**ifference **of G**aussian (DoG) approach, instead of Laplacian of Gaussian can also be used for edge detection. The DoG is an approximation of LoG, but, as Lowe states - *"...the approximation has almost no impact on the stability of extrema detection..."*, where extrema are the possible location of Key-Points. The reason for this approximation is mainly to bring down the computation time for building the Scale-Space.

Scale-Space built using the DoG is as shown in Figure 2.2. In this example, only one

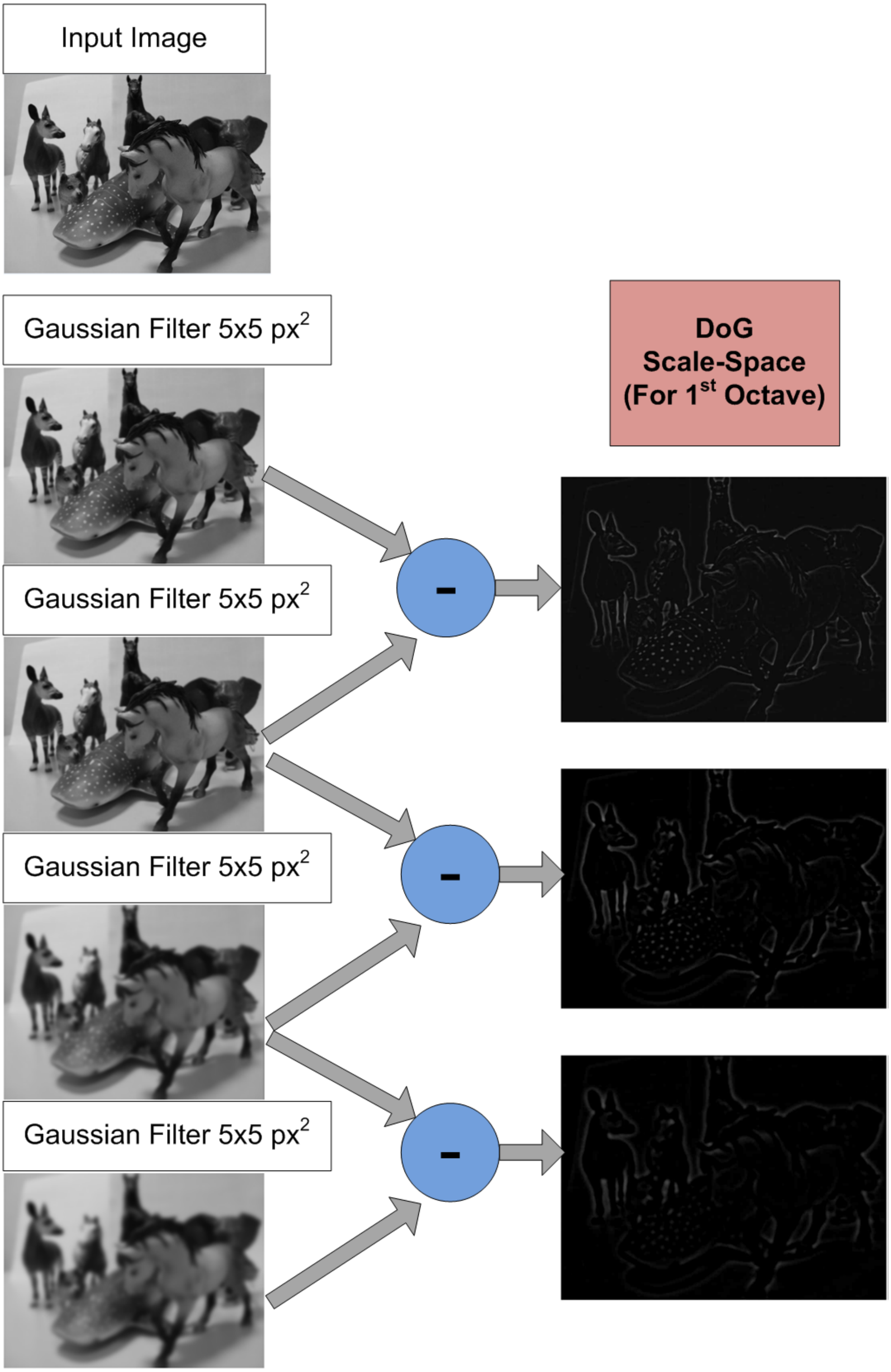

Input Image
Gaussian Filter 5x5 px^2
Gaussian Filter 5x5 px^2
Gaussian Filter 5x5 px^2
Gaussian Filter 5x5 px^2
DoG Scale-Space (For 1st Octave)
-
-
-

Figure 2.2: *A representation of the Scale-Space obtained by applying the DoG filter at different scales of the input image. Here only one Octave is considered. On the LHS, the given image is successively blurred with a Gaussian Filter of given size. On the RHS, the DoG of consecutive response layers forms the Scale-Space.*

scale (One Octave) is considered, however, in reality the input image is scaled several times (Several Octaves), after which the Gaussian Blurring of increasing filter sizes are applied across each Octave. The difference of two such consecutive response layers (or DoG), becomes a part of the Scale-Space.

Lowe's SIFT algorithm has four stages:

Stage 1: Scale-Space extrema detection

The first stage is computing the potential interest points, across all scales. This is accomplished by building the Difference-Of-Gaussian (DoG) Scale-Space as shown in Figure 2.2. The points so identified are invariant to scale and rotation.

Stage 2: Key-Point localization

The feature point detector only provides a set of possible feature point. A detailed model of all the Key-Points are created by localising to the nearest pixel and scale, where the Key-Points were found. Key-Points which are resistant to image distortions are selected.

Stage 3: Orientation assignment

Based on the local image gradients, SIFT will then compute and assign to each of the stable Key-Points one or more orientations.

Stage 4: Key-Point Descriptor

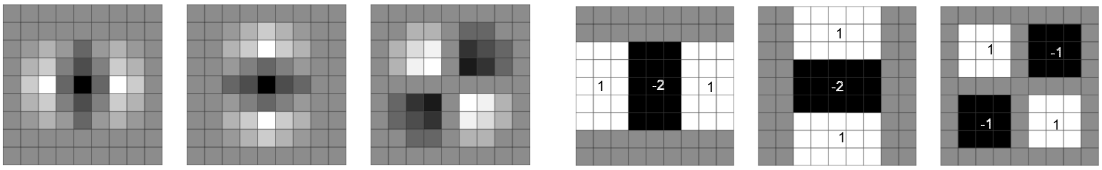
Around each Key-Point's locality, image gradients are measured and these measurements are transformed into a descriptor which has tolerance for shape distortion and illumination changes.

One important point to be mentioned before closing this section is - the Gaussian filter sizes and their contribution to the computation time. As mentioned earlier, in the SIFT approach the filter sizes increase with the Octaves. In any Octave, the entire given image is successively convoluted with a filter of a given size. When the Octaves change, the filter sizes also change. The higher the octave, the larger the filter size, this increases the number of computations and hence the computation time of the convolution operation. In other words, the computation time for the convolution of an entire image with the Gaussian filter of size $rows_1$, $cols_1$ is less than that for the same operation with a Gaussian filter of size $rows_2$, $cols_2$, where ($rows_1 < rows_2$) and ($cols_1 < cols_2$).

2.1.3 Speeded-Up Robust Features or SURF

Speeded-**U**p **R**obust **F**eatures (SURF) uses a different and a far more intensive filter approximation when compared with DoG approach in SIFT. Created by Bay et al [19] to compute the Key-Points in an image, in this method a **Box Filter** or Mean Filter responses are used. In the paper [19] it is claimed by the authors that, while building the LoG Scale-Space *"...the importance of the Gaussian seems to have been somewhat overrated..."*. So, they use a simpler alternative - the Box Filters computed using **Integral Images**. It is shown in the results of the paper that - for a given image, the number of feature points extracted using both DoG approach in SIFT and simple box filter approach in SURF are comparable.

The approximations achieved by box filters can be viewed in Figure 2.3, where Figure 2.3(a) shows a 9×9 discrete and cropped **Gaussian second order partial derivatives** in x, y, and xy-directions with the $\sigma = 1.2$ and Figure 2.3(b) shows their equivalent 9×9 **box filter approximations**.



(a) The 9×9 Gaussian second order partial derivatives in x, y and xy directions.

(b) The weighted 9×9 box filter approximation of the 9×9 Gaussian Filters, the 1's, -1's and -2's are the weights assigned to those regions and the grey regions are of value 0.

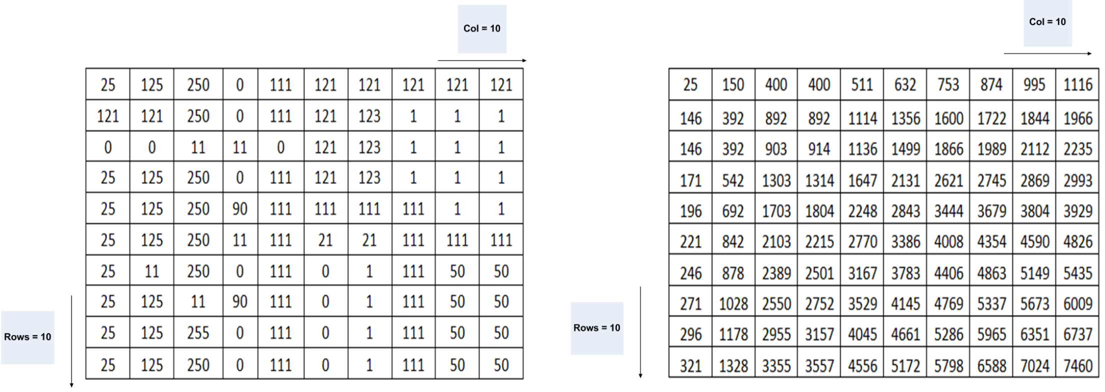
Figure 2.3: *Box filter approximations of the Gaussian Filters, from [1].*

2.1.3.1 Integral Images

The Box Filter responses can be computed very quickly by using an **Integral Image**. An integral image, which is also known as **Summed Area Tables**, has the same dimensions as the given image. In the integral image a value at any position (x, y) is the sum of all the pixel values above and to its left in the given image. It is computed with the Equation 2.1.1 and it can be well understood from Figure 2.4.

$$sum(x, y) = \sum_{x' \leqslant x, y' \leqslant y} i(x', y') \tag{2.1.1}$$

In SURF there are three types of filters used in the three directions **x**, **y** and **xy** as depicted in Figure 2.5. They are the approximations of the second order partial derivatives of Gaussian filter responses in LoG. In order to understand how to compute the responses of these box filter approximations, consider the D$_{xx}$ filter in Figure 2.5, in this, one must compute the area within the region ABCD and then subtract that area with the area covered by the region A'B'C'D'. These area computations can be easily carried out using the integral image, as demonstrated in the example Figure 2.6 and irrespective of the filter sizes every response can be computed with **only three additions**.


Col = 10
Rows = 10
Col = 10
Rows = 10

25	125	250	0	111	121	121	121	121	121
121	121	250	0	111	121	123	1	1	1
0	0	11	11	0	121	123	1	1	1
25	125	250	0	111	121	123	1	1	1
25	125	250	90	111	111	111	111	1	1
25	125	250	11	111	21	21	111	111	111
25	11	250	0	111	0	1	111	50	50
25	125	11	90	111	0	1	111	50	50
25	125	255	0	111	0	1	111	50	50
25	125	250	0	111	0	1	111	50	50

(a) Given image of Size 10×10.

25	150	400	400	511	632	753	874	995	1116
146	392	892	892	1114	1356	1600	1722	1844	1966
146	392	903	914	1136	1499	1866	1989	2112	2235
171	542	1303	1314	1647	2131	2621	2745	2869	2993
196	692	1703	1804	2248	2843	3444	3679	3804	3929
221	842	2103	2215	2770	3386	4008	4354	4590	4826
246	878	2389	2501	3167	3783	4406	4863	5149	5435
271	1028	2550	2752	3529	4145	4769	5337	5673	6009
296	1178	2955	3157	4045	4661	5286	5965	6351	6737
321	1328	3355	3557	4556	5172	5798	6588	7024	7460

(b) Integral Image of size 10×10*, formed from the given image.*

Figure 2.4: *Integral Image.*

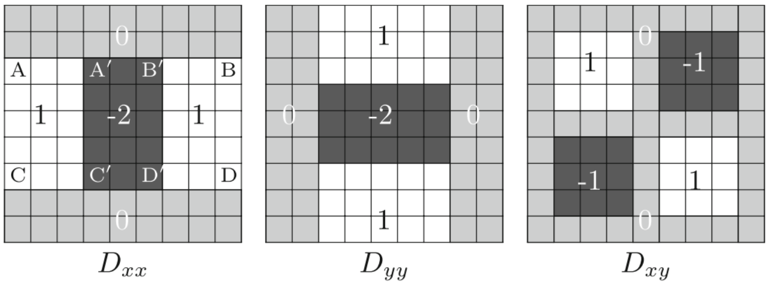
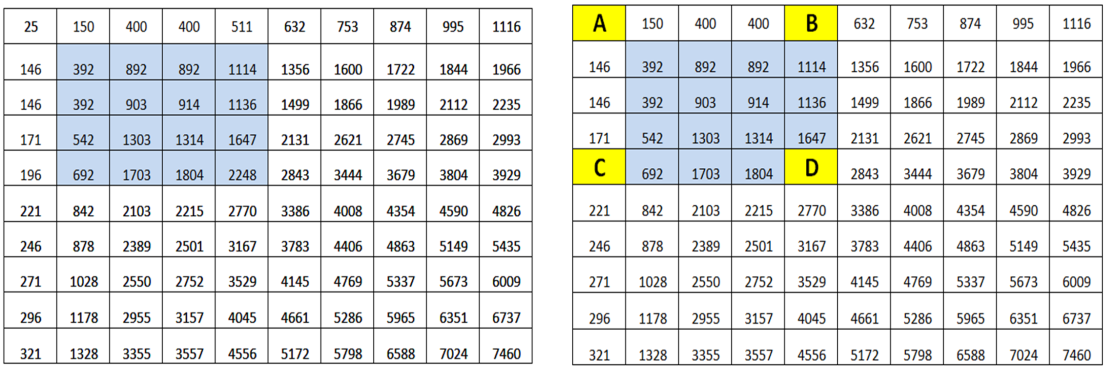


Figure 2.5: *The three Box Filter approximations,* D_{xx}*,* D_{yy} *and* D_{xy} *of the second order derivatives of Gaussian Filter Responses, from [2].*



25	150	400	400	511	632	753	874	995	1116
146	392	892	892	1114	1356	1600	1722	1844	1966
146	392	903	914	1136	1499	1866	1989	2112	2235
171	542	1303	1314	1647	2131	2621	2745	2869	2993
196	692	1703	1804	2248	2843	3444	3679	3804	3929
221	842	2103	2215	2770	3386	4008	4354	4590	4826
246	878	2389	2501	3167	3783	4406	4863	5149	5435
271	1028	2550	2752	3529	4145	4769	5337	5673	6009
296	1178	2955	3157	4045	4661	5286	5965	6351	6737
321	1328	3355	3557	4556	5172	5798	6588	7024	7460

(a) To compute the ***area*** *in the* 4×4 *region (blue region).*

A	150	400	400	B	632	753	874	995	1116
146	392	892	892	1114	1356	1600	1722	1844	1966
146	392	903	914	1136	1499	1866	1989	2112	2235
171	542	1303	1314	1647	2131	2621	2745	2869	2993
C	692	1703	1804	D	2843	3444	3679	3804	3929
221	842	2103	2215	2770	3386	4008	4354	4590	4826
246	878	2389	2501	3167	3783	4406	4863	5149	5435
271	1028	2550	2752	3529	4145	4769	5337	5673	6009
296	1178	2955	3157	4045	4661	5286	5965	6351	6737
321	1328	3355	3557	4556	5172	5798	6588	7024	7460

(b) The ***area*** *(blue region) = A - B - C + D =* ***25 - 511 - 196 + 2248 = 1566****.*

Figure 2.6: *Integral Image based box filter response computation.*

Just like in SIFT, even in SURF also filters of increasing sizes are used. But unlike Gaussian Filters used in SIFT, whose computation time depends on their sizes, in SURF, irrespective of the filter sizes the responses of the box filters are calculated in a constant amount of time. In other words, the computation time for the convolution of an entire image with the box filter of size $rows_1$, $cols_1$ is the same as that for the same operation with a box filter of size $rows_2$, $cols_2$, where $(rows_1 < rows_2)$ and $(cols_1 < cols_2)$. This is a predominant factor which makes SURF compute its features faster than SIFT.

Just like in SIFT, even SURF feature extraction algorithm has 4 stages - Scale-Space analysis, Key-Point localization, Orientation assignment, and Key-Point Descriptor generation. These four stages are explained in detail in the next Chapter 3.

2.2 The comparison between SIFT and SURF

In this section, a comparative study is conducted between the two very popular feature point extraction algorithms - SIFT and SURF, in order to find out which of them is most appropriate for achieving **fast** and yet **robust** Key-Point features which can be used in a real-time object identification application,

2.2.1 Parameters

As explained in Chapter , a real-time object identification application gives very good results, if **(a)** it can handle the disturbing conditions effectively and **(b)** it has very short **Processing Time**.

So, the SIFT and SURF algorithms are evaluated by taking into consideration the following parameters:

1. Scale
2. Blur
3. Orientation
4. Illumination
5. Viewpoint and
6. Processing Time

The comparison is made based on the results from the implementations of SIFT [28] and SURF [1] algorithms. To do this, first, a benchmark image is considered, on which all the disturbing conditions (Parameters 1 - 5) are applied at different levels. Then the undisturbed and the disturbed benchmark images are sent as inputs to both the applications; their performances are measured based on the obtained **repeatability** values.

For the detection stage, the repeatability is defined as - a ratio of the matching pairs obtained between the two input images (Benchmark, Disturbed) and the mean of the number of points extracted from the input images, and can be expressed as in Equation 2.2.1[29].

$$ratio = \frac{MatchingPairs}{Mean(Features_1, Features_2)} \tag{2.2.1}$$

Where,

- $MatchingPairs$ - Matching Interest Points between the Interest Points of Image 1 and Image 2.
- $Features_1$ - Number of Interest Points in Image 1 and
- $Features_2$ - Number of Interest Points in Image 2.

In the following figures, Figure 2.7 is a benchmark image and Figure 2.8 has a collection of all the images obtained by applying the disturbing parameters at different levels on the benchmark image. All these images are Gray Scale and are of 640 × 480 pixels dimension.

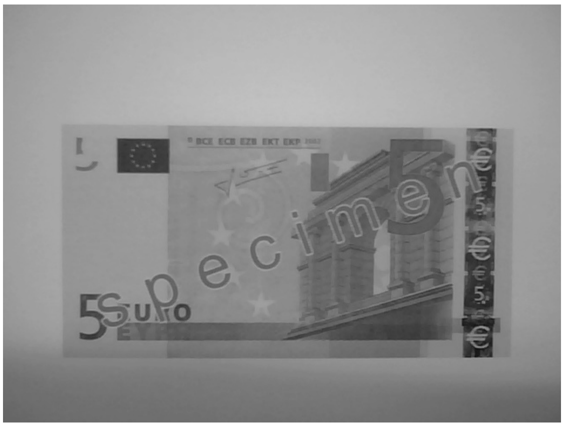
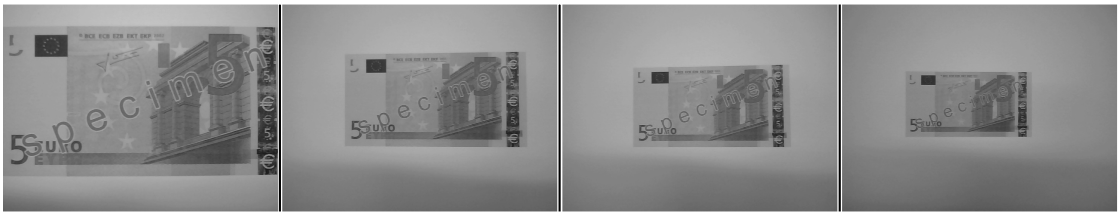
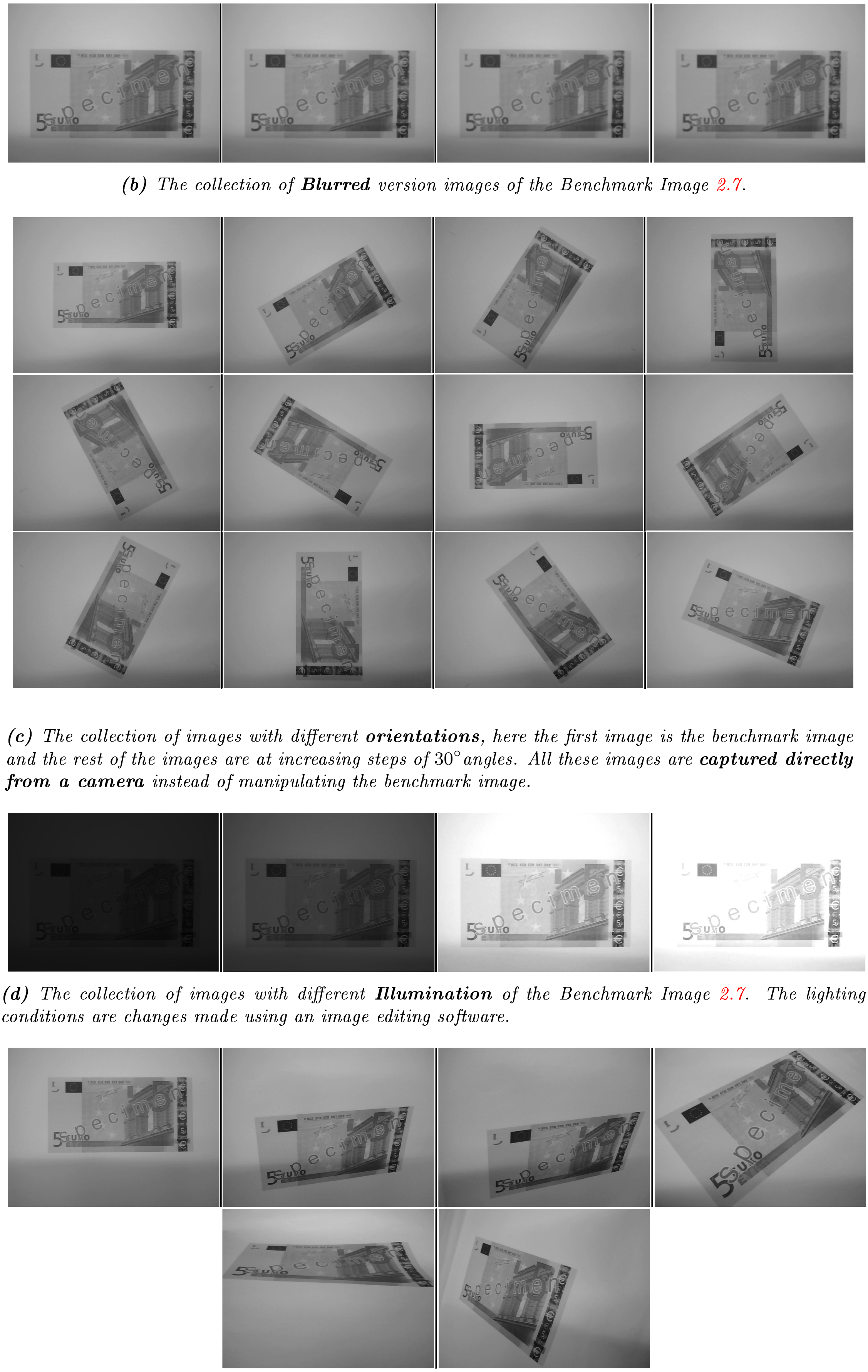


Figure 2.7: *The Benchmark Image used for comparative study.*



*(a) The collection of **Scaled** version images of the Benchmark Image 2.7. Here the first image is scaled up version while the rest are scaled down version of the benchmark image.*

Figure 2.8: *The collection of distorted images used for comparative study.*



*(**b**) The collection of **Blurred** version images of the Benchmark Image 2.7.*

*(**c**) The collection of images with different **orientations**, here the first image is the benchmark image and the rest of the images are at increasing steps of 30° angles. All these images are **captured directly from a camera** instead of manipulating the benchmark image.*

*(**d**) The collection of images with different **Illumination** of the Benchmark Image 2.7. The lighting conditions are changes made using an image editing software.*

*(**e**) The collection of images which have **viewpoint** changes. The first image has only scaling change, while the rest of the images have significant changes in the viewpoint.*

Figure 2.8: *The collection of distorted images used for comparative study. (con't)*

2.2.2 Results of the Repeatability Test

A pair of benchmark and disturbed images are given as inputs to both the algorithms (SIFT and SURF). The results thus obtained from the two algorithms are used to calculate the repeatability for both the algorithms. The repeatability results for all the benchmark - disturbed image combinations are tabulated[1] and plotted.

The graphs having repeatability results for all the 5 disturbing conditions - scaling, blurring, orientation, illumination, and viewpoint changes are presented in Figure 2.9, from which one can observe the following:

- **Scaling changes** - Figure 2.9(a): It is clear from the results that, for the scaling changes, although the SURF Key-Points show a similar trend as the SIFT Key-Points, SIFT clearly outperforms SURF. For the first image, where the disturbance is up-scaling, both SIFT and SURF perform poorly. But, when the images are downscaled - second and third images, both the algorithms perform better. After a certain point - the fourth image, where the image is severely scaled-down, the repeatability values for both the algorithms goes down.

- **Blurring changes** - Figure 2.9(b): From the results, it is clear that, as and how the blurring intensity increases the detection capability of both the algorithms continue to decrease. However, the SURF Key-Points show far more resilience to blurring than the SIFT Key-Points.

- **Orientation changes**- Figure 2.9(c): The results show that, once again the detection capability of both SIFT and SURF algorithms show a similar trend for orientation changes, but the SIFT detection capability is better than SURF.

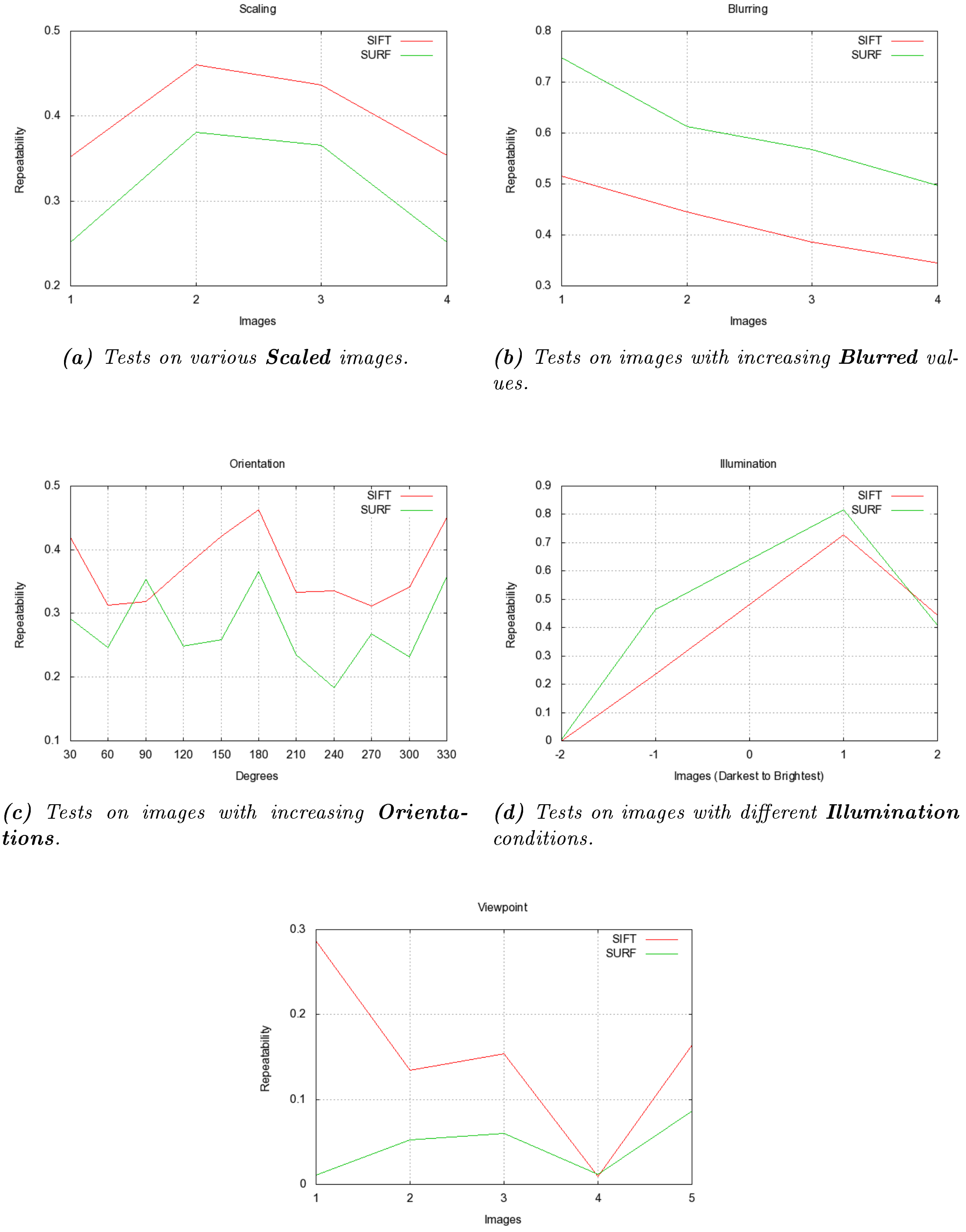
- **Illumination changes** - Figure 2.9(d): The results show that, when the image has very low illumination - the left most image, both SIFT and SURF show very low detection capability. As the illumination goes higher the algorithms perform better, but SURF performs better than SIFT.

- **Viewpoint changes** - Figure 2.9(e): The results show that, the SIFT Key-Points show great resilience to viewpoint changes when compared to SURF.

Bearing in mind that SIFT is far more precise because of the nature of the Gaussian Filters, it uses to build the Scale-Space, while SURF uses Box-Filter approximations of the Gaussian Filter responses (refer to 2.1.3). From the above results it is clear that both SIFT and SURF algorithms have capabilities to handle the disturbing conditions very well. While SIFT shows better resilience to Scaling, Orientation and Viewpoint changes than SURF; SURF handles the Blurring and Illumination conditions better. So, either SIFT or SURF algorithms may be used to build a real-time object identification application.

The next important quality expected from a Key-Point extraction algorithm is its required **Processing Time**, which is addressed in the next section.

[1] For a list of detailed results refer to Appendix 7.1.



(a) Tests on various ***Scaled*** *images.*

(b) Tests on images with increasing ***Blurred*** *values.*

(c) Tests on images with increasing ***Orientations****.*

(d) Tests on images with different ***Illumination*** *conditions.*

(e) Tests on images with different ***Viewpoints****.*

Figure 2.9: *Results of the comparative repeatability tests conducted between SIFT and SURF algorithms.*

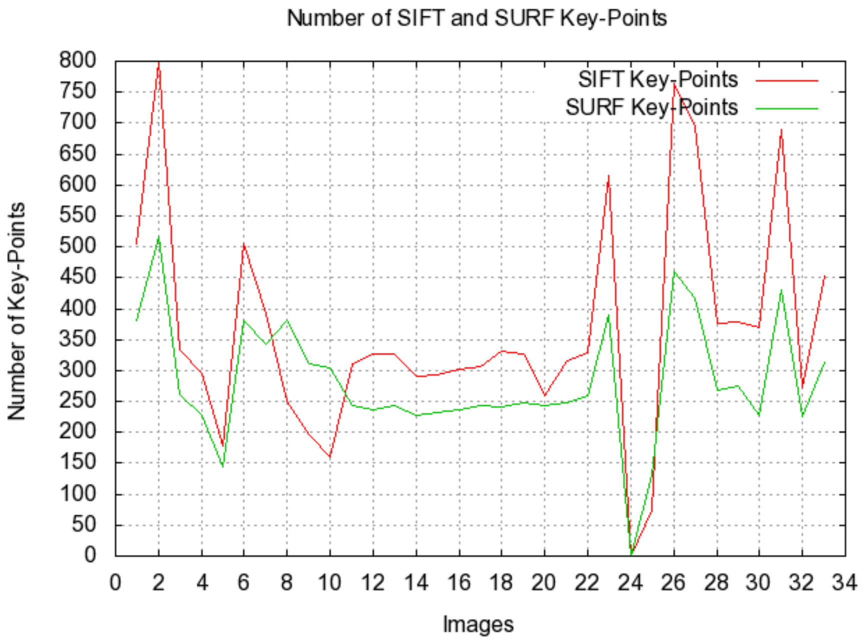
2.2.3 Processing Times

In this section the **Processing Time** needed to compute the SIFT and SURF Key-Points are measured. A set of 33 sample images are considered, this set is a collection of all the images from Figure 2.8 and Figure 2.9.

Both the applications run on a Linux/Pentium 4 system [2] and the time durations between the start and the end of computing Key-Points are measured using *gettimeofday()* function.

The results of such measurements conducted while computing the Key-Points for 33 images are as shown in Figure 2.10. Where Figure 2.10(a) shows the **number of Key-Points** extracted from each of the images by both the algorithms and Figure 2.10(b) shows the corresponding Processing Time needed to compute those features.

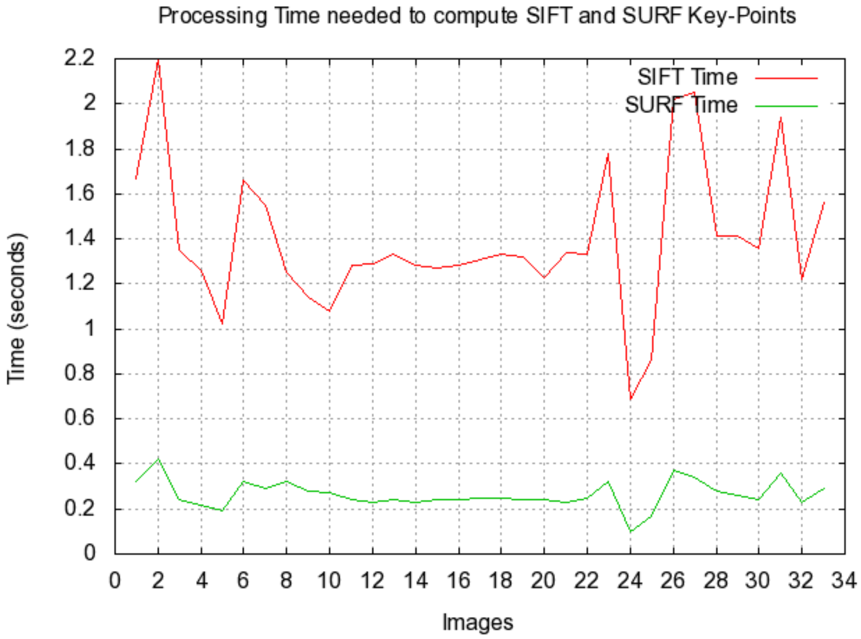
In the case of SIFT, the minimum time needed is **0.69s** and the maximum time needed is **2.2s**. While, in the case of SURF, the minimum time needed is **0.1s** and the maximum time needed is **0.42s**. It is clear from these results that SURF needs lesser time to compute the features when compared to SIFT. Even at such instances where SIFT and SURF have both computed the same number of features or SURF has computed more features than SIFT, like with images numbered - 8, 9, 10, 11, 24 and 25, the time needed by SURF is far less than the one required by SIFT.



(a) The number of SIFT and SURF features extracted.

Figure 2.10: *Processing Time needed by SIFT and SURF algorithms to compute features for 33 images.*

[2] Intel Pentium 4 CPU 2.80GHz, 1.49GB RAM.



(b) Processing Time needed to compute the SIFT and SURF Key-Points.

Figure 2.10: *Processing Time needed by SIFT and SURF algorithms to compute features for* ***33 images****. (con't)*

Algorithm	**Filter Technique**	**Additions**	**Multiplications**
SURF	Integral Image with Box Filter	$2+3$	1
SIFT	2D Gauss	N^2	N^2-1
	Two pass 1-D Gauss	$2 \star N - 2$	$N+2$
	Recursive Gauss	6	14

Table 2.1: *Comparison of various Scale-Space building techniques (calculations per filter response) [5].*

The performance differences in Processing Times can be attributed to how the Scale-Space is built in these two algorithms.

In SURF, the number of computations needed to compute a box filter response are - **4 memory accesses**, **5 additions** [3] and **1 multiplication**, these numbers are constant

[3] Two Additions used in computing the integral image value for one position and 3 additions to compute the box filter response.

irrespective of the size of the filter.

In SIFT, the number of computations needed to compute a Gaussian filter response depends on the method used to compute it. There are several methods used to compute a filter response, as shown in the Table 2.1 which is derived from the study conducted by Jan-Mark et al [30] and Grabner et al [5]. Out of the three methods to compute the filter responses, the first two **depend on the filter sizes** -

1. **2D-Gauss**: The complexity in 2-D convolution method is $O(N^2)$, where **N represents the filter size**, this makes the 2-D convolution method the slowest.

2. **Two pass 1-D Gauss**: This method is better than the previous because it leads to linear costs in the filter size.

3. **Recursive Gauss**: This takes a constant number of computations - 6 additions and 14 multiplications.

As mentioned earlier in this chapter, the filter sizes in SIFT increases with the number of Octaves, the larger the filter sizes, the higher the number of computations needed to compute their responses.

From these methods, it is clear that no matter which approach is used to build the Scale-Space in SIFT, the number of computations needed to compute filter responses are far higher when compared to SURF. In terms of processing time, it is this aspect of SURF which makes it faster and more attractive than SIFT.

So, from the Processing Time measured for both SIFT and SURF algorithms, it is clear that SURF is the algorithm best suited for building a real-time object identification application.

2.3 Conclusion

From the experiments conducted in Section 2.2, it is evident that both SIFT and SURF Key-Points are invariant to several kinds of constantly changing conditions. Both outperform each other in handling certain disturbances. Even under conditions where one performs poorer than the other, they still show similar trends as the other. Taking into account the nature of approximation carried out in SURF it is expected from SURF to show lesser accuracy than SIFT. And for the same reason, when it comes to Processing Times, which is expected to be as low as possible, SURF clearly outperforms SIFT by great margins. Even on instances where SURF is computing equal or greater number of Key-Points as SIFT, SURF still needs lower processing times.

In order to achieve SIFT like accurate Key-Points, computed at a far lesser processing time, SURF is a viable alternative; especially in situations where speed is a critical factor. *On the basis of the above results,* ***SURF algorithm*** *is chosen to be used in building the real-time object identification application.* In the next chapter a step-by-step account of the SURF algorithm, optimizations and the processor platform details are covered.

The Algorithm Implementation and Optimization 3

In the previous chapter two algorithms SIFT and, SURF were studied and on the basis of the study the SURF algorithm was chosen to be used in building the real time object identification. This chapter covers the next research goal, a study of -

- The Algorithm Implementation and
- The Algorithmic Optimization

3.1 The SURF Algorithm

In this project, SURF algorithm was implemented by following the papers written by Bay et al [19], Chris Evans [1] and Nan Zhang [2]. The following section covers the details of the four steps in SURF algorithm and the subsequent section introduces the algorithm optimization details.

3.1.1 The Four Steps of SURF Algorithm

The SURF Key-Points extraction is carried out in four steps -

3.1.1.1 Step 1: Scale-Space Analysis

In this step a Scale-Space is built and analysed for possible **extrema locations** across all scales.

In the case of SIFT's Scale-Space, each Octave is built by successively applying filters of different sizes on a previous layer starting from the input image. The filter sizes and the scaling down sizes increase with every Octave, as indicated in Figure 3.1(a).

Where as, in SURF, building a Scale-Space is little different, every layer is computed by applying a Box Filter of different size on input image. Filter sizes increase both within and across Octaves. However, the only value which remains constant within an Octave is Scale, scaling-down value changes only across Octaves, as shown in Figure 3.1(b). Box filters referred here are approximations of second order Gaussian derivatives in x, y, and xy directions, represented as D_{xx}, D_{yy}, and D_{xy} are as shown in Figure 3.2. In Figure 3.1(b), the starting filter size is 9×9 which is applied at lowest scale of the image and with every next interval, filter sizes increase until the last Interval in fourth Octave where its size will be 195×195. The convolution of these box filters with input image can be efficiently computed using Integral Images (Refer to Section 2.1.3.1 for details.).

Building an Integral Image is an essential step in SURF algorithm, an Integral Image constructed once at the beginning of Scale-Space construction is used to compute approximated Gaussian responses - D_{xx}, D_{yy}, and D_{xy} at all Intervals of Scale-Space very efficiently. The steps involved in constructing an Integral Image are given in Algorithm 1

Algorithm 1 Algorithm used in computing the Integral Image of a given Input Gray-Scale Image.

```
$Position = 0$
$PartialIntegralImage[ImageHeight][ImageWidth]$
function IntegralImageCompute(IntegralImageArray, InputImage)
    for $i = 1$ to ImageHeight do
        for $j = 1$ to ImageWidth do
            $Position = i \times ImageHeight + j$
            $PartialIntegralImage[i][j] = PartialIntegralImage[i][j-1] + InputImage[Position-1]$
        end for
    end for
    for $i = 1$ to ImageHeight do
        for $j = 1$ to ImageWidth do
            if $i > 0$ then
                $IntegralImageArray[i][j] = IntegralImageArray[i-1][j] + PartialIntegralImage[i][j]$
            else
                $IntegralImageArray[i][j] = PartialIntegralImage[i][j]$
            end if
        end for
    end for
end function
```

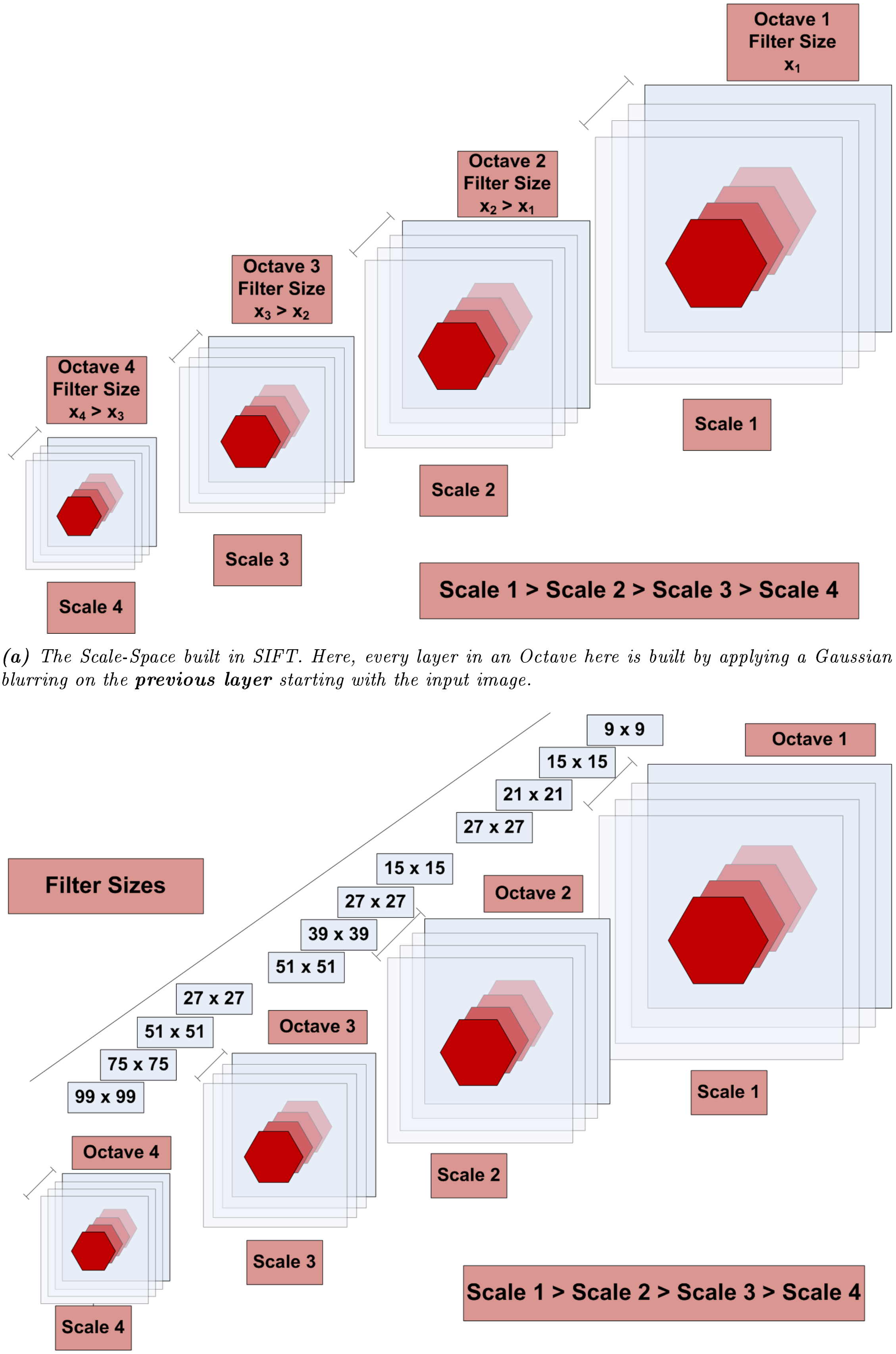
The responses Dxx, Dyy, and Dxy computed using Integral Image and Box filters are then used to compute, what are known as approximated normalized determinant of Hessians, as proposed by Bay et al [19] and given by Equation (3.1.1).

$$det(H_{approx}) = D_{xx} \times D_{yy} - (0.9 \times D_{xy})^2 \tag{3.1.1}$$

The determinant of Hessians or interest points, are computed for all the positions of the input image, for all the scales and filter sizes. In the algorithm implementation, they are stored in the *ResponsePyramid* array. *If there are four Octaves and each Octave has four intervals then the ResponsePyramid will have sixteen layers of determinant of Hessains.*

3.1.1.2 Step 2: Key-Point Localization

Not all interest points computed in Scale-Space analysis are scale and rotation invariant. Weak interest points contribute nothing significant in the matching process, in-fact they become an overhead in that process. Key-Point localization is a two step process, in the first step, only such interest points are chosen which are scale and rotation invariant or in



*(a) The Scale-Space built in SIFT. Here, every layer in an Octave here is built by applying a Gaussian blurring on the **previous layer** starting with the input image.*

*(b) The Scale-Space built in SURF. Here, every layer in an Octave is built by applying a box filter of unique size **ONLY** on the **input image**.*

***Figure 3.1:** SIFT and SURF approaches for building a Scale-Space.*

Algorithm 2 Algorithm used in building a Scale-Space.

```
function BuildScaleSpace(ResponsePyramid, LaplacianPyramid)
    for $i = 0$ to OCTAVES do
        for $j = 0$ to INTERVALS do
            $//Get\ layer\ details\ : Rows, Columns, LobeSize, FilterSize, BorderSize,$
            $//StepSize, NormFactor$
            for $row = 0$ to ImageHeight do
                for $col = 0$ to ImageWidth do
                    $//Compute\ the\ D_{xx}, D_{yy}, D_{xy}$
                    $//Normalize\ the\ D_{xx}, D_{yy}, D_{xy}$
                    $nDxx = D_{xx} * NormFactor$
                    $nDyy = D_{yy} * NormFactor$
                    $nDxy = D_{xy} * NormFactor$
                    // Compute the Normalized Determinant Hessian responses
                    $ResponsePyramid[row][col] = nD_{xx} * nD_{yy} - 0.81 * (nD_{xy} * nD_{xy})$
                    // Compute the Sign of Laplacian
                    $LaplacianPyramid[row][col] = (nD_{xx} + nD_{yy} >= 0\ ?\ 1 : 0)$
                end for
            end for
        end for
    end for
end function
```

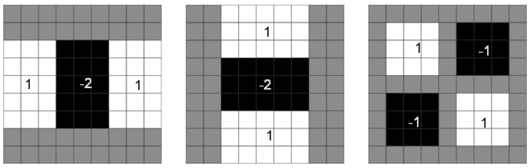


Figure 3.2: Box filter approximations of the Gaussian Filters D_{xx}, D_{yy}, and D_{xy}, from [1].

other words **strong interest points** and in the second step, the chosen interest points are localised across scales.

Excluding or filtering out weak interest points is carried out at two levels. In the first level, all interest points within a layer are passed through a threshold test. In this test, interest points which are above a threshold value are accepted and the rest are discarded. Choosing a threshold value is left as a decision to the algorithm designer, the higher the threshold value chosen, the fewer but stronger interest points are accepted for further processing.

The second level of filtering is called Non-Maximum Suppression. It is carried out across three layers with different scales. Here, interest points from three different scales are considered, as shown in Figure 3.3. Taking the middle value (Red) in the middle layer into consideration, a search for maximum value within that layer, which constitutes

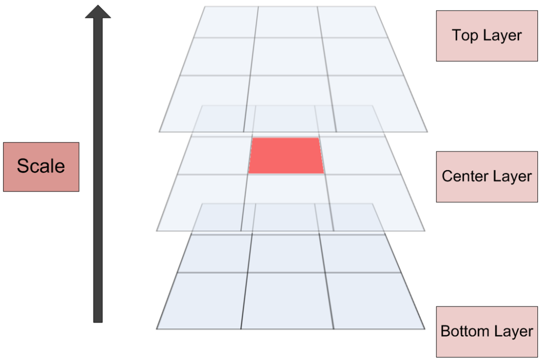


Figure 3.3: *Representation of Non-Maximum Suppression.*

a comparison with 8 neighbouring values and then across two layers - above and below, which constitutes a comparison with 18 values, is conducted. Interest Points which pass this test are then passed on for localization.

The last step in the Key-Point localization is to interpolate the nearby data to determine the position and the scale of the interest points to a sub-pixel accuracy, denoted by $\hat{p}$.

For an interest point at a location **p** = (x, y, s) (s is the scale), a sub-pixel distance value $\hat{p}$ is calculated, such that, when it is added to the location **p** the resultant location will give an accurate estimate of the interpolated position of the interest point **p**. The theory behind this method is explained by Brown et al [31], in practice the elements of $\hat{p}$, given by the equation 3.1.2, can be represented by matrices as given below. So, the sub-pixel accuracy $\hat{p}$ can be computed by three matrix operations - (a) Inverse, (b) Multiply, and (c) Negate.

$$\hat{p} = -\frac{\delta^2 H^{-1}}{\delta x^2}\frac{\delta H}{\delta x} \tag{3.1.2}$$

$$\frac{\delta^2 H^{-1}}{\delta x^2} = \begin{bmatrix} p_{xx} & p_{yx} & p_{sx} \\ p_{xy} & p_{yy} & p_{sy} \\ p_{xs} & p_{ys} & p_{ss} \end{bmatrix}$$

$$\frac{\delta H}{\delta x} = \begin{bmatrix} p_x \\ p_y \\ p_s \end{bmatrix}$$

Where, $p_x = \frac{\delta p}{\delta x}$, $p_y = \frac{\delta p}{\delta y}$, $p_y = \frac{\delta p}{\delta s}$, $p_{xx} = \frac{\delta^2 p}{\delta x \delta x}$, $p_{yy} = \frac{\delta^2 p}{\delta y \delta y}$, $p_{ss} = \frac{\delta^2 p}{\delta s \delta s}$, $p_{yx} = \frac{\delta^2 p}{\delta x \delta y}$, $p_{sx} = \frac{\delta^2 p}{\delta s \delta x}$, $p_{sy} = \frac{\delta^2 p}{\delta s \delta y}$, $p_{xy} = p_{yx}$, $p_{xs} = p_{sx}$ and $p_{sy} = p_{ys}$.

Algorithm 3 gives the details of Key-Point localization step. In Step 3, localized interest points are assigned with reproducible orientation information.

In Step 3, an orientation the localized interest points are assigned with a reproducible orientation information.

Algorithm 3 Algorithm used in Key-Point localization.

```
function NonMaximumSupression(ResponsePyramid)
    for $i = 0$ to OCTAVES do
        for $j = 0$ to INTERVALS do
            //Get Toplayer details : TopLayerHeight, TopLayerWidth
            //Get Centerlayer details : CenterLayerHeight, CenterLayerWidth
            //Get Bottomlayer details : BottomLayerWidth, BottomLayerWidth
            for row = 0 to TopLayerHeight do
                for col = 0 to TopLayerWidth do
                    //Get the CenterLayer's − CenterValue
                    if CenterValue > THRESHOLD then
                        //
                        //Do a Non-Maximum Supression with
                        //8 values within the center layer and
                        //18 values from the top and bottom layers.
                        //
                        //Do an Interpolation.
                        //Carry out the Matrix −
                        //(a) Inverse of $\frac{\delta^2 H^{-1}}{\delta x^2}$ (b) Multiplication with $\frac{\delta H}{\delta x}$ and
                        //(c)Negate the results.
                        //Results of the interpolation − $\hat{x}, \hat{y}$ and $\hat{s}$
                        //Generate the InterestPoint.
                        $InterestPoint.x = x + \hat{x}$
                        $InterestPoint.y = y + \hat{y}$
                        $InterestPoint.s = s + \hat{s}$
                        //Assign the Orientation
                        AssignOrientation(InterestPoint);
                        //Generate the Descriptor or SURF Key-Point
                        GenerateDescriptor(InterestPoint);
                    end if
                end for
            end for
        end for
    end for
end function
```

3.1.1.3 Step 3: Orientation Assignment

In the previous step, the interest points were localized to a sub pixel accuracy in terms of (x, y, scale). In this step, every interest point which has passed the previous tests are assigned with a reproducible orientation information to achieve invariance to image rotation. The value orientation is very important in computing the final interest point description, which is expressed in sixty four floating point values.

Assigning an orientation detail to every interest point is carried out in two steps. In the first step, a circular region of radius 6 × *scale* around each interest point is considered

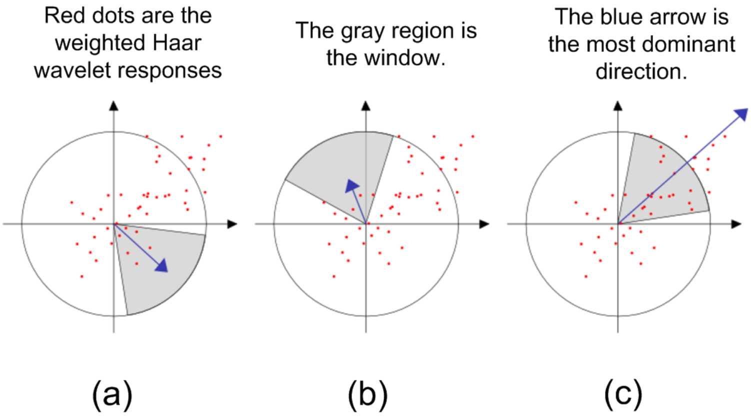

Red dots are the weighted Haar wavelet responses
The gray region is the window.
The blue arrow is the most dominant direction.
(a)
(b)
(c)

Figure 3.4: *Pictorial representation of Orientation computation and Dominant Orientation determination.*

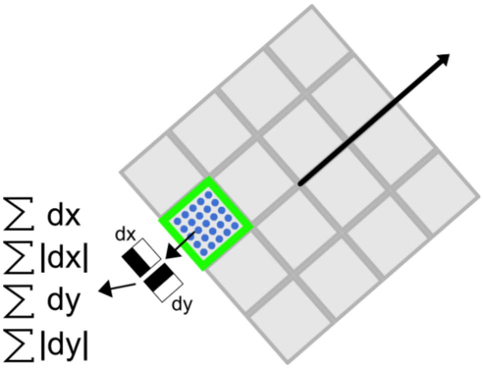


Figure 3.5: *A pictorial representation, which shows how the description of a interest point is computed.*

and within this region the Haar wavelet responses of size $4 \times scale$ in x and in y directions are computed. Responses so obtained are weighted with a Gaussian centred around an interest point and plotted as vector points along x and y coordinates. In step two, a window of size $\frac{\pi}{3}$ is rotated around an interest point and the points which are covered within the window are summed-up. The most dominant result of such summing actions is considered as dominant orientation of the interest point, which is used in calculating the description of the interest point. These two steps are pictorially represented in Figure 3.4 and the algorithm to compute Orientation is given in Algorithm 4.

3.1.1.4 Step 4: Key-Point Descriptor Generation

The last step in Key-Point generation is to give a **description** to all localised Key-Points with orientation information. The Key-Point description is expressed in 64 values.

Algorithm 4 Algorithm used in Key-Point localization.

```
function CalculateOrientation(InterestPoint)
    //Get the x and y coordinates of the InterestPoint.
    //Step 1: Haar wavelet responses in x and y directions.
    RESPONSES responses[]
    float Orientation
    for $i = -6$ to 6 do
        for $j = -6$ to 6 do
            //Check if the point is within the circular region considered.
            if $i * i + j * j < 36$ then
                Get the Haar-x and Haar-y directions
                responses[count]:x = x * gauss
                responses[count]:y = x * gauss
                Count++
            end if
        end for
    end for

    Step 2: Calculate the Dominant Orientation.

    for i = 0 to $2 * \pi$ do
        for n = 0 to Count do
            sumX + = responses[n].x
            sumY + = responses[n].y
            Orientation = getangle(sumX; sumY )
        end for
    end for
    InterestPoint.orientation = Orientation
end function
```

In this step, a square region which is divided into sixteen sub-squares is considered around the center of every interest point. This square is aligned along the orientation computed in previous step. Every sub-square is sampled at twenty five (5×5) regularly spaced points. Like in the previous step, Haar-x and Haar-y wavelet responses are computed at every 25 points within a sub-square. These responses are then applied with Gaussian weights. From every sub-square region, four vectors - two in x (d_x, $|d_x|$) and two in y (d_y, $|d_y|$) co-ordinates are computed. The summation of all the four values from all the 25 samples gives rise to four vectors ($v = (\sum d_x, \sum |d_x|, \sum d_y, \sum |d_y|)$) from one sub-square. So, from the entire square region, which has 16 sub-squares there are $16 \times 4 = 64$ values which forms the description of an interest point. This is as shown in Figure 3.5 and Algorithm 5.

Algorithm 5 Algorithm used to compute the Key-Point Description.

```
function CalculateDescription(InterestPoints)
    The Outer While Loop selects one of the 16 sub-squares
    while i < 16 do
        while j < 16 do
            The Inner For Loops are used to get 25 samples within a sub-square
            for $k = 0$ to 25 do
                for $l = 0$ to 25 do
                    $//GettheCo - ordinatesofrotatedsquare$
                    $//Co - ordinatex; Co - ordinatey$
                    $//GettheHaarxandHaaryresponses$
                    $dx = Haarx * gauss$
                    $dy = Haary * gauss$
                    $adx = abs(Haarx) * gauss$
                    $ady = abs(Haary) * gauss$
                end for
            end for
            InterestPoints.descriptor[count + +] = dx
            InterestPoints.descriptor[count + +] = dy
            InterestPoints.descriptor[count + +] = adx
            InterestPoints.descriptor[count + +] = ady

            i++
            j++
        end while
    end while
end function
```

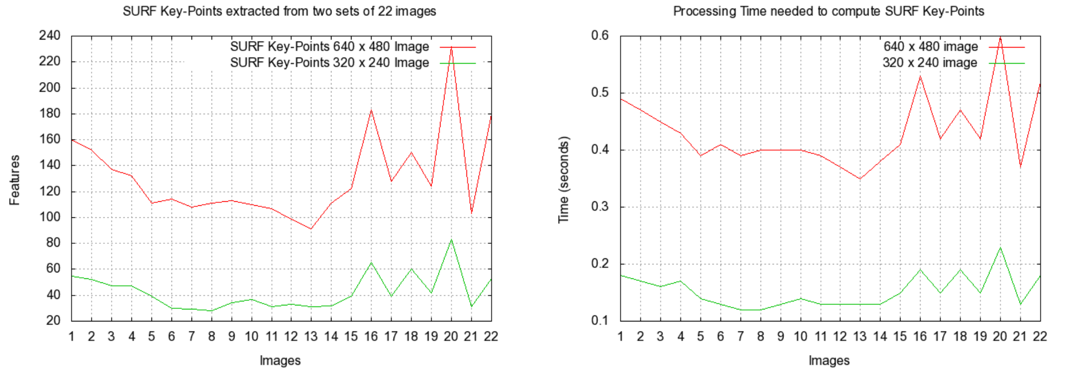
3.1.2 Processing time

The SURF feature extraction algorithm implemented according to the description in the previous section is tested to find out processing time needed to compute features from two sets of images. The images of the first set are of 640 × 480 pixels dimension and the images from second set are 320 × 240 pixels variants of the same images in the first set. These images are borrowed from Section 2.8. The system considered for these tests is once again, a Linux/Pentium 4 system[1]. Figure 3.6(a) shows the number of Key-Points extracted from both sets of images and Figure 3.6(b) shows the processing time needed by the algorithm to compute those features.

The trend shown by the algorithm in extracting Key-Points and also the processing time needed from both variants of image sets is similar. Higher the number of Key-Points extracted, greater the time consumed.

The previous section ended with results from SURF implementation. The processing time measured are on a desktop system, which is far more superior in terms of resources,

[1] Intel Pentium 4 CPU 2.80GHz, 1.49GB RAM.



(a) Number of SURF features extracted.

(b) Processing time needed by SURF to extract features.

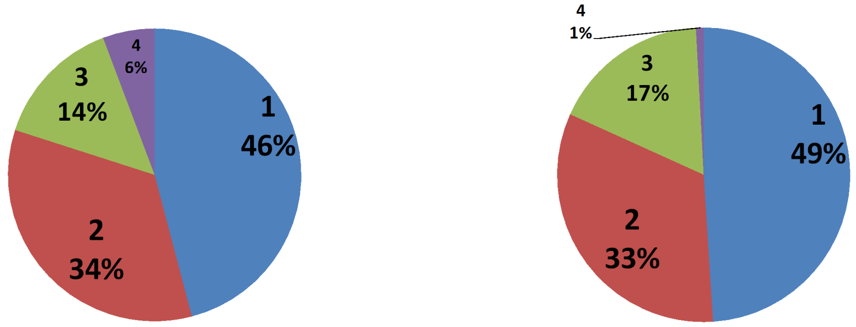
Figure 3.6: *SURF Key-Points extracted from two sets of **22 images**.*

when compared to the embedded platform - i.MX515EVK on which the algorithm will finally run. The first attempt to optimize the algorithm is carried out in this section.

3.1.3 Optimization 1: Algorithm Optimization

The study on Scale-Space in the earlier sections showed that a Scale-Space is divided into several Octaves and each Octave in turn is a collection of several intervals. Two important factors start to increase across the intervals and Octaves -(a) Filter sizes - As and how the number of intervals increase, the filter sizes also increases and (b) Sampling-Step - To achieve a step-pyramid like shape of Scale-Space, the Sampling-Step value is increased by a factor of 2 for every Octave. From Bay et al [19], the authors of SURF, it seems that, in their implementation they have used 4 Octaves and 4 intervals i.e., $4 \times 4 = 16$ intervals and Sampling-Step starts from 2 and ends at 16, i.e., $Sampling - Step = [2; 4; 8; 16]$, which means that in the first Octave where Sampling-Step is 2, the box filters are moved to every alternate pixel (From left to right), in the second Octave where Sampling-Step is 4, the box filters are moved to every 4th pixel and so on for the rest of the Octaves. It was observed during the study conducted in this project that, fewer number of Key-Points are extracted from higher Octaves. This phenomenon of decreasing number of Key-Points towards the higher Octaves can be attributed to increasing filter sizes and increasing Sampling-Step values. Both of these factors accelerate the movement of the filters to the end of the image. Which implies that, fewer computations are carried out in higher Octaves, this is precisely the reason why the number of Key-Points extracted from higher Octaves decrease sharply. This is demonstrated in Figure 3.7(a) which shows how the Key-Points extracted from previous steps are distributed across 4 Octaves; out of 2878 Key-Points extracted from the first set of images, 80% of the points, i.e., 2301 points are extracted from first and second Octaves. Similarly, Figure 3.7(b) shows that, out of 937 points extracted from the second set of images, 82% of the points, i.e., 766 points are extracted from first and second Octaves. On the basis of these results, in this

SURF implementation, instead of going in a conventional way of computing four Octaves, only first two Octaves are considered for computation. The impact on processing time due to this optimization is the reduction of the processing time between 40 - 90 ms as shown in Figure 3.8.



(a) Key-Points extracted from first set of images. ***Eighty percent*** *of the points were extracted from* ***first two Octaves****.*

(b) Key-Points extracted from second set of images. ***Eighty two percent*** *of the points were extracted* ***from first two Octaves****.*

Figure 3.7: *Distribution of SURF Key-Points extracted from two sets of* ***22 images*** *across four Octaves.*

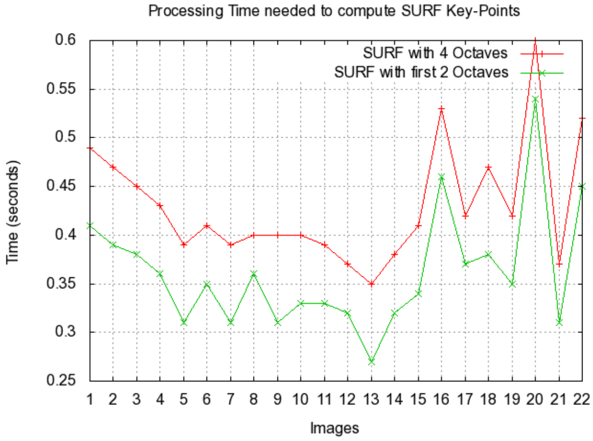


Figure 3.8: *Comparison showing the time difference between SURF with 4 Octaves and SURF with only first 2 Octaves.* ***Optimization 1*** *has decreased the processing time between* ***40 - 90 ms****.*

Once again it is worth repeating an important point, the improvement in processing time of SURF achieved by optimization, can prove to be a lot more significant on the i.MX515EVK platform. In the following section, where the Hardware platform -

i.MX515EVK is introduced, the contribution of this optimisation to the processing time is once again measured.

3.2 Conclusion

The focus of this chapter was on the implementation and optimization of the SURF algorithm. The optimization carried out in this chapter was based on the study conducted in this project. Observing the results of that study it was realized that, out of a given number of Key-Points extracted from an image, it was found that the first two Octaves are responsible for extracting more than 80% of the Key-Points. The reason for this sudden reduction in the number of Key-Points was also found out. The SURF algorithm is designed have larger filters and higher stepping values in higher Octaves, the reduction in the number of Key-Points can be attributed to such large filter sizes and stepping values, both of which are responsible for accelerating the movement of the filters to the end of the image, which implies that fewer computations are carried out in the higher Octaves. Therefore, a decision was made in this project to modify the algorithm to not compute Key-points in the higher Octaves.

An obvious outcome in cutting down on the number of computations carried out is an improvement in processing time (40 - 90ms)[2], however, a hidden benefit from this approach is an improvement in both static and runtime memory requirements, and also lowering the power utilization of the application.

The speed up achieved here can be seen as quite significant when the algorithm is running on the i.MX515EVK. The details of the implementation of SURF for i.MX515EVK is explained in the next chapter. Where once again measurements are conducted to find out the impact of Optimization 1 in that environment.

[2] On an system with Intel Pentium 4 CPU 2.80GHz, 1.49GB RAM.

4 Hardware Platform: i.MX515 Evaluation Kit

In the previous chapter SURF was implemented and the first optimization was carried out on the algorithm. This chapter covers the details of i.MX515EVK embedded platform and how the processor's resources, especially the SIMD unit is efficiently used to improve the processing time of SURF_ARM_NEON.

The Embedded Platform - **Freescale's i.MX515 Evaluation Kit (i.MX515EVK)** is the platform on which the final real-time object identification application is intended to run on. The block diagram of the evaluation kit is as shown in Figure 3.1. This EVK runs on Linux OS (Ubuntu distribution) with supporting Linux Board Support Packages (BSP) developed and provided by Freescale.

Some of the important features [1] of this platform, which are relevant to this project are listed below:

- CPU: Freescale's **i.MX515** at a clock speed of 800 MHz.
- Memory: 512 MB DDR-RAM.
- *For the entire list of peripherals refer to Appendix B: i.MX515EVK Details 7.2.*

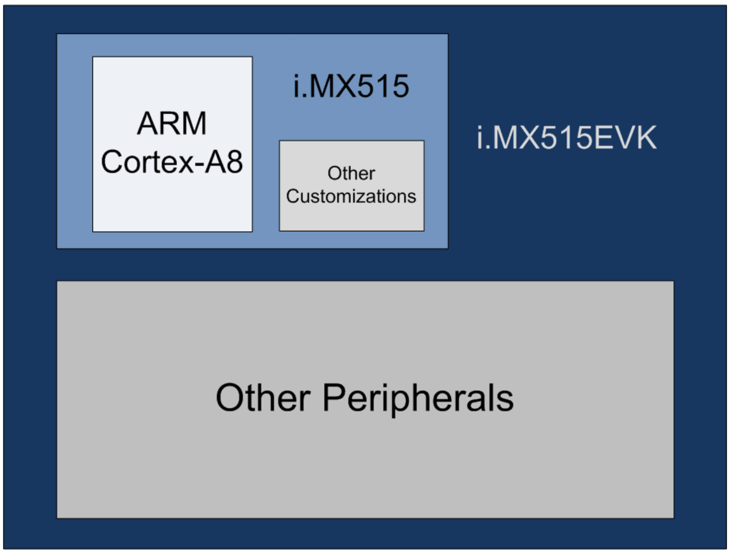


Figure 4.1: *A block diagram of Freescale's i.MX515 evaluation kit.*

[1] For the functional block diagram of the i.MX515EVK refer to Appendix B: i.MX515EVK Details.

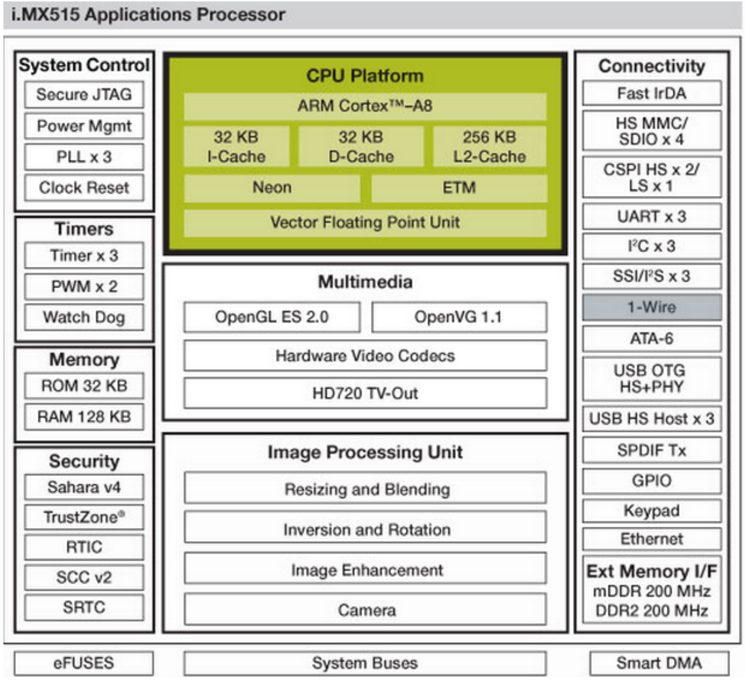


Figure 4.2: *Functional block diagram of i.MX515 Processor, borrowed from the Freescale's i.MX515 web page.*

4.1 Fresscale's i.MX515 Processor

The **i.MX515 is a Multimedia Processor** which is one of Freescale's latest additions to their growing multimedia-focused products; offering a High Processing Performance, at a very low power consumption which can be attributed to the core of this processor, the **ARM Cortex-A8**.

The functional block diagram of the i.MX515 processor, which shows the implementation details of the ARM Cortex-A8 processor and other co-processors are as shown in Figure 3.2.

The processor is suitable for applications such as:

- Netbooks
- Handheld devices such as
  - Portable Media Devices
  - Smart Phones
  - Navigation Devices
- Gaming consoles

4.2 ARM Cortex-A8 Processor

The ARM architecture was originally developed by Acorn Computers in 1985. The ARM Core is built on a RISC architecture philosophy, which is aimed at delivering simple and yet powerful instructions that complete execution within a single clock cycle. The ARM processor has been specifically designed to be small in order to reduce power consumption and extend battery operation, this is one of the main reasons which makes it a favourite amongst handheld device manufacturers. As of 2007, more than 90% of all mobile handsets shipped contained ARM Processors. Many of the top semiconductors manufacturing companies around the world produce products based on ARM processors, which are used in various applications, such as Mobile Phones, Net Books, Digital TVs, Set-Top boxes, Automotive entertainment[32] [33].

The ARM Cortex-A8 core is a 32-bit, dual-issue, in-order type processor, with dynamic branch predictor.

Some of the key features of this processor are listed below.

- It operates at a Clock Speed of 800Mz.
- 32 KB Instruction and Data Caches.
- A unified 256 KB L2 Cache
- A Vector Floating Point Unit (VFP)
- A SIMD unit called NEON
- Instruction Set Architecture (ISA) support
  - ARM - For Cortex-A8 processor, ARM instruction set provides the definitive and complete 32-bit instructions, using this instruction set gives best results in terms of **performance**.
  - Thumb - The Thumb instruction set is an extension to the 32-bit ARM architecture and it is used to obtain a high code density. It is a subset of the most commonly used 32-bit ARM instructions which have been compressed into **16-bit wide operation codes**. These 16-bit instructions when decoded eventually enable the same functions as their 32-bit ARM instruction equivalents. This instruction set was primarily introduced to cater such situations where program memory is a constraint. Compared to full ARM up to 30 percent code size reduction is achieved, however what is achieved in low system memory use, is lost in performance [34].
  - Thumb2 - The Thumb2 is a superset of Thumb instruction set. Several, new 16-bit instructions were introduced into Thumb2, but the speciality of Thumb2 lies in the new 32-bit instructions introduced, which were once again derived from ARM instructions. Although the performance is not completely at the level of pure ARM instructions, it is definitely better than Thumb instructions [3]. A comparison of code density and performance achieved by using all three instruction sets is presented in Figure 4.3.

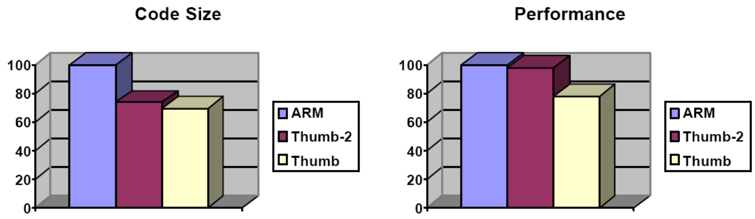


Figure 4.3: *Comparison between code density and performance achieved using ARM, Thumb and Thumb2, from [3].*

- VFPv3 Floating Point - These instructions are used to program the Vector Floating Point Unit (vfpv3). In simple words vfpv3 is a floating point hardware accelerator. The purpose of VFP is to speed up half, single and double precision floating point operations. The name Vector here is a misnomer, the VFP actually has no parallel architecture. It performs one operation on one set of inputs and returns one output.

- NEON - These instructions are used to program NEON - the SIMD unit of the Cortex-A8 processor. This is a true vector processor. Programs targeted to NEON can be written directly in Assembly or by making use of NEON Intrinsics, which are a C wrapper around assembly instructions. Some of the powerful NEON instructions are used extensively to program critical parts of the SURF algorithm implementation of this project.

The **13 stage Integer pipeline** and **10 stage NEON pipeline** view of ARM Cortex-A8 is as given in Figure 4.4.

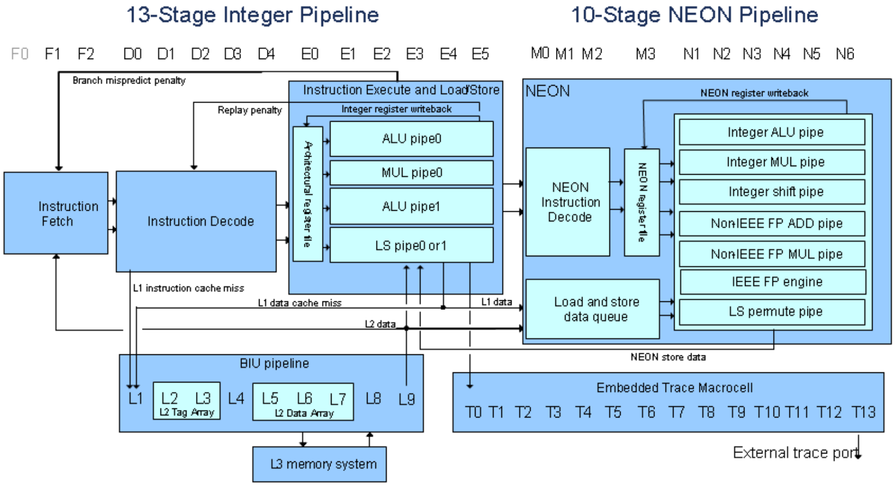


Figure 4.4: *Functional block diagram of i.MX515 Processor, from the Freescale's i.MX515 web page.*

4.2.1 Development Tools

The development tools used in this project were from the GNU tool chain.

- GCC 4.4.1
- GNU Binutils
  - as
  - ld
  - gprof
  - objdump
- GNU make

4.3 SURF Algorithm for ARM Cortex-A8 Processor

This section starts of by explaining how SURF algorithm is built for ARM Cortex-A8 processor and then the focus is entirely devoted to its optimizations. Optimization techniques are successively applied to eventually push the algorithm to reach to an optimum state, suitable to be used for a real-time object identification application.

Because of a full fledged operating system such as Ubuntu 9.10 running on i.MX515EVK, software building process was carried out directly on i.MX515EVK rather than using expensive Cross Compilers on a host system. By making use of the development tools mentioned in previous section, SURF Key-Points extraction algorithm was built for ARM Cortex-A8 processor, from hence forth this version of SURF is referred to as **SURF_ARM**.

The compiler options provided to *gcc* compiler to build **SURF_ARM** are as given below:
gcc* -*O3 $-mcpu =$ ***cortex-a8* -*marm*** $-mfloat - abi =$ ***softfp*** $-mfpu =$ ***vfpv3***
Where,

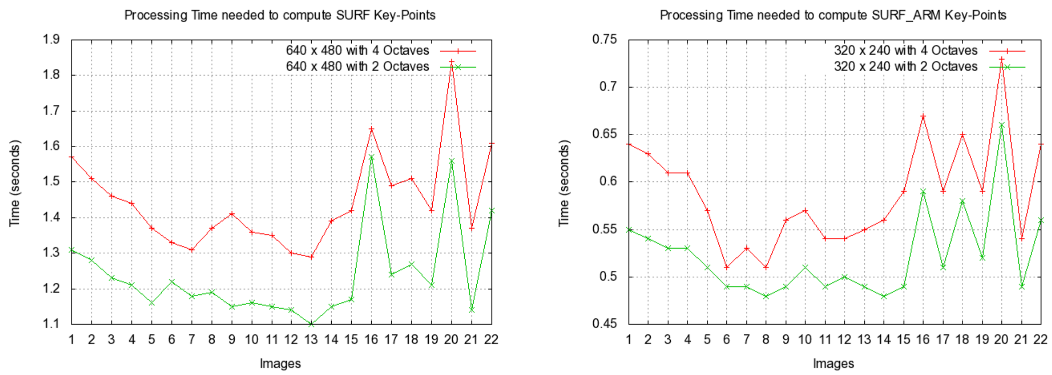
- $-O3$: Flag which indicates *gcc* to carry out the highest level of code optimizations, for the entire list of optimizations carried out here, refer to gcc 4.4.1 manual.
- $-mcpu$: It is the name of the CPU, i.e., **cortex-a8** in this case.
- $-marm$: It is used to indicate to the compiler to emit ARM instructions.
- $-mfloat-abi$: **softfp** used here allows the generation of code for hardware floating-point unit.
- $-mfpu$: **vfpv3** is to indicate to the compiler to make use of VFPv3 for floating point operations.

4.3.1 Processing Time needed to compute Key-Points using SURF_ARM

To get a perspective on where processing time[2] needed by **SURF_ARM** stands, the two sets of images used in time measurement tests of section 3.1.2 are used to measure the processing time on i.MX515EVK, as shown in Figure 4.5. For 640 × 480 sized images, optimized SURF_ARM takes between 1.1 - 1.57 seconds to process and for 320 × 240 sized images, it takes between 0.48 - 0.66 seconds.

Observing the processing time needed for both sets it is clear that achieving a good frame rate in the real-time object identification is an up hill task and the implementation of SURF_ARM has to go through a lot of optimizations.

Unlike the algorithm optimization carried out in **Optimization 1** (Section: 3.1.3), optimizations carried out from henceforth are not on algorithm, they are rather on the way SURF_ARM is implemented by making the best use of the processor capabilities.



(a) For a set of twenty two, 640 × 480 sized images, Optimization 1 has brought down between ***80 - 280 ms*** *of processing time.*

(b) For a set of twenty two, 320 × 240 sized images, Optimization 1 has brought down between ***20 - 100 ms*** *of processing time.*

Figure 4.5: *Processing time needed to compute Key-Points using SURF_ARM. It also shows a difference between SURF_ARM with 4 Octaves and SURF_ARM with* ***only first 2 Octaves****.*

4.3.2 Optimization 2: Make use of NEON

This section covers details about optimizations carried out on SURF_ARM by making use of NEON - the SIMD co-processor of the Cortex-A8 processor.

4.3.3 NEON

Single **I**nstruction **M**ultiple **D**ata (SIMD) computers are such computers where multiple processing elements execute the same instruction while processing distinct data elements.

[2] The time durations between the start and the end of computing Key-Points are measured using *gettimeofday()* function.

SIMD is one of the four classes in the taxonomy of computer design as proposed by Flynn, the others being SISD, MISD, and MIMD. Multimedia and digital signal processing applications typically spend significant portions of their execution time in loops applying same arithmetic and logical operations on a large chunk of 8-bits or 16-bits datatypes and such cases are classic situations where data level parallelism can be exploited. In a SIMD processor this is achieved by packing several such small data elements into its registers and a common operation is then applied on the registers as shown in Figure 4.6. The data elements could be of 8-bits, 16-bits, and 32-bits in length and the SIMD registers into which they are packed could be of 32-bits, 64-bits, and 128-bits in length.

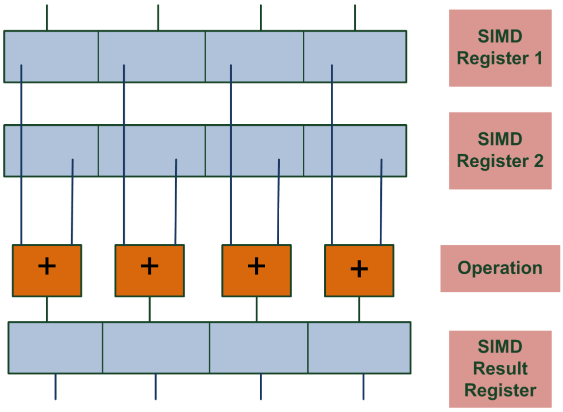


Figure 4.6: *Pictorial representation of a SIMD processor operating on 2 registers.*

NEON is a SIMD processor integrated with the ARM Cortex-A8 processor. It contains 32 64-bit registers as shown in figure and the NEON unit can view same register bank in two views as shown in Figure 4.7:

- Sixteen 128-bit quadword registers, Q0-Q15.
- Thirty-two 64-bit doubleword registers, D0-D31.

The NEON instructions support 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers. It also supports 32-bit single precession floating point values.

4.3.4 Methods available to program NEON

Three methods are currently available to program NEON as follows -

1. Using *gcc* Flags for Auto Vectorization

The first attempt made to vectorize the SURF_ARM was to use compiler's - *gcc*, ***automatic vectorization*** option. To do this, the *compile line* used to compile the source code appeared like this -

gcc -O3 $-mcpu =$ ***cortex-a8 -marm*** $-mfpu =$ ***neon*** $- ftree - vectorize$

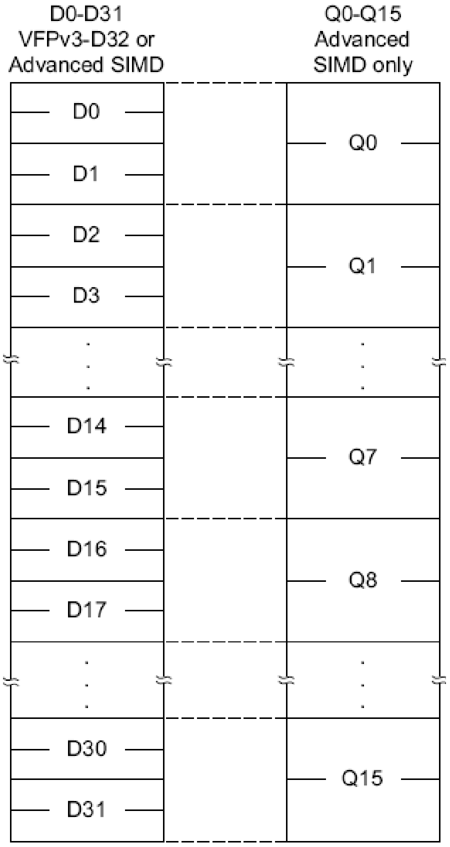


Figure 4.7: *Two views of NEON's 32 64-bit registers.*

In the above line, first of all, the compiler needs to be told to use NEON ($-mfpu = \boldsymbol{neon}$), then by passing the flag $-ftree-vectorize$ we ask the compiler to auto vectorize the code. However, in this project it was observed that gcc 4.4.1 is not matured enough to vectorize any portions of SURF_ARM for NEON at all.

To test *gcc*, *Matrix Multiplication*, one of the best candidates which can be vectorized and also one of the most frequently used operations in Non-Maximum Suppression (Algorithm: 3) of SURF_ARM was compiled using the auto-vectorization flags. Upon comparing the resulting disassembled code with regular non-NEON, $-mfpu = \boldsymbol{vfpv3}$ code it was found that there was no difference at all between the two (For a detailed look at the disassembled views refer to Appendix C: NEON Tests 7.3.) Several, potential candidates, such as - parts of Matrix Inverse computation, Integral Image computation, and SURF Descriptor generation, which can be hand vectorized were given to *gcc* and in all of those cases *gcc* **failed** to generate NEON code.

2. Using Assembly

Using Assembly, is a direct way of programming NEON. The GNU assembling tool - **as** can be directly provided with ARM assembly code and it generates the object code for

NEON.

An example assembly code for NEON, which loads a set of four consecutive 32-bit values from memory into q0 register and four more 32 bit values from consecutive memory locations into q1 register, then adds the contents of both registers and stores the results back into memory is as shown below.

```
.text
.global _do_it4_again

_do_it4_again:
  # r0: Pointer to X Co-ordinates
  # r1: Pointer to Y Co-ordinates
  # r2: Pointer to the result
  push          {r4-r7,lr}

  vld1.32     {q0}, [r0]
  vld1.32     {q1}, [r1]
  vadd.32     q0, q0, q1
  vst1.32     [r2], q0

  pop           {r4-r5, pc }
```

3. Using NEON Intrinsics

NEON Intrinsics are a C wrapper around NEON assembly instructions. Intrinsics are the easiest way in which NEON can be programmed. Unlike while using assembly instructions, using intrinsics can ease the developer from issues such as register allocation and interlock. The example given below shows how NEON intrinsic can be used.

```
float32x4_t vAddition(float32x4_t input)
{
  return(vaddq_f32(input, input));
}
```

Where, **float32x4_t** is the vector which is of 128-bits wide type, having 4 individual **float** values and **vaddq_f32** is the NEON addition intrinsic to add 2 float32x4_t type values.

4.3.5 SURF_ARM profiling

In this section SURF_ARM is profiled on i.MX515EVK to find out in which parts of SURF_ARM does the processor spend most of its time. This is an in depth analysis of SURF_ARM carried out to find potential areas for improvement.

Profiling is carried out using GNU's profiling tool ***gprof*** (Refer to ***-pg*** option in GCC). An excerpt of *gprof* profiler report, which shows the most Processing-Time consuming functions is given in Table 4.1. The complete profiler report is in Appendix D: Profiler Report (7.4). This report was generated when SURF_ARM extracted 129 Key-Points from a 640 × 480 dimension image.

Function Name	Time Spent (%)	Calls
BoxFilterCompute()	69.24	3,383,284
BuildDescriptor()	12.89	129
BuildResponseLayers()	11.11	1
IntegralImageCompute()	2.78	1
MatrixInverseMultiply()	2.57	146
GetOrientation()	1.43	129

Table 4.1: *An excerpt of* **gprof** *report, when SURF_ARM was given a* 640 × 480 *sized image as input.*

4.3.6 Candidates chosen for Vectorization

On the basis of the profiler report, the candidates chosen for vectorization are *BuildDescriptor()*, *IntegralImageCompute()*, and *MatrixInverseMultiply()*. The *BoxFilterCompute()* as such is a very simple function, the amount of time spent is maximum there because of the very high number of calls made to it from *BoxScaleSpace()*.

1. Box Filtering using NEON

The significance of box filtering and how it brings down the computation time of SURF in general was learnt from previous chapters. Looking at the profiler report generated in the previous section it is clear that, during the execution of SURF_ARM, the function - *BoxFilterCompute()* was called the highest number of times and the processor spends maximum amount of its time in that function.

In this section, an innovative method, which involves NEON is explored to compute box filter responses more efficiently, which optimizes SURF_ARM implementation.

```
float BoxFilterCompute(int row, int col, unsigned int num_rows, unsigned int num_cols)
{
  float area = 0;

  // Compute the Co-Ordinates of A, B, C and D
  int temp_row = row + num_rows;
  int temp_col = col + num_cols;

  int r1 = row - 1;
  int r2 = temp_row - 1;

  int c1 = col - 1;
  int c2 = temp_col - 1;

  // Compute Box Filter Response. area = A - B - C + D

  area = integral_image[r1][c1] - integral_image[r1][c2] - integral_image[r2][c1] + integral_image[r2][c2];

  return area;
}
```

While computing Key-Points, box filters are used in three distinct steps of SURF_ARM (a) Building Scale-Space (b) Orientation Assignment, and (c) Building Descriptor. For an image of given dimensions, out of these three steps, the **Scale-Space building is a constant and never changing step**, so the number of *BoxFilterCompute()* function calls from within Scale-Space building is always a **constant number**. Where as, when it comes to the other two steps, the number of *BoxFilterCompute()* calls are dependent **on the content of a given image**, i.e., the number of feature points localized and not on image dimensions.

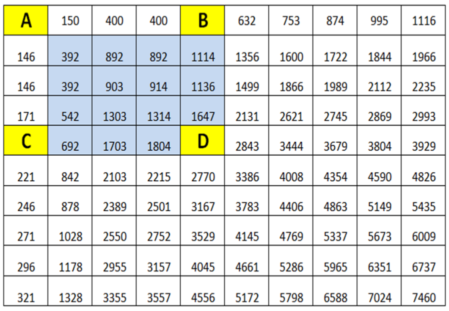
The distribution of *BoxFilterCompute()* function calls for **two images of** 640 × 480 **and two images of** 320 × 240 dimensions, is as given below in Table 4.2.

Image	Dimension	Features	Total Calls	Building Scale-Space	Orientation Assignment and Descriptor building
1	640 × 480	129	3,383,284	2,658,304	724,980
2	640 × 480	153	3,518,164	2,658,304	859,860
1	320 × 240	44	816,624	569,344	247,280
2	320 × 240	54	872,824	569,344	303,480

Table 4.2: *Distribution of BoxFilterCompute() function calls across* ***three steps*** *of SURF_ARM.*

The sheer number of times box filtering is called makes it **the most time consuming function** and the only way any performance enhancement can be brought to the SURF_ARM implementation is by cutting down the number of calls made to that function, which can be achieved using NEON.

The *BoxFilterCompute()* function is a very simple function as shown below, it can be used to compute response of the box region(blue) as shown in Figure 4.8.



A	150	400	400	B	632	753	874	995	1116
146	392	892	892	1114	1356	1600	1722	1844	1966
146	392	903	914	1136	1499	1866	1989	2112	2235
171	542	1303	1314	1647	2131	2621	2745	2869	2993
C	692	1703	1804	D	2843	3444	3679	3804	3929
221	842	2103	2215	2770	3386	4008	4354	4590	4826
246	878	2389	2501	3167	3783	4406	4863	5149	5435
271	1028	2550	2752	3529	4145	4769	5337	5673	6009
296	1178	2955	3157	4045	4661	5286	5965	6351	6737
321	1328	3355	3557	4556	5172	5798	6588	7024	7460

Figure 4.8: *Pictorial representation of computing box(Blue) response, computed using Integral Image.*

NEON Instructions suitable for performing box filtering

To vectorize some of the redundant DSP operations, NEON has some very effective instructions in its instructions set. One best example of a redundant DSP operation in image processing is a **Color to Grayscale conversion**. Color to Grayscale conversion is a simple and straight forward step as shown in Figure 4.9, but carrying out in a **serial fashion** can be very time consuming, in the event of an availability of a vector processor with large width registers (32-bits, 64-bits or 128-bits) and good instructions, **COLOR to GRAYSCALE** conversion can be carried out at a faster rate. The NEON SIMD processor has **128-bit registers** and it has just the right kind of instructions - **interleaving loads/stores**, which can be used to convert 8 $RGB \rightarrow GRAYSCALE$ pixels, simultaneously.

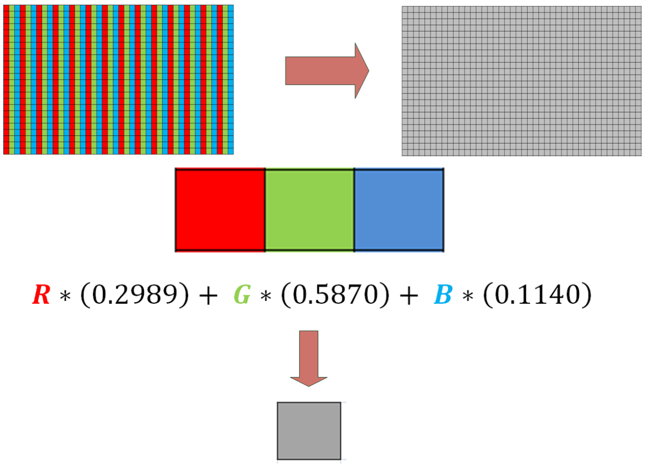


Figure 4.9: *A pictorial representation of a 24-bit Color to Grayscale image conversion. The constants used to multiply R, G and B values are standard values used in* $RGB \rightarrow GRAYSCALE$ *conversion.*

The instruction ***vld3.u8*** can be used to load pixel information, located at every third memory locations, in this example R, G, and B values, into three separate NEON registers, which gives a freedom to operate upon individual registers, here they are multiplied with R, G and B multiplying factors, respectively.

Although these concepts were added to NEON with the intention to solve redundant DSP related functions, **in case of SURF_ARM they can be adopted to compute several box filter responses simultaneously**.

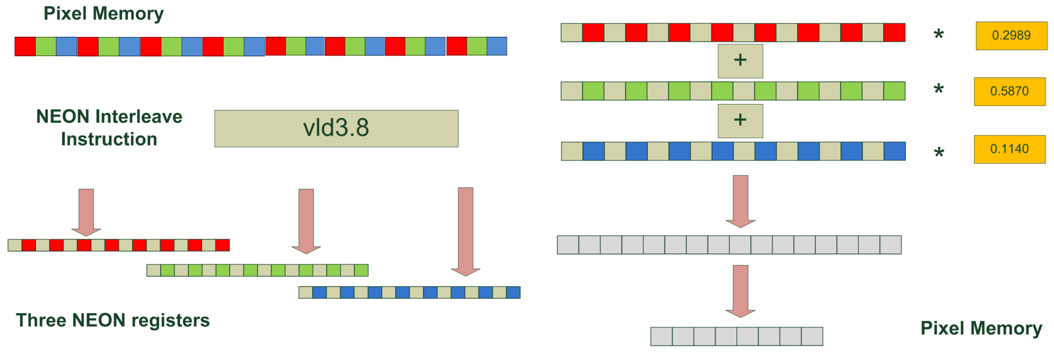
In NEON, 4 interleaving load/store instructions are available, which are:

- **No interleaving load/store instructions** - They are used to load/store data from/to consecutive memory locations, e.g., *vld1.32*, *vld1.16*, *vld1.8*, *vst1.32* etc.

- **Alternate interleaving load/store instructions** - They are used to load/store data from/to every alternate memory locations, e.g., *vld2.32*, *vld2.16*, *vst2.32*, etc.

- **Every third interleaving load/store instructions** - They are used to load/store data from/to every third memory locations e.g., *vld3.32*, *vld3.16*, *vst3.32*, etc.

- **Every forth interleaving load/store instructions** - They are used to load/store data from/to every forth memory locations e.g., *vld4.32*, *vld4.16*, *vst4.32*, etc.

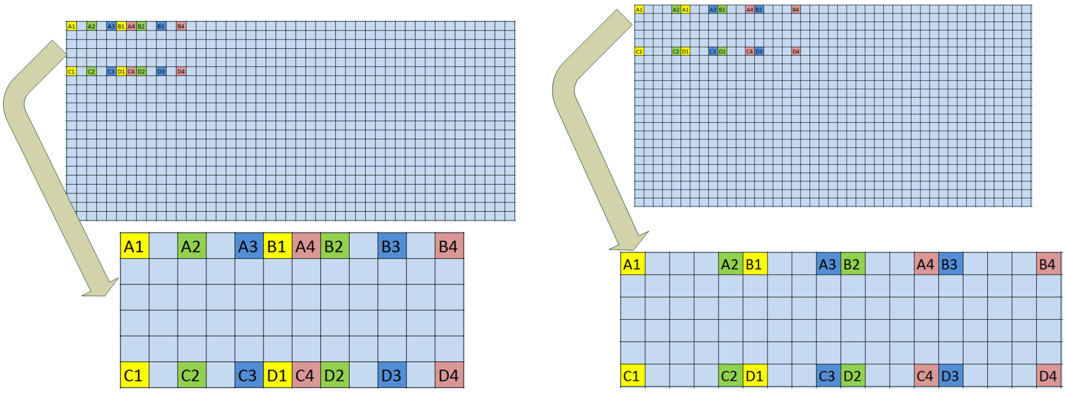


(a) By using NEON's ***interleave instruction vld3.u8****. pixel values at every third memory locations were loaded into three separate 128-bits wide NEON registers.*

(b) The three NEON registers having individual color values are multiplied with three multiplication factors, which will then be added together to get ***8 Grayscale values.***

Figure 4.10: *Figure depicting how NEON registers and interleaving instructions can be used to perform RGB → GRAYSCALE conversion of 8 pixels.*

To compute box filter responses, it is important to recollect that at the end of Optimization 1 (Section: 3.1.3) it was decided that as **only 2 Octaves** are responsible for computing more than 80 percent of interest points, the last two octaves were dropped from computation. In Octaves 1 and 2 the **Scaling Down** values chosen for first and second Octaves were **2 and 4**, respectively, i.e., while computing **Octave 1** the box filters are moved to every alternate position of integral image, as shown in Figure 4.11(a) and while computing **Octave 2** the box filters are moved to every 4^{th} position of integral image, as shown in Figure 4.11(b).



(a) In Octave 1, box filters are moved to every alternate position of Integral Image.

(b) In Octave 2, box filters are moved to every 4^{th} position of Integral Image.

Figure 4.11: *Pictorial representation of how box filters are positioned and moved in Octave 1 and Octave 2.*

Registers used to store the values

Box filter responses are computed using integral image and the integral image has data of 32-bits floating point values and because the largest register size in NEON is 128 bits wide (**Q** registers 4.7), 4 32-bits integral image data values can be loaded into these registers, which in turn implies that 4 box filter responses can be computed simultaneously, as shown in Figure 4.12.

Instructions used to compute the responses

Due to two unique scaling values used in Octave 1 and Ocatve 2, to compute box filter responses in Octave 1 the *vld2.* and *vst2.* variant interleaving instructions are used and in Octave 2 *vld4.* and *vst4.* variant interleaving instructions are used. The C code with NEON intrinsics which is used to implement this is as given below and a representation of it is as shown in Figure 4.12.

```
float32x4_t BoxFilterCompute(int row, int col, int num_rows, int num_cols,
    unsigned int step_size, float three, int r, int c)
{

  int temp_row = row + num_rows;
  int temp_col = col + num_cols;

  int r1 = row - 1 ;
  int r2 = temp_row - 1;
  int c1 = col - 1;
  int c2 = temp_col - 1;

  float32x4_t v128_areas;

  float32_t *tl = NULL;
  float32_t *tr = NULL;
  float32_t *bl = NULL;
  float32_t *br = NULL;

  tl = (float32_t *)integral_image[r1] + c1;
  tr = (float32_t *)integral_image[r1] + c2;
  bl = (float32_t *)integral_image[r2] + c1;
  br = (float32_t *)integral_image[r2] + c2;

  struct box_input_structure box_filter_variables;

  switch (step_size) {
  case 2:

    box_filter_variables.ptr1 = tl;
    box_filter_variables.ptr2 = tr;
    box_filter_variables.ptr3 = bl;
    box_filter_variables.ptr4 = br;

    v128_areas = do_it2(tl, tr, bl, br, r, c);

  break;
  case 4:

    box_filter_variables.ptr1 = tl;
    box_filter_variables.ptr2 = tr;
    box_filter_variables.ptr3 = bl;
    box_filter_variables.ptr4 = br;

    _do_it4_asm(&box_filter_variables, &v128_areas);

  break;
    }

  static int x =0 ;

  if(three == 3.0)
  {
    float32x4_t v128_three = vdupq_n_f32(3.0);
    v128_areas = vmulq_f32(v128_areas,  v128_three);

  }
  return v128_areas;

}
static inline float32x4_t do_it2(float32_t *tl, float32_t *tr, float32_t *
    bl, float32_t * br, int r, int c) {
  float32x4x2_t top_left, top_right, bottom_left, bottom_right;
  float32x4_t A, B, C;

  top_left = vld2q_f32(tl);
  top_right = vld2q_f32(tr);
  bottom_left = vld2q_f32(bl);
  bottom_right = vld2q_f32(br);

  A = vsubq_f32(*top_left.val, *top_right.val);
  B = vsubq_f32(A, *bottom_left.val);
  C = vaddq_f32(B, *bottom_right.val);

  return C;
}

static inline float32x4_t do_it4(float32_t *tl, float32_t *tr, float32_t *
    bl, float32_t *br, int r, int c) {
  float32x4x4_t top_left, top_right, bottom_left, bottom_right;
  float32x4_t A, B, C;

  top_left = vld4q_f32(tl);
  top_right = vld4q_f32(tr);
  bottom_left = vld4q_f32(bl);
  bottom_right = vld4q_f32(br);

  A = vsubq_f32(*top_left.val, *top_right.val);
  B = vsubq_f32(A, *bottom_left.val);
  C = vaddq_f32(B, *bottom_right.val);

  return C;
}
```

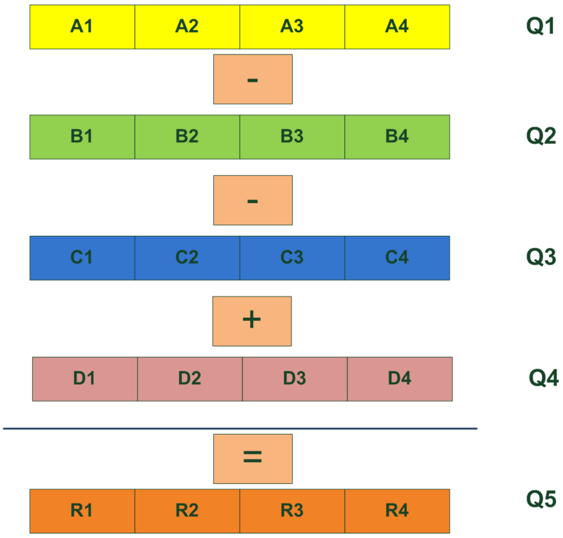


Figure 4.12: *Pictorial representation of how 4* ***box filter*** *responses are computed simultaneously.*

Image	Dimension	Building Scale-Space	Orientation Assignment and Descriptor building	Total
1	640 × 480	845,821	724,980	1,570,801
2	640 × 480	845,821	859,860	1,705,681
1	320 × 240	142,336	247,280	445,816
2	320 × 240	142,336	303,480	389,616

Table 4.3: *Distribution of BoxFilterCompute() function calls after vectorization using NEON, across **three steps** of SURF_ARM. The BoxFilterCompute() function calls within building Scale-Space is brought down by 4 **times**.*

With this optimization step the number of calls made to *BoxFilterCompute()* have been brought down by 4 times. The new number of calls made to *BoxFilterCompute()* are as shown in Table 4.3.

2. Integral Image computation using NEON

The next potential candidate for vectorization is **Integral Image** computation. Using NEON to compute Integral Image is not a straight forward step, the code written to compute an Integral Image has to be a combination of ARM and NEON instructions, the reason for this is the need for changing datatypes.

Building an Integral Image is the first and foremost step of SURF algorithm. The equation to build an integral image is as given in equation 2.1.1.

In simple words, every pixel of an integral image is a sum of pixel data at the corresponding position of the input image and all pixel values above and to the left of that position. The pixel data of input image is represented by 256 colors (8-bits), which means that the integral image value at position $(0, 0)$ is same as the input image pixel value at position $(0, 0)$, which could be a value between $(0 - 255)$ (represented by 8-bits). However, the integral image value at position $(1, 1)$ is a summation of all the pixel values of input image at $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$. Depending on pixel values at those positions the resulting value of the summation could be larger than $(0 - 255)$, which can't be represented by using only 8-bits. If all the pixel values of input image with dimensions (640×480) are white (value 255), then the value at position $(639, 479)$ of the integral image will be a summation of all the pixel values of input image, which is 78,336,000 and this value can only be represented using 32-bit datatype.

In situations where datatypes need to be changed, such as the one explained above, NEON processor cannot be used effectively. There should be an intermediate step which makes the datatypes homogeneous, this intermediate step steals away some performance. There are few NEON datatype conversion instructions available, but they are designed only to handle $FloatingPoint \leftrightarrow FixedPoint$ datatype conversions -

- float32x2_t vcvt_f32_u32(uint32x2_t)

- int32x4_t vcvtq_s32_f32 (float32x4_t)

The C code used to compute Integral Image is as given below;

```

#define PREFETCH_SIZE 64

void IntegralImageCompute(unsigned char *input_img)
{

  // Variables used in computing the integral image
  float partial_int[IMAGE_HEIGHT][IMAGE_WIDTH];

  unsigned char image_data[PREFETCH_SIZE];
  uint8x16_t vtemp128;

  int i, j, position = 0;

  //*************Type conversion using ARM*****************

  for(i=0;i<IMAGE_HEIGHT;i++)
  {
    position = i * IMAGE_WIDTH;
    partial_int[i][0] = *(input_img + position);

    for(j=1;j<IMAGE_WIDTH;j++)
    {
      position = i * IMAGE_WIDTH + j;

      //*************NEON Pre-fetching*************
      vtemp128 = vld1q_u8((input_img + (position)));
      vst1q_u8(image_data, vtemp128);

      vtemp128 = vld1q_u8((input_img + (position + 16)));
      vst1q_u8(image_data+16, vtemp128);

      vtemp128 = vld1q_u8((input_img + (position + 24)));
      vst1q_u8(image_data+24, vtemp128);

      vtemp128 = vld1q_u8((input_img + (position + 32)));
      vst1q_u8(image_data+32, vtemp128);

      for(k=0; k<PREFETCH_SIZE; k++)
      {
        partial_int[i][j] = partial_int[i][j - 1] + image_data[k];
        j++;
      }
    }
  }

  //*****************NEON PART************************

  float32x4_t temp1, temp2, temp3, temp4;

  for(i=0;i<2;i++)
  {
    for(j=0;j<IMAGE_WIDTH;j+=4)
    {
      if(i == 1)
      {
        temp1 = vaddq_f32(vld1q_f32(integral_image[i - 1] + j), vld1q_f32(
            partial_int[i] + j));
        vst1q_f32(integral_image[i] + j, temp1);
      }
      else
        vst1q_f32(integral_image[0] + j, vld1q_f32(partial_int[0] + j));
    }
  }

  int NEW_HEIGHT = IMAGE_HEIGHT - 2;

  for(i=2;i<NEW_HEIGHT;i+=4)
  {
    for(j=0;j<IMAGE_WIDTH;j+=4)
    {
      temp1 = vaddq_f32(vld1q_f32(integral_image[i - 1] + j), vld1q_f32(
          partial_int[i] + j));
      vst1q_f32(integral_image[i] + j, temp1);

      temp2 = vaddq_f32(vld1q_f32(integral_image[i] + j), vld1q_f32(
          partial_int[i + 1] + j));
      vst1q_f32(integral_image[i + 1] + j, temp2);

      temp3 = vaddq_f32(vld1q_f32(integral_image[i + 1] + j), vld1q_f32(
          partial_int[i + 2] + j));
      vst1q_f32(integral_image[i + 2] + j, temp3);

      temp4 = vaddq_f32(vld1q_f32(integral_image[i + 2] + j), vld1q_f32(
          partial_int[i + 3] + j));
      vst1q_f32(integral_image[i + 3] + j, temp4);
    }
  }
}
```

Experiments show that Vectorization of Integral Image computation brought down the computation time needed to build an Integral Image by around 38ms.

3. Other Vectorized Parts

Other functions of SURF_ARM which were vectorized are as follows:

- *BuildDescriptor()*: A loop in the last part of Building Descriptor was optimized. In SURF_ARM a series of 64 division operations had to be carried out, by making use of NEON this number was brought down by 4 times. However, no change in processing time was observed due to vectorization.

```
  float32x4_t v128_temp_len = vdupq_n_f32(1/len);
  \\ Due to the absence of any division NEON instruction,
  \\ instead of division by len, 1/len is used for multiplication

  int i_d;

  for(i_d = 0; i_d < DESCRIPTOR_SIZE; i_d+=4)
  {

    vst1q_f32(descriptor + i_d, vmulq_f32(vld1q_f32(descriptor + i_d),
         v128_temp_len));

  }
```

- *MatrixInverseMultiply()*

  A 3 × 3 matrix multiplication was vectorized, once again there was no change observed in processing time of matrix multiplication in SURF_ARM and SURF_ARM_NEON.

```

inline void mat_inverse_and_multiple(float H[][3], float dD[][1],
    float result[][1])
{
  float inverse[3][3];
  unsigned char i, j, k;
  if(inverse_matrix(H, inverse))
  {

    float32x4_t v128_result;

    v128_result = vmulq_n_f32(vld1q_f32(inverse[0]), dD[0][0]);
    v128_result = vmlaq_n_f32(v128_result, vld1q_f32(inverse[1]), dD
        [1][0]);
    v128_result = vmlaq_n_f32(v128_result, vld1q_f32(inverse[2]), dD
        [2][0]);

    v128_result = vnegq_f32(v128_result);

    vst1q_f32(result, v128_result);
  }
}
```

4. Pre-fetching

Integral image computation is an excellent and only part of SURF algorithm where the concept of pre-fetching data can be useful.

Over several decades now, Microprocessor performance has been improving dramatically by many scales, however, over this period there has always been a disparity between the improvement in Microprocessor's performance and the main memory dynamic RAM's performance. This difference is popularly known as "Processor-Memory Performance Gap" as depicted in Figure 4.13. The impact of this performance difference is an increase in processor's idle time waiting for memory to respond back with data. Several techniques have been invented to over come this latency, at the hardware level **Cache Hierarchy** is one such method, however, even in spite of having hierarchical caches there are still stalls in processing due to data unavailability. Fortunately this problem could be over come by using a technique known as **data pre-fetching**.

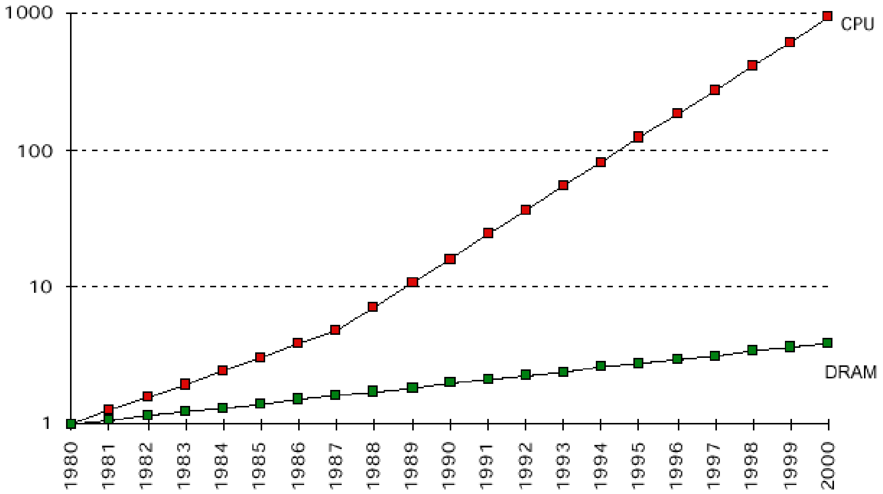


Figure 4.13: *Processor-Memory performance gap, borrowed from [4]*

While computing an Integral Image it was explained, how due to datatype mismatch between input data and final result NEON processor could not be used effectively and how this was overcome by making use of ARM instructions and building data of homogeneous datatype, which will then be used to build the Integral Image effectively using NEON. During this first step of Integral Image building, input image pixel data accessing is carried out sequentially, every time the processor needs the next pixel it has to fetch it. This waiting period can be overcome by pre-fetching a large packet of pixels into the memory hierarchy in one go.

Due to its **sequential locality** and due to large 128-bits registers of NEON and $16 \times 4 = 64$ bytes of input image pixel data are accessed into the memory hierarchy using NEON.

Looking at the Cortex-A8's architecture diagram in Figure 4.4 it is clear that NEON is logically located at the downstream of ARM. The sequence of events which go through during pre-fetching process are:

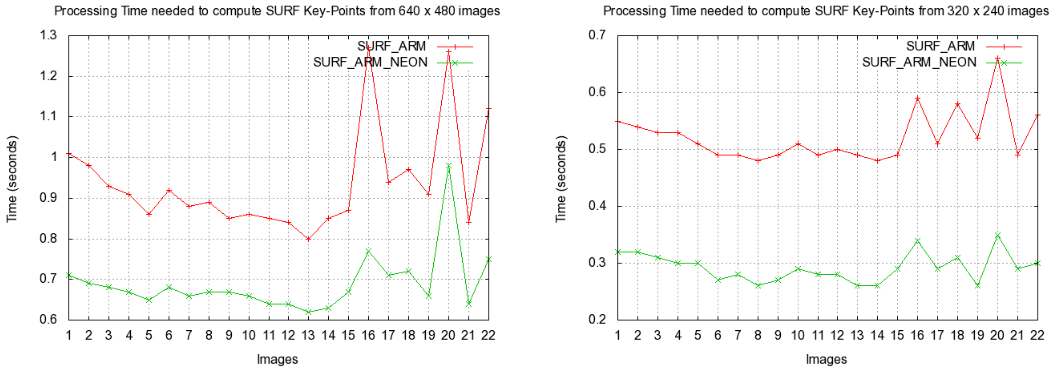
- The compiler issues a stream of instructions, which is a mixture of NEON and ARM instructions. When these instructions go through the ARM pipeline first, the NEON instructions go untouched, while the ARM instructions start to fill into the ARM pipeline and wait for data fetch to happen.

- The stream of NEON instructions which come out of the ARM unit starts to fill into the instruction queue of the NEON unit and then go through the NEON pipeline. Once the NEON pipeline is full, the first NEON instruction will be executed and the first set of 16 pixels will be fetched into the memory hierarchy.

- When the first 16 pixels are fetched, the ARM unit will be free to use them to build the partial integral image and the NEON is busy fetching the data, effectively, this introduces parallel operations in time between ARM and NEON.

The experiments conducted in this section shows that Pre-fetching reduces the Integral Image computation time by about 21ms.

From henceforth, the SURF_ARM version which has parts of it vectorized will be referred to as **SURF_ARM_NEON**. The processing time difference measured between the two versions while computing the Key-Points is as given below. For this test, once again the same two sets of twenty-two images are considered, as in previous tests. From the results as shown in Figure 4.14 it is clear that, by vectorizing parts of the algorithm has improved the processing time needed for images of 640 × 480 dimension by upto 33% and for images of 320 × 240 dimension by upto 46%.



(a) Processing time measured for images of dimensions 640 × 480. *A gain of 21 - 33% was achieved in processing time due to vectorization.*

(b) Processing time measured for images of dimensions 320 × 240. *A gain of 41 - 46% was achieved in processing time due to vectorization.*

Figure 4.14: *A comparison showing the performance gain achieved by **SURF_ARM_NEON** over **SURF_ARM**.*

4.4 Conclusion

The focus of this chapter was on the implementation and optimization of the SURF algorithm.

The reduction in number of computations carried out in Optimization 1 (Section 3.1.3) achieved a drop of **80 - 280 ms** of processing time for a 640 × 480 sized image and **20 - 100 ms** for a 320 × 240 sized image, when executed on the i.MX515EVK embedded platform.

To meet the objective of this project, which is to extract SURF Key-Points at real-time on an embedded system with limited resources, two strategies of optimizations were applied on SURF implementation:

- **Vectorizing Parts of SURF Algorithm** - The second level of optimization was on the basis of profiler report. From the report it was learnt that, nearly 75% of SURF_ARM time was spent in computing Box-Filter responses. For an image from which 129 Key-Points were extracted by SURF_ARM, the number of calls made to *BoxFilterCompute()* function was **3,383,284** times. Just the sheer number of times this function was called makes it a significant contributor to processing time. Once again by segregating the number of calls made to *BoxFilterCompute()* function, it was found out that, for an image of 640 × 480 dimensions, from which 129 Key-Points were extracted, **2,658,304** number of calls were made to *BoxFilterCompute()* function **while building the Scale-Space**. By making use of NEON uniquely in this situation, the number of calls made to *BoxFilterCompute()* while building Scale-Space, were dropped by **4 times** i.e., from **2,658,304** to **845,821**. The final SURF_ARM_NEON implementation takes only (a) 980ms to extract 199 features for 640 × 480 dimension images and (b) 350ms to extract 75 features for 320 × 240 dimension images.

- **Pre-fetching Data** - While computing Integral Image the sequential locality of image data can be exploited to pre-fetch 64 bytes of image pixel data in one transaction and while the ARM processor is busy using this pixel data, NEON will be busy fetching the data. This helps in overcoming memory latency. Pre-fetching image data brought down around 21ms of processing time of the Integral Image computation.

The new processing time achieved due to vectorization will indeed be sufficient to compute SURF Key-Points from images with 320 × 240 ***dimensions at real-time, at a frame rate of about 3 frames per second.*** The following chapter shows how the SURF Key-Points can be used in object identification. following chapter SURF_ARM_NEON is used in building a real-time object identification application.

5 A Real-Time Object Identification Application

In the previous chapters, the SURF optimization was carried out at two levels: (a) Algorithm optimization, and (b) Vectorization of important parts of SURF implementation. As a result the SURF_ARM_NEON implementation which makes use of the NEON vector facility can compute Key-Points from images of dimensions 320×240 pixels at a rate of around 3 frames per second, suitable for a real-time object identification application.

In this chapter, details of how SURF_ARM_NEON could be used to build a Real-Time object identification application are covered. A new algorithm is introduced which is used to compute matches between two sets of Key-Points. Both the SURF algorithm and the Key-Points matching algorithm are used to build a real-time object identification application, while the task chosen to be solved using this application is the identification of Euro Currency notes from real-time input image frames captured by a camera.

5.1 The Euro Currency Notes Identification Application

The **Euro Currency Notes Identification** application is designed to identify the seven Euro currency notes: 5, 10, 20, 50, 100, 200, and 500, irrespective on what side they are presented in front of the camera.

At the core of this application there are two algorithms: (a) **SURF Key-Points extraction algorithm**, which is used to extract SURF Key-Points from a given input image frame and (b) **Key-Points matching algorithm**, to which, two sets of Key-Points extracted from two images are given as inputs, to determine if they both match or not.

The block diagram of this application is presented in Figure 5.1. The functionality of this application is achieved in two steps, as described in the following two sections -

5.1.1 Step 1: Extracting SURF Key-Points

The objective of the first step in this application is to extract SURF Key-Points at real-time from image frames captured by a camera.

Input Camera

The camera chosen for this project is a Logitech C200 Web-Camera [35]. Some of its features relevant to this project are as given below:

- VGA sensor (640 x 480 pixels)
- Video capture up to 30 frames per second (with recommended systems)

- Hi-Speed USB 2.0
- Manual focus

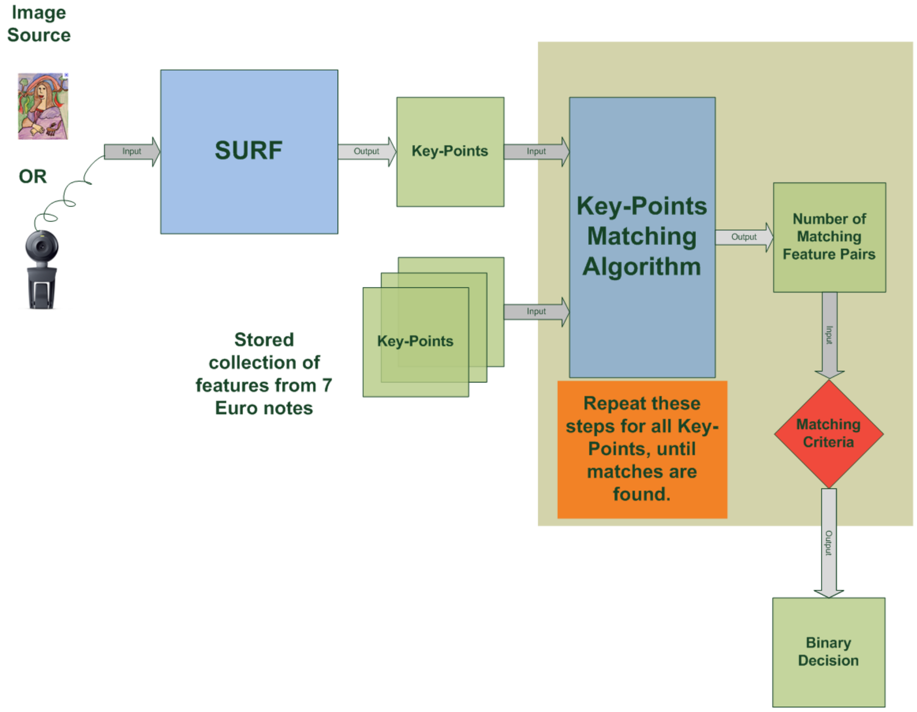


Image Source
OR
Input
SURF
Output
Key-Points
Input
Key-Points Matching Algorithm
Output
Number of Matching Feature Pairs
Input
Key-Points
Stored collection of features from 7 Euro notes
Repeat these steps for all Key-Points, until matches are found.
Input
Matching Criteria
Output
Binary Decision

Figure 5.1: Block diagram of Euro Currency Notes Identification Application.

By definition [36] a VGA camera provides 640 × 480 pixel images with 256 colors, however the SURF implementation used in Euro Currency Notes Identification application is designed to handle only Grayscale images, so, the $RGB \rightarrow GRAYSCALE$ conversion is carried out at the software level. This conversion is done very efficiently using NEON as explained in Section 4.3.6.

OpenCV

In the context of input image source it is also important to learn about how image frames are captured and displayed on a screen.

Developed by Intel, OpenCV (**O**pen Source **C**omputer **V**ision) libraries are used for a variety of programming functions of real-time computer vision. It provides an interface to a USB camera and to display the captured frames on a screen. The OpenCV libraries are not platform specific so they can be built for an i.MX515 processor using open source tool chain. Because of a full fledged Ubuntu 9.10 operating system running on i.MX515EVK, OpenCV libraries could be built directly on the i.MX515EVK platform.

Some basic functionalities necessary for this application, such as:

1. Capturing image frames from camera and displaying them on screen.

2. Displaying text on image frame.

3. Resizing of images from 640 × 480 → 320 × 240.

are carried out using OpenCV functions.

A function which makes use of OpenCV functions **to capture 100 frames** from a USB camera, **resize** them from 640 × 480 → 320 × 240 and **display** the frames on a screen is as shown below.

```

#include<cv.h>
#include<highgui.h>

void CaptureResize()
{
  IplImage *temp_color_frame = NULL;
  IplImage *current_gray_frame = NULL;
  IplImage *temp_current_gray_frame = NULL;

  CvCapture *capture = 0;
  int camera_index = CAMERA_INDEX;
  unsigned int frame_counter = 0;

  if(CameraSetup(camera_index, &capture))
  {

    temp_color_frame = NULL;

    temp_current_gray_frame = cvCreateImage(cvSize(640, 480), IPL_DEPTH_8U, 1);
    current_gray_frame = cvCreateImage(cvSize(320, 240), IPL_DEPTH_8U, 1);

    cvNamedWindow("Current Frame", CV_WINDOW_AUTOSIZE);
    cvMoveWindow("Current Frame", WINDOW_POSITION, 0);

    ///////////////////////////////////////////////////////////////
    // Configure the fonts needed to display text on the image frame
    ///////////////////////////////////////////////////////////////
    CvFont font, font1;
    cvInitFont(&font, CV_FONT_HERSHEY_SIMPLEX, 0.3, 0.3, 0, 1, CV_AA);
    cvInitFont(&font1, CV_FONT_HERSHEY_SIMPLEX, 1.0, 1.0, 0, 1, CV_AA);

    while(frame_counter < 100)
    {
      // Capture a color frame from the camera
      temp_color_frame = cvQueryFrame(capture);

      if(temp_color_frame != NULL)
      {
        // Convert the RGB -> GRAYSCALE
        ColorConversion(temp_color_frame, temp_current_gray_frame);

        cvResize(temp_current_gray_frame, current_gray_frame, CV_INTER_LINEAR);
        cvShowImage("Current Frame", current_gray_frame);

        frame_counter++;
      }
    }
    cvDestroyWindow("Current Frame");

    cvReleaseCapture(&capture);
  }
  else
  {
    error = CAMERA_NOT_SETUP;
  }

  return 0;
}
```

The ***IplImage*** is a structure used to store all data related to an image frame. The structure of IplImage is as shown below.

```
typedef struct _IplImage
{
    int  nSize;
    int  ID;
    int  nChannels;
    int  alphaChannel;
    int  depth;
    char colorModel[4];
    char channelSeq[4];
    int  dataOrder;
    int  origin;
    int  align;
    int  width;
    int  height;
    struct _IplROI *roi;
    struct _IplImage *maskROI;
    void  *imageId;
    struct _IplTileInfo *tileInfo;
    int  imageSize;
    char *imageData;
    int  widthStep;
    int  BorderMode[4];
    int  BorderConst[4];
    char *imageDataOrigin;
}
IplImage;
```

The OpenCV function ***cvQueryFrame();*** to capture a RGB format image frame from the USB camera and load into an **IplImage** type data structure.

After a color image frame is captured from the camera, two intermediate steps are necessary, which are:

- **Color to Garyscale conversion:** In this project the SURF algorithm is designed to handle only Grayscale images hence, the color image captured from the camera must be converted to grayscale by using NEON-optimized color conversion function $ColorConversion(temp\_color\_frame, temp\_current\_gray\_frame)$;. Color conversion using NEON was explained in detail, in the previous chapter (Refer to Section 4.3.6).

- **Scaling:** From the results of the tests conducted in the last section of the previous chapter, it was noticed that, only for images of 320 × 240 pixels dimension that SURF_ARM_NEON could extract features at a frame rate of around 3 frames per second. So, scaling down of input image frames from 640 × 480 to 320 × 240 must be carried out. This is achieved by using OpenCV's $cvResize(temp\_current\_gray\_frame, current\_gray\_frame, CV\_INTER\_CUBIC)$; function.

With the end of resizing, the SURF algorithm will start processing the frame to extract Key-Points.

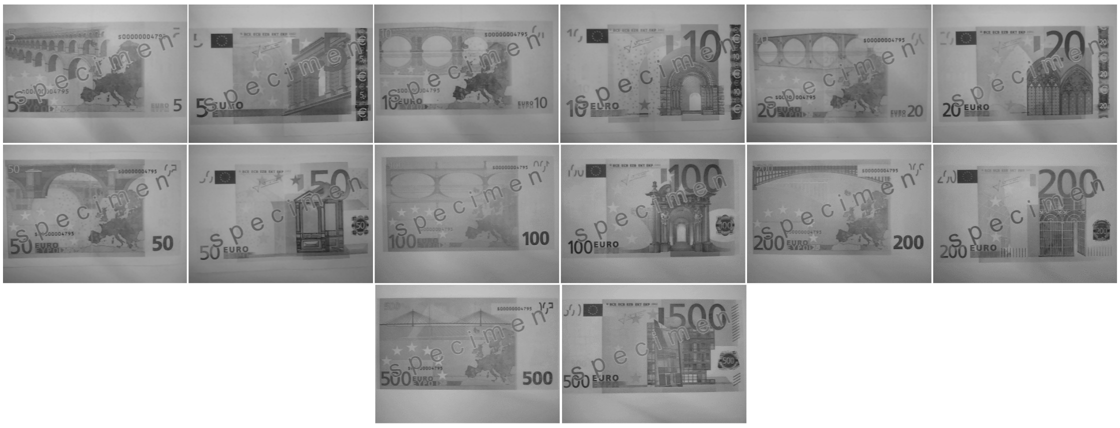


Figure 5.2: *Fourteen **Model Images** of 7 Euro currency notes from which **Model Key-Points** are extracted on a one-by-one fashion. These Key-Points are then made as a part of the application which will be used during the matching phase.*

5.1.2 Step 2: Matching the Key-Points and Decision Making

After the SURF Key-Points are extracted from a given frame, they are given to a matching algorithm. The matching algorithm takes the description of every Key-Point from the current image frame and match it against the description of every Key-Point in a collection of Key-Points known as **Model Key-Points**. The model Key-Points collection is always part of the application and they are extracted in a one-by-one fashion from all $7 \times 2 = 14$ sides of Euro currency notes, known as Model Images, as shown in Figure 5.2.

5.1.2.1 Key-Points Matching Algorithm

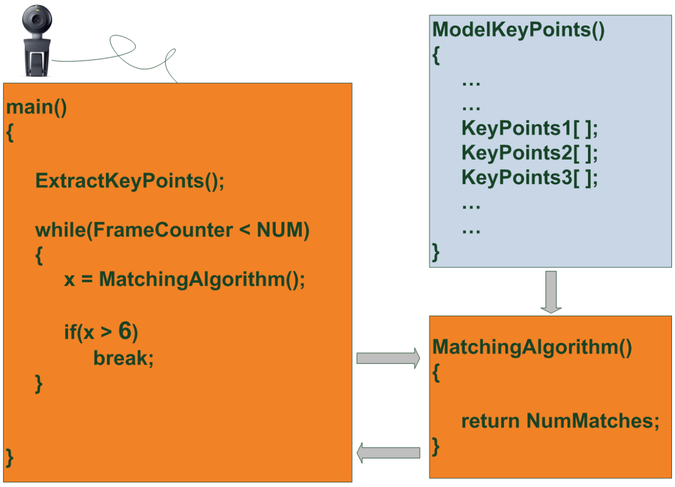
Every Key-Point extracted by the SURF algorithm is expressed in 64 floating point values, which describe a point and its surroundings.

A Key-Point of one image is said to be matched with a Key-Point of another image based on how similar their descriptions are. The similarity is measured by calculating the aggregate difference or also known as **R**oot-**S**um-**S**quare (RSS) [37] of the 64 floating point valued description of one Key-Point with the 64 floating point values description of another Key-point. The formula used to compute RSS between two points P_1 and P_2, is as follows:

$$difference = \sum_{i=0}^{63} \sqrt{(P_1.descriptor[i])^2 - (P_2.descriptor[i])^2} \tag{5.1.1}$$

The difference between every Key-Point of the model Key-Points collection and every Key-Point of the current frame is computed and the Key-Point pairs which have the lowest value of difference are considered to be **matching pairs**. The matching algorithm returns the number of matching pairs found between Key-Points of a model image and the Key-Points found in the current frame. From extensive tests conducted it was found out that, greater than **6** matching pairs found confirms that the model image and the current image frame are the same.

This entire operation of matching can be clearly understood from the block diagram in Figure 5.3.



```
main()
{

    ExtractKeyPoints();

    while(FrameCounter < NUM)
    {
        x = MatchingAlgorithm();

        if(x > 6)
            break;
    }

}
```

```
ModelKeyPoints()
{
    ...
    ...
    KeyPoints1[ ];
    KeyPoints2[ ];
    KeyPoints3[ ];
    ...
    ...
}
```

```
MatchingAlgorithm()
{

    return NumMatches;
}
```

Figure 5.3: *Block diagram describing how matching algorithm is interfaced with Key-Points extraction algorithm and how the result from matching algorithm is used in the application.*

The C code used in computing the Matching Key-Points is as given below:

```
unsigned int ExtractMatchingFeatures(struct MATCHING_PAIRS *matching_pairs,
    struct INTEREST_POINT *model, int number_model_i_points, struct
  INTEREST_POINT *target, int number_target_i_points)
{
  int i, j, k;
  float diff;
  float diff_square;
  unsigned int number_of_matches_found = 0;

  for(i=0; i< number_model_i_points; i++)
  {
    float besterror = FLT_MAX;
    INTEREST_POINT best_possible_matched_point = NULL;

    for(j=0; j< number_target_i_points; j++)
    {
      if(model[i].laplacian == target[j].laplacian)
      {
        ////////////////////////////////////////////////////////////
        // Calculating the RSS for difference between every point of the
        // model and every point in the target
        ////////////////////////////////////////////////////////////
        diff = 0;
        diff_square = 0;

        for(k=0; k<DESCRIPTOR_SIZE;k++ )
        {
          diff = (model[i].surf_descriptor[k] - target[j].surf_descriptor[k
             ]);
          diff_square += (diff * diff);
        }

        diff = sqrt(diff_square);

        if(diff < besterror)
        {
          besterror = diff;
          best_possible_matched_point = target[j];
        }
      }
    }

    if(best_possible_matched_point != NULL)
    {
      matching_pairs[number_of_matches_found].x_model = model[i].x;
      matching_pairs[number_of_matches_found].y_model = model[i].y;
      matching_pairs[number_of_matches_found].x_target =
          best_possible_matched_point.x;
      matching_pairs[number_of_matches_found].y_target =
          best_possible_matched_point.y;
      number_of_matches_found++;
    }

  }

  return number_of_matches_found;
}
```

Upon profiling the real-time object identification application, it was found that, the *ExtractMatchingFeatures()* is one of the significant contributors to the computation time.

To match a set of **71** Key-Points extracted from an image, with **503** Key-Points in the Key-Points collection of the application, the matching algorithm would take close to **97ms**. By looking at its implementation it was found that critical parts of this function can be vectorized. So, once again parts of the matching algorithm were vectorized and the C code with which NEON was programmed to compute the cumulative difference between two Key-Points is as shown below. With this optimization the matching algorithm's computation time was dropped to **42ms**, a 56.7% improvement.

The C code for matching algorithm, which is optimized by using NEON intrinsics is as given below.

```
unsigned int ExtractMatchingFeatures(struct MATCHING_PAIRS *matching_pairs, struct INTEREST_POINT *model, int number_model_i_points, struct INTEREST_POINT *target, int number_target_i_points)
{
  int i, j, k;
  float diff;
  float diff_square;
  unsigned int number_of_matches_found = 0;

  for(i=0; i< number_model_i_points; i++)
  {
    float besterror = FLT_MAX;

    INTEREST_POINT best_possible_matched_point = NULL;

    float32x4_t v128_model = vld1q_f32(model[i].surf_descriptor);

    for(j=0; j< number_target_i_points; j++)
    {
      if(model[i].laplacian == target[j].laplacian)
      {
        ////////////////////////////////////////////////////////////////
        // Calculating the RSS for difference between every point of the
        // model and every point in the target
        ////////////////////////////////////////////////////////////////
        float32x4_t v128_diff, v128_diff_square;

        int k;
        v128_diff = vsubq_f32(v128_model, vld1q_f32(target[j].surf_descriptor));
        v128_diff_square = vmulq_f32(v128_diff, v128_diff);

        for(k=4; k<DESCRIPTOR_SIZE;k=k+4)
        {

          v128_diff = vsubq_f32(vld1q_f32(model[i].surf_descriptor+k), vld1q_f32(target[j].surf_descriptor+k));
          v128_diff_square = vmlaq_f32(v128_diff_square, v128_diff, v128_diff);
        }

        float32x2_t v64_result_temp;
        v64_result_temp = vadd_f32(vget_low_f32(v128_diff_square), vget_high_f32(v128_diff_square));

        float32_t diff_square;
        diff_square = vgetq_lane_f32(v64_result_temp, 0) + vget_lane_f32(
            v64_result_temp, 1);

        diff = sqrt(diff_square);

        if(diff < besterror)
        {
          besterror = diff;
          best_possible_matched_point = target[j];
        }
      }
    }

    if(best_possible_matched_point != NULL)
    {
      matching_pairs[number_of_matches_found].x_model = model[i].x;
      matching_pairs[number_of_matches_found].y_model = model[i].y;
      matching_pairs[number_of_matches_found].x_target =
          best_possible_matched_point.x;
      matching_pairs[number_of_matches_found].y_target =
          best_possible_matched_point.y;
      number_of_matches_found++;
    }

  }

  return number_of_matches_found;
}
```

5.2 Application Performance

Now that the real-time object identification application is completely implemented, its performance measurement can be conducted. Which is carried out by taking into consideration the following two aspects:

- The Capacity to handle the disturbing conditions, and
- The Processing time or the Frame rate.

Apart from this, other measurements such as (a) Utilized Memory and (b) Energy consumption, two very important properties associated with any Embedded Application, are also measured in this section.

5.2.1 Robustness to Changing Conditions

In this section, a test on the application's robustness to changing conditions or disturbances such as - **scale**, **blur**, **orientation**, **illumination**, and **viewpoint** is carried on.

For this test a hard copy of a 50 Euro currency note specimen is considered (Refer to Figure 5.2). This is subjected to various disturbing conditions, while these disturbing conditions are being applied on the specimen, using the camera 100 continuous image frames are captured and each image frame is given as an input to the application. The SURF algorithm of the application extracts the Key-Points and the matching algorithm then match these Key-Points with the Model Key-Points. The results tabulated below are an average of all the results obtained across 100 frames. Any value larger than 6 in the **Matches Found** column indicates a **true match** between a captured frame and the model image.

Scaling Level	**Avg. number of Key-Points Extracted**	**Avg. number of Matches Found**
-2 Down Scale	49	0
-1	65	9
0 Normal	59	10
1	64	8
2	104	7
3	62	10
4 Up Scale	78	0

Table 5.1: *Performance Measurement:* ***Scaling***

Blur Level	**Avg. number of Key-Points Extracted**	**Avg. number of Matches Found**
0 No Blur	66	9
1	58	12
2	43	7
4 Extreme	50	2

Table 5.2: *Performance Measurement:* ***Blurring***

Illumination	**Avg. number of Key-Points Extracted**	**Avg. number of Matches Found**
Darker	64	2
Dark	69	7
Normal	55	7
Bright	71	11
Brighter	60	10
Brightest	52	4

Table 5.3: *Performance Measurement:* ***Illumination***

Viewpoint	**Avg. number of Key-Points Extracted**	**Avg. number of Matches Found**
View 1	64	7
View 2	69	4
View 3	55	0
View 4	71	0
View 5	60	1
View 6	52	0

Table 5.4: *Performance Measurement:* ***Viewpoint***

Orientation	**Avg. number of Key-Points Extracted**	**Avg. number of Matches Found**
+180°	49	0
+135°	49	4
+90°	50	9
+45°	47	7
0°	52	8
−45°	60	7
−90°	62	3
−135°	50	1
−180°	49	7

Table 5.5: *Performance Measurement:* ***Orientation***

With the end of Viewpoint change tests, all the tests related to how well the application can handle the disturbing conditions at real-time have been completed.

Ultimately, the performance of an objected-identification application which makes use of Key-Points depends on the nature of Key-Points it extracts, from the results it is clear that the application is able to handle the scaling (Refer to Table 5.1), **blurring** (Refer to Table 5.2), a majority of **orientation** changes (Refer to Table 5.5) and **illumination** (Refer to Table 5.3) changes effectively, however, on the basis of SURF Key-Points which do not perform well with viewpoint changes as seen in Chapter 2, it is expected that the application performs poorly with **viewpoint** changes (Refer to Table 5.4).

5.2.2 Processing Time

Once again, to measure the frame rate, 100 image frames are captured continuously from the camera and they are given as inputs to the application, the average time taken by the application to process them is considered as the average frame rate at which the application operates.

Two versions of real-time object identification application are built, the first version, built around using the SURF_ARM and the second version based on the

SURF_ARM_NEON. The results of this tests are presented in Table 5.6.

Application Name	**Average fps** 640 × 480	**Average fps** 320 × 240
With SURF_ARM	0.89	1.79
With SURF_ARM_NEON	1.42	**3.12**

Table 5.6: *Performance Measurement:* ***Processing Time***

Looking at the processing time it is clear that the application built using SURF_ARM_NEON almost doubles the average processing time for image frames of both sizes and the application built using SURF_ARM_NEON for scaled down images (640 × 480 → 320 × 240) is the best version amongst the four. There is a further scope in improving this which is explored in the next section

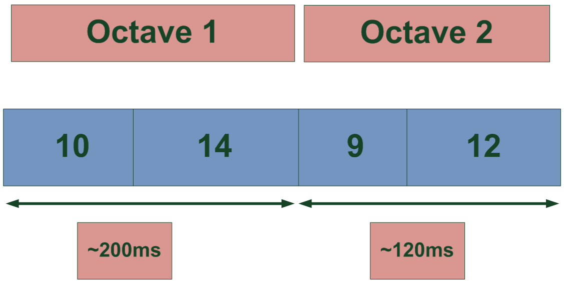


Figure 5.4: *Distribution of 45 Key-Points and ∼320ms of processing time across 2 Octaves for a* 320 × 240 *image.*

5.3 Optimization 3: Progressive SURF Key-Points Extraction and Matching

Progressive SURF is an innovative way in which matching Key-Points extracted from SURF algorithm can be carried out even more effectively.

In any object identification application which makes use of Key-Points, the conventional method of operation is to extract Key-Points from all Octaves and then send them to the matching algorithm, which then tries to match all those Key-points with Key-Points of other images. Here a different approach is used which is referred to as **Progressive Matching** in this project. The idea of progressive matching is that, because Key-Points extraction is distributed across Octaves, matching of Key-Points can also be carried out at stages. As soon as some number of Key-Points are extracted by the algorithm, those points can already be used for matching.

It was learnt in previous chapters that, within an Octave Key-Points are extracted in two iterations and if in an algorithm implementation there are 4 Octaves then there are $4 \times 2 = 8$ points where matching of Key-Points could be carried out. In the SURF_ARM or SURF_ARM_NEON implementations used in the application of this project, there are 2 Octaves, which means that there are **4 points** where matching of Key-Points could be carried out. If the Key-Points extracted after first iteration are sufficient to be used in establishing a match, then the burden of computing more Key-Points can be given up and the next frame can be captured for processing.

For example, the distribution of 45 Key-Points and around 320ms processing time between 2 Octaves from a 320 × 240 image frame is as shown in Figure 5.4. In this example, right at the point when 10 Key-Points are computed, they can be used for matching and if the matching algorithm is already able to establish a match, nearly 220ms of processing time and the additional energy drawn by the application to compute those extra Key-Points can be saved.

Although this method sounds very good in cutting down on processing time, the stages at which matching algorithm is called must be chosen carefully, because, if there are 2 Octaves and matching is carried out at the end of every iteration within an Octave and if a match was unable to be established at the end of a certain early iteration, then by the end of second Octave the matching algorithm would have already been invoked four times and still no match would have been established. Keeping this reason in mind, in SURF_ARM and SURF_ARM_NEON the matching was carried out only at the end of every Octave rather than at the end of every iteration within an Octave, so even in worst case scenarios, the matching algorithm would have been called only twice.

Several tests presented in Figure 5.5 were conducted where it was noticed that, when considerable number of frames were matched right by the end of first Octave, then the total time taken to process 100 frames was drastically reduced. This time is nearly 20% lesser when compared with time taken by the application with SURF_ARM_NEON with normal matching and about 56% lesser than SURF_ARM with progressive matching.

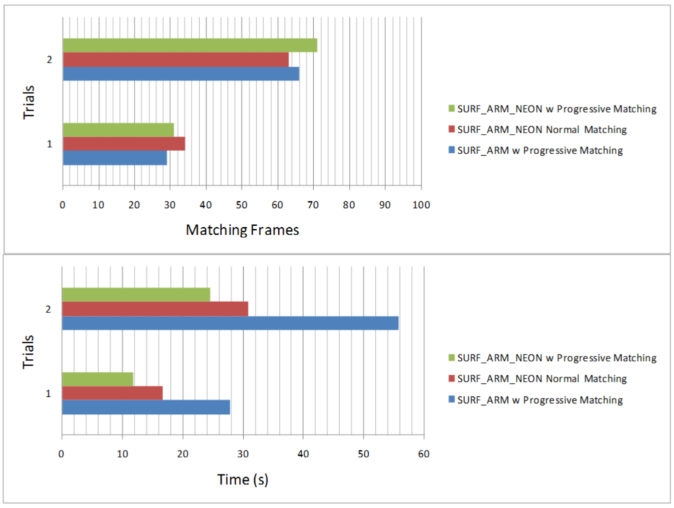


Figure 5.5: *Progressive search improves the total time taken to process frames. In* ***Trial 1 50 frames*** *were considered and* ***Trial 2 100 frames*** *were considered.*

By the end of progressive matching the average frame rate of the real-time object identification application was around 3.5 - 4 frames per second.

5.4 Other Measurements

In this section, the application Memory and Energy consumption are discussed.

5.4.1 Memory Requirements

The memory requirements can be classified into two types, (a) Application Size and (b) Runtime Memory Requirements.

5.4.1.1 Application Size

The real-time object identification application consists of three parts, (a) **SURF algorithm**, (b) **Matching Algorithm**, and (c) **the collection of all Model Key-Points**. The Table 5.7 shows the sizes of all the three individual parts of the application. Once again there are two versions of applications considered here, one with SURF_ARM and one with SURF_ARM_NEON. The slight increase in application size with SURF_ARM_NEON can be attributed to compiler's inability to optimize the code in these functions as they have large parts of NEON code written using NEON intrinsics.

Application Name	**SURF Algorithm** (bytes)	**Matching Algorithm** (bytes)	**Model Key-Points** (bytes)	**Total Size** (bytes)
With SURF_ARM	34401	476	138828	173795
With SURF_ARM_NEON	34828	517	138828	174173

Table 5.7: *Memory Requirements:* ***Application Size***

5.4.1.2 Run-Time Memory Requirements

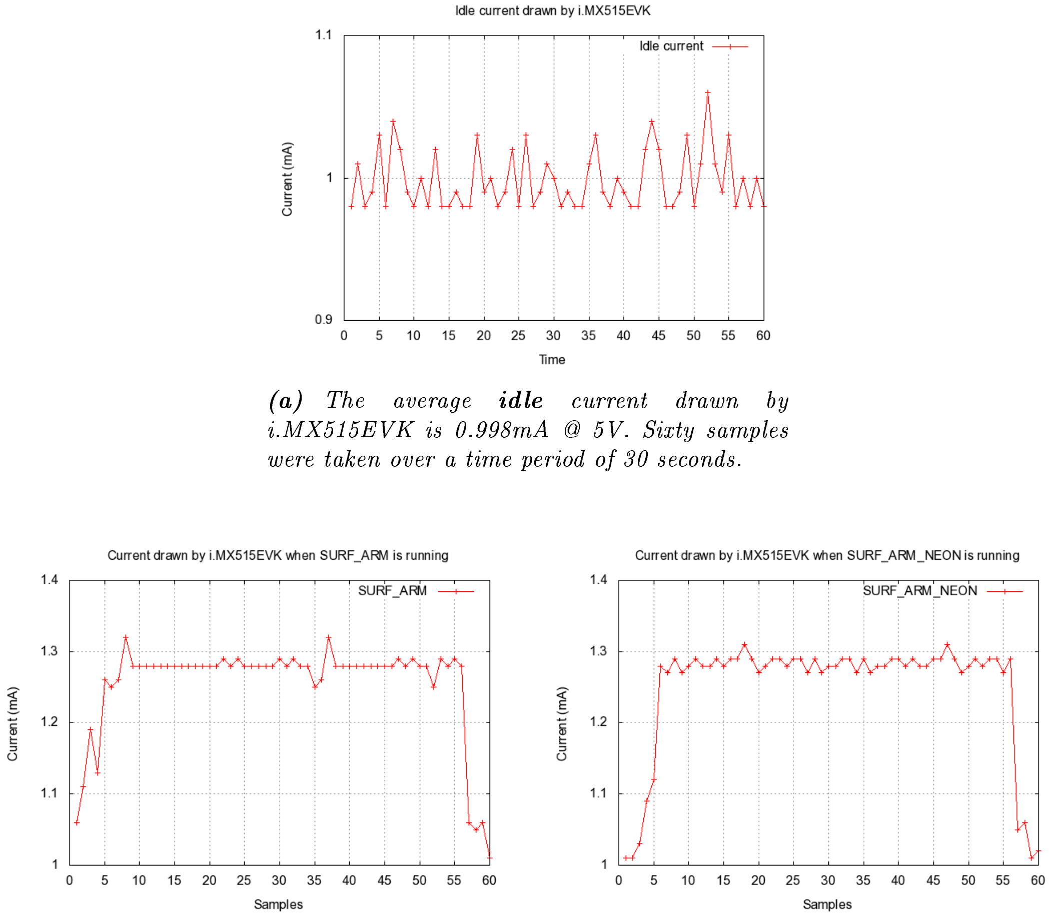
On the i.MX515EVK there is a 512MB DDR-RAM, in this section, the run-time usage of this memory by the application is recorded. The Htop tool in Ubuntu OS gives an insight into all the process running and their current memory usage.

To measure the memory usage once again, the application is made to capture and process 100 frames and the average memory usage of this application during this period is given in Table 5.8.

Although in case of 640 × 480 sized image frames have 4 times more pixels when compared with the image frames of 320 × 240 sized image frames, the run-time memory requirement of the former one is only about 1.4 times higher.

Application Name	Approximated Memory Usage 640 × 480 MB	Approximated Memory Usage 320 × 240 MB
With SURF_ARM	6.16	4.42
With SURF_ARM_NEON	6.16	4.42

Table 5.8: *Memory Requirements:* ***Runtime memory***



(a) *The average* ***idle*** *current drawn by i.MX515EVK is 0.998mA @ 5V. Sixty samples were taken over a time period of 30 seconds.*

(b) *The average current drawn by i.MX515EVK when SURF_ARM was running is* ***1.254mA*** *@ 5V. Sixty samples were taken over a time period of 30 seconds.*

(c) *The average current drawn by i.MX515EVK when SURF_ARM_NEON was running is* ***1.248mA*** *@ 5V. Sixty samples were taken over a time period of 30 seconds.*

Figure 5.6: *Current drawn by i.MX515EVK.*

5.4.2 Energy Utilizations

The i.MX515EVK is powered by a 5V power supply. In its idle[1] state the average current drawn during by the board is ∼0.998mA.

When the SURF_ARM is made to run on the platform the current drawn by the board is higher than during the idle state. The processor utilization is at 98% and the average current drawn by the kit during that time is ∼1.254mA. Similarly, when SURF_ARM_NEON runs with 98% processor utilization the average current drawn is once again ∼1.248mA. Which means that the average additional current drawn by applications with both versions of algorithms is **0.256mA** and **0.25mA**, respectively, which is about 25% increase.

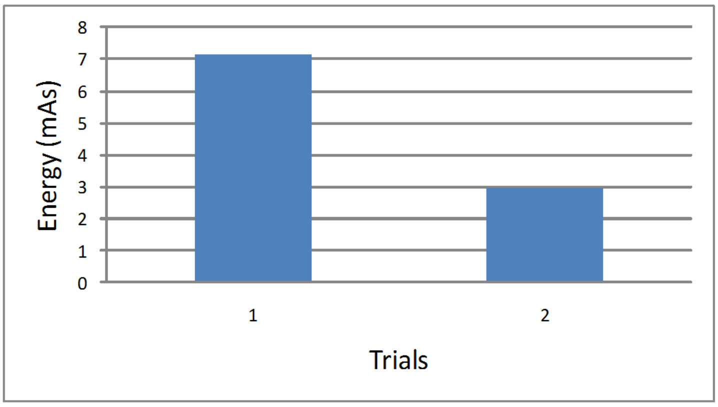


Figure 5.7: *Average Energy consumed by SURF_ARM and SURF_ARM_NEON. The trials were conducted for applications with SURF_ARM and SURF_ARM_NEON to process 100 frames. Around 58% of lesser energy is consumed by the application when NEON is used.*

In the context of energy utilization, an important question arises regarding the usage of NEON: Does the NEON utilization consume more energy? The answer to this is yes, to use NEON more energy is definitely needed, however, to quote a specific figure is not possible at this moment due to the absence of any mentioning of it in the i.MX515 processor literature. Upon observing the pipeline architecture (Refer to Figure 4.4) of ARM Cortex-A8 it is easy to assume NEON to be a part of ARM processor core and to act more as an instruction set extension and less as an independent core, so, using NEON only needs very insignificant amount of additional energy. This can be observed upon comparing the results from the conducted tests, where the current drawn by the kit is measured when only ARM (Refer to Figure 5.6(b)) and ARM+NEON (Refer to Figure 5.6(c)) are used.

Although from the current drawn results it can be concluded that a very small additional power is needed to utilize NEON, the total amount of energy used-up when NEON

[1] During this period USB peripherals such as Key-board, Mouse and Camera were connected to the board.

is turned ON should be comparatively lesser because more work is done by using NEON during shorter amount of time.

For example, taking the cases of applications with SURF_ARM and SURF_ARM_NEON + Progressive-Matching in Section 5.3 into consideration, the time needed by the application with SURF_ARM is **27.9s** and the average current drawn by the application during this time is **0.256mA**, so, the energy consumed by the application is **7.1424mAs**. On the other hand, the time needed by SURF_ARM_NEON with progressive matching is **11.82s** and the average current drawn during this time by application is **0.25mA**, so, the energy consumed is only **2.955mAs**, which is around 58.5% decrease in energy consumption.

5.5 Conclusion

The details of how a real-time object identification application which is intended to identify 7 Euro currency notes were discussed in this chapter. This application embeds two algorithms:

- **Key-Points extraction algorithm**: SURF_ARM_NEON is used to extract Key-Points at around 3fps.
- **Key-Points matching algorithm**: A Key-Points matching algorithm which is based on Root of Sum of Squares(RSS) is used to find matches between Key-Points extracted from two images.

Once again while implementing the matching algorithm it was found that, crucial part of that algorithm can be vectorized. The NEON vectorized matching algorithm takes nearly **56%** lesser computation time to complete matching of 71 Key-Points against a collection of 501 Key-Points.

The nature of the Key-Points decide how well the application performs in handling the disturbing conditions, unlike SIFT Key-Points, the SURF Key-Points are not very good at handling viewpoint changes, which reflects in the application's poor performance to view point changes.

One of the very important achievements while building the real-time object identification application was, showing that it is possible to progressively match Key-Points as and how they are computed, rather than computing all Key-Points in one go and then use them to match them; which is how it is done currently in applications which make use of Key-Points. From several tests conducted here, it was found that when the matching algorithm is able to find considerable number of matching frames right at the end of Key-Points extracted from the first Octave, the total time taken by the application to process 100 frames decreases by upto **20%**. This is a further boost to the application as this impacts the application's throughput (3.5 - 4 frames per second) directly and it also improves the runtime memory and energy utilization of the application.

NEON unit is extensively used in this application to bring up the speed and a question which arises as an outcome of this excessive usage is, does NEON contributes significantly in increasing the energy consumption of the application? NEON is basically an electronic hardware unit and additional energy is definitely needed to use it. But during the tests conducted it was found that, the current drawn by the application which makes use of NEON is so minuscule that it is practically unobservable and the average amount of current consumed over a significant duration of time by the application which makes use of NEON unit is similar to the application which makes of only ARM unit. Nearly 58% of energy reduction is found when NEON version of the application is used, because, the amount of work done while making use of NEON is far higher. The following chapter is the concluding chapter, it gives an outlook of the whole project work carried out and it also gives the details for future research that can be carried out in this field.

Conclusions and Future Work 6

In this project a study was conducted to find out, how a real-time object identification can be built starting from a feature extraction algorithm. This chapter gives a summary and conclusions of the work carried out and also possible future work related to this project.

6.1 Summary and Conclusions

Image features are similar to what frequencies or keywords are for sound and text. Extracting information from images can be very valuable in obtaining the knowledge of the content in the image.

Images contain various kinds of features such as - lines, corners or edges, color information, Key-Points and so on. Every one of these features are useful to address certain specific problems. **Key-Points** are useful in situations where visual disturbances occur, because, unlike other features, Key-Points contain description of a point and its surroundings and the algorithms developed to extract the Key-Points are resilient to changes such as - scale, orientation, illumination, blurring and viewpoints.

To choose an algorithm for this project, by keeping the possible image disturbances such as scaling, orientation, blurring, illumination and viewpoint changes are parameters, both SIFT and SURF were tested to find out how well they can perform in those conditions. From the tests it was found that, both SIFT and SURF perform well in handling certain disturbing conditions.

Both SIFT and SURF are computationally intensive algorithms and if the application which is built around it is running on an embedded platform with limited resources, achieving real-time Key-Point extraction becomes a challenging task. Implementations of both the algorithms were put to test to on a Linux/Pentium 4 system [1]. From these tests it was found that SURF has very low computation time when compared to SIFT. While, the average time needed by SIFT to compute features from 33 images was **1.4s**, the average time needed by SURF to compute almost same number of Key-Points from same set of images was only **0.26s**. On the basis of the results from these tests, the SURF algorithm was chosen over SIFT. The reduction in processing time of SURF can be attributed to the nature of filters used in building the Scale-Space. By making use of Integral Images the box filter responses were computed with only **5 additions** and **1 multiplication**.

The SURF algorithm was implemented following the four steps, Scale-Space Analysis, Key-Point Localization, Orientation Assignment and Key-Point Descriptor Generation. While studying and implementing the SURF algorithm, it was learnt that out of a given

[1] Intel Pentium 4 CPU 2.80GHz, 1.49GB RAM.

number of Key-Points computed by SURF algorithm, more than 80% of them are extracted from the first two Octaves alone. The reason for the sudden decrease in the Key-Points from the higher Octaves can be attributed to the large filter sizes and higher stepping values in the higher Octave, which accelerate the movement of box filters towards the end of the image. Which implies that, fewer computations are made in these higher Octaves, which in turn result in fewer Key-Points being extracted from them. Therefore, the algorithm was modified to restrict from computing the Key-Points from Higher Octaves. By cutting down these computations, nearly **80 - 280ms** of processing time was improved for images with 640 × 480 pixels dimension and **20 - 100 ms** of processing time was improved for images with 320 × 240 pixels dimension. This forms the Algorithmic Optimization of SURF.

The platform chosen for this project was Freescale's i.MX515EVK, some of the important features [2] of this platform, which are relevant to this project are listed below:

- CPU: Freescale's **i.MX515** at a clock speed of 800 MHz.
- Memory: 512 MB DDR-RAM.
- Ubuntu 9.10 OS
- *For the entire list of peripherals refer to Appendix B: i.MX515EVK Details 7.2.*

The **i.MX515 is a Multimedia Processor** which is one of Freescale's latest additions to their growing multimedia-focused products; offering a High Processing Performance, at a very low power consumption which can be attributed to the core of this processor, the **ARM Cortex-A8**. The processor is aimed at applications such as:

- Netbooks
- Handheld devices such as
  - Portable Media Devices
  - Smart Phones
  - Navigation Devices
- Gaming consoles

The ARM Cortex-A8 core is a 32-bit, dual-issue, in-order type processor, with dynamic branch predictor.

Some of the key features of this processor are listed below.

- It operates at a Clock Speed of 800Mz.
- 32 KB Instruction and Data Caches.
- A unified 256 KB L2 Cache
- A Vector Floating Point Unit (VFP)

[2] For the functional block diagram of the i.MX515EVK refer to Appendix B: i.MX515EVK Details 7.2.

- A SIMD unit called NEON
- Instruction Set Architecture (ISA) support
  - ARM
  - Thumb2
  - VFPv3 Floating Point
  - NEON

Because of a full fledged operating system such as Ubuntu 9.10 running on i.MX515EVK, software building process was carried out directly on i.MX515EVK rather than using expensive Cross Compilers on a host system. By making use of the development tools mentioned in previous section, SURF Key-Points extraction algorithm was built for ARM Cortex-A8 processor, from hence forth this version of SURF is referred to as **SURF_ARM**.

The SURF algorithm is built for this project and named as SURF_ARM, which takes 1.1 - 1.57 seconds to process 640×480 pixels sized images and 0.48 - 0.66 seconds for 320×240pixels sized images. These computation times are too high to achieve a good frame rate, the SURF_ARM implementation is optimized. During this optimization phase special attention was paid to NEON and due to the compiler's inability to automatically vectorize the parts of the algorithm for NEON, they had to be hand coded for NEON.

After profiling the algorithm implementation it was found that around 70% of processor's time is spent in Box filtering. Box Filtering as such is a very simple function, but the sheer number of times it is called, which is around **3,383,284**[3] makes it the costliest function. Out of the 3,383,284 calls made, 2,658,304 were made while building the Scale-Space. In this project, by making use of NEON's interleaving instructions innovatively, this number was brought down by **4 times** to 845,821. This reduction in number can be termed as the most significant contributor in improving the computation time of the algorithm.

Other functions which were vectorized during this process were:

- **Integral Image computation** with pre-fetching of image data: This brought down the computation time of Integral Image by upto 40%.
- **Descriptor Computation and Matrix Multiplication**: No significant improvement due to vectorization of these functions.

By the end of this step of vectorization, the computation time needed to extract Key-Points for images of 320×240 dimensions was brought down to 300 - 350ms, a gain of 41 - 46%, which translates to around 3 frames per second. This version of SURF was called as SURF_ARM_NEON.

Next, by making using the SURF_ARM_NEON a Real-Time Euro currency notes Identification application was built. The objective of this application is to continuously capture image frames from a camera and identify the 7 Euro currency notes irrespective of the face in which they are presented in front of the camera. Key-Points from the

[3] For 640×480 size image.

$7 \times 2 = 14$ sides of images are extracted on a one-by-one fashion and they are stored within the application. When Key-Points from input image frames are extracted they are given to a matching algorithm which carries out a matching process between the freshly extracted Key-Points with the Key-Points within the application. The matching algorithm takes the 64 floating point values in the description of every Key-Point from the current image frame and match it against the description of every Key-Point in a collection of Key-Points known as **Model Key-Points**. The model Key-Points collection is always part of the application and they are extracted in a one-by-one fashion from all $7 \times 2 = 14$ sides of Euro currency notes, known as Model Images. A Key-Point of one image is said to be matched with a Key-Point of another image based on how similar their descriptions are. The similarity is measured by calculating the aggregate difference or also known as **R**oot-**S**um-**S**quare (RSS) of the 64 floating point valued description of one Key-Point with the 64 floating point values description of another Key-point. Key-Point pairs with the lowest value of difference are considered to be **matching pairs**. The matching algorithm returns the number of matching pairs found between Key-Points of a model image and the Key-Points found in the current frame. From extensive trials conducted it was found out that, greater than **6** matching pairs found confirms that the model image and the current image frame are the same.

Basic functions such as image resizing, displaying them on the screen, placing text on the image frame were carried out by using OpenCV libraries for real-time computer vision were used.

Upon profiling the real-time object identification application, it was found that, the *ExtractMatchingFeatures()* is one of the significant contributors to the computation time.

To match a set of **71** Key-Points extracted from an image, with **503** Key-Points in the Key-Points collection of the application, the matching algorithm would take close to **97ms**. By looking at its implementation it was found that critical parts of this function can be vectorized. So, once again parts of the matching algorithm were vectorized. With this optimization the matching algorithm's computation time was dropped to **42ms**, a 56.7% improvement.

The performance of the real-time object identification is measured by conducting tests to see how well it is able to handle changing Scale, Illumination, Orientation, Blurring and Viewpoint. From these tests it was found that, except for the Viewpoint changes, the application can handle all the changing conditions very well. The lack of the ability to handle viewpoint changes can be attributed to the SURF Key-Points, compared to SIFT algorithm, SURF is not resilient to changes to viewpoint.

On an average, the application was able to process 320×240 pixels sized image frames at a rate of **3.12 frames per second**.

While studying the SURF algorithm it was learnt that the Key-Points extracted by SURF in each Octave are unique in nature, the motive behind progressive matching arises from this study. The idea here is, when the Key-Points extracted from an Octave are provided to matching algorithm and if it is able to extract enough number of matching Key-Point pairs to decide that the two input images match or not, further Key-Point computations are not needed any longer and processing of next frame can be started instantly. The contribution of this approach was a significant amount of reduction in computation time when considerable number of image frames were matched by the end of first Octave. In one of the trials it was found that the total time taken for processing 100 frames was reduced by nearly 29% due to progressive matching.

By the end of progressive matching the average frame rate of the real-time object identification application was around **3.5 - 4 frames per second**.

The application size of the Euro Currency notes identification is **34KB** and it was utilizing around **4.42MB** of runtime memory.

Towards the end of the project another important question was answered, which is, if excessive usage of NEON needed more energy or not. NEON is an additional hardware unit and to power it additional energy is certainly needed. However, from the tests conducted it was found that practically an unobservable amount of additional current was drawn when NEON was used. But on the other hand it was seen that in a given amount of time greater amount of work was completed when NEON was used. So, for such minute increment in the power consumption and such high throughput achieved, it is not wrong to conclude that the NEON usage in fact brings down the overall energy consumption.

The energy consumed by the application with SURF_ARM_NEON + Progressive matching, to process 100 frames is around **2.955mAs**.

6.2 Recommendations for Future Work

The recommendations for future work in this direction is classified into two parts, they are:

1. Accuracy

To improve the accuracy of the Key-Points further, an algorithm known as **Fast approximated SIFT** can be tried [5]. This algorithm is a fusion of concepts proposed in both SIFT and SURF algorithms. The authors have recorded that in their experiments, the Fast approximated SIFT algorithm takes eight times lesser processing time when compared to SIFT algorithm and the Key-Points show better repeatability performance over the SURF Key-Points.

2. Speed

Algorithmic Improvements

In some applications such as face recognition, rotation invariance is not required, so the descriptor computation can be simplified by excluding the computation of orientation information. This might considerably improve the processing time. Such a version of SURF is called as Upright SURF (U-SURF)[19].

Future Processors

The ARM Cortex-A8 processor is only the second of the four processors in the **Cortex-A family**. ARM has already launched the next two processors in its Cortex-A series known as **Cortex-A9** and **Cortex-A15** processors.

A Cortex-A9 is a 2Ghz capable processor with upto 4 scalable coherent cores as shown in Figure 6.1. Each core of Cortex-A9 has its individual floating point and NEON unit. This processor is targeted for next generation of handheld devices which have - Full HD camcorders, High resolution touch screen displays, High performance 3D graphics, multiple cameras, multiple operating systems and so on. ST Ericsson have already used Cortex-A9 processor for building a **Smartphone Platform** known as U8500 [38].

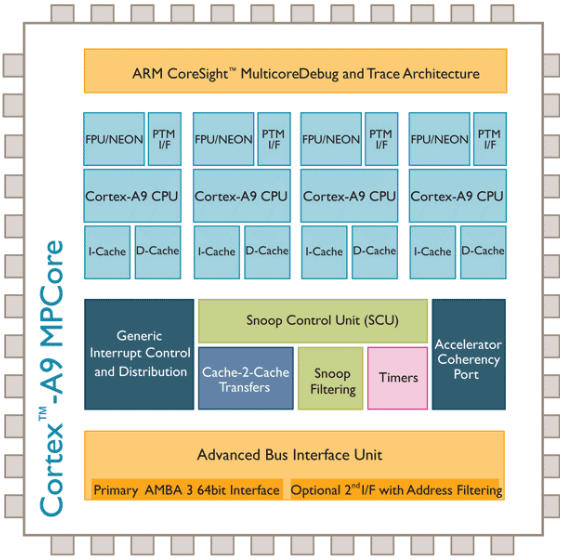


Figure 6.1: *ARM Cortex-A9 processor, which is 2Ghz capable processor with upto 4 scalable cores.*

By using such multicore processors SURF Key-Points computations can be carried out in parallel[2]. Candidate parts of the SURF algorithm which can be computed in parallel are -

- **Building Scale-Space**: Unlike in SIFT where every layer in its Scale-Space must be built by applying Gaussian Filters on a previous layer, which makes it a serial process, Scale-Space building in SURF can be carried out in parallel, because every layer of SURF Scale-Space is built by applying box filters directly on the input image. Like in the case of Cortex-A9 processor where every core has a NEON and a FPU unit of its own, the number of box filters needed to build the Scale-Space can once again be computed very effectively using NEON as described in this project.

- **Generating Key-Points**: Once a Scale-Space is built, Key-Points from each Octave can be computed using individual cores. If the algorithm designer chooses to have 4 Octaves in his/her design and if the processor has four cores, Key-Points from each Octave can be computed simultaneously.

- Apart from computing SURF Key-Points in parallel, their unique nature which facilitated in implementing progressive matching in the real-time object identification application, can also be carried out in parallel.

Bibliography

[1] C. Evans, "Notes on the opensurf library," Tech. Rep. CSTR-09-001, University of Bristol, January 2009.

[2] N. Zhang, "Computing parallel speeded-up robust features (p-surf) via posix threads.," in *ICIC (1)* (D.-S. Huang, K.-H. Jo, H.-H. Lee, H.-J. Kang, and V. Bevilacqua, eds.), vol. 5754 of *Lecture Notes in Computer Science*, pp. 287–296, Springer, 2009.

[3] "Improving arm code density and performance." http://arch.eece.maine.edu/ece471/images/3/32/Thumb-2CoreTechnologyWhitepaper-Final4.pdf.

[4] C. Carvalho, "The gap between processor and memory speeds.,"

[5] M. Grabner, H. Grabner, and H. Bischof, "Fast approximated sift.," in *ACCV (1)* (P. J. Narayanan, S. K. Nayar, and H.-Y. Shum, eds.), vol. 3851 of *Lecture Notes in Computer Science*, pp. 918–927, Springer, 2006.

[6] C. G. G. ROBERT DESIMONE, THOMAS D. ALBRIGHT and C. BRUCE, "Stimulus-selective properties of inferior temporal neurons in the macaque," 1984.

[7] K. Tanaka, "Mechanisms of visual object recognition: monkey and human studies," 1997.

[8] Google, "Google goggles application for android." http://www.google.com/mobile/goggles/#text.

[9] T. Libor, "Sharpstitch a c# image stitching library." http://sharpstitch.sourceforge.net/.

[10] "Sony cameras panorama."

[11] "Canon cameras panorama views."

[12] *Real-time eye blink detection with GPU-based SIFT tracking*, 2007.

[13] S. Se and P. Jasiobedzki, "Stereo-vision based 3d modeling for unmanned ground vehicles," 2007.

[14] S. N. Sinha, J. michael Frahm, M. Pollefeys, and Y. Genc, "Gpu-based video feature tracking and matching," tech. rep., In Workshop on Edge Computing Using New Commodity Architectures, 2006.

[15] T.-K. Kim and R. Cipolla, "Gesture recognition under small sample size," in *ACCV'07: Proceedings of the 8th Asian conference on Computer vision*, (Berlin, Heidelberg), pp. 335–344, Springer-Verlag, 2007.

[16] C. ting Hsu and M. chou Shih, "Content-based image retrieval by interest points matching and geometric hashing."

[17] M. Bokeloh, A. Berner, M.Wand, H.-P. Seidel, and A. Schilling, "Symmetry detection using feature lines." http://www.gris.uni-tuebingen.de/people/staff/bokeloh/project_symmetry2.html.

[18] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, p. 91, 2004.

[19] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, "Speeded-up robust features (SURF)," *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346 – 359, 2008. Similarity Matching in Computer Vision and Multimedia.

[20] D. Marr and E. Hildreth, "Theory of edge detection," *Proceedings of the Royal Society of London Series B*, vol. 207, pp. 187–217, 1980.

[21] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, 1986.

[22] C. Harris and M. Stephens, "A combined corner and edge detector," in *Proceedings of the 4th Alvey Vision Conference*, pp. 147–151, 1988.

[23] R. Szeliski, "Computer vision: Algorithms and applications." https://web.engr.oregonstate.edu/~hess/publications/siftlib-acmmm10.pdf.

[24] J. H. Elder and R. M. Goldberg, "Image editing in the contour domain.," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 23, no. 3, pp. 291–296, 2001.

[25] S. N. Sinha, D. Steedly, R. Szeliski, M. Agrawala, and M. Pollefeys, "Interactive 3d architectural modeling from unordered photo collections.," *ACM Trans. Graph.*, vol. 27, no. 5, p. 159, 2008.

[26] "Scale invariance wikipedia page." http://en.wikipedia.org/wiki/Scale_invariance.

[27] K. Mikolajczyk, T. Tuytelaars, C. Schmid, A. Zisserman, J. Matas, F. Schaffalitzky, T. Kadir, and L. Gool, "A comparison of affine region detectors," *International Journal of Computer Vision*, vol. 65, pp. 43–72, November 2005.

[28] R. Hess, "An open-source sift library." https://web.engr.oregonstate.edu/~hess/publications/siftlib-acmmm10.pdf.

[29] O. G. Luo Juan, "A comparison of sift, pca-sift and surf." http://www.cscjournals.org/csc/manuscript/Journals/IJIP/volume3/Issue4/IJIP-51.pdf.

[30] J.-M. Geusebroek, A. W. M. Smeulders, and J. van de Weijer, "Fast anisotropic gauss filtering.," *IEEE Transactions on Image Processing*, vol. 12, no. 8, pp. 938–943, 2003.

[31] M. Brown and D. G. Lowe, "Invariant features from interest point groups.," in *BMVC* (P. L. Rosin and A. D. Marshall, eds.), British Machine Vision Association, 2002.

[32] C. W. Andrew N. Sloss, Dominic Symes, *ARM System Developer's Guide: Designing and Optimizing System Software*. Elsevier, 2004.

[33] "Five billionth arm processor for mobile devices." http://www.arm.com/about/newsroom/16535.php.

[34] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel, "A code compression system based on pipelined interpreters.," *Softw., Pract. Exper.*, vol. 29, no. 11, pp. 1005–1023, 1999.

[35] "Logitech webcam c200." http://www.logitech.com.

[36] "Vga definition." http://www.pcmag.com/encyclopedia_term/0,2542,t%3DVGA&i%3D53801,00.asp.

[37] "Root-sum-square (rss) calculations of digital timing delays." http://www.klabs.org/richcontent/General_Application_Notes/SDE/RSS.pdf.

[38] "St ericsson u8500 - the smartphone platform." http://www.stericsson.com/platforms/U8500.jsp.

Appendicies 7

7.1 A: Results of the comparative repeatability tests conducted between SIFT and SURF algorithms.

Disturbance Type						
Scaling						
SIFT						
Value	**Image Name**	**Features Found**		**Matches Obtained**		**Performance**
	Original.png	505				
1	Scale1.png	800		230		0.352490421
2	Scale2.png	334		193		0.460071514
3	Scale3.png	296		175		0.436953808
4	Scale4.png	179		121		0.35380117
SURF						
Value	**Image Name**	**Features Found**		**Matches Obtained**		**Performance**
	Original.bmp	380				
Scale1	Scale1.bmp	517		113		0.251950948
Scale2	Scale2.bmp	261		122		0.380655226
Scale3	Scale3.bmp	227		111		0.365733114
Scale4	Scale4.bmp	144		66		0.251908397

Disturbance Type						
Blurring						
SIFT						
Value	**Image Name**	**Features Found**		**Matches Obtained**		**Performance**
	Original.png	505				
1	Blur1.bmp	391		231		0.515625
2	Blur2.bmp	250		168		0.445033113
3	Blur3.bmp	195		135		0.385714286
4	Blur4.bmp	161		115		0.345345345
SURF						
Value	**Image Name**	**Features Found**		**Matches Obtained**		**Performance**
	Original.bmp	380				
Scale1	Blur1.bmp	342		270		0.747922438
Scale2	Blur2.bmp	380		233		0.613157895
Scale3	Blur3.bmp	310		196		0.568115942
Scale4	Blur4.bmp	304		170		0.497076023

Disturbance Type						
Illumination						
SIFT						
Value	**Image Name**	**Features Found**		**Matches Obtained**		**Performance**
	Original.bmp	615				
-2	dark2.bmp	0		0		0
-1	dark1.bmp	74		81		0.235123367
1	bright1.bmp	763		501		0.727140784
2	bright2.bmp	693		290		0.443425076
SURF						
Value	**Image Name**	**Features Found**		**Matches Obtained**		**Performance**
	Original.bmp	390				
-2	dark2.bmp	1		1		0.00511509
-1	dark1.bmp	131		121		0.464491363
1	bright1.bmp	459		346		0.815076561
2	bright2.bmp	418		165		0.408415842

Disturbance Type						
Orientation						
SIFT						
Value	**Image Name**	**Features Found**		**Matches Obtained**		**Performance**
	Original.bmp	312				
30	Ori1.bmp	327		134		0.419405321
60	Ori2.bmp	326		100		0.313479624
90	Ori3.bmp	290		96		0.318936877
120	Ori4.bmp	293		112		0.370247934
150	Ori5.bmp	301		129		0.420880914
180	Ori6.bmp	306		143		0.462783172
210	Ori7.bmp	331		107		0.33281493
240	Ori8.bmp	326		107		0.335423197
270	Ori9.bmp	259		89		0.3117338
300	Ori10.bmp	315		107		0.341307815
330	Ori11.bmp	328		144		0.45
SURF						
Value	**Image Name**	**Features Found**		**Matches Obtained**		**Performance**
	Original.bmp	243				
30	Ori1.bmp	237		70		0.291666667
60	Ori2.bmp	244		60		0.246406571
90	Ori3.bmp	227		83		0.353191489
120	Ori4.bmp	231		59		0.248945148
150	Ori5.bmp	237		62		0.258333333
180	Ori6.bmp	244		89		0.36550308
210	Ori7.bmp	241		57		0.23553719
240	Ori8.bmp	247		45		0.183673469
270	Ori9.bmp	243		65		0.267489712
300	Ori10.bmp	248		57		0.232179226
330	Ori11.bmp	260		90		0.357852883

Disturbance Type						
Viewpoint						
SIFT						
Value	**Image Name**	**Features Found**		**Matches Obtained**		**Performance**
	Original.bmp	376				
1	View1.bmp	379		108		0.286092715
2	View2.bmp	369		50		0.134228188
3	View3.bmp	690		82		0.153846154
4	View4.bmp	273		3		0.009244992
5	View5.bmp	453		68		0.164053076
SURF						
Value	**Image Name**	**Features Found**		**Matches Obtained**		**Performance**
	Original.bmp	268				
1	View1.bmp	274		3		0.011070111
2	View2.bmp	228		13		0.052419355
3	View3.bmp	431		21		0.060085837
4	View4.bmp	225		3		0.012170385
5	View5.bmp	313		25		0.08605852

SIFT

Image Name	Features Found	Processing Time
Original.png	505	1.67
Scale1.png	800	2.2
Scale2.png	334	1.35
Scale3.png	296	1.26
Scale4.png	179	1.02
Original.png	505	1.66
Blur1.bmp	391	1.55
Blur2.bmp	250	1.25
Blur3.bmp	195	1.14
Blur4.bmp	161	1.08
Original.bmp	312	1.28
Ori1.bmp	327	1.29
Ori2.bmp	326	1.33
Ori3.bmp	290	1.28
Ori4.bmp	293	1.27
Ori5.bmp	301	1.28
Ori6.bmp	306	1.31
Ori7.bmp	331	1.33
Ori8.bmp	326	1.32
Ori9.bmp	259	1.23
Ori10.bmp	315	1.34
Ori11.bmp	328	1.33
Original.bmp	615	1.78
dark2.bmp	0	0.69
dark1.bmp	74	0.86
bright1.bmp	763	2.02
bright2.bmp	693	2.05
Original.bmp	376	1.41
View1.bmp	379	1.41
View2.bmp	369	1.36
View3.bmp	690	1.94
View4.bmp	273	1.22
View5.bmp	453	1.56

SURF

Image Name	Features Found	Processing Time
Original.bmp	380	0.32
Scale1.bmp	517	0.42
Scale2.bmp	261	0.24
Scale3.bmp	227	0.22
Scale4.bmp	144	0.19
Original.bmp	380	0.32
Blur1.bmp	342	0.29
Blur2.bmp	380	0.32
Blur3.bmp	310	0.28
Blur4.bmp	304	0.27
Original.bmp	243	0.24
Ori1.bmp	237	0.23
Ori2.bmp	244	0.24
Ori3.bmp	227	0.23
Ori4.bmp	231	0.24
Ori5.bmp	237	0.24
Ori6.bmp	244	0.25
Ori7.bmp	241	0.25
Ori8.bmp	247	0.24
Ori9.bmp	243	0.24
Ori10.bmp	248	0.23
Ori11.bmp	260	0.25
Original.bmp	390	0.32
dark2.bmp	1	0.1
dark1.bmp	131	0.17
bright1.bmp	459	0.37
bright2.bmp	418	0.34
Original.bmp	268	0.28
View1.bmp	274	0.26
View2.bmp	228	0.24
View3.bmp	431	0.36
View4.bmp	225	0.23
View5.bmp	313	0.29

7.2 B: i.MX515EVK Details

i.MX Applications Processors

Evaluation Kit (EVK) for the i.MX51 Applications Processor

Price. Performance. Personality.

Energy Efficient Solutions by Freescale™

Overview

Freescale delivers the cost-effective i.MX51 evaluation kit, allowing customers to develop, debug and demonstrate their next great product without compromising performance. As part of our new price, performance and personality series, the evaluation kit is designed to support all the features of the device in a small, single-board design to enable designers to complete a development platform at a low price point of less than an estimated $700USD. The i.MX51 EVK has two optional add-on modules: an LCD module and an expansion board which includes a camera, TV out, keypad and UART. Based on a powerful ARM Cortex™-A8 core, the i.MX51 EVK delivers extreme performance and low power consumption, helping developers design products that meet today's demands for energy efficiency.

A range of connectivity options makes the i.MX51 EVK suitable for developing many different types of user applications. The provided board support packages (BSP) for Linux® OS and Windows® Embedded CE enable rapid prototyping which helps to speed up the processor selection process and quickly deliver a demo into the hands of the project stakeholders. The i.MX51 EVK includes two SD cards: one pre-loaded with Linux and the other with Windows Embedded CE. Both options support a wide range of automotive, consumer, general embedded and industrial applications.

Key Benefits

- Explore multiple connectivity options with the i.MX51 applications processor: display, touch screen, USB, SDIO, Ethernet and others
- Investigate usage of the video and graphics through the hardware accelerated video processing unit, OpenGL® ES 2.0 and OpenVG™ 1.1 graphics processing units
- Develop with the MC13892 power management chip from Freescale that supports power sequencing of the i.MX51 device and output rails to supply power to external components such as memories and other system peripherals
- Use proven design examples and software drivers to reduce hassles associated with design-in of key connectivity and power management options
- Enable rapid prototyping of human-machine interfaces (HMI) via the on-board digital visual interface (DVI) peripheral that allows the EVK to interface to a standard PC monitor
- Boot from SD, SPI or NAND flash

Performance

With the i.MX51 EVK, designers have access to key features needed for an end design offering hardware functionality and connectivity required for developing many applications, such as portable media players, mobile Internet devices, smartbooks, gaming consoles, ebooks, media phones, digital photo frames, high-end appliances, video and navigation, security and surveillance, medical and factory automation. With production-ready software components, an optimized OS and a system-validated BSP, designers have the tools to test and maximize the performance of the applications they have developed. Software and hardware engineers can also download this code to the target EVK to test and validate their software and to run and evaluate performance metrics. The ability to have all communications ports working (serial, USB) and to debug over JTAG and Ethernet is essential for product development. The EVK also provides boot select switches, which provide the user with the option to override the default boot setting of the CPU.

Personality

Freescale's EVK for the i.MX51 applications processor allows designers to quickly prototype and demonstrate the results of their development efforts in a small, portable design the size of a 5 x 5 portrait, giving confidence to project decision makers that the product is that much closer to production. Develop user-interactive software and display your product-specific graphical data on a high-quality, touch screen-enabled 7" WVGA LCD available as an add-on module to the EVK. Connect additional input and output peripherals such as a camera, TV out, keypad and UART with the expansion board add-on module. With the Freescale i.MX51 EVK, prototyping and development are simplified to improve time to market.

i.MX51 EVK Key Features

CPU

- i.MX51 applications processor
- 4 x 128 MB DDR2
- 4 MB SPI NOR
- PMIC: Freescale MC13892
- NAND and EIM header

freescale™ semiconductor

Peripherals

- 7" WVGA touch screen LCD display (add-on module)
- Two LVDS connectors
- DVI-I connector
- Two SD/MMC card slots
- USB host x2/USB OTG x1
- Ethernet port
- Mini PCI Express®
- SATA HDD connector
- SIM card connector
- Keyboard connector
- Mic input, stereo headphone output (jack), V2IP headphone
- Speaker connector
- USB camera connector
- PS-2 TP connector
- RGB output through DVI-I connector
- Expansion header
- Ambient light sensor footprint
- FM receiver footprint
- Expansion board (add-on module) with camera, TV out, keypad and UART

Debug

- Debug serial port
- JTAG
- Reset, boot switches
- Debug LED
- Power source
- Power on/off button
- Power measurement header

Software Development Kit

- Optimized and validated for both Linux and Windows Embedded CE operating systems
- Integrated and validated BSP for the i.MX51 EVK feature set
- Highly optimized software that is coded by Freescale processor experts
- Consistent application programming interface (API) and frameworks across software packages
- Evaluation and production software packages available through a streamlined, Web-based licensing and delivery system
- Freescale development tools, test streams and documentation provided

Part Number	Operating Systems	MSRP (USD)
MCIMX51EVKJ	Linux and Windows Embedded CE	$699.00

Part Number	Peripheral	Features	MSRP (USD)
MCIMX51LCD	i.MX51 LCD Module	WVGA with resistive touch screen	$250.00
MCIMX51EXP	i.MX51 Expansion Board	-CMOS camera -TV out -Keypad -UART	$200.00

The MC13892 Power Management and User Interface IC (PMUI IC)

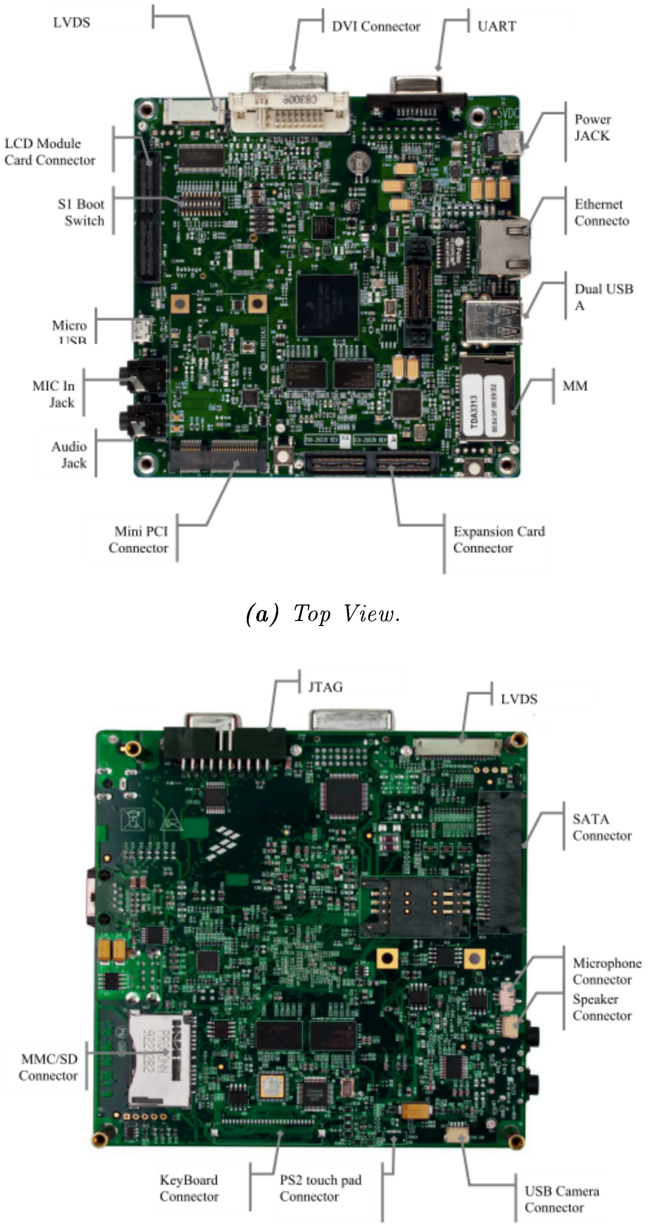
The MC13892 PMUI IC is designed for use with the i.MX51 applications processor requiring a highly integrated, bi-directional power management IC and communications device. Features include:

- Battery charging system for wall charging and USB charging
- 10-bit ADC for monitoring battery and other inputs
- Four adjustable output buck converters
- 12 adjustable output low drop outs (LDO) with internal and external pass devices
- Two boost converters
- Serial backlight drivers
- Power control logic with processor interface and event direction
- Real-time clock and crystal oscillator circuitry
- Touch screen interface
- SPI/I²C bus interface

i.MX51 Evaluation Kit assembled with LCD and expansion boards

i.MX51 Evaluation Kit

Learn More: For current information about Freescale products and documentation, please visit **www.freescale.com/imx51evk**.



(a) Top View.

(b) Bottom View.

Figure 7.1: *The Top and bottom views of the i.MX515EVK.*

7.3 C: NEON test

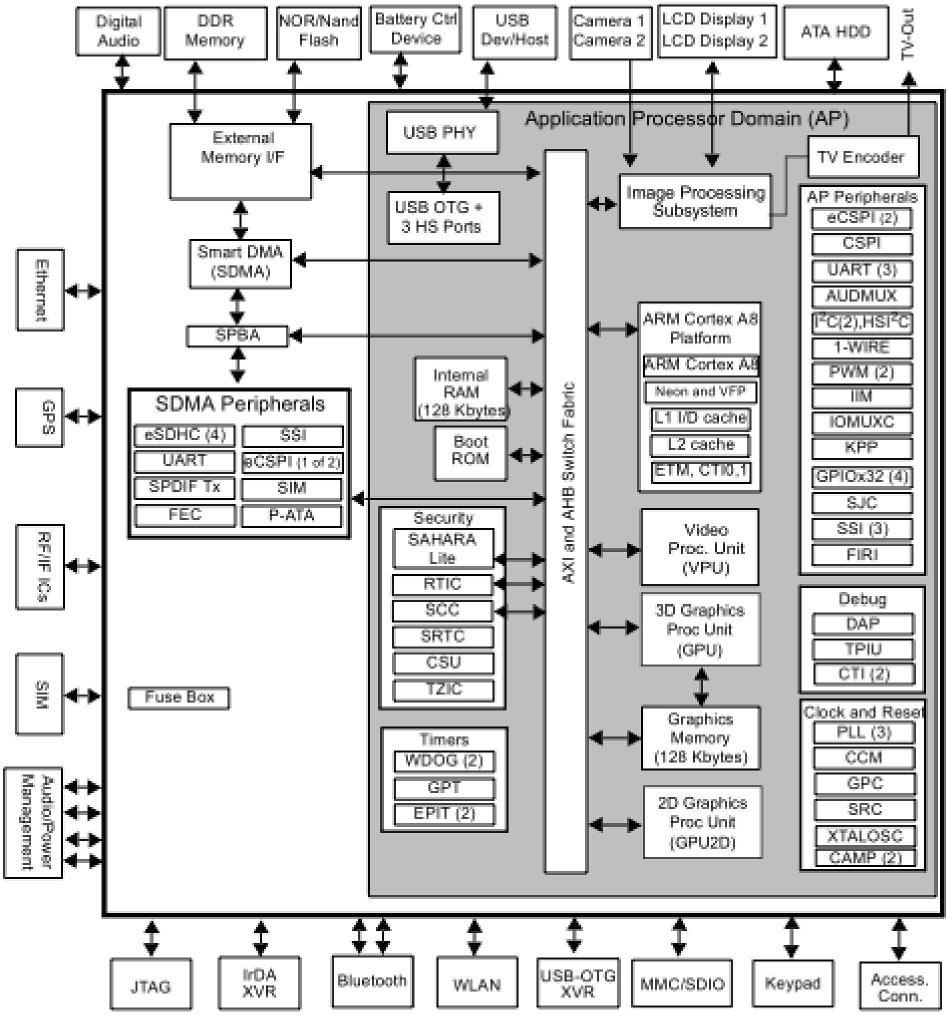
```c
inline void MatrixMultiplication(float H[][3], float dD[][1], float result[][1])
{
    unsigned char i, j, k;

    for(i=0;i<3;i++)
    {
        for(j=0;j<1;j++)
        {
            result[i][j] = 0;
        }
    }

    for(i=0;i<3;i++)
    {
        for(j=0;j<1;j++)
        {
            for(k=0;k<3;k++)
            {
                result[i][j] += H[i][k] * dD[k][j];

            }
        }
    }
}
```

```
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
; With gcc -vfpu=vfpv3
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

Disassembly of section .text:

000083fc <MatrixMultiplication>:
    83fc:   e52db004    push    {fp}        ; (str fp, [sp, #-4]!)
    8400:   e28db000    add fp, sp, #0
    8404:   e24dd01c    sub sp, sp, #28
    8408:   e50b0010    str r0, [fp, #-16]
    840c:   e50b1014    str r1, [fp, #-20]
    8410:   e50b2018    str r2, [fp, #-24]
    8414:   e3a03000    mov r3, #0
    8418:   e54b3007    strb    r3, [fp, #-7]
    841c:   ea000013    b   8470 <MatrixMultiplication+0x74>
    8420:   e3a03000    mov r3, #0
    8424:   e54b3006    strb    r3, [fp, #-6]
    8428:   ea00000a    b   8458 <MatrixMultiplication+0x5c>
    842c:   e55b3007    ldrb    r3, [fp, #-7]
    8430:   e1a02103    lsl r2, r3, #2
    8434:   e51b3018    ldr r3, [fp, #-24]
    8438:   e0823003    add r3, r2, r3
    843c:   e55b2006    ldrb    r2, [fp, #-6]
    8440:   eddf7a4d    vldr    s15, [pc, #308] ; 857c <MatrixMultiplication+0x180>
    8444:   ee171a90    vmov    r1, s15
    8448:   e7831102    str r1, [r3, r2, lsl #2]
    844c:   e55b3006    ldrb    r3, [fp, #-6]
    8450:   e2833001    add r3, r3, #1
    8454:   e54b3006    strb    r3, [fp, #-6]
    8458:   e55b3006    ldrb    r3, [fp, #-6]
    845c:   e3530000    cmp r3, #0
    8460:   0afffff1    beq 842c <MatrixMultiplication+0x30>
    8464:   e55b3007    ldrb    r3, [fp, #-7]
    8468:   e2833001    add r3, r3, #1
    846c:   e54b3007    strb    r3, [fp, #-7]
    8470:   e55b3007    ldrb    r3, [fp, #-7]
    8474:   e3530002    cmp r3, #2
    8478:   9affffe8    bls 8420 <MatrixMultiplication+0x24>
    847c:   e3a03000    mov r3, #0
    8480:   e54b3007    strb    r3, [fp, #-7]
    8484:   ea000036    b   8564 <MatrixMultiplication+0x168>
    8488:   e3a03000    mov r3, #0
    848c:   e54b3006    strb    r3, [fp, #-6]
    8490:   ea00002d    b   854c <MatrixMultiplication+0x150>
    8494:   e3a03000    mov r3, #0
    8498:   e54b3005    strb    r3, [fp, #-5]
    849c:   ea000024    b   8534 <MatrixMultiplication+0x138>
    84a0:   e55b3007    ldrb    r3, [fp, #-7]
    84a4:   e1a02103    lsl r2, r3, #2
    84a8:   e51b3018    ldr r3, [fp, #-24]
    84ac:   e0821003    add r1, r2, r3
    84b0:   e55b0006    ldrb    r0, [fp, #-6]
    84b4:   e55b3007    ldrb    r3, [fp, #-7]
    84b8:   e1a02103    lsl r2, r3, #2
    84bc:   e51b3018    ldr r3, [fp, #-24]
    84c0:   e0823003    add r3, r2, r3
    84c4:   e55b2006    ldrb    r2, [fp, #-6]
    84c8:   e083c102    add ip, r3, r2, lsl #2
    84cc:   ed9c7a00    vldr    s14, [ip]
    84d0:   e55b2007    ldrb    r2, [fp, #-7]
    84d4:   e1a03002    mov r3, r2
    84d8:   e1a03083    lsl r3, r3, #1
    84dc:   e0833002    add r3, r3, r2
    84e0:   e1a03103    lsl r3, r3, #2
    84e4:   e1a02003    mov r2, r3
    84e8:   e51b3010    ldr r3, [fp, #-16]
    84ec:   e0823003    add r3, r2, r3
    84f0:   e55b2005    ldrb    r2, [fp, #-5]
    84f4:   e083c102    add ip, r3, r2, lsl #2
    84f8:   eddc6a00    vldr    s13, [ip]
    84fc:   e55b3005    ldrb    r3, [fp, #-5]
    8500:   e1a02103    lsl r2, r3, #2
    8504:   e51b3014    ldr r3, [fp, #-20]
    8508:   e0823003    add r3, r2, r3
    850c:   e55b2006    ldrb    r2, [fp, #-6]
    8510:   e083c102    add ip, r3, r2, lsl #2
    8514:   eddc7a00    vldr    s15, [ip]
    8518:   ee667aa7    vmul.f32    s15, s13, s15
    851c:   ee777a27    vadd.f32    s15, s14, s15
    8520:   ee173a90    vmov    r3, s15
    8524:   e7813100    str r3, [r1, r0, lsl #2]
    8528:   e55b3005    ldrb    r3, [fp, #-5]
    852c:   e2833001    add r3, r3, #1
    8530:   e54b3005    strb    r3, [fp, #-5]
    8534:   e55b3005    ldrb    r3, [fp, #-5]
    8538:   e3530002    cmp r3, #2
    853c:   9affffd7    bls 84a0 <MatrixMultiplication+0xa4>
    8540:   e55b3006    ldrb    r3, [fp, #-6]
    8544:   e2833001    add r3, r3, #1
    8548:   e54b3006    strb    r3, [fp, #-6]
    854c:   e55b3006    ldrb    r3, [fp, #-6]
    8550:   e3530000    cmp r3, #0
    8554:   0affffce    beq 8494 <MatrixMultiplication+0x98>
    8558:   e55b3007    ldrb    r3, [fp, #-7]
    855c:   e2833001    add r3, r3, #1
    8560:   e54b3007    strb    r3, [fp, #-7]
    8564:   e55b3007    ldrb    r3, [fp, #-7]
    8568:   e3530002    cmp r3, #2
    856c:   9affffc5    bls 8488 <MatrixMultiplication+0x8c>
    8570:   e28bd000    add sp, fp, #0
    8574:   e49db004    pop {fp}        ; (ldr fp, [sp], #4)
    8578:   e12fff1e    bx  lr
    857c:   00000000    .word   0x00000000
```

```
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
; With gcc -vfpu=neon -ftree-vectorize
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\


Disassembly of section .text:

000083fc <MatrixMultiplication>:
    83fc:   e52db004    push    {fp}        ; (str fp, [sp, #-4]!)
    8400:   e28db000    add fp, sp, #0
    8404:   e24dd01c    sub sp, sp, #28
    8408:   e50b0010    str r0, [fp, #-16]
    840c:   e50b1014    str r1, [fp, #-20]
    8410:   e50b2018    str r2, [fp, #-24]
    8414:   e3a03000    mov r3, #0
    8418:   e54b3007    strb    r3, [fp, #-7]
    841c:   ea000013    b   8470 <MatrixMultiplication+0x74>
    8420:   e3a03000    mov r3, #0
    8424:   e54b3006    strb    r3, [fp, #-6]
    8428:   ea00000a    b   8458 <MatrixMultiplication+0x5c>
    842c:   e55b3007    ldrb    r3, [fp, #-7]
    8430:   e1a02103    lsl r2, r3, #2
    8434:   e51b3018    ldr r3, [fp, #-24]
    8438:   e0823003    add r3, r2, r3
    843c:   e55b2006    ldrb    r2, [fp, #-6]
    8440:   eddf7a4d    vldr    s15, [pc, #308] ; 857c <MatrixMultiplication+0x180>
    8444:   ee171a90    vmov    r1, s15
    8448:   e7831102    str r1, [r3, r2, lsl #2]
    844c:   e55b3006    ldrb    r3, [fp, #-6]
    8450:   e2833001    add r3, r3, #1
    8454:   e54b3006    strb    r3, [fp, #-6]
    8458:   e55b3006    ldrb    r3, [fp, #-6]
    845c:   e3530000    cmp r3, #0
    8460:   0afffff1    beq 842c <MatrixMultiplication+0x30>
    8464:   e55b3007    ldrb    r3, [fp, #-7]
    8468:   e2833001    add r3, r3, #1
    846c:   e54b3007    strb    r3, [fp, #-7]
    8470:   e55b3007    ldrb    r3, [fp, #-7]
    8474:   e3530002    cmp r3, #2
    8478:   9affffe8    bls 8420 <MatrixMultiplication+0x24>
    847c:   e3a03000    mov r3, #0
    8480:   e54b3007    strb    r3, [fp, #-7]
    8484:   ea000036    b   8564 <MatrixMultiplication+0x168>
    8488:   e3a03000    mov r3, #0
    848c:   e54b3006    strb    r3, [fp, #-6]
    8490:   ea00002d    b   854c <MatrixMultiplication+0x150>
    8494:   e3a03000    mov r3, #0
    8498:   e54b3005    strb    r3, [fp, #-5]
    849c:   ea000024    b   8534 <MatrixMultiplication+0x138>
    84a0:   e55b3007    ldrb    r3, [fp, #-7]
    84a4:   e1a02103    lsl r2, r3, #2
    84a8:   e51b3018    ldr r3, [fp, #-24]
    84ac:   e0821003    add r1, r2, r3
    84b0:   e55b0006    ldrb    r0, [fp, #-6]
    84b4:   e55b3007    ldrb    r3, [fp, #-7]
    84b8:   e1a02103    lsl r2, r3, #2
    84bc:   e51b3018    ldr r3, [fp, #-24]
    84c0:    e0823003    add r3, r2, r3
    84c4:    e55b2006    ldrb    r2, [fp, #-6]
    84c8:    e083c102    add ip, r3, r2, lsl #2
    84cc:    ed9c7a00    vldr    s14, [ip]
    84d0:    e55b2007    ldrb    r2, [fp, #-7]
    84d4:    e1a03002    mov r3, r2
    84d8:    e1a03083    lsl r3, r3, #1
    84dc:    e0833002    add r3, r3, r2
    84e0:    e1a03103    lsl r3, r3, #2
    84e4:    e1a02003    mov r2, r3
    84e8:    e51b3010    ldr r3, [fp, #-16]
    84ec:    e0823003    add r3, r2, r3
    84f0:    e55b2005    ldrb    r2, [fp, #-5]
    84f4:    e083c102    add ip, r3, r2, lsl #2
    84f8:    eddc6a00    vldr    s13, [ip]
    84fc:    e55b3005    ldrb    r3, [fp, #-5]
    8500:    e1a02103    lsl r2, r3, #2
    8504:    e51b3014    ldr r3, [fp, #-20]
    8508:    e0823003    add r3, r2, r3
    850c:    e55b2006    ldrb    r2, [fp, #-6]
    8510:    e083c102    add ip, r3, r2, lsl #2
    8514:    eddc7a00    vldr    s15, [ip]
    8518:    ee667aa7    vmul.f32    s15, s13, s15
    851c:    ee777a27    vadd.f32    s15, s14, s15
    8520:    ee173a90    vmov    r3, s15
    8524:    e7813100    str r3, [r1, r0, lsl #2]
    8528:    e55b3005    ldrb    r3, [fp, #-5]
    852c:    e2833001    add r3, r3, #1
    8530:    e54b3005    strb    r3, [fp, #-5]
    8534:    e55b3005    ldrb    r3, [fp, #-5]
    8538:    e3530002    cmp r3, #2
    853c:    9affffd7    bls 84a0 <MatrixMultiplication+0xa4>
    8540:    e55b3006    ldrb    r3, [fp, #-6]
    8544:    e2833001    add r3, r3, #1
    8548:    e54b3006    strb    r3, [fp, #-6]
    854c:    e55b3006    ldrb    r3, [fp, #-6]
    8550:    e3530000    cmp r3, #0
    8554:    0affffce    beq 8494 <MatrixMultiplication+0x98>
    8558:    e55b3007    ldrb    r3, [fp, #-7]
    855c:    e2833001    add r3, r3, #1
    8560:    e54b3007    strb    r3, [fp, #-7]
    8564:    e55b3007    ldrb    r3, [fp, #-7]
    8568:    e3530002    cmp r3, #2
    856c:    9affffc5    bls 8488 <MatrixMultiplication+0x8c>
    8570:    e28bd000    add sp, fp, #0
    8574:    e49db004    pop {fp}        ; (ldr fp, [sp], #4)
    8578:    e12fff1e    bx  lr
    857c:    00000000    .word   0x00000000
```

7.4 D: *gprof* Profiler Report

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
 69.24      0.17     0.17  3383284     0.00     0.00  box_area_compute
 12.89      0.31     0.05      129     0.39     1.03  get_descriptor
 11.11      0.35     0.04        1    40.01   210.05  build_response_layers
  2.78      0.36     0.01        1    10.00    10.00  integral_image_compute
  2.57      0.36     0.00      146     0.00     0.00  mat_inverse_and_multiple
  1.43      0.36     0.00      129     0.00     0.05  get_orientation
  0.00      0.36     0.00        8     0.00     0.00  alloc_2D_float
  0.00      0.36     0.00        8     0.00     0.00  alloc_2D_uchar
  0.00      0.36     0.00        8     0.00     0.00  free_2D_float
  0.00      0.36     0.00        8     0.00     0.00  free_2D_uchar
  0.00      0.36     0.00        2     0.00     0.00  file_check
  0.00      0.36     0.00        2     0.00     0.00  file_close
  0.00      0.36     0.00        1     0.00     0.00  alloc_pixels
  0.00      0.36     0.00        1     0.00     0.00  bmp_check
  0.00      0.36     0.00        1     0.00     0.00  bmp_get_image_details
  0.00      0.36     0.00        1     0.00     0.00  bmp_read_pixels
  0.00      0.36     0.00        1     0.00     0.00  bmp_verify_image_size
  0.00      0.36     0.00        1     0.00   350.08  build_scalespace
  0.00      0.36     0.00        1     0.00     0.00  free_pixels

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.
 
 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
       else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this 
       function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
       the function in the gprof listing. If the index is
       in parenthesis it shows where it would appear in
       the gprof listing if it were to be printed.
FF
```



[H]

Figure 7.2: *Functional block diagram of i.MX515EVK, borrowed from the i.MX515EVK data sheet.*