A Hardware Architecture for Scale-space Extrema Detection

HAMZA BIN IJAZ

KTH Information and
Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

ROYAL INSTITUTE
OF TECHNOLOGY

A Hardware Architecture for Scale-space Extrema Detection

HAMZA BIN IJAZ
hamzai@kth.se

Master's Thesis at KTH School of ICT and Robert Bosch GmbH
Supervisors:
Arne Zender, Robert Bosch GmbH
Fernando Ortiz Cuesta, Robert Bosch GmbH
Examiner:
Mark T. Smith, KTH School of ICT

TRITA-ICT-EX-2012:183

Abstract

Vision based object recognition and localization have been studied widely in recent years. Often the initial step in such tasks is detection of interest points from a grey-level image. The current state-of-the-art algorithms in this domain, like Scale Invariant Feature Transform (SIFT) and Speeded Up Robust Features (SURF) suffer from low execution speeds on a GPU(graphic processing unit) based system. Generally the performance of these algorithms on a GPU is below real-time due to high computational complexity and data intensive nature and results in elevated power consumption. Since real-time performance is desirable in many vision based applications, hardware based feature detection is an emerging solution that exploits inherent parallelism in such algorithms to achieve significant speed gains. The efficient utilization of resources still remains a challenge that directly effects the cost of hardware.

This work proposes a novel hardware architecture for scale-space extrema detection part of the SIFT algorithm. The implementation of proposed architecture for Xilinx Virtex-4 FPGA and its evaluation are also presented. The implementation is sufficiently generic and can be adapted to different design parameters efficiently according to the requirements of application. The achieved system performance exceeds real-time requirements (30 frames per second) on a 640 x 480 image. Synthesis results show efficient resource utilization when compared with the existing known implementations.

Keywords: SIFT, computer vision, scale invariance, scale-space extrema detection, FPGA, feature detection, gaussian smoothing, separable convolution.

Acknowledgments

First of all, thanks to God, the Almighty, for having made everything possible for me by giving me strength and courage to do this task.

A sincere gratitude to my supervisors Fernando Ortiz Cuesta and Arne Zender for giving me the opportunity to conduct my thesis at Robert Bosch GmbH and their continued support and valuable inputs during the course of my work. Friendly and thoughtful discussions with Fernando were the key to some of the major milestones achieved in this work. Special thanks to Arne whose help enabled me to focus on the right directions.

I am grateful to my supervisor/examiner at KTH Royal Institute of Technology Prof. Mark T. Smith for responding to all my queries, giving feedbacks, guidance and advices throughout the course of my masters at KTH.

Also many thanks to Thilo Grundmann and Carsten Dolar at Robert Bosch GmbH for there expert opinions during our thought provoking meetings.

Finally I would like to acknowledge the unconditional support of my mother and father, for all their love and faith in me and for their never-ending encouragement and patience.

Hamza Ijaz

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This Master thesis aims to propose a generic, dedicated hardware architecture for real-time detection of scale invariant interest points in the image. Feature detection is widely used in vision systems for various applications, however they are computationally intensive, requiring considerable amount of resources in terms of time, power and silicon. Scale invariant feature transform SIFT proposed by Lowe [1] [2] is one of the most popular techniques used in image matching and object recognition applications. SIFT is computationally expensive and resource hungry which makes it challenging to be utilized for real time applications.

This work is carried out as part of a requirement to complete, Master of Science in Design and Implementation of ICT Products and Systems at the School of Information and Communication Technology, KTH Royal Institute of Technology in Stockholm, Sweden. The content of this work is defined at Robert Bosch GmbH in Hildesheim, Germany, by the Corporate Sector Research and Advance Engineering, which is in charge of designing, testing and exploring systems, components and technologies. Its innovations consistently aim to achieve an improvement in the quality of life.

This chapter gives the motivation for carrying out this work, explains the problem area, and finally outlines the structure of remaining sections of this report.

1.1 Motivation

SIFT (Scale Invariant Feature Transform) is a commonly used algorithm in robotic applications to solve problems of object recognition, tracking and camera based robot localization. Object recognition with SIFT features makes possible the matching of objects with independence of their size and position in the captured image. The extraction and processing of SIFT features on real time is still a challenge. Current software based SIFT implementations typically involve the use of high power GPU to achieve less than real-time performance, which is not desired in many em-

bedded applications. Due to the elevated power consumption of graphic cards, the need for an dedicated hardware accelerator becomes evident.

In order to achieve the desired results a new design needs to be proposed consisting of hardware accelerators based on FPGA. A complete hardware implementation of a SIFT like algorithm is beyond the scope of this master thesis work. This work focus on the interest point detection which is the initial and considered to be the most computationally intensive part in SIFT. The nature of this part in the SIFT algorithm is best suited for the FPGA based design.

1.2 Problem Definition and Specifications

One important question in video processing is to decide which algorithm parts should run on software processors and which ones should be implemented in hardware, for instance FPGA or ASIC. A dedicated Computer Vision Camera with a Virtex-4 VFX60 FPGA is the chosen platform for this work. The camera comprises an FPGA for demanding image processing task i.e. feature detection and a Texas Instruments TI OMAP35x/ARM Cortex-A8 based system-on-chip for efficient software implementation of high-level image processing i.e. feature description.

The scope of this master thesis is to implement part of the SIFT software algorithm in hardware on a FPGA. The designed architecture should comprise dedicated hardware units for the computationally intensive part of the SIFT algorithm, *Scale-space extrema detection*. The design must be synthesizable, adaptable, generic and demand few FPGA resources. It may be adaptable to different number of octave and scales.

The input images are provided by a camera with a VGA format 640x480 with an 8-bit grey scale image and a hardware module which stores the images from the camera on the external RAM. In order to access and store the input images and output data, a pre-developed cache hardware module will be utilized. The output data will be stored on the external RAM memory connected with the FPGA using a multi-port memory controller IC provided by Xilinx. The minimum speed requirement from the target application is 15 frames/images per second for an image size of 640x480 pixels. However the desired goal of this work is to achieve a speed of 30 frames per second and it will be referred to as real-time in this report.

1.3 Organization

The rest of this report is organized as follows.
Chapter 2 (Scale-space Extrema Detection) gives background for the readers of this report covering related concepts, details of target algorithm, software simulation approach and simulation results.

Chapter 3 (Hardware Design and Implementation) starts with a brief overview of hardware related concepts followed by a discussion while reviewing related work and then gives a system overview followed by design details of the proposed hardware architecture and finally the implementation details.
Chapter 4 (Verification) provides the verification method employed for testing the implemented design.
Chapter 5 (Evaluation) evaluates and gives discussion on the results. It also enlists synthesis results and compares them with the state of the art.
Chapter 6 (Conclusion) concludes this work with the summary of overall contributions. It also gives possible enhancements and directions for future work.

Chapter 2

Scale-space Extrema Detection

Features found in an image contain properties that make them suitable for matching differing images of a similar object or a scene. Detecting good features is a complex problem. [3] gives a discussion on the properties of the ideal local feature. The desired properties from a feature depend on the actual application.

Scale Invariant Feature Transform (SIFT) is an image descriptor developed by David Lowe [1] [2] for image-based matching. The features produced by SIFT are invariant to image scaling and rotation, and partially invariant to change in illumination and 3D camera viewpoint [2]. In practice SIFT descriptor has proven to be very useful for applications involving image matching and object recognition. SIFT uses Scale-space extrema as candidate features. This work concentrates on the Scale-space extrema detection with focus on dedicated hardware implementation.

This chapter first gives an overview of the Gaussian and its properties which play a major role in building the scale-space. After that the details of steps involved in Scale-space extrema detection are presented. Followed by that, a simulation approach is discussed and some results are presented.

2.1 The Gaussian kernel and its properties

The Gaussian kernel is defined in 1-D as,

$$G(x, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x^2)/2\sigma^2} \tag{2.1}$$

Where σ determines the width of kernel and is often refer to as the *inner scale* or in short *scale*. In statistical terms it is referred to as the *standard deviation* while the σ^2 is referred to as the *variance*. The term $\frac{1}{\sqrt{2\pi}\sigma}$ in front of the equation 2.1 is the normalization constant and it comes due the fact that the integral over the exponential function is not unity. Taking this constant in the equation makes it a

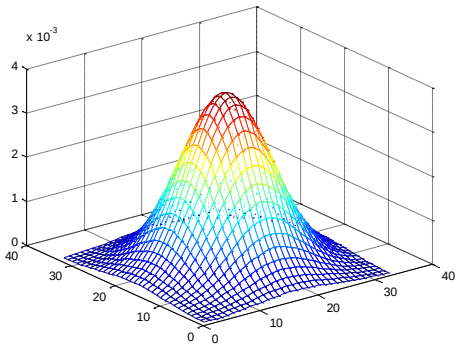


Figure 2.1: An example of a Gaussian curve with $\sigma = 6.44$ in 3D

normalized kernel with the integral unity for every σ. This means that increasing the σ value effectively decreases the height of kernel while increasing its width as observed in the figure 2.10, where five kernels are plotted on the same axes. This characteristic of this kernel is used to control the smoothing of an image while preserving the overall intensity. As the σ gets greater the width of the kernel increases and hence the neighbouring boundary around the pixel increases while the process of smoothing. Therefore when a grey scale image is blurred with a normalized kernel the average grey level of the image remains the same. This property is known as *average grey level invariance*.

2.1.1 Self Similarity

The convolution of two Gaussian kernels result in a wider Gaussian kernel whose variance is the sum of the variances of the constituting kernels. This property is known as *Self Similarity*. Since the convolution with a Gaussian is a linear operation, small Gaussians can be convolved repetitively to get a large blurring step. This phenomena is also know as the *cascade filtering* or *cascade smoothing*.
A classic example of self symmetry is the pascal triangle as shown in the figure 2.2. As can be seen, in order to obtain the effect of wider kernel small kernels can be

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5  10  10   5   1
```

Figure 2.2: Pascal Triangle

cascaded, e.g. kernel 2 and kernel 3 can be cascaded to get the effect of kernel 5

(2+3). So the effect of convolution is the same whether convolved with the larger kernel or with multiple small kernels.

2.1.2 Symmetry

The Gaussian kernel is symmetric. We will see how this property can be utilized to save the number of computations in chapter 3.

2.1.3 Separability

Another very distinctive property of Gaussian is that, a higher dimensional kernel can be described as a regular product of one-dimensional kernels hence making the higher dimensional kernels *separable*. As a consequence a product of Gaussian functions gives a higher dimensional Gaussian function. An important application is the speed improvement when implementing numerical separable convolution. In chapter 3 it will be explained in detail how a 2-D Gaussian convolution can be replaced by a cascade of 1-D convolutions requiring far fewer multiplications.

2.1.4 Size of the Gaussian kernel

The Gaussian kernel gets larger with greater standard deviation σ, alternatively the scale factor. Theoretically, the size of Gaussian kernel ranges from $-\infty$ to $+\infty$. In practice however it has a negligible value in the range beyond 5σ. The size of kernel often directly effects computational complexity and is desired to be smaller for increased efficiency. As an example a $\sigma = 3.22$ results in a kernel size of 17 (3.22 x 5 = 16.1 approximated to 17).

2.2 Scale-Space Extrema Detection

The original SIFT Descriptor [1] [2] is computed using the image intensities around the locations of interest in the image which are often referred to as *Interest Points*. These interest points are obtained from the scale-space which are detected from the *Difference of Gaussians* (DoG). Figure 2.3 shows where the extrema detection part fits in SIFT along with other major steps involved.

A *scale-space* is a stack of 2D images, where *scale* is the third dimension. The process of detecting extrema is further divided into three steps as indicated in the figure 2.3. The first step is to construct a Gaussian pyramid through repeated smoothing and sub-sampling. The Gaussian pyramid is then used to compute the difference of Gaussian (DoG) pyramid. From this DoG pyramid, maxima and minima are detected that are known as scale-space extrema, also referred as keypoints or interest points.

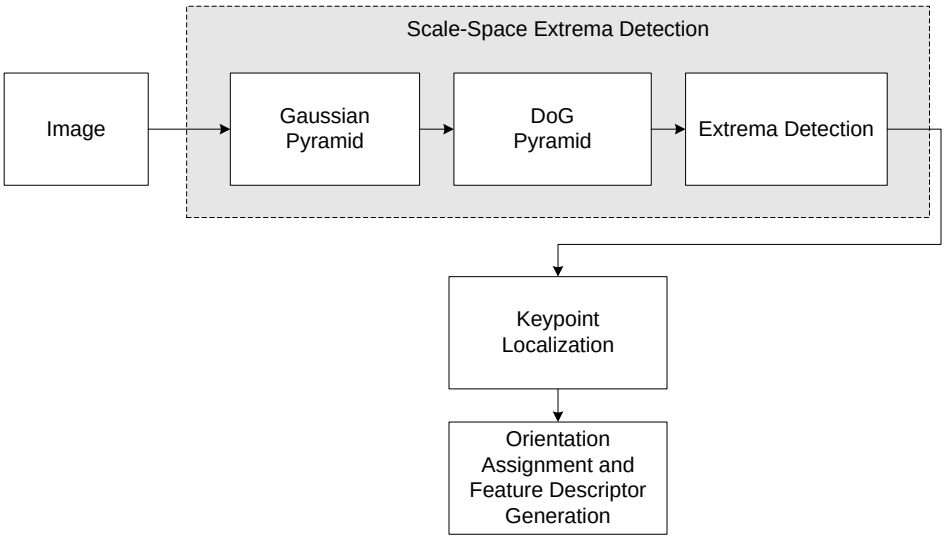


Figure 2.3: Steps in SIFT Algorithm

2.2.1 Gaussian Pyramid Construction

SIFT uses Gaussian function as a scale space kernel based on the results shown by Koendrink [4] and Lindeberg [5]. The scale-space $L(x, y, \sigma)$ of an image is defined as.

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{2.2}$$

Where $I(x, y)$ is an input image intensity level at the pixel location determined by x and y coordinates, $*$ is a convolution operator in x and y, while $G(x, y, \sigma)$ is the variable scale Gaussian defined as.

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \tag{2.3}$$

σ signifies the scale factor which determines width of the kernel. It is explained in detail in section 2.1.

The convolution of image with the variable scale Gaussian kernel produces a stack of blurred images with the amount of blur dependent on scale factor. The scale of Gaussian kernel is varied using a constant multiplicative factor k which is computed by the relation $k = 2^{1/s}$, where s is the number of scales that can contain keypoints. It will become more clear in section 2.4.2 where details of simulation are given. The stack of Gaussian blurred images produced, makes one octave. A Gaussian pyramid is composed of multiple octaves with different image sizes. For each successive octave the image is down sampled or up sampled. The number of octaves and scales directly affect the robustness and complexity. A higher number of either scales or octaves offers higher robustness to matching scale changes at the

cost of higher complexity. A Gaussian pyramid consisting of five Gaussian scales and three octave is shown in the figure 2.4.

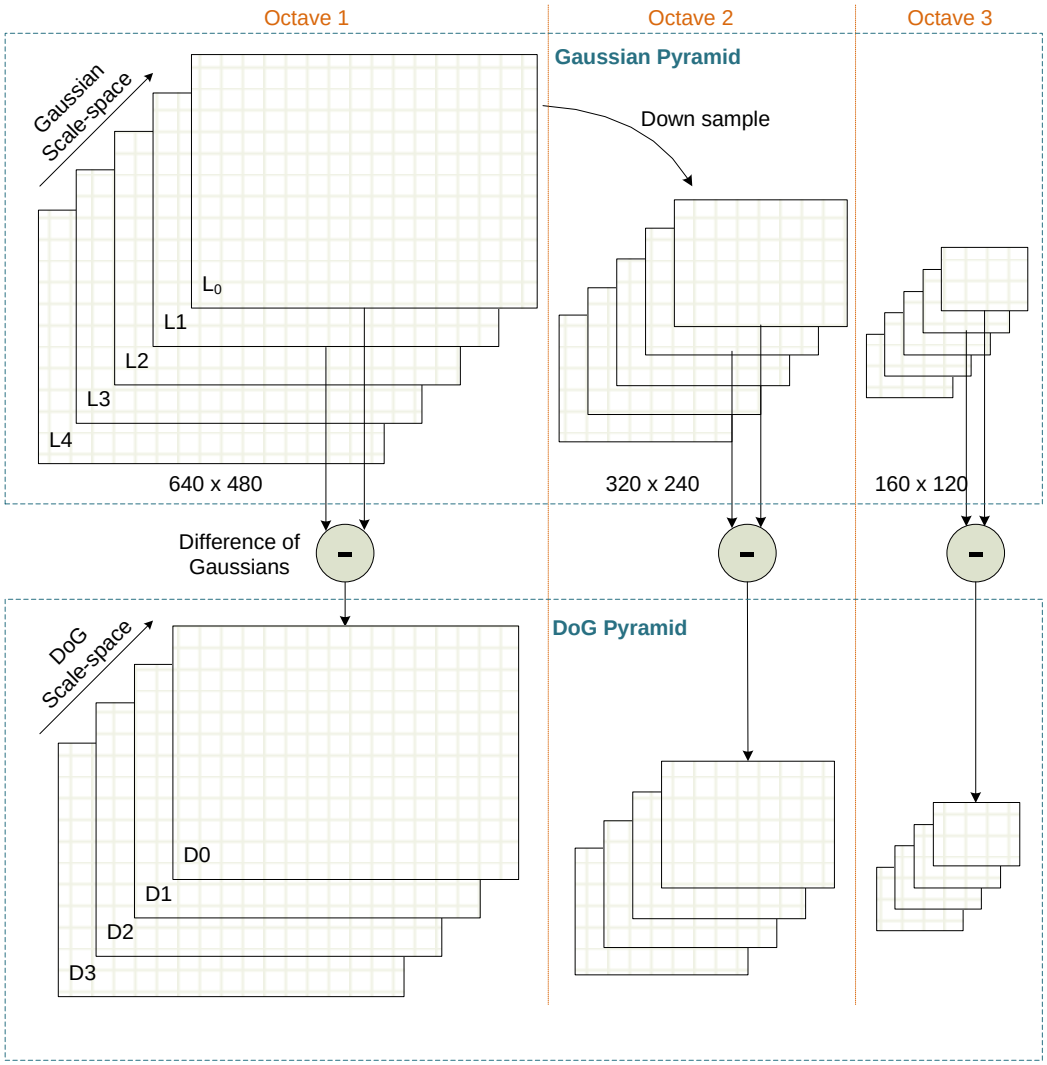


Figure 2.4: Gaussian and DoG pyramids

2.2.2 Difference of Gaussian Pyramid Construction

Once the Gaussian pyramid is constructed the next step is to compute the DoG pyramid. This step is significantly improved in terms of computational cost by Lowe's method where it is computed by taking the difference between the two adjacent scales separated by a constant multiplicative factor k, compared to the Lindberg's method of scale normalized Laplacian of Gaussian [6]. According to Lowe, Lindeberg's scale normalized Laplacian of Gaussian $\sigma^2\nabla^2 G$ closely approximates the

difference of Gaussian function. Lindeberg showed that for true scale invariance the factor of σ^2 is required to normalize the Laplacian.

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) \tag{2.4}$$
$$= L(x, y, k\sigma) - L(x, y, \sigma) \tag{2.5}$$

Where 2.5 defines the difference operations. It can be seen that convolution can be performed after taking the difference of Gaussian filtered images to get the DoG scale space directly saving one image convolution and hence multiplications that are expensive. However the Gaussian scale space is needed later for the descriptor generation part of the algorithm. Moreover there is no significant gain in terms of hardware, that will become evident in chapter 3.

2.2.3 Extrema Detection

Candidate features are detected from the DoG scale-space by finding the local extrema. In order to detect the local maxima or minima from $D(x, y, \sigma)$ each sample in the DoG image is compared to its eight neighbours in the current image and nine corresponding neighbours at the adjacent scales. A sample point is selected as a candidate keypoint if it is either larger or smaller than all these neighbours. These extrema are computed for all the octaves in the similar manner. In figure 2.5 symbol 'x' corresponds to a *keypoint*.

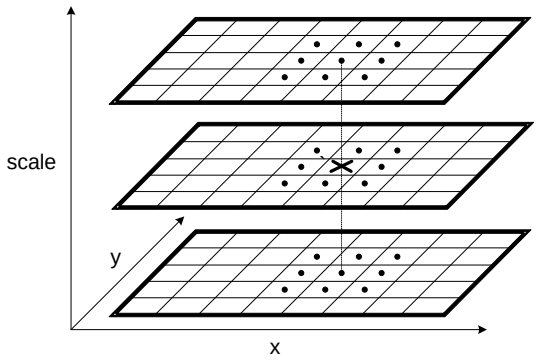


Figure 2.5: Detection of Local Maxima and Minima

This approach gives rise to a lot of keypoints depending on the image size. Often quality of keypoints is more important than the quantity for reliable object recognition. The features that have greater probability to be found in the other version of the image exhibit better quality. For ensuring the stability of keypoints, the local extrema are compared with a minimum threshold value. The extrema that have relatively higher minima or maxima value pass the threshold test and are considered, while the weak keypoints having low contrast are discarded, resulting in less but more stable candidates.

2.3 Image Convolution

This section gives details of image convolution using 2D kernels and then 1D kernels in the case of separable kernel as that of the Gaussian.
Figure 2.6 shows the concept of 2D image convolution considering a case of 3x3 kernel. A 3x3 window (comprising nine pixels) is taken from the image in a sliding fashion starting from top left and ending at bottom right. Each of the windows gets convolved with 3x3 Gaussian kernel by multiplying pixel values with corresponding kernel values and summing up all the results as illustrated by the figure.

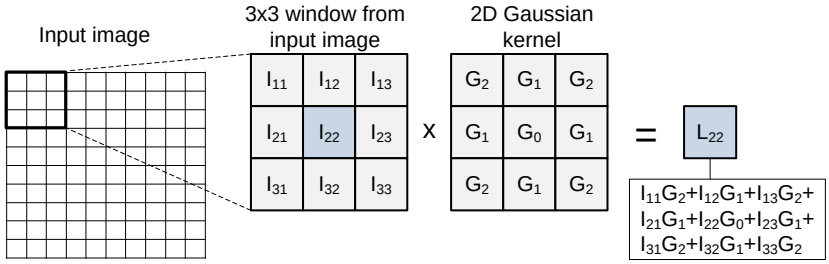


Figure 2.6: Image convolution

Separable convolution as clear from the name is broken into two steps that can be referred to as vertical convolution and horizontal convolution. As illustrated in figure 2.7 considering the same case of 3x3 kernel now divided into two 1D kernels, in separable convolution first each vertical sub-window from the 3x3 window in the image gets convolved with the vertical 1D kernel. The result of this convolution is a horizontal window which then gets convolved with the horizontal kernel to get the final result of 2D convolution.

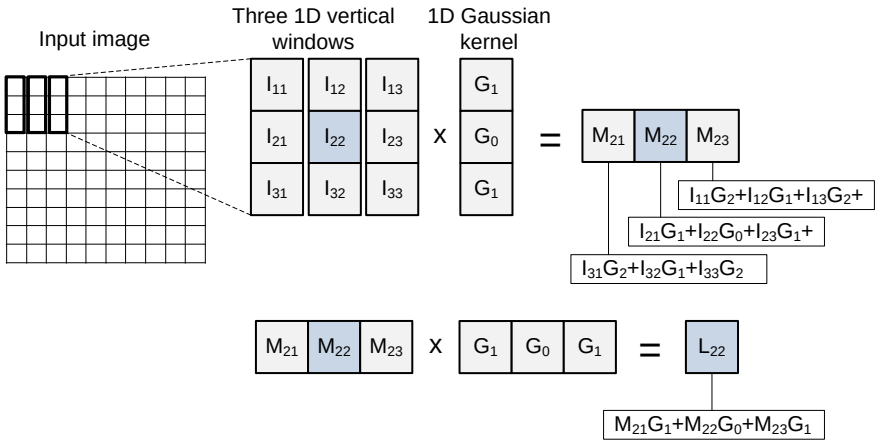


Figure 2.7: Separable convolution

The difference between 2D convolution and separable convolution is clear in terms of the number of computations involved. In case of 2D convolution n^2 multiplications are required to compute a single result, where n is the width or height of the kernel. While in 1D convolution $2n$ multiplications are required for the same computation. Clearly the separable convolution outperforms in terms of the computational complexity and this becomes significant when larger kernels are used.

2.4 Simulation

This section describes the simulation method with details of scale-space extrema detection. The simulation is done in *MathWorks MATLAB* (Version 7.11.0.584 (R2010b), 32-bit(win32)). The focus of implementation is not to obtain the exact results as obtained by Lowe in the original implementation of SIFT but to verify the functionality and reliability of the method sufficiently, before the proposition of a dedicated architecture for implementation in hardware which is the primary goal of this work. The software simulation results will be taken as reference for the evaluation of implementation results.

2.4.1 Overview of Method

A hybrid approach has been adapted. Andrea developed an open implementation of the SIFT detector and descriptor [7] to produce results compatible to Lowe's version. This library is being utilized for the later parts of SIFT algorithm to visualize the results. Figure 2.8 illustrates the method employed.

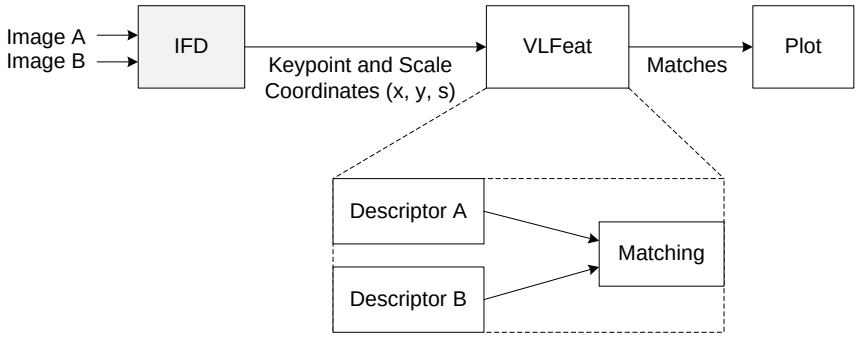


Figure 2.8: Simulation Method

The IFD (image feature detector) detects the candidate keypoints from an image, which are then passed to VLFeat [8] block to generate the associated descriptor vector which is then used to perform matching between two versions of the image. The matches are then plotted on top of both images to visualize the results. Following sections give the details of the IFD block. Details about the VLFeat library

setup and related functions can be found online[1]. Figure 2.9 illustrates the flow of MATLAB program for the scale-space extrema detection.

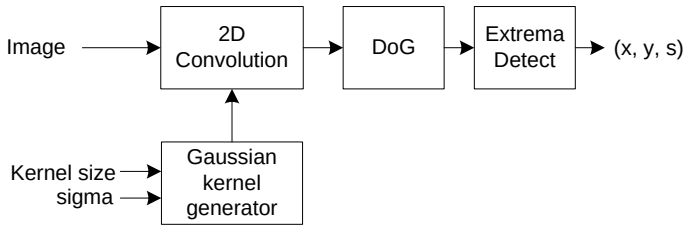


Figure 2.9: Details of IFD block

2.4.2 Simulation Details

At first the Gaussian scale-space is created by generating a series of smoothed images with a series of σ values. The σ domain is quantized in logarithmic steps that form an octave, which are further divided into sub-levels as explained earlier in section 2.2.1. The value of σ in terms of o octave and s scale is given by $\sigma(o, s) = \sigma_0 2^{o+s/S}$, where σ_0 is the base scale level e.g. $\sigma_0 = 1.61$ and S signifies the total number of scales.
At each successive octave the image is down-sampled by a factor of two by simply discarding every next row and column from the image for the current octave. Since the Gaussian kernel is symmetric (see 2.1), 1D kernels are generated by the Gaussian kernels generator function, based on the prior smoothing factor σ and the kernel size. Kernel size is calculated based in the Gaussian standard deviation criteria (discussed in 2.1). The repeatability of keypoint detection continues to increase with prior smoothing factor σ [2], however increasing σ means larger kernels which are not efficient to implement in hardware. Hence there is a trade off in selecting σ for efficient implementation. This work follows a conservative approach and takes the prior smoothing factor according to the experimental results of Lowe which suggested $\sigma = 1.6$ and above for achieving a close to optimal repeatability rate. k is calculated from the relation $k = 2^{1/s}$, where s is the number of scales that can contain features. Here $s = 2$ is chosen, which gives good balance between kernel size, stability of the detected extrema and the number of detected extrema. This results in five Gaussian smoothed images as $s + 3$ images are needed to produce s scales from which extrema are detected. Table 2.1 gives the σ for all the scales while figure 2.10 shows the corresponding generated kernels (from the narrowest to the widest respectively).

The convolution block, convolves the Gaussian kernels with the input image to produce Gaussian scale-space.

[1] http://www.vlfeat.org

	Value
σ_0	1.61
σ_1	2.27
σ_2	3.22
σ_3	4.55
σ_4	6.44

Table 2.1: Smoothing factors for various scales

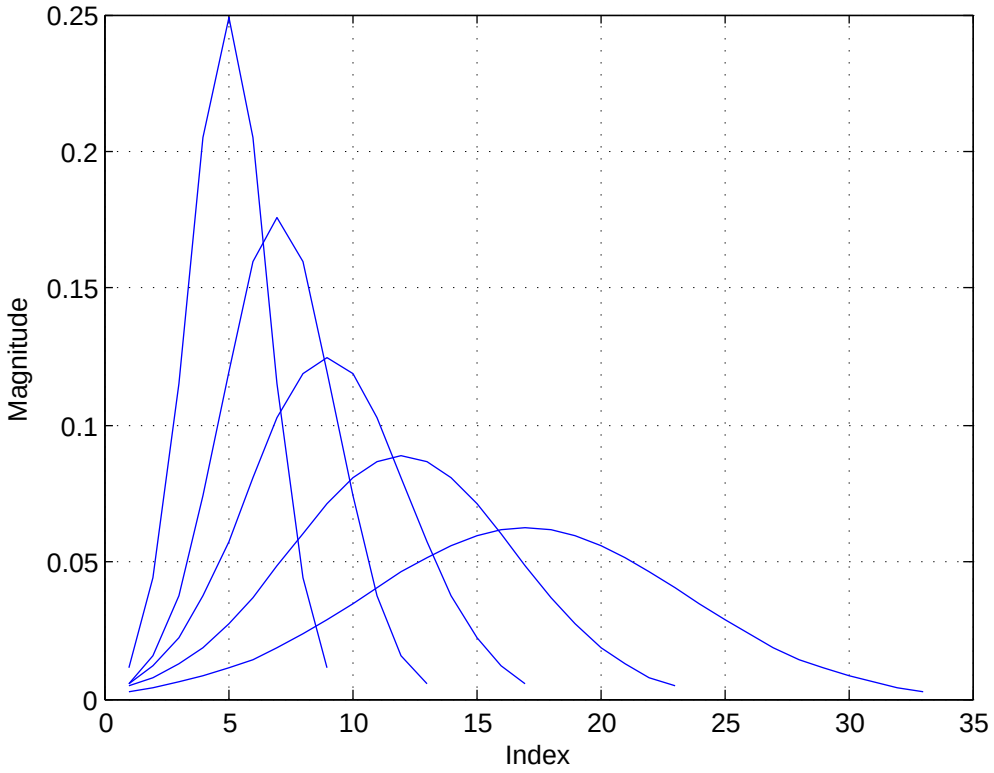


Figure 2.10: Double Precision (64-bit) Gaussian kernels

Vertical convolution is followed by the horizontal convolution using the generated 1D kernels. Figure 2.11 illustrates the Gaussian filtered images with a 3D graph showing the effect of increasing sigma. It can be seen how the smoothing effect increases with increase in sigma while the details get suppressed.

Adjacent Gaussian scales are subtracted to get the DoG scale-space. Finally the extrema are detected from the DoG scales. Figure 2.12 illustrates all the steps (considering direct filtering approach) with an image taken from the INRIA im-

age dataset for Feature Detector Evaluation [9]. The detected extrema are shown as white point on the binary image with circles drawn around them for increased visibility. They indicate the location of extrema in the image coordinates. The resultant two images give the coordinate also indicate the respective level in the scale-space.

In the next step these detected extrema coordinates along with the scale coordinates are used to generate the descriptor by the VLFeat block. Two VLFeat library functions are used to calculate and match the descriptors. The detected features are passed to the VLFeat function that returns the descriptor. This whole process is repeated for the transformed image. The descriptor from both images are passed to another VLFeat function which returns the matches between the two images based on Euclidean distance. Finally the matched features are plotted to visualize matches results under various image transformations. An example of the matched features is shown in figure 2.13.

2.4.3 Evaluation Data set

Several pairs of images have been taken form INRIA [9] dataset (with different transformation) for evaluation. Figure 2.14 shows the original pair of images selected from the Data set.

2.4.4 Simulation Results

The result of applying all the steps discussed in the simulation method on the set of images is illustrated in figure 2.15. All the images are resized and cropped to VGA size 640 x 480 to make it consistent with the input image specifications for this work. The results will be compared with the implementation results for evaluation purpose.

For the purpose of comparison between simulation results and implementation results this work utilizes visual inspection method to find the approximate correct matches among the total number of matches. For a detailed evaluation however, standard evaluation procedures should be adapted and is a proposed extension of this work which is briefly described here.

Performance is often measured by the Repeatability rate i.e. the percentage of points that are simultaneously present in two images of a same scene. [10] gives a detailed account of repeatability criterion. Other method that has gained more popular in evaluation of detectors is the recall and precision [12] [13]. Recall and precision are based on the number of correct and false matches for a pair of images. A *recall* vs 1-*precision* graph is used for the evaluation. Recall 2.6 is given by the number of correctly matched regions with respect to the number of corresponding regions between two images of the same scene. While 1-*precision* 2.7 is given by

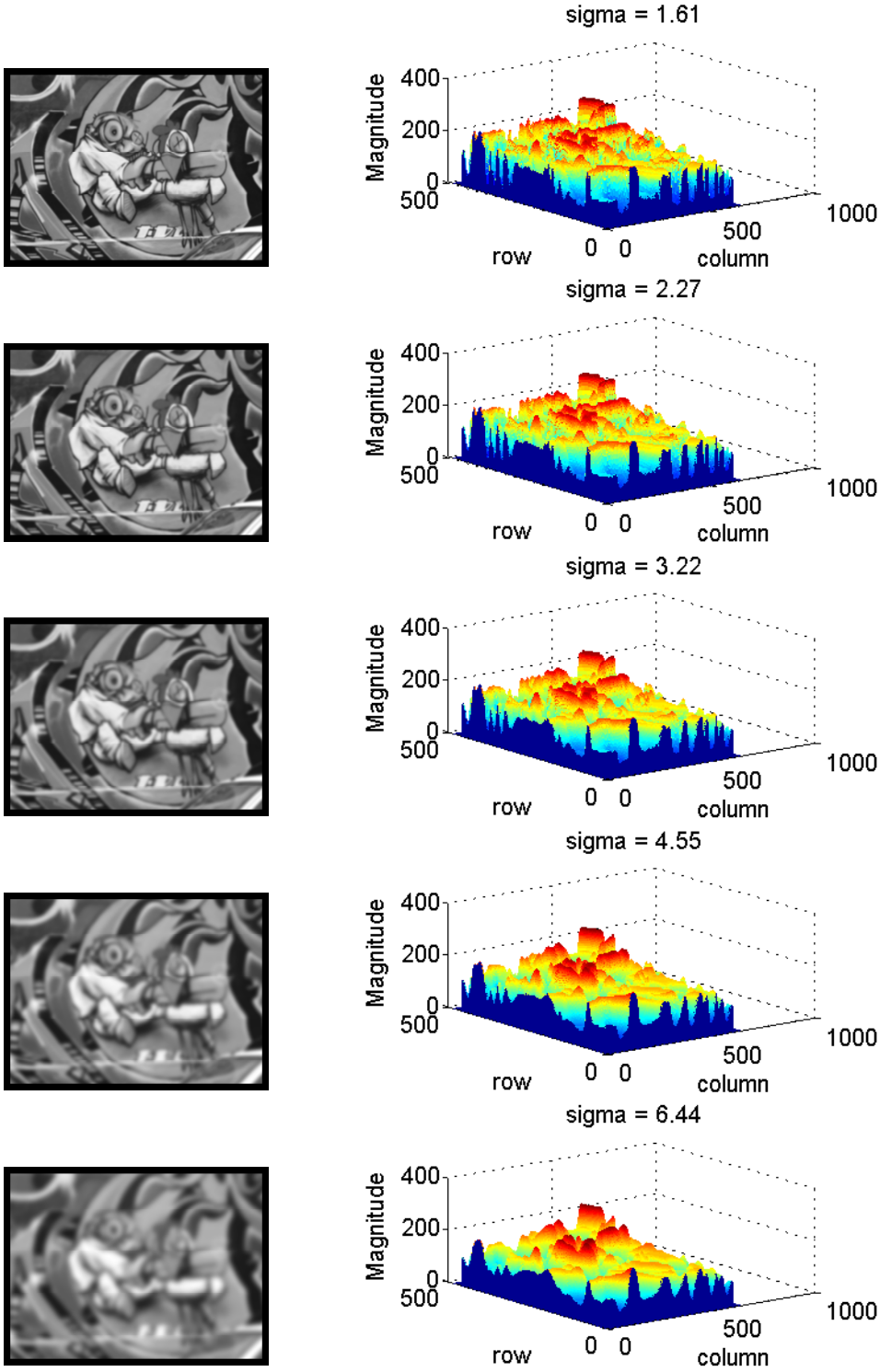
the number of *false-positives* with respect to the total number of matches (correct or false).

$$recall = \frac{numberofcorrect - positives}{totalnumberofpositives(correspondences)} \tag{2.6}$$

$$1 - precision = \frac{numberoffalse - positives}{totalnumbermatches} \tag{2.7}$$

A *correct-positive* is a match where two keypoints correspond to the same physical location. While a *false-positive* signifies that the two keypoints come from different physical locations.

The matching algorithm uses Euclidean distance between two feature vectors to determine if the two vectors belong to the same keypoint in different images. Distance threshold is an important parameter in matching process. Changing the distance threshold affects the number of *false-positives* and allows to select appropriate trade-off between number of *false-positives* and *false-negatives*.



(a) Gaussian filtered Images [9]

(b) Gaussian blur effect

Figure 2.11: Gaussian Filtering

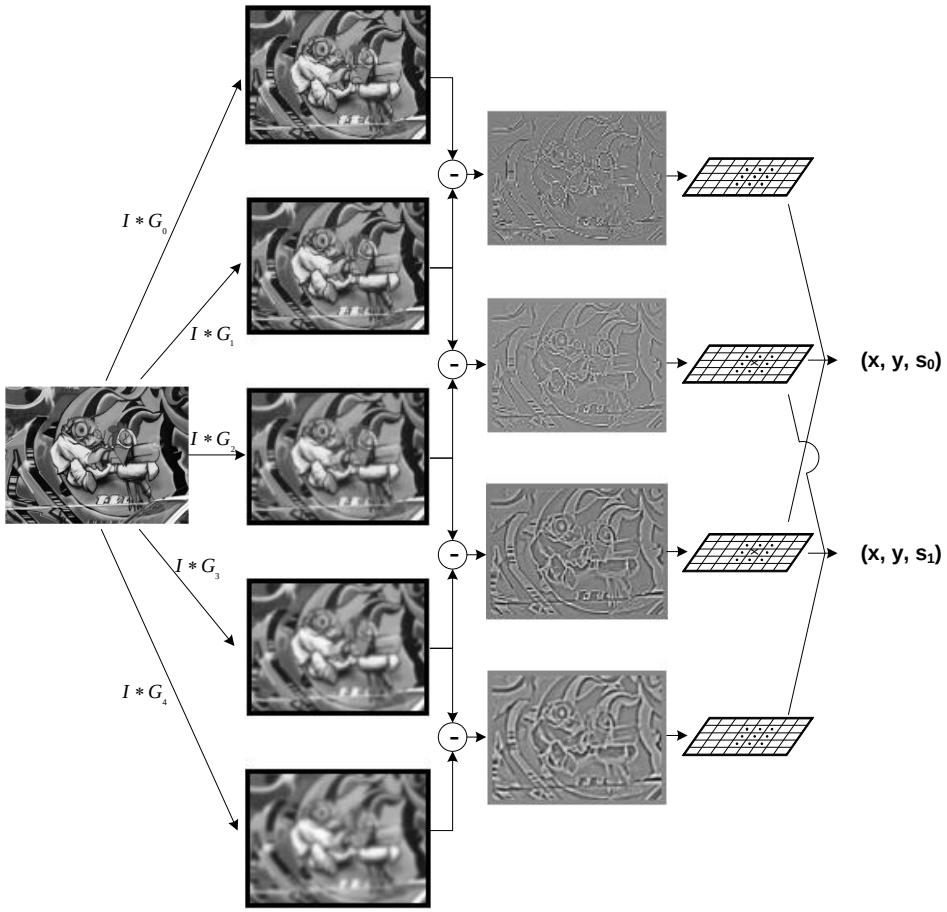


Figure 2.12: SIFT based extrema detection

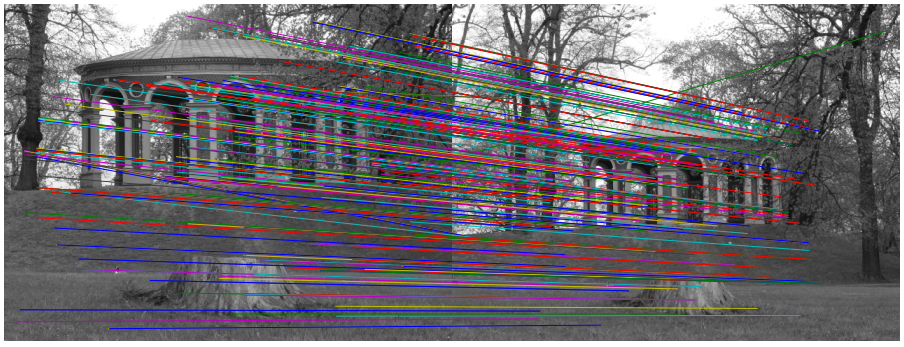


Figure 2.13: Matching Results: Two images of a scene taken with different Zoom level (640 x 480 image size)

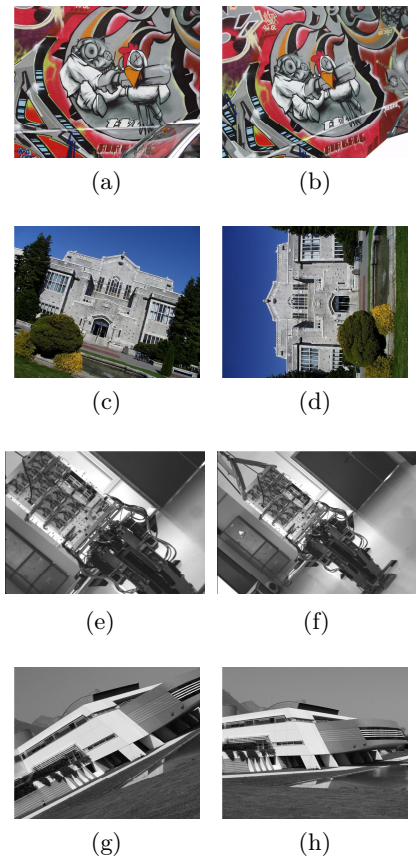
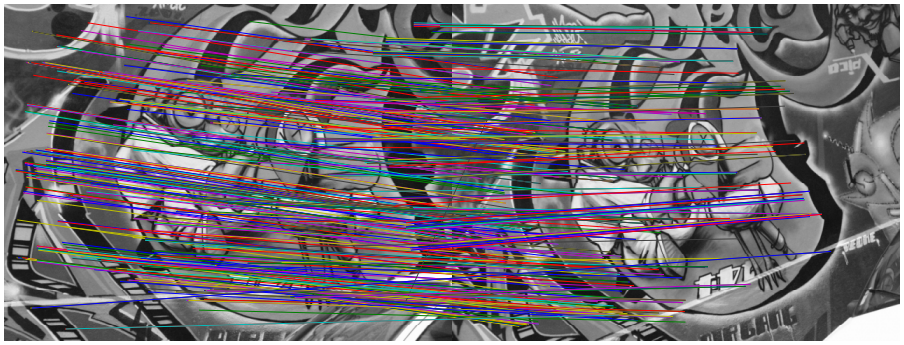
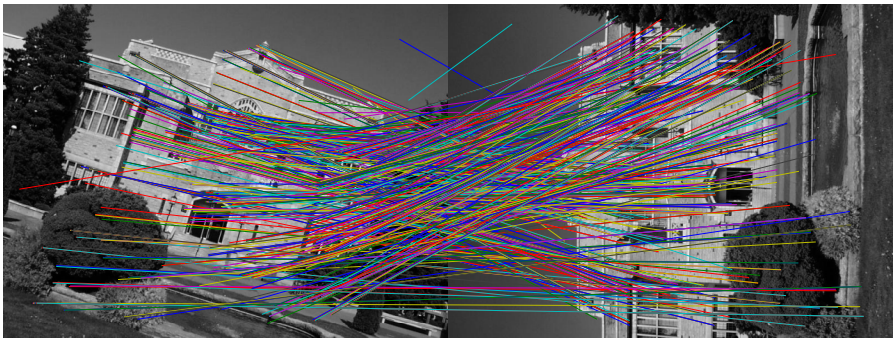


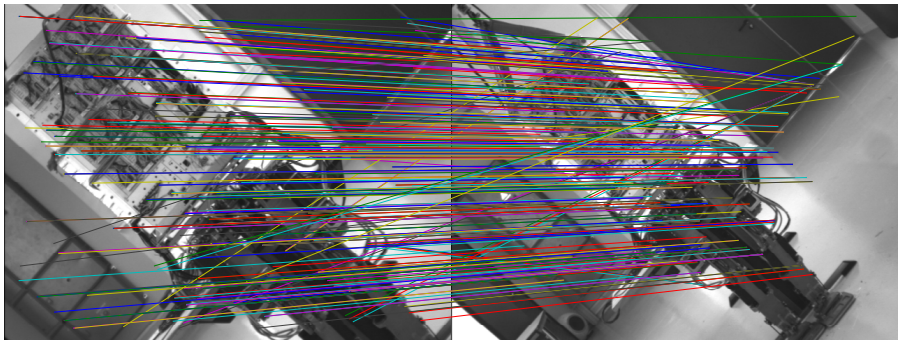
Figure 2.14: Data set. Example of images used for evaluation, (a)(b) and (c)(d) Viewpoint, (e)(f) Zoom, (g)(h) Rotation + zoom



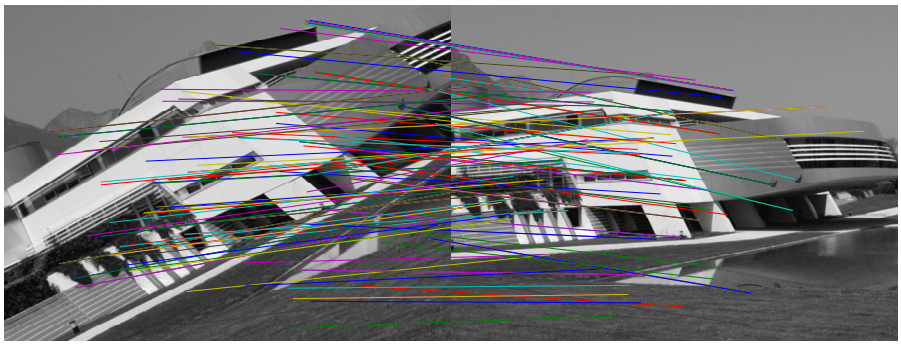
(a) Matching results on graffiti images taken from different viewpoints



(b) Matching results for images taken from different viewpoints



(c) Matching results for images with different Zoom levels



(d) Matching results for images with Rotation and Zoom variation

Figure 2.15: Software Simulation Matching Results

Chapter 3

Hardware Design and Implementation

This chapter starts with basic concepts related to Field Programmable Gate Arrays (FPGA) including a description of Xilinx Virtex-4 FPGA (device XC4VFX60) to prepare the reader for the following sections if necessary. It then give a discussion with a review of possible solutions that are adapted for the problem at hand. Followed by that it briefly describes the target system at higher level and finally lays down design and implementation details of the proposed hardware architecture.

3.1 Overview of Hardware related concepts

FPGA is composed of predefined resources with programmable interconnects. The specifications that play a major role in designing an application utilizing an FPGA are number of configurable logic blocks (CLBs), number of fixed functional blocks such as multipliers and size of memory resources such as block RAM that is embedded in the FPGA.

All the resources on an FPGA that can perform logic functions are known as logic resources. They are grouped in slices to make configurable logic blocks (CLBs) consisting of number of LUTs, flip flops and multiplexers. CLBs are the main logic resource for implementing both sequential and combinatorial circuits.

A LUT is made of logic gates hard-wired on FPGA. They provide a fast way to produce output for a logic operation by using predefined list of outputs for all possible input combinations. The LUT implementation differ from one FPGA family to another. A flip-flop represents a single bit capable of maintaining two stable states. A multiplexer is a circuit that selects between multiple inputs, and outputs the selected input.

3.1.1 Virtex-4 FPGA

Virtex-4 devices are user programmable gate arrays with various configurable elements and embedded cores optimized for high-density and high-performance system designs [14]. In Virtex-4 FPGA a single CLB is made up of four Slices interconnected. Table 3.1 summarizes the logic resources in one CLB while table 3.2 summarizes the logic resources available in all CLBs of device XC4VFX60. Further details related to Xilinx Virtex-4 FPGA can be found in the *Virtex-4 FPGA User Guide* [15].

Slices	LUTs	Flip-Flops	MULT_ANDs	Arithematic & Carry Chains	Distributed RAM	Shift Registers
4	8	8	8	2	64 bits	64 bits

Table 3.1: Logic Resources in one CLB of Virtex-4 FPGA

Device	**Slices**	**LUTs**	**Flip-Flops**
XC4VFX60	25,280	50,560	50,560

Table 3.2: Logic Resources in all CLBs of Virtex-4 FPGA

3.1.2 Block RAM

Block RAM also referred as BRAM is a Random Access Memory that is embedded throughout the FPGA for storing data. Xilinx Virtex-4 (device XC4VFX60) has 232 block RAMs. Each block RAM stores 18 Kbits of data. It can be utilized in different possible configurations. The Xilinx CORE Generator System *Coregen* (introduced in next sections) provides option to configure the BRAM in different ways. Virtex-4 Block RAMs also contain optional programmable FIFO logic for increased device utilization.

3.1.3 Xilinx CORE Generator System

Coregen is an easy to use tool with a Graphical User Interface, that delivers parameterizable COREs optimized for Xilinx FPGA. It is included with all editions of Xilinx ISE Design Suite. It has an extensive library including memories, storage elements, DSP functions, math functions and various basic elements.

3.2 Discussion and Review

Real-time processing of images require high computational power. The amount of processing required per pixel determines the overall processing requirement for a

particular image processing algorithm.

The main reason to use reconfigurable hardware instead of a general purpose processor is to achieve higher performance with a better trade-off for cost. However in order to achieve such gains, the parallelism in algorithm has to be exploited for the target hardware. The most crucial part in SIFT based feature detection is the computation of the Gaussian Scale-space. It is the starting point on which the later design relies on.

There are mainly two approaches to generate the Gaussian pyramid. One method is using cascade filtering approach where the Gaussian filtered images are computed from the input image by recursive convolution. The other one generates all the Gaussian filtered images directly from the input image. Theoretically both produce the same results. However they have their own advantages and disadvantages. This section gives a discussion on these methods considering the related work.

In cascade filtering approach, the Gaussian scale-space is generated by successive convolutions of the image with Gaussian filters. The next blurring step is dependent on the result of previous step. The main reason for adapting this approach is to use small Gaussian kernels and save large multiplications. However there is a cost. Since the computation of next Gaussian scale is dependent on the previous blurred image and similarly the following scales require the result from the previous convolution, a pipelined architecture is usually employed and the intermediate results from the scales have to be buffered. This buffer requirement increases with the increase in number of scales.

Another implication becomes evident in the next step when the DoG is computed. The computation of DoG requires multiple scales, hence the result of the initial scales also have to be buffered until the next scale is being computed. Furthermore the extrema detection as discussed in the previous chapter, requires the result from three DoG scales which means even more buffers. Another implication is that this buffer requirement increases considerably with the increase in the number of scales. The exact buffer requirement depends on the architecture and can vary considering the logic vs complexity trade-off. A pipelined design saves resources by working only on a specific pixels at a time that are required for the result.

To achieve a better utilization of resources an ROI (Region of Interest) based approach is used in [16]. Bonata [17] also utilized the cascaded filtering approach. The exact buffer requirement for the feature detection stage is not clear enough from their design but their synthesis results for the feature detection stage show a moderate Block RAM utilization. However other logic resources are considerably high.

Yet another implementation of the SIFT based detector [18] uses the similar

approach. They introduce octave interleave processing to save resources by using the same hardware for multiple octaves. This is the primary reason there implementation show comparatively better utilization of resources than [17] however the Block RAM usage in there proposed architecture exceeds two times more.

One of the goals of this architecture is that it should be generic and scalable with low expense of hardware resources. At the same time it has to meet the real-time (30 fps) processing requirements. The cost has a direct relation to accuracy of the system for reliable operation. On the other hand in general, the systems that require fast computation often have some room for accuracy degradation. Nevertheless the determination of lower bound to accuracy is important and varies from application to application. There also exists applications that require high precision with low processing requirement (low frame rate). Although this is not true for all cases but in general it is natural to have a trade off between time, accuracy and resources and it directly effects the cost of hardware. The Gaussian scale-space is often limited by the scales and octaves due to the limited resources and timing requirements. On the other hand increasing these parameter not necessarily increases the quality of the results. In the case of pipelined cascaded Gaussian architecture, if the number of Gaussian scales and/or octaves are increased, the resource requirement increases considerably. Nevertheless there is an obvious advantage in terms of the constant throughput (after a certain latency determined by the pipeline).

3.3 System Overview

Figure 3.1 shows a block diagram of the system with interface between the major blocks. The image taken from the camera resides in external memory. To access the image data from this external memory a stream cache module (provided by Bosch) has being utilized. The slave controller module contains system configuration registers and is connected to the IFD_HW block. It generates the start signal for the IFD_HW along with the starting address of the image and the address for writing back the results (based on the size of input image). Finally the IFD_HW module contains the hardware accelerator for the proposed architecture of Scale-space extrema detection.

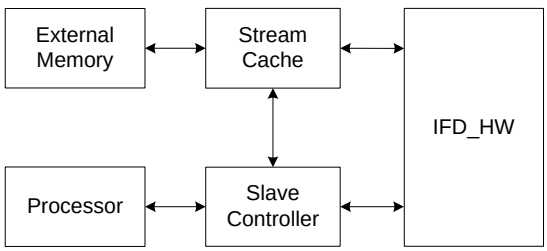


Figure 3.1: System overview

The SIFT description generation engine is controlled by an ARM Cortex 8 processor. The input image pixels are loaded to the IFD_HW block for processing, using a stream cache interface (provided by CR/AEM5[1] at Bosch)

3.4 Stream Cache Module

Stream cache is used for image data communication with the external memory. The top level module has an interface with stream cache module and the slave controller which also connects to the stream cache. Figure 3.2 shows the stream cache interface signals with inputs on the left and outputs on the right.

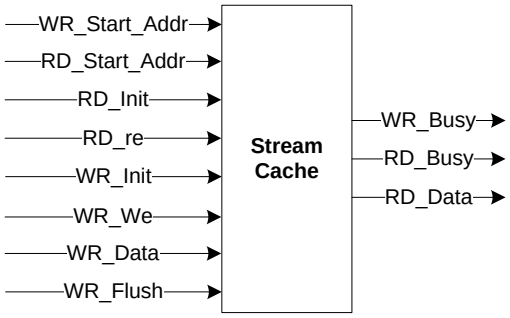


Figure 3.2: Stream cache interface

Figure 3.3 illustrates the operation of stream cache module under different phases.

The P_ctrl_start_sl signal from slave controller notifies the controller (IFD2_Ctrl) that it can initiate communication with the stream cache. The starting address for memory read is set through RD_Start_Addr. To start initialization the controller asserts WR_Init signal. Following that a low on the WR_Busy line indicates that initialization is done and controller can perform read/write operations.
During the initialization operation, 64 bytes packet is loaded into the stream cache from external memory. This data is stored in the Block RAM. Subsequently, when the number of bytes in the internal memory gets lower than some constant value, the stream cache loads new packet from the external memory.

For reading the image pixel data, read request signal RD_re is asserted for one cycle for a single word read and for multiple cycles in case of continuous reading. The stream cache returns the word in the next clock cycle on RD_Data port. Read request can not be made when RD_Busy is asserted and is illustrated figure 3.3.

The write operation is similar to the read operation except the fact that data is written to the port in the same cycle when write enable WR_we is asserted. The

[1]Corporate Research/Advanced Engineering Multimedia

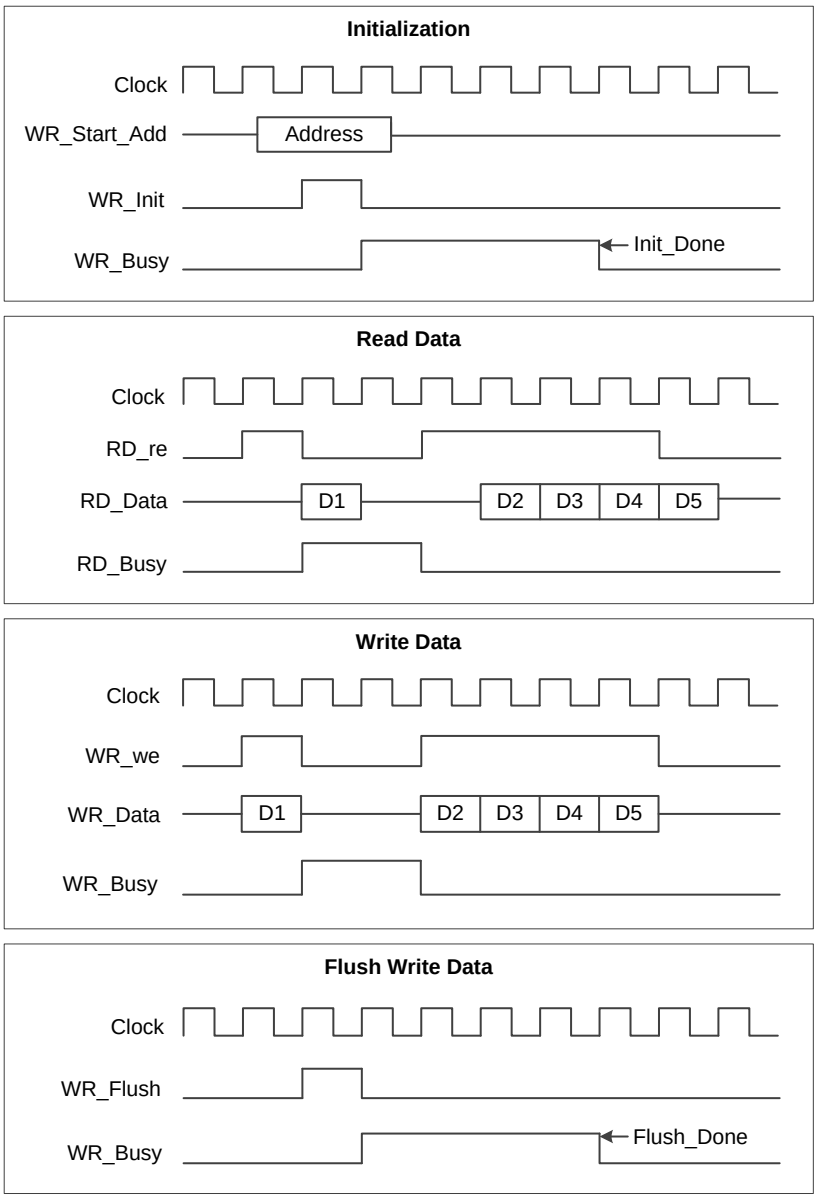


Figure 3.3: Stream cache - Initialization, Read, Write and Flush operation

starting address of write memory location is set through WR_Start_Addr. When the number of bytes in the internal write memory becomes 64, they are written to the external memory if the bus is available. In case the data bus is busy the output data can be written to internal buffer until the bus gets free.

The flush operation is used to empty the cache. When the WR_Flush signal is asserted for a single clock cycle, all the bytes present is the cache are written to the

external memory. The write flush completion is indicated by the WR_Busy signal going low.

3.5 Proposed Architecture for Scale-Space Extrema Detection

Whenever dedicated hardware is considered, exploiting parallelism becomes crucial. However there is always a cost to parallelism in terms of increase in the required silicon. The optimal utilization of resources while meeting timing and precision specifications is always a challenge in the design of dedicated hardware architectures. Several factors arise when the overall system (utilizing the design) is taken into account. Some of the challenges are optimal bus utilization and memory accesses. The SIFT feature detector works on several scales at a time which are but a blurred copy of the original image. This adds a considerable overhead specially in the case where the GPU are employed for processing. Reconfigurable architectures on the other hand give the designer a free hand to exploit there inherent parallel nature.

Direct filtering approach is being adapted in the proposed architecture. The scale-space can be divided into two dimensions, in terms of levels and octaves. It can be seen that both the dimensions have common inter processing. In other words if we consider a single octave, there are several scales that are computed using the same steps. Secondly if we consider all the octaves, each octave require the same computational steps. The question however is of the time. If speed becomes a critical factor then there have to be multiple units working in parallel. A careful analysis is necessary before deciding how much parallelism is required to meet the timing specifications while utilizing the resources in an optimal way. This method has been adapted for the design choices and the design space is explored considering the time, resources and precision while keeping the design generic in terms of algorithm design parameters and precision requirements. As a consequence the resulting architecture is expected to be scalable with less expense in terms of resources and gives acceptable trade-off in terms of timing requirements.

For the timing criteria and to make a comparison of performance (with existing architectures) nominal values are chosen for the design parameters that are already known to produce acceptable results from the simulation in previous chapter. Once this criteria is met we consider the divergence from this criteria and evaluate the results to see if they meet the expected bounds. Since the proposed architecture is generic, it can be evaluated for different parameters with minor changes.

Figure 3.4 gives an overview of the interconnections between the major elements of system and figure 3.5 shows the block diagram of proposed architecture for IFD hardware which adapts the direct filtering approach.

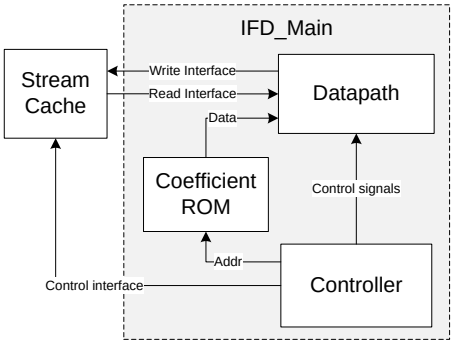


Figure 3.4: System Design

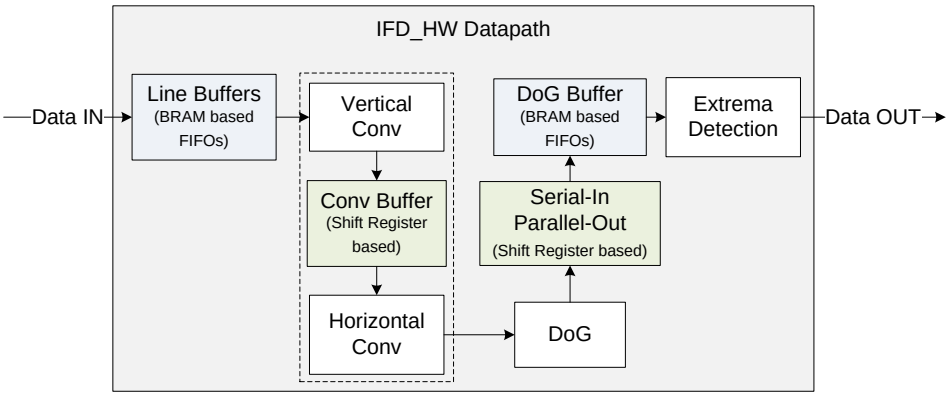


Figure 3.5: Block digram of the proposed hardware architecture for Scale-Space Extrema Detection

To comply with the simulation in earlier chapter the system is configured for five scales (Gaussian scales). The following explanation will consider this parameter for the ease of understanding.

Separability of Gaussian kernel is exploited to divide the 2D convolution into two 1D convolutions performed in a pipeline fashion. In order to save the multiplications Gaussian symmetry is exploited and symmetric addition is employed before multiplication with the kernel. There is a single hardware unit for Gaussian filtering, split into half for separable convolution and is pipelined. For a single pixel input at a time, maximum throughput and minimum latency is achieved by initially loading a pixel every cycle, and every five cycles (equal to the number of Gaussian scales) after the buffer gets full. At first the vertical convolution is performed in a time multiplexed fashion. At every cycle the pixels are convolved with a different Gaussian kernel to generate the appropriate blurring for the scale directly. The re-

sult is stored in registers shown as *Conv Buffer* in figure 3.5 (further details of this these registers are covered in section 3.5.3). Hence in every clock cycle one result is generated for one scale and in five cycles one result is generated for all the scales. The second 1D convolution module cannot start until it gets the required number of pixels (same as that of the first convolution block). Therefore these intermediate results from the first convolution modules are saved in the registers. The size of these registers depend on the number of scales, the size of largest kernel and the bit width of the result. These registers can also be switched with Block RAM but it would not be an optimal usage of Block RAM. However it can be decided based on the availability of either of the resource. The kernel coefficients are stored in a ROM. In this case five kernels are required to produce five scales. As a result the pixels for all five scales are computed with a gap of single cycle.

The next step is Difference of Gaussian to produce DoG scale space. Since the adjacent Gaussian scales are subtracted to get the DoG scales the proposed architecture uses a single subtracter for all the scales with one additional register. The result from the DoG module is then saved in the Block RAM based buffers after a serial to parallel conversion module. Once the buffers get full the extrema detection module is activated. In current implementation the extrema detection module only detects the maxima as the candidate keypoints.

The interface to stream cache give rise to one complexity. It can get busy and the architecture has to take account of this busy status. There are two possible case for the occurrence of busy status. one is at the time of reading the data from the stream cache and the other is at the time of writing back the results. The reading and writing interfaces are independent of each other. The interface has separate dedicated busy signals for reading and writing hence the control is distributed. When the stream cache gets busy it asserts the busy signal. Two signals are used in the modules. The data enable signal tells the module when there is valid data on it's input ports. Based on the input data valid an output data valid signal is generated according to the latency of the module. This data valid signal serves as the data enable for the subsequent module in the pipeline. Thus it propagates along the pipeline until the final result is calculated with the final data valid signal. To handle the busy at the writing side the input side is simply paused by not reading further pixels until the writing side gets available. Fairly simple, but due to the propagative nature of valid data in the pipeline, there can be a valid output result that needs to be written in the cache while it is busy resulting in data loss. To handle this case an almost busy signal is used instead (from the stream cache interface). Basically it gets asserted a few cycles earlier than the actual busy which makes sure that the valid result out of the pipeline gets written. This approach avoids the use of extra buffers and complexity required to preserve the outputs when the write side is busy. The limitation however is a loss of constant throughput depending on the behaviour of busy. In general the penalty is negligible and the average throughput still remains the same. This penalty can be avoided by

using an extra FIFO at the output. If we look at the nature of the output, we can see that there will not be extrema detected all the time. Rather there will be few cases when the extrema are detected in the consecutive cycles, coupled with the fact that we are only detecting the maxima. So a relatively small FIFO should suffice, however size of this FIFO depends on the worst case busy waiting time and also on the maximum number of extrema that can be detected during this worst case busy time. This small additional FIFO overhead will ensure that the input throughput gets preserved independent of the status of output resource availability.

3.5.1 Input line buffers

The number of input image lines required at a time is determined by the *kernel size* parameter which is set to the largest Gaussian kernel size needed to produce the most blurred scale. All the kernels are zero extended to make their size constant. The line buffers can be implemented in different ways. In this design Block RAM based FIFO's are used as input line buffers. The FIFO's are generated using the Xilinx Coregen utility.

The Xilinx Virtex-4 BRAM can be utilized in different possible configurations. Each of the Block RAM contains optional address sequencing and control circuitry to operate as a built-in FIFO memory. The FIFO can be 4K deep and 4 bits wide, or 2Kx9, 1Kx18 or 512x36 [15]. In this case the width of image is 640 pixels of 8 bits each. Xilinx Coregen utility provides option to configure the width and depth of FIFO. Hence the depth of FIFO should be at least 640. Thus a possible option is to generate 16 bit wide and 1024 words deep FIFO's by configuring the Block RAM for 1K deep and 18 bits FIFO. However since one pixel only needs 8 bits, the other 10 bits remain free. To optimize this usage two image lines (adjacent rows) are concatenated in one such FIFO. Figure 3.6 shows the basic structure of input line buffer made up of several BRAM based FIFO's. The number of FIFO's required are dependent on the kernel size. The kernel size is always *odd* and since in this design the current pixel is directly fed into the processing block, the number of lines that need to be buffered remain one less than the *kernel size* which is *even* and therefore the number of FIFO's required becomes *half* of the *kernel size less one*. As an example, in case of 13x13 *kernel* 6 BRAM are required for the image line buffering.

The usage of BRAM can be fully optimized in case of 512x512 image with BRAM configuration of 32 bit width and 512 word depth with concatenation. Another possible optimization in case of 640x480 sized image would be to take the image column wise, which means a FIFO depth of 480 word would be sufficient. This will make the BRAM usage *half*, if the chosen kernel size (largest kernel) is the multiple of four, and will require one additional BRAM in the case when it is not the multiple of four. The same 13x13 *kernel* will now require only 3 BRAM's. It becomes more significant as the kernel size increases. For this implementation a FIFO depth of

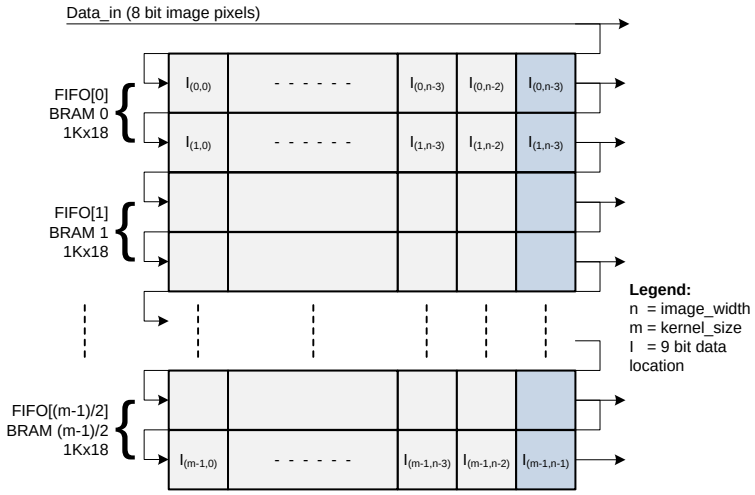


Figure 3.6: Input Line Buffers using Block RAM based FIFO's configure for 1K deep and 18 bits wide

1024 is considered to make possibility for testing larger images. A better solution however would be to have all the possible required FIFO configuration already generated and decide for the one that makes usage of BRAM optimal based on the image size parameters.

3.5.2 Scale Processing

The proposed architecture calculates different scales with the same hardware in a time multiplexed fashion. As mentioned earlier the input pixel is loaded into the line buffers after every s cycles, where s defines the number of Gaussian scales within one octave. At the initial stage when line buffer gets full all the FIFO output lines contain valid data hence the first vertical window starts getting processed. The output of line buffers remain valid until these s clock cycles. In each of these s cycles the pixels are convolved with the respective 1D Gaussian kernels. The result of convolution is buffered for the horizontal convolution.

3.5.3 Convolution in Hardware

The convolution module is composed of four steps. Symmetric property of Gaussian kernel as described in the background chapter is exploited to save number of multiplications. At first the symmetric pixels in the input image column are added while the central pixel is passed as it is. The result is then multiplied with the Gaussian coefficients. This reduces the number of kernel coefficients required to be stored in the ROM to one more than half of the original number of coefficients. The result of multiplication is then summed up using an adder tree. Finally the result is normalized with the sum of the complete 1D kernel. Figure 3.9 illustrate

1D convolution block.

Although the benefits of using symmetric additions are obvious however there is a little trade-off. Addition before multiplication increases the bit-with of the inputs by one which means larger multiplier. Nevertheless when it comes to large kernels as in the proposed architecture several multipliers are saved which justifies the use of symmetric addition well.

Convolution with a large kernel generate big numbers and there addition requires even larger bit-width. Bit-width directly impacts the register resources required between the two 1D convolutions. The next convolution stage which takes this resulting bit with and applies the same process again results in larger bit-width. Normalization averages the results after every convolution and bit growth can be reduced at the expense of little accuracy degradation.

For the case of this implementation integer arithmetic is considered. The double precision Gaussian kernels are generated using MATLAB and are scaled up by a constant factor 1024. The resulting kernel is rounded to get rid of the decimal with some loss of some precision. Figure 3.7 shows the integer approximated kernels as a substitution to the double precision kernels shown in figure 2.10 earlier in section 2.4.1. The simulation results in MATLAB with double precision kernels and integer approximated kernels show that the results are not degraded significantly which is observed by the number of features matched and number of false positives. However there is an obvious gain in terms of hardware utilization and complexity. Figure 3.8 shows an example of matching results with approximated kernels.

The integer approximated kernels are then convolved with the image using separable convolution. After each 1D convolution the result is scaled down by the same constant factor by simple shifting. Figure 3.9 illustrates a 1D convolution block in case of a kernel with 5 coefficients, showing all the operations mentioned in this section.

Since the second 1D convolution requires multiple rows hence the result of first 1D convolution is saved in the buffer until sufficient rows are available for the horizontal convolution.

This buffer comprises of simple shift register connected together as shown in the figure 3.10. The horizontal convolution works in the similar way as the vertical convolution. Figure 3.11 illustrates the design of 2D convolution module for the case of a 5x5 kernel made from two 1D blocks along with an intermediate buffer for a simple case of a single case. While figure 3.12 illustrates the same case (with a 5x5 kernel) showing connection with input line buffers and an intermediate shift register required for preserving the results of for all the scales.

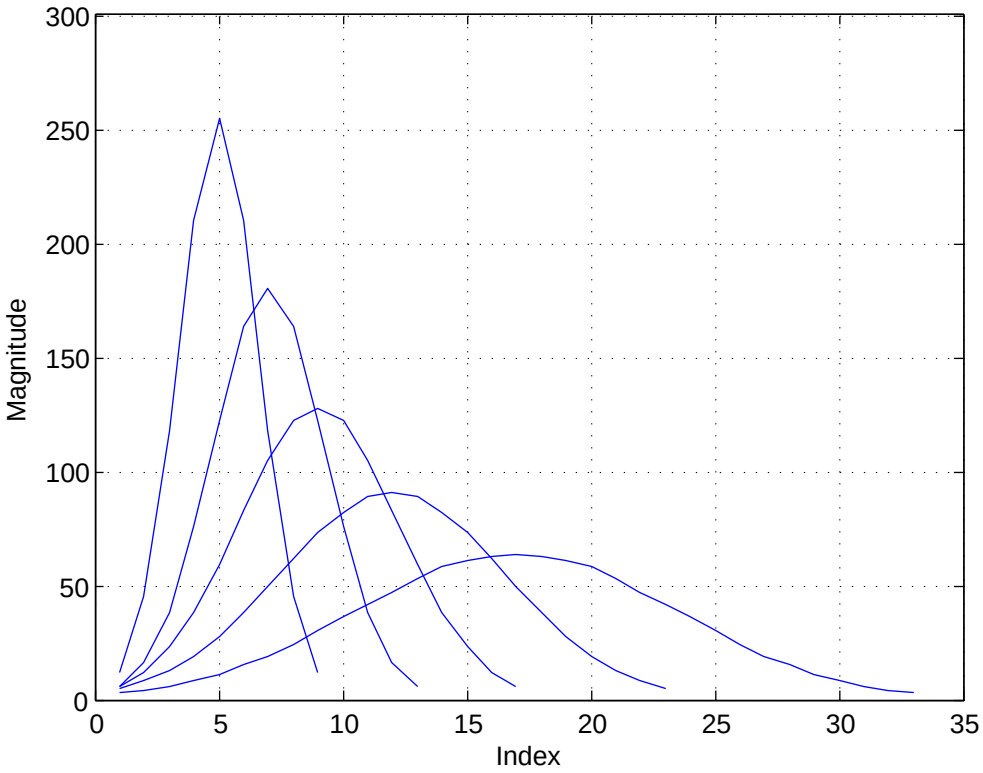


Figure 3.7: Integer approximated Gaussian kernels

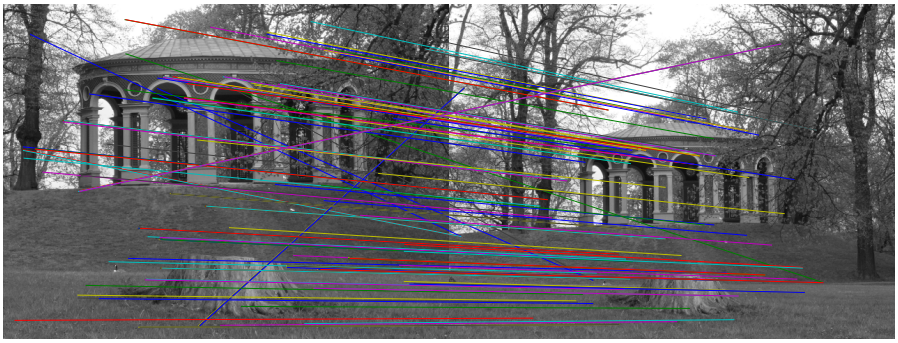


Figure 3.8: Matching results with approximated kernels

For producing all the blurring steps different kernels are convolved with the same input image pixels. The coefficients are saved in a ROM and are given to the convolution block every cycle.

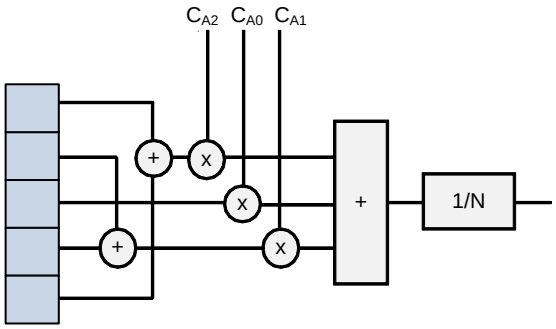


Figure 3.9: 1D Convolution block

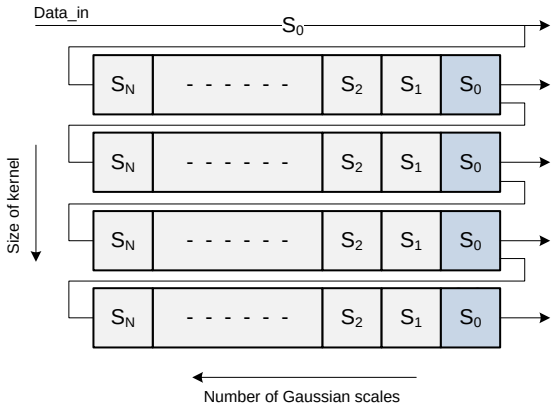


Figure 3.10: Buffer to store the first convolution result

3.5.4 Boundaries

Whenever convolution is performed, boundaries of the image have to be considered. There exist few methods to preserve the boundaries by calculating the boundary pixels based on a certain criteria. The boundaries are effected with the size of the convolution kernel. This implementation discards the boundary pixels by not calculating the pixels at the boundaries. Since the image size in the current implementation is focused for 640x480, neglecting the boundaries of image is not undesirable. This simplifies the hardware. Consequently, the effective area of image where the features are detected decreases depending on the size of the kernel. The size of greatest kernel determines the number of boundary pixels that are neglected. Following analysis will illustrate the effect on the boundary after every step in the algorithm. This is important for the implementation point of view and for the calculation of the exact location of features.

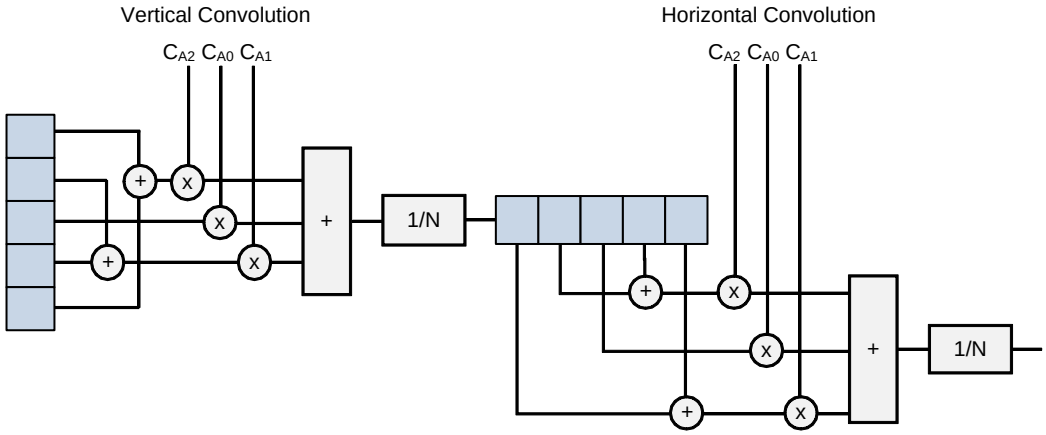


Figure 3.11: 2D Convolution using two 1D blocks

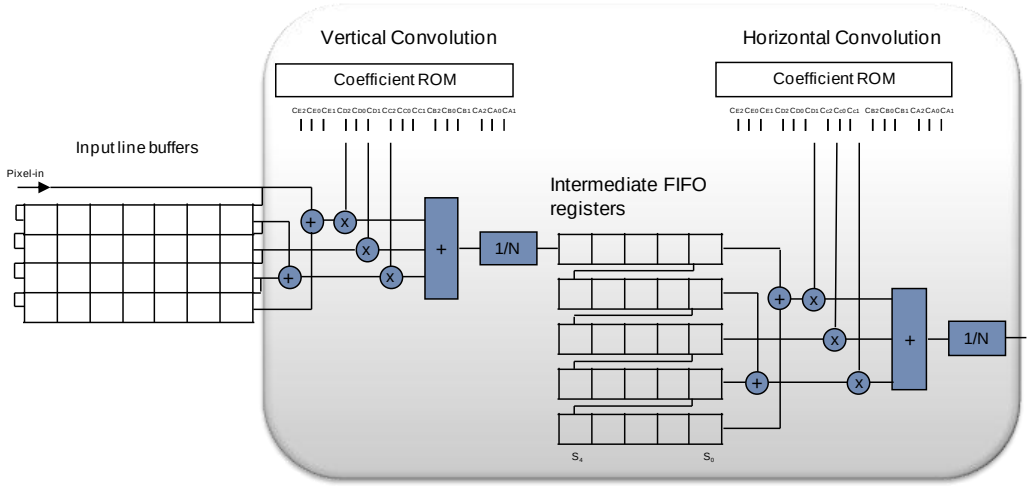


Figure 3.12: Convolution hardware in case of multiple scales

Original image characteristics

Suppose the image characteristics are denoted as follows.
image size - image_size
image width - image_width
image height - image_height
size of the kernel - kernel_size
Column range : 0 to $(image\_width - 1)$
Row range : 0 to $(image\_height - 1)$

Characteristics after first convolution

effective width: image_width (remains the same)
effective height: image_height - (kernel_size-1)

Column range : 0 to $(image\_width - 1)$
Row range : $(kernel\_size - 1)/2$ to $(image\_height - (kernel\_size - 1)/2)$

Characteristics after second convolution

effective width: image_width - (kernel_size-1)
effective height: image_height - (kernel_size-1)

Column range : $(kernel\_size - 1)/2$ to $(image\_width - (kernel\_size - 1)/2))$
Row range : $(kernel\_size - 1)/2$ to $(image\_height - (kernel\_size - 1)/2)$

Characteristics after Extrema Detection

effective width: image_width - (kernel_size-1)
effective height: image_height - (kernel_size-1)-2

Column range : $(kernel\_size - 1)/2 + 1$ to
$(image\_width - (kernel\_size - 1)/2) - 1)$
Row range : $(kernel\_size - 1)/2 + 1$ to $(image\_height - (kernel\_size - 1)/2 - 1)$

3.5.5 Octave processing

This section gives details how the octaves are being handled in the proposed architecture. As mentioned earlier, the octave logic is multiplexed. The only difference in the processing of successive octave is in the image size. After each octave the image is sampled down by taking every next column and every next row. The resultant image gets four times smaller containing 1/4th of the pixels of it's predecessor. The proposed architecture requires the reading of the image data again for the subsequent octaves. The decrease in number of pixels decreases the memory access required for the next octaves. It can be seen that the number of times the image is read from external memory has a direct relation to the number of octaves in this case.

The choice of implementation for the octave processing is dependent on the interface with the external memory and the representation of the image in the memory. We simplify this fact by assuming that all the octave images are stored in the external memory one after the other. This way the octaves are processed seamlessly without requiring extra address or pixel handling. While processing of octaves, there are several factors that are affected due to the change in input image row size. This requires additional control logic. For example the size of line buffers the size of the DoG output buffer gets affected. This is accomplished by changing

the characteristics of the row counter logic based on the current octave number. All the associated read/write controls for the buffers are handled with this additional information.

3.5.6 Difference of Gaussian

The difference of Gaussian is computed by simple subtraction. As explained earlier one subtraction unit in hardware is used for all the DoG calculations. Figure 3.13 shows the architecture of this unit. The output from convolution block gives one result each scale in a round robin fashion. By exploiting this round robin fashion of the results the DoG is calculated by use of a single delay register. The result from the current scale is stored in the register and is subtracted from the next scale in the following cycle. Figure 3.14 illustrates the operation of this module.

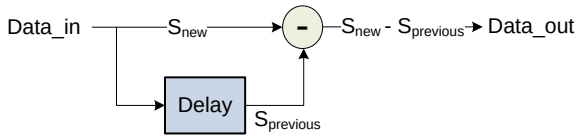


Figure 3.13: Difference of Gaussian unit

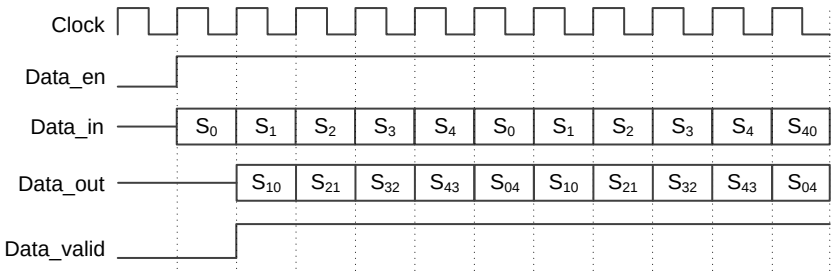


Figure 3.14: DoG module operations

The control signal *Data_in* indicates valid data at the input port of this module, while *Data_valid* signal indicates valid data on the output port. In *Data_out* waveform S_{10} indicate the difference operation $S_1 - S_0$

3.5.7 Serial to Parallel Conversion

The DoG module explained in the previous section generates single result at a time. This data is given out in the parallel from through a simple shift register and control logic. The data from DoG output moves into this shift register based on *Data_en* signal and once all the valid results are present at the output the *Data_valid* is asserted in that cycle. Figure 3.15 shows the architecture of this module while figure 3.16 illustrates the operation of this module.

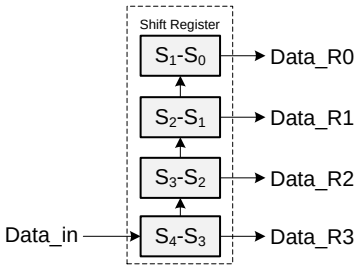


Figure 3.15: Serial In Parallel Out module

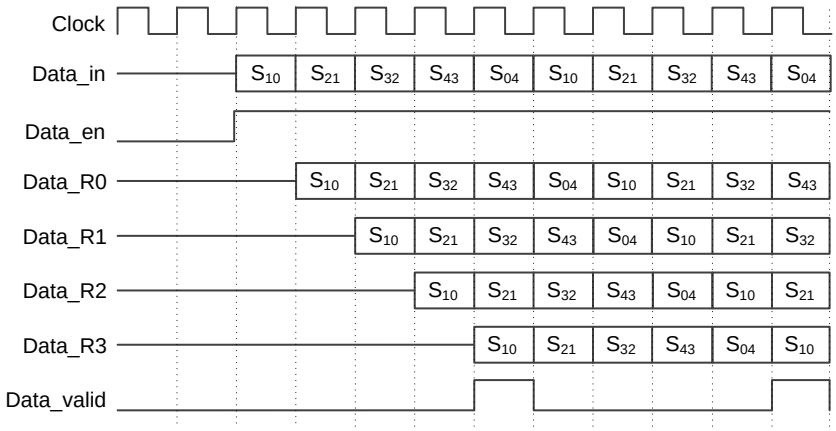


Figure 3.16: Serial In Parallel Out module operation

3.5.8 The DoG buffer

The extrema detection stage requires a 3x3 neighbouring window for detecting Maxima. A FIFO based buffer is needed to store the result of difference of Gaussian and to align the result of pixel according to the new image size after dropping the boundary pixels as explained in the section 3.5.4. The depth of this FIFO depends on the effective image size after the difference of Gaussian stage. The DoG operation does not effect the boundaries as it is a single pixel operation. Hence the size is determined from the output of convolution block. Figure 3.17 depicts how this buffer looks like.

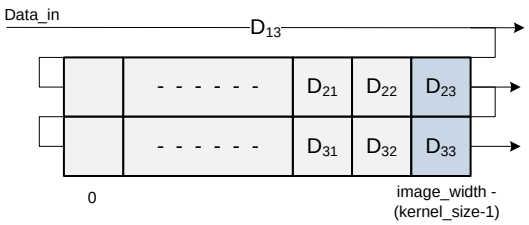


Figure 3.17: A single Difference of Gaussian result buffer

The FIFO is implemented using Block RAM in similar way to the line buffers as explained in section 3.5.1. Notice again that the first row is processed on the fly, without the need of buffers. Hence only two FIFO buffers are needed. Now the question is about the width of these buffers. It depends on the range of the result after the DoG operation. Since this implementation tends to be generic for the evaluation of different bit widths, a conservative approach is used and a larger width is considered. The same FIFO core which is utilized in line buffers is employed for these buffers. However a complete FIFO width is reserved for a single line of the DoG buffer. Therefore each DoG result can have a bit width of up to 16 bits. This however can be optimized down to half (8 bits) in most cases reducing the block RAM requirement to half by concatenation. In [17] only 5 bits are used to store the difference of Gaussian result (without the sign bit). They claim that this reduction in the bit width is a good trade-off. These observations can be considered for further optimizing the current resource utilization mentioned in chapter 5.

Figure 3.18 illustrates the integration of DoG, SIPO and DoG buffer along with the convolution block.

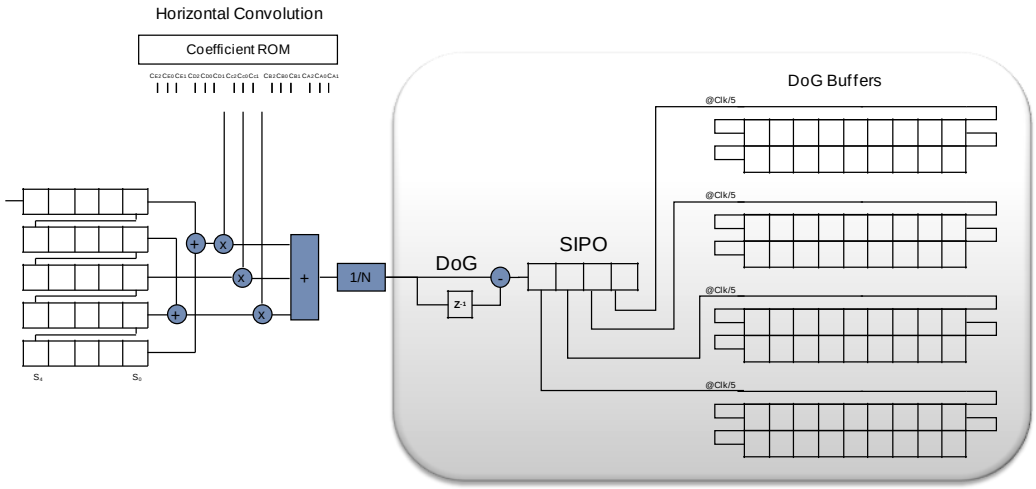


Figure 3.18: DoG, SIPO and DoG buffers

3.5.9 Extrema Detection

The design of maxima detector is pipelined. Figure 3.19 illustrates the design of maxima detector hardware.

Each DoG buffers output consists of three values that constitute one column of a 3x3 neighbouring window required for detection of extrema. After every n clock cycles (where n is the number of Gaussian scales) a new column gets available at

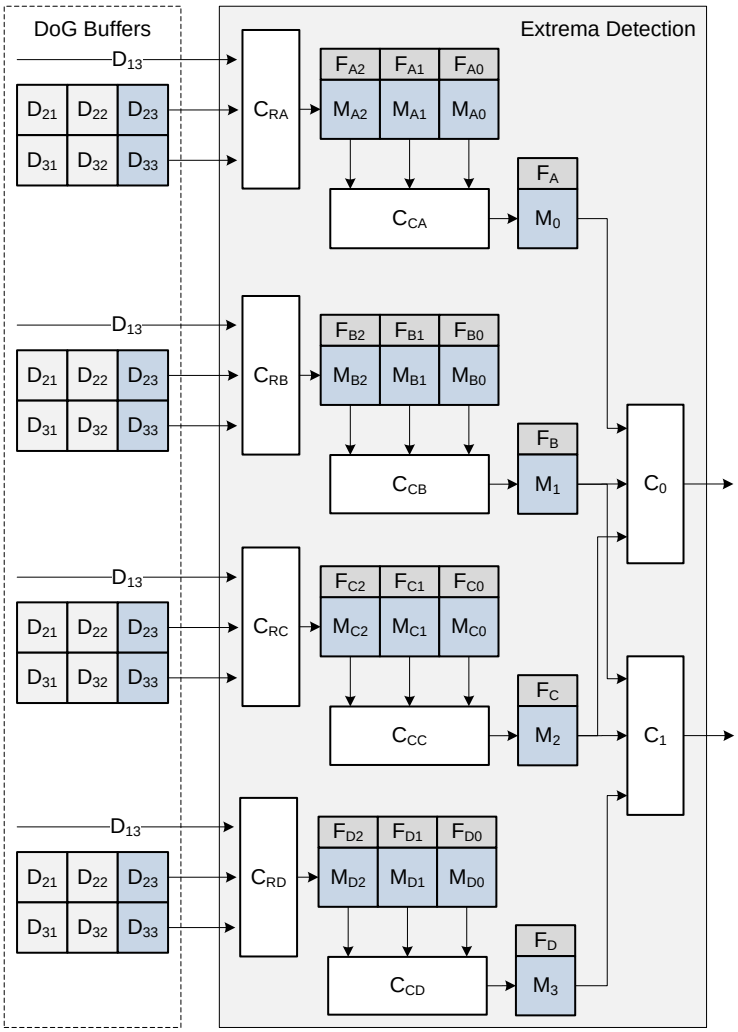


Figure 3.19: Pipelined design of Extrema Detection hardware

the output of DoG buffer. The extrema detection module hence runs on a lower clock i.e. *sys_clk/n*.

The number of comparators required for current design are directly proportional to the number of DoG scales and hence the design parameter *S*.

Extrema Output Format

The extrema detection module outputs the coordinates of detected maxima along with the scale and octave information encoded in a 32-bit word. Figure 3.20 illustrates the output format.

Figure 3.20: Feature Keypoint output format

The x and y pixel coordinates range is determined from the maximum image size. For the image size of 640x480 a minimum of 10-bits are required for the y coordinate and 9-bits for the x coordinate. This combination exceeds 16-bit hence a 32-bit write bus is used. To utilize these 32-bit properly x and y both are made consistent with 10-bit width. The scale coordinate only tells the scale index of the corresponding keypoint coordinates. It is encoded with a maximum of 8-bits (11 Gaussians) each bit representing whether the current coordinate belong to its respective scale or not. This way only one set of coordinate information serves for the keypoints that may be detected in multiple scales simultaneously. The octave is represented using 2-bits (binary encoded) hence can represent a maximum of four octaves. Four octaves are considered to be maximum for an image size of 640x480 as the fifth octave would produce an image which is too small for filtering, besides having lost much of it's information content. The 4 byte feature information is written back to the memory through the stream cache interface. A hardware efficient approach requiring a smaller write bus is proposed in the chapter 6.

Chapter 4

Verification

This chapter gives the details of the approach used to verify the functionality of the implemented hardware.

4.1 Approach

The approach adapted for verification is shown in Figure 4.1.

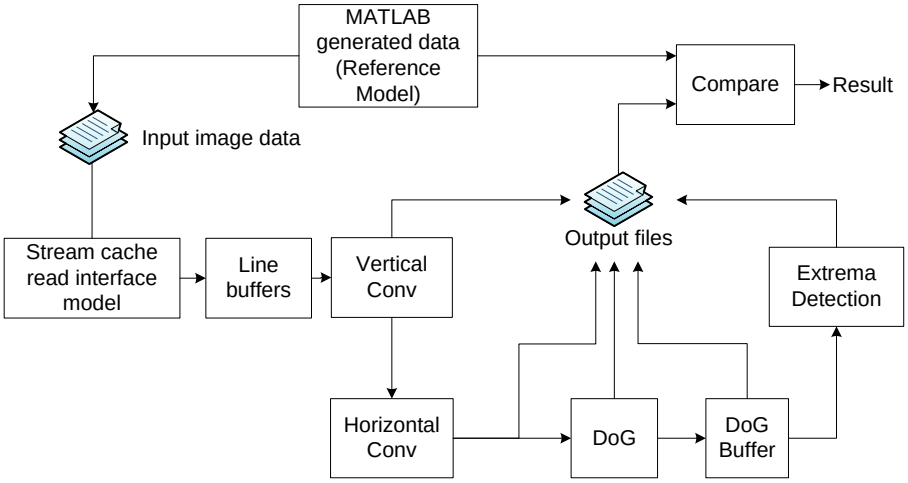


Figure 4.1: Modular verification method

As discussed earlier the algorithm was first implemented in MATLAB (Version 7.11.0.584) as a proof of concept to evaluate the results of the algorithm. This implementation was considerably modified to establish a relation between the hardware modules and the software functions. The verification process is broken down into phases along the data-path. As shown in the figure 4.1 the results of all the parts are saved into text files. Starting from the convolution blocks followed by the DoG and DoG buffer block and finally the extrema detection block. This approach is being

adapted as the hardware consists of a chain of processes pipelined. In MATLAB this is fairly straight forward since all the parts are divided into functions. In VHDL it is accomplished in the Top level interconnect module where all the modules are instantiated. A non-synthesizable code is written inside this module to write the data output from various modules to the files by monitoring the corresponding data valid output signal.

The hardware simulation is performed using Mentor Graphic's ModelSim Simulator (Version 10.0c). Since the target hardware is primarily designed to run on the system clock of 100 MHz, therefore for all simulation purposes a system clock frequency of 100 MHz is considered. A test bench with a stream cache model is made to simulate the design before interfacing to the actual stream cache interface and the system. This model enables standalone testing of the complete hardware. The input image is first converted to a single column data and saved in a text file using MATLAB. This stream of input image pixels is then accessed by the top level test bench module in VHDL. Every time a read request is received from the controller, a new pixel is read from this file and pushed into the data-path. The timing and functionality of each hardware module is verified through simulation waveforms. The comparison of results generated from the MATLAB implementation and the hardware simulation ensures the correctness of the results.

Chapter 5

Evaluation

This chapter give details of the implementation results in terms of timing characteristics and resource utilization. The impact of generic design parameters on resource utilization and timing requirements has been discussed. It also provides a comparison of results of the proposed architecture with known implementations. Furthermore it compares the results of software implementation with that of hardware and finally a discussion is made on the results and observations to highlight the advantages and limitations of the current implementation.

5.1 Timing Characteristics

For the calculation of timing requirements 100 MHz is considered which is the specified frequency of the system where IFD is suppose to be integrated.

$$\begin{aligned}
\text{Clock frequency} &= 100\ \text{MHz}(\text{Period} = 10\text{ns}) \\
\text{Total image processing time} &= \text{Number of pixels} \times \text{Time required for one pixel} \\
\text{Time required for one pixel} &= 5 \times 10\text{ns} = 50\text{ns} \\
\text{Time per frame} &= \text{T}_{O1} + \text{T}_{O2} + \text{T}_{O3} \\
&= (640 \times 480 \times 50\text{ns}) + (320 \times 240 \times 50\text{ns}) + (160 \times 120 \times 50\text{ns}) \\
&= 20.16\text{ms} + \text{Latency}
\end{aligned}$$

The time to process a single image comes out to be less than 21ms, which means that the design can achieve a frame rate of about 48 fps which exceeds the desired 30 fps specification. Hence the IFD can be run on lower frequency e.g. 75 MHz while still meeting the desired real-time(30 fps) requirements with a frame rate of 37 frames per second (time per frame = 26.88ms + Latency).

5.2 Synthesis Results

This section gives an overview of synthesis results for the target FPGA Xilinx Virtex-4 XC4VFX60 (Package: 12ff672, Speed grade: -11) [19]. Xilinx ISE (version

9.2i) has been used for the synthesis of design. An overview of synthesis results for IFD hardware is presented in table 5.1. The input image size of 640x480 is considered with five Gaussian levels and a kernel size of 33. The effect of changing number of octaves is negligible since the resources are time shared among the octaves.

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	3395	25280	13 %
Number of Slice Flip Flops	3015	50560	5 %
Number of 4 input LUTs	5516	50560	10 %
Number of FIFO16/RAMB16s	24	232	10 %
Number of GCLKs	1	32	3 %
Minimum period: 7.581ns (Maximum Frequency: 131.909MHz)			
Minimum input arrival time before clock: 6.201ns			
Maximum output required time after clock: 7.701ns			
Maximum combinational path delay: 5.501ns			

Table 5.1: Overall Hardware Synthesis results

The Maximum operating frequency of hardware (as indicated by synthesis results) comes out to be 131.909 MHz which meets the minimum requirements of 100 MHz.

5.2.1 Effect of S parameter on the Kernel Size and Logic Resources

The size of kernel is effected by the S parameter. It can be observed from figure 5.1 that kernel size gets smaller with increase in number of scales due to the fact that the scaling constant becomes smaller. This shrinking behaviour of the largest kernel decreases the number of coefficients required for Gaussian filtering.

In figure 5.2 the utilization of logic resources is shown for the same range of S parameter. From $S = 2$ to $S = 4$ there is a sharp decrease in kernel size and has its effect on the utilization mainly due to the number of multipliers and adders required. However at the same time the parallel Extrema Detection Logic increases with the increase in scales. Moreover the register requirement for buffering the intermediate results also starts rising again when scales are increased without a significant change in kernel size. This is behaviour can be observed from the graph.

5.2.2 BRAM Utilization and Kernel Size

BRAM utilization is dependent on two types of buffers. The line buffers and the DoG buffers. There are two factors effecting the BRAM requirement. The kernel

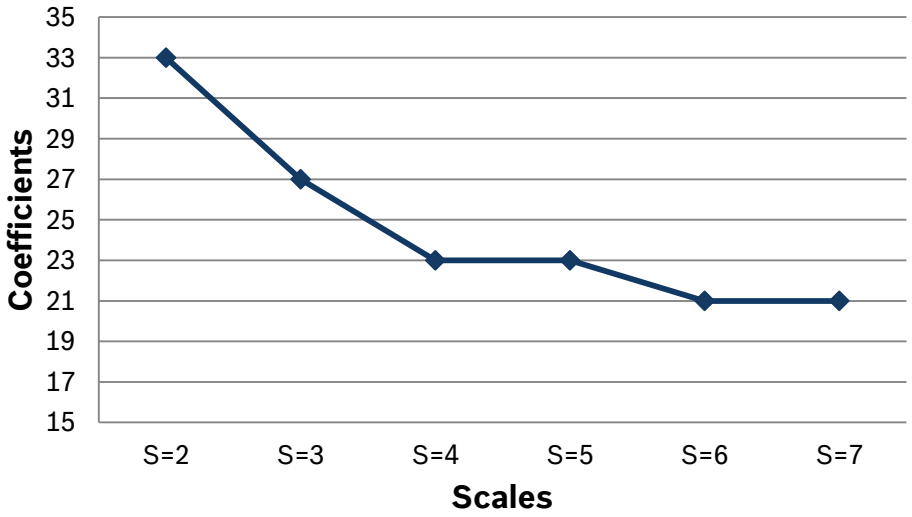
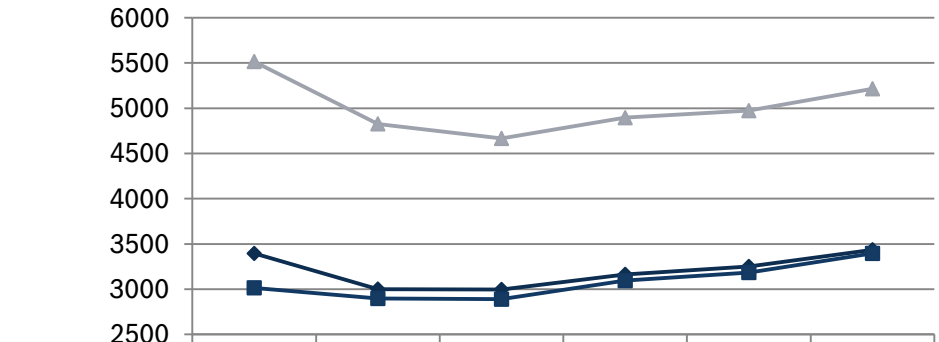


Figure 5.1: Scales vs the kernel size



	S=2	S=3	S=4	S=5	S=6	S=7
Slices	3395	3001	2997	3164	3251	3436
Flip Flops	3015	2897	2889	3094	3184	3395
LUTs	5516	4827	4666	4895	4973	5215

Figure 5.2: Effect of S parameter on Logic resources

size and the number of scales. The number of line buffers required depends on the size of largest kernel while the number of buffers required after the DoG stage directly depend on the S factor. Since each DoG buffer requires two BRAMs the general relation between S and number of DoG buffers becomes $2S$. Where S is the number of DoG scales. BRAM utilization in case of different number of scales is depicted in figure 5.3. At $S = 3$ and $S = 4$ the utilization is optimal, while the variation of BRAMs from minimum S ($S = 2$) S to maximum S ($S = 7$) is only four BRAMs.

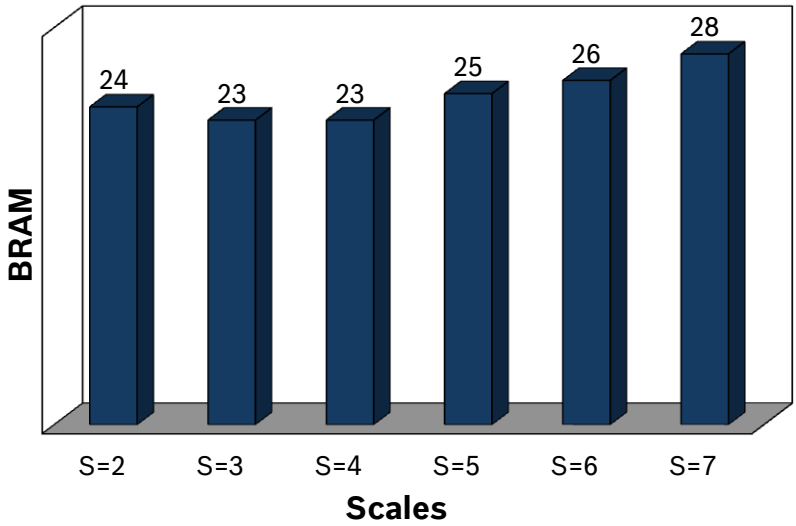


Figure 5.3: BRAM

Figure 5.4 shows the effect of S on kernel size and BRAM in the same graph. The difference in BRAM is not so significant here. Therefore the optimal choice would be at $S = 4$ where there is lowest utilization of resources combined (also considering the results in figure 5.2).

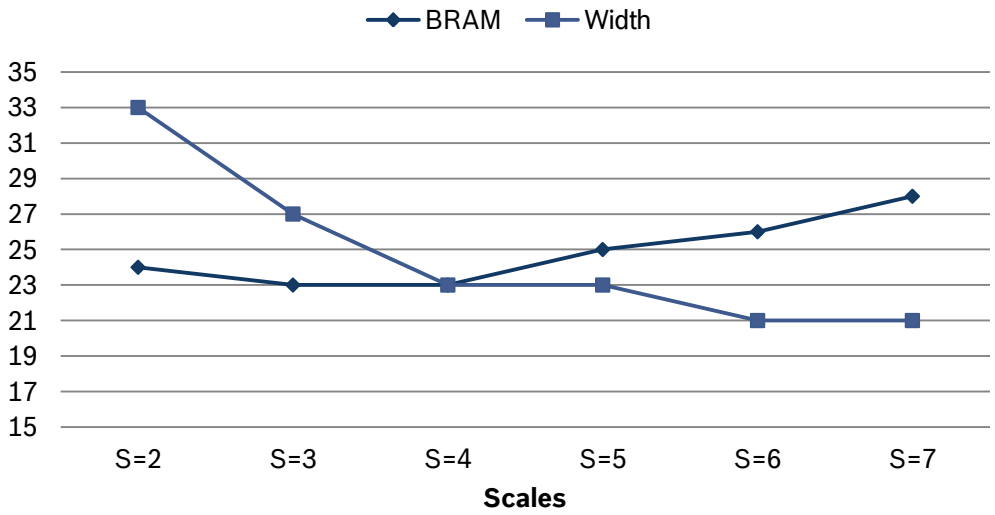


Figure 5.4: Kernel size and BRAM usage

5.2.3 Comparison with other implementations

In this section the results (in terms of required resources) of implementation are compared with two existing implementations. In [18] the authors have made a comparison of resources utilized by their implementation with the resources taken by that of Bonata et al. system [17]. The same table of comparison is taken and extended with the results of proposed architecture in table 5.2.

Resources	**Proposed** O = 3, S = 6, k = 27	**DoG part of [17]** O = 4, S = 6, k = 7	**[18]** O = 3, S = 6, k = 7
Slices	3001	5068	-
Flip-flops	2897	6028	7256
Look Up Tables	4827s	6880	15137
Block RAM	23 (0.4 Mb)	120 (2.1 Mb)	0.91 Mb

Table 5.2: Comparison with other Implementations

The system in Chang et al. [18] targets 320x240 sized images. The implementation is done on a Xilinx Virtex II Pro FPGA (XC2VP30-5FF1152) [20]. They have used Xilinx System Generator [21] and MathWorks Simulink [22] for the implementation. They claim that there system uses a clock rate of 50 MHz, taking 3 ms to detect scale-space extrema in an image and is thus able to process 330 frames per second.

In [17] a complete implementation of SIFT algorithm has been demonstrated which requires 33 ms for a 320x240 image. The hardware blocks are implemented in Handel-C [23] and development platform used is based on Altera Stratix II 2 S60 FPGA [24].

The proposed architecture considers the image size of 640x480 pixels in contrast to the image size of 320x240 pixels which is considered by both the implementations that are compared in table 5.2. The image size of 320x240 pixels will require nearly half block RAM resources with the proposed architecture. Moreover, it can process a 320x240 image in 5.1 ms or in other words up to 196 frames per second.

5.3 Detector Performance

Figure 5.5 illustrates the implementation results similar to that presented in chapter 2 (software simulation results).

The number of matches from software simulation and hardware implementation are compared in table 5.3. For the first two scenes that belong to viewpoint

category the number of matches have dropped by 23.6 percent and 34.2 percent respectively, while for the other two images with zoom and rotation transformation it has increased to 13.9 percent and 22.8 percent. It can be observed that the number of false positive matches have also increased slightly due to the error introduced by approximations. Tests have shown that number of false positives decrease by slightly increasing the distance threshold in the matching process at the expense of comparatively less overall matches.

Scene	**Matches**	
	Simulation	Implementation
graffiti (640 x 480)	487	372
ubc (640 x 480)	627	412
bip (640 x 480)	322	374
inria (640 x 480)	135	175

Table 5.3: Comparison of results from simulation and hardware implementation

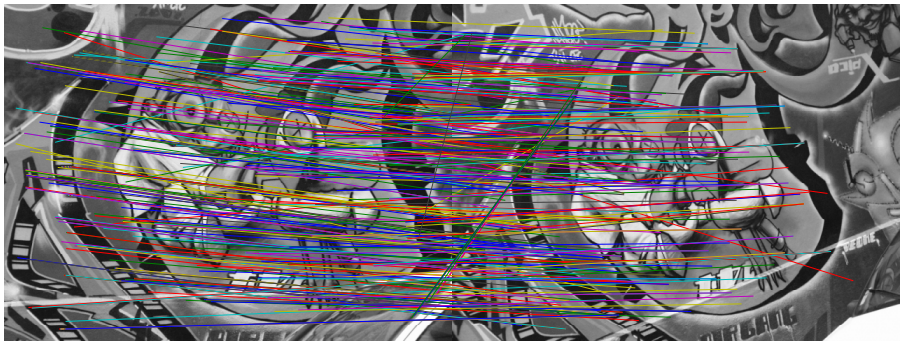
5.4 Discussion on Results

The synthesis results show that the proposed architecture require less resources and achieves the real-time (30 fps) processing requirements. Speed can be further increased by using higher frequency as the system is able to achieve maximum frequency of 131.909MHz. Furthermore it is also possible to increase the frequency beyond 131.909MHz by introducing more pipeline stages at the expense of more resources however increasing the frequency directly impact the power requirements. In applications that require lower frame rate a lower clock can be used to get more efficiency in terms of power and resources.

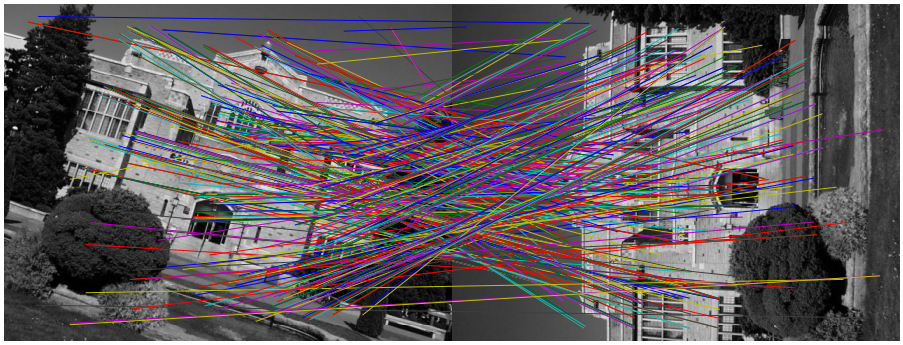
In section 5.2.2 it is shown that S=4 is a good choice, however there exists another dimension, the number of detected keypoints, which is directly proportional to S. At S=4 the number of detected keypoints will increase significantly. Hence there will be a trade off in selection of the parameters. It also suggests for a refinement of the detected keypoint to reject those which are unstable. This would not only reduce the number of detected keypoints but also increase the repeatability rate and hence the reliability of detected features.

The result show that the performance of detector is slightly degraded due to hardware optimizations. However there is a significant gain in terms of hardware resource utilization as a result. The performance can be increased with the expense of greater hardware resources depending on the application requirement. Moreover it has been noted that by increases the distance threshold in descriptor match-

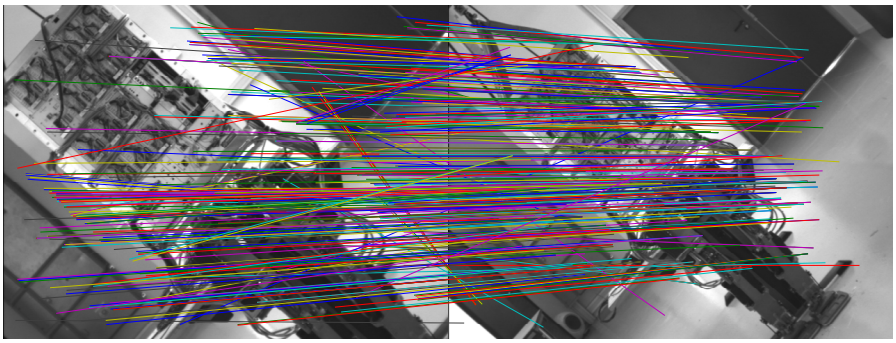
ing process the number of false positives are reduced with less number of overall matches.



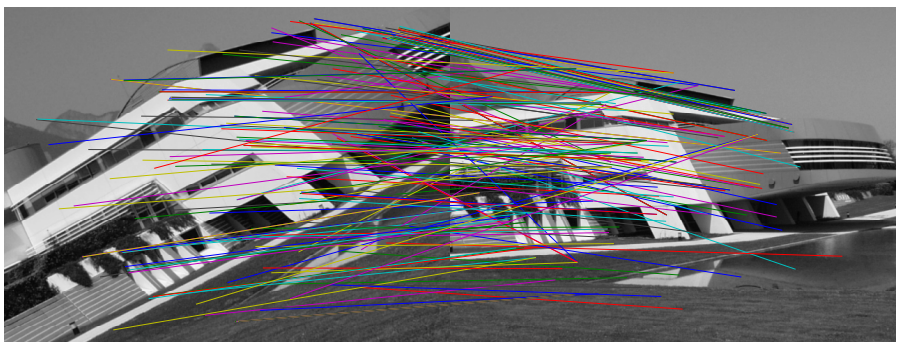
(a) Matching results on graffiti images taken from different viewpoints



(b) Matching results for images taken from different viewpoints



(c) Matching results for images with different Zoom levels



(d) Matching results for images with Rotation and Zoom variation

Figure 5.5: Implementation Matching results

Chapter 6

Conclusion

A pipelined hardware architecture is proposed for scale-space extrema detection part of Scale Invariant Feature Transform (SIFT). The proposed architecture is implemented on a Xilinx Virtex 4 FPGA. Preliminary results are presented which show the capability of supporting real-time processing on an image size of up to 640 x 480. Scale space extrema detection is first simulated in software and is evaluated using a hybrid approach.

The results presented in chapter 5 meets and exceeds the performance specifications of the desired application. It shows relatively low resource utilization that can be achieved through an inexpensive FPGA. By carefully analysing the desired performance and level of parallelism required it is shown that the FPGA based solution of a feature detection algorithm based on SIFT algorithm is capable of real-time performance with highly effective resource utilization. Moreover it is also observed from the results that the proposed architecture is scalable in terms of design changes with slight overhead in the resource requirements while still meeting the real-time constraints. Optimal values of design parameters for an efficient hardware utilization are also discussed based on the synthesis results using different values of design parameters. Nevertheless the proposed hardware architecture shows encouraging results, there are a few limitations that can be overcome with a cost of more time. The future work section highlights some of the possible solutions and gives a direction for further work in this area.

6.1 Future Work

During the course of this thesis many potential ideas came up for possible improvements that could result in a better solution. Not all of them were considered during the implementation due to limited amount of time. This section is dedicated to the discussion of limitations in the current architecture, possible solutions to overcome these limitations and directions for future work.

6.1.1 Reducing stream cache modules

The current implementation utilizes two stream cache modules. One for reading the input image pixels and the other for writing the output. Since the current design of stream cache interface requires that the read and write side bus width be the same, this limits the design to have different read bus width and write bus width configuration. There are two solutions to this limitation.

Reading multiple pixels from the cache

By reading a 32-bit word i.e. four pixels at a time would make both bus widths common and thus only one stream cache module is required for both read and write since both interfaces are mutually exclusive. A parallel to serial converter will be required to interface the 32-bit stream cache interface to 8-bit input line buffers.

Decrease the output bus width

The current scheme of output requires 32-bit wide bus. If the output can be decoded in less number of bits it will allow the use of a smaller bus. Section 6.1.3 proposes a scheme to reduce the bus width to 16-bits. If the proposed scheme is considered a 16-bit read bus will be required with a parallel to serial converter.

6.1.2 Resource Efficient Extrema Detector

The current extrema detection hardware module uses dedicated set of comparators for every DoG scale. So when the scales are increased comparators increase accordingly. Since this module runs at lower clock in a pipelined fashion, there could be a way to reuse one set of comparators for several scales by using the normal system clock with additional control logic and multiplexing to save the number of comparators required.

6.1.3 Efficient output feature format

Section 3.5.9 explains the output format in detail. Here a new scheme is proposed in order to decrease the bus width required. In case of a 640x480 image there are 640 columns and 480 rows. The column information requires 10 bits. For the case of row a single bit can be used for indicating the start of a new row by inverting this bit every time the new row starts. In this way 8 bits are preserved at the cost of little overhead at the decoding side which has to maintain a simple 9-bit counter by monitoring the row bit. With this scheme 11 bits are utilized for the coordinate information. From the rest 5 bits, 3 bits (binary encoded to allow maximum of 8 scales or 11 Gaussians) are used for the scale coordinate while 2-bit (binary encoded) are used to represent octave index (maximum 4 octaves). A 16-bit bus interface will be required in this case which is half compared to the current scheme.

6.1.4 Minimization of BRAM usage

Although the proposed architecture shows efficient BRAM utilization, it can be further optimized by dividing the image into smaller parts and utilizing a larger width and smaller length BRAM. The effect of partitioning the image have to be investigated. An important consideration would be the boundaries of the partitions that lie inside the boundaries of the image. Considering inner boundaries might become necessary unlike current design where the boundaries have been neglected. One approach could be to overlap the partitions sufficiently in order to avoid the boundary effect, however such procedure will require additional control in hardware. Furthermore, to make the architecture generic for the best utilization of BRAM with respect to the different image sizes the image size generics should be used to determine the optimal BRAM configurations.

6.1.5 Use of Dedicated Multipliers from DSP Slices

The Virtex-4 FPGA contain cascadable embedded XtremeDSP Slices with dedicated multipliers, integrated Adders and accumulators. These blocks when utilized intelligently can increase efficiency and speed of the system while replacing a significant amount of logic resources needed for large multipliers. They can be a good alternative when the FPGA targeted for a specific application contain these resources that are otherwise free.

6.1.6 Extended Evaluation and Reliability testing on a larger database

Evaluation of a feature detector is of great importance. This work evaluates the results by applying them on a matching problem. There exists methods for evaluation of feature detectors and a possible future work could be to make a standard evaluation procedure for evaluating the architecture and tuning it to the best parameters based on the evaluation results by showing that it satisfies the criteria well enough for the desired application. Chapter 2 briefly describes the evaluation method as a food for thought.

6.1.7 Evaluation of the effect of bit-width on the hardware resources vs the accuracy

The accuracy of a feature detector degrades in an attempt to decrease the required hardware resources. A thorough analysis of bit-width would be very useful in order to make the best trade-off between logic utilization and accuracy demanded by the application under consideration. The effect of bit-width on logic utilization can be easily observed by changing the parameters in this implementation. There is more work required on measuring the accuracy of detector under different conditions and parameters.

Appendix A

Appendix

A.1 Functions written in MATLAB

```
function [features] = IFD2_algo(im, sigma0, S, SD_criteria, sf
    )

k = 2^(1/S)

num_of_kernels = S+3

gauss_kernels = gen_kernels(sigma0, S, SD_criteria, sf)

for i = 1 : num_of_kernels
    [v_smooth(:,:,i) im_smooth(:,:,i)] = ifd_conv1d(im,
       gauss_kernels(i,:), sf);
end

threshold = 0;
scale = sigma0*(k.^(linspace(1,S,S)));
[features] = extrema_detect(DoG, S, scale, threshold)

disp('Number of Extrema:');
disp(length(features));
```

Listing A.1: Main Algorithm function in MATLAB

```
function [gauss_kernels_mat] = gen_kernels(sigma0, S,
   SD_criteria, sf)

k = 2^(1/S)
num_of_kernels = S+3
```

```
sigma = zeros(1,num_of_kernels);
sigma(1) = sigma0;

for i = 1 : length(sigma)-1
    sigma(i+1) = sigma(i) * k
end

for i = 1 : length(sigma)
    width = ceil(sigma(i)*SD_criteria);
    if mod(width,2) == 0
        k_size(i) = width + 1
    else
        k_size(i) = width
    end
end

for i = 1 : num_of_kernels
    gauss_kernels{i} = ifd_gaussian(sigma(i),k_size(i),sf)
end

gauss_kernels_mat = zeros(num_of_kernels, max(k_size))

for i = 1 : num_of_kernels
    gauss_kernels_mat(i, (((max(k_size)+1)/2) - (k_size(i)-1)
       /2) : (((max(k_size)+1)/2) + (k_size(i)-1)/2) ) =
       gauss_kernels{i}
end

plot(gauss_kernels{1})
hold on
for i = 2 : num_of_kernels
    plot(gauss_kernels{i})
end
hold off
grid on
xlabel('Index');
ylabel('Magnitude');
```

Listing A.2: Kernel Generation function in MATLAB

```
function [gaus_kernel] = ifd_gaussian(sigma,k_size,sf)

indp = 1 : (k_size-1)/2;
ind = cat(2,fliplr(indp),0,indp);
gaus_kernel = (1/(sigma*sqrt(2*pi)))*exp(-((ind.^2)/(2*sigma
    ^2)));

gaus_kernel = gaus_kernel/sum(gaus_kernel); % Normalize
```

```
if sf > 1
    gaus_kernel = round(gaus_kernel .* sf);
end
```

Listing A.3: Gaussian Generation function in MATLAB

```
function [v_smooth im_smooth] = ifd_conv1d(im, kernel_1D, sf)

if(ndims(im) > 2)
  error('im must be a two dimensional matrix') ;
end

k_size=size(kernel_1D);
if(k_size(1) > 1)
  error('Kernel must be single dimensional row vector') ;
end

if(mod(k_size(2),2) == 0)
  error('Kernel must have odd number of indexes') ;
end

[r,c]=size(im);
kernel_size=length(kernel_1D);
kernel_ind_ofs=(kernel_size-1)/2;

sum_kernel_1D = sum(kernel_1D);

v_smooth=zeros(r,c);

% Vertical convolution
for i=1:c
    for j=1:r-kernel_size+1
        v_smooth(j+kernel_ind_ofs,i) = sum(im(j:j+kernel_size
            -1,i).* kernel_1D');
    end
end
if sf > 1
    v_smooth = round(v_smooth./sf);
end
im_smooth=zeros(r,c);

% Horizontal convolution
for i=1:r
    for j=1:c-kernel_size+1
        im_smooth(i,j+kernel_ind_ofs) = sum(v_smooth(i,j:j+
            kernel_size-1).*kernel_1D);
    end
```

```
end
if sf > 1
    im_smooth = round(im_smooth./sf);
end
```

Listing A.4: Separable Convolution in MATLAB

```
function [features] = extrema_detect(DoG, S, scale, threshold)

[nrows,ncols] = size(DoG(:,:,1));

index = 1;
num_of_min_key = 0;
num_of_max_key = 0;
flag_min = 0;
flag_max = 0;

for i = 2:S+1
  for r = 1 : nrows-2;
    for c = 1 : ncols-2;
      flag = 0;
      if (DoG(r+1,c+1,i) > (DoG(r:r+2, c:c+2,i-1)+threshold))
        if (DoG(r+1,c+1,i) > (DoG(r:r+2, c:c+2,i+1)+threshold)
            )
          flag = 1;
          for l = 0 : 2
            for m = 0 : 2
              if (l ~= 1)&&(m ~= 1)
                if (DoG(r+1,c+1,i) > (DoG(r+l, c+m, i)+
                    threshold))
                  flag_max = 1;
                else
                  flag = 0;
                  flag_max = 0;
                  break;
                end
              end
            end
          end
        end
      end
      if(flag == 1)
        if(flag_max==1)
          num_of_max_key=num_of_max_key+1;
        else
          error('extrema error');
        end
        key_im(r+1, c+1) = 255;
```

```
            key_loc(1:2, index) = [(c+1) (r+1)];
            key_loc(3, index) = scale(i-1);
            index = index + 1;
          end
        end
      end
      figure;
      imshow(key_im);
end
features = key_loc;

disp('Number of Maxima:');
disp(num_of_max_key);
```

Listing A.5: Extrema Detection function in MATLAB

A.2 VHDL Entity Declarations

```
ENTITY IFD2_Main IS
     GENERIC (
       G_APPL_AWIDTH   : natural := 32;
       G_APPL_DWIDTH   : natural := 32;
       G_SYSBUS_AWIDTH : natural := 32
       );
  PORT (
    sys_clk                       : IN std_logic;
    reset_n                       : IN std_logic;
    P_ctrl_start_sl               : IN  std_logic;
    P_ctrl_imgInAbsStartAddr_slv  : IN  std_logic_vector(
       G_SYSBUS_AWIDTH-1 DOWNTO 0);
    P_ctrl_imgOutAbsStartAddr_slv : IN  std_logic_vector(
       G_SYSBUS_AWIDTH-1 DOWNTO 0);
    P_ctrl_ready_sl               : OUT std_logic;
    P_WR_Start_Addr_slv           : OUT std_logic_vector(
       G_SYSBUS_AWIDTH-1 DOWNTO 0);
    P_WR_Init_sl                  : OUT std_logic;
    P_WR_Almost_Full_sl           : IN  std_logic;
    P_WR_Flush_sl                 : OUT std_logic;
    P_WR_Data_slv                 : OUT std_logic_vector(
       G_APPL_DWIDTH-1 DOWNTO 0);}
    P_WR_we_sl                    : OUT std_logic;
    P_RD_Start_Addr_slv           : OUT std_logic_vector(
       G_SYSBUS_AWIDTH-1 DOWNTO 0);
    P_RD_Init_sl                  : OUT std_logic;
    P_RD_Busy_sl                  : IN  std_logic;
    P_RD_Data_slv                 : IN  std_logic_vector(
       G_APPL_READ_DWIDTH-1 DOWNTO 0);
```

```
    P_RD_re_sl                      : OUT std_logic
    );
END IFD2_Main;
```

Listing A.6: VHDL Top module for the IFD Hardware

```
entity IFD2_Ctrl is
    port  (
      clk : IN std_logic;
      rst : IN std_logic;
      P_ctrl_start_sl                : IN  std_logic;
      P_ctrl_imgInAbsStartAddr_slv   : IN  std_logic_vector(
         G_SYSBUS_AWIDTH-1 DOWNTO 0);
      P_ctrl_imgOutAbsStartAddr_slv  : IN  std_logic_vector(
         G_SYSBUS_AWIDTH-1 DOWNTO 0);
      P_ctrl_ready_sl                : OUT std_logic;
      P_WR_Start_Addr_slv            : OUT std_logic_vector(
         G_SYSBUS_AWIDTH-1 DOWNTO 0);
      P_WR_Init_sl                   : OUT std_logic;
      P_WR_Almost_Full_sl            : IN  std_logic;
      P_WR_flush_sl                  : OUT std_logic;
      P_WR_we_sl                     : OUT std_logic;
      P_RD_Start_Addr_slv            : OUT std_logic_vector(
         G_SYSBUS_AWIDTH-1 DOWNTO 0);
      P_RD_Init_sl                   : OUT std_logic;
      P_RD_Busy_sl                   : IN  std_logic;
      P_RD_re_sl                     : OUT std_logic;
      P_Data_en_sl                   : OUT std_logic;
      P_Data_out_Rd_en_sl            : OUT std_logic;
      P_Line_buff_wr_en_sl           : OUT std_logic;
      P_Line_buff_rd_en_sl           : OUT std_logic;
      P_norm_coef1_valid_sl          : IN  std_logic;
      P_norm_coef2_valid_sl          : IN  std_logic;
      P_Data_Valid_HCBuff_sl         : IN  std_logic;
      P_Data_Valid_ExD_sl            : IN std_logic;
      P_Addr_Conv_Coef1              : OUT INTEGER range 0 to
         ram_depth-1;
      P_Addr_Norm_Coef1              : OUT INTEGER range 0 to
         num_of_kernels-1;
      P_Addr_Conv_Coef2              : OUT INTEGER range 0 to
         ram_depth-1;
      P_Addr_Norm_Coef2              : OUT INTEGER range 0 to
         num_of_kernels-1
      );
end IFD2_Ctrl;
```

Listing A.7: IFD Controller module

```
entity IFD2_Conv_1D is
  port (
  clk             : IN  std_logic;
  din_C1          : IN  array_IN_A;
  data_conv_coef  : IN  std_logic_vector (rom_width-1 downto
     0);
  data_norm_coef  : IN  std_logic_vector (norm_coe_bit_width-1
      downto 0);
  rst             : IN  std_logic;
  data_en         : IN  std_logic;
  norm_coef_valid : OUT std_logic;
  data_valid      : OUT std_logic;
  dout_C1         : OUT std_logic_vector(R3_bit_width_A +
     norm_coe_bit_width-1 downto 0)
  );
end IFD2_Conv_1D;
```

Listing A.8: A single 1D Convolution module

```
entity IFD2_DoG is
port (
  clk             : IN  std_logic;
  rst             : IN  std_logic;
  din_DoG         : IN  std_logic_vector(R3_bit_width_B +
     norm_coe_bit_width-1 downto 0);
  data_en         : IN  std_logic;
  data_valid      : OUT std_logic;
  buff_rd_en      : OUT std_logic;
  DoG_buff_full   : OUT std_logic;
  dout_DoG        : OUT array_DoG_out);
end IFD2_DoG;
```

Listing A.9: Difference of Gaussian module

```
entity IFD2_Extrema_Detect is
  port (
  clk         : IN  std_logic;
  rst         : IN  std_logic;
  din         : IN  array_all_DoG_buff;
  data_en     : IN  std_logic;
  data_valid  : OUT std_logic;
  x_dout      : OUT std_logic_vector(locat_out_bit_width-1
     downto 0);
  y_dout      : OUT std_logic_vector(locat_out_bit_width-1
     downto 0);
  s_dout      : OUT std_logic_vector(scale_out_bit_width-1
     downto 0)
```

```
  );
end IFD2_Extrema_Detect;
```

Listing A.10: Extrema Detection module

```
entity IFD2_Extrema_Detect is
  port (
  clk          : IN  std_logic;
  rst          : IN  std_logic;
  din          : IN  array_all_DoG_buff;
  data_en      : IN  std_logic;
  data_valid   : OUT std_logic;
  x_dout       : OUT std_logic_vector(locat_out_bit_width-1
     downto 0);
  y_dout       : OUT std_logic_vector(locat_out_bit_width-1
     downto 0);
  s_dout       : OUT std_logic_vector(scale_out_bit_width-1
     downto 0)
  );
end IFD2_Extrema_Detect;
```

Listing A.11: Extrema Detection module

Bibliography

[1] David G. Lowe, *Object recognition from local scale-invariant features*, Proc. 7th International Conference on Computer Vision (ICCV'99)(Corfu, Greece): 1150-1157 (1999).

[2] David G. Lowe, *Distinctive image features from scale-invariant keypoints*, International Journal of Computer Vision, 60(2), 91-110 (2004).

[3] T. Tuytelaars, K. Mikolajczyk, *Local Invariant Feature Detectors: A Survey*, Foundations and Trends in Computer Graphics and Vision, Vol. 3, No. 3, 177-280 (2007).

[4] Koenderink J.J., *The structure of images*, Biol. Cybern. 50, 363-370 (1984).

[5] Lindeberg T., *Feature detection with automatic scale selection*, International Journal of Computer Vision, 30, 79-116 (1988).

[6] Lindeberg T., *Scale-space theory: A basic tool for analysing structures at different scales*, Journal of Applied Statistics, 21(2):225-270 (1994).

[7] A. Vedaldi, *An open implementation of SIFT detector and descriptor*, UCLA CSD, Tech. Rep. 070012, (2007).

[8] A. Vedaldi, B. Fulkerson, *VLFeat: An Open and Portable Library of Computer Vision Algorithms*, vlfeat.org, (2008).

[9] Feature Detector Evaluation Sequences, `http://lear.inrialpes.fr/people/mikolajczyk/Database/index.html`

[10] C. Schmid, R. Mohr, C. Bauckhage, *Evaluation of Interest Point Detectors*, In Proceedings of the International Conference on Computer Vision, 230-235, Bombay, (1998).

[11] Brown, M.; Lowe, D.G, *Invariant features from interest points groups*, In Proceedings of British Machine Vision Conference, Cardiff, Wales, 656-665 (2002).

[12] Y . Ke, R. Sukthankar, *PCA-SIFT: A more distinctive representation for local image descriptors*, In Proceedings of the Conference on Computer Vision and Pattern Recognition, Washington, USA, 511-517, (2004).

[13] Mikolajczyk, K.; Schmid, C. *A performance evaluation of local descriptors*, Pattern Analysis and Machine Intelligence, IEEE Transactions on , 27(10), 1615-1630 (2005)

[14] Xilinx Virtex-4 Family Overview, `http://www.xilinx.com/support/documentation/user_guides/ug070.pdf`

[15] Xilinx Virtex-4 FPGA User Guide, `http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf`

[16] Kosuke Mizuno, Hiroki Noguchi, Guangji He, Yosuke Terachi, Tetsuya Kamino, *A Low-Power Real-Time SIFT Descriptor Generation Engine for Full-HDTV Video*, IEICE Transactions, (2011).

[17] Vanderlei Bonato, Eduardo Marques, and George A., *A Parallel Hardware Architecture for Scale and Rotation Invariant Feature Detection*, IEEE Transaction on Circuits and Systems for Video Technology, 18(12), 1703-1712 (2008).

[18] L. Chang and J. Hernández-Palancar, *A Hardware Architecture for SIFT Candidate Keypoints Detection*, In CIARP, pp.95-102, (2009).

[19] Virtex-4 FPGA Data Sheet: DC and Switching Characteristics, `http://www.xilinx.com/support/documentation/data_sheets/ds302.pdf`

[20] Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, `http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf`

[21] Xilinx System Generator for DSP, `http://www.xilinx.com/tools/sysgen.htm`

[22] MathWorks Simulink, `http://www.mathworks.se/products/simulink/`

[23] Handel-C Language Reference Manual Celoxica, `http://www.celoxica.com`

[24] *Stratix II Device Handbook*, San Jose, CA: Altera, (2007).

TRITA-ICT-EX-2012:183

www.kth.se