MASTER

Accelerating SIFT feature extraction with a vector DSP
a feasibility study

Sathyanarayana Prasad, A.

Award date:
2014

Technische Universiteit Eindhoven University of Technology

Department of Mathematics and Computer Science
The System Architecture and Networking group

Accelerating SIFT Feature extraction with a vector DSP- A feasibility study

Master Thesis

Arjun Sathyanarayana Prasad

ERICSSON

Supervisors:
Prof. Dr. Ir. C. H. (Kees) Van Berkel
Ir. D. (David) Van Kampen
Dr. Ir. W. E. H. (Wim) Kloosterhuis
Prof. Dr. K. G. W. (Kees) Goossens

This thesis work has been carried out at Ericsson. The content of this report is confidential until August 29, 2015

Acknowledgments

I would like to express my deepest gratitude to Professor Dr. Ir. Kees Van Berkel from TU, Eindhoven for providing me an opportunity to conduct this thesis work under him. He has been motivating and has guided me to carry out this thesis work in a fruitful manner.

I am extremely grateful to David van Kampen from Ericsson for tutoring me throughout this thesis work. He has given me valuable suggestions and feedback in each and every stage of my work. His willingness to give time so generously is immensely appreciated.

Special thanks to Dr. Ir. Wim Kloosterhuis from Ericsson for advising me in the critical stages of the thesis work and providing me the constructive feedback. He has made time to review my work even with a busy schedule which is much appreciated.

Lastly, I would also like to extend my thanks to Ericsson and specially the EVP team for being helpful and accommodating for this period. This has been a great learning experience.

Arjun Sathyanarayana Prasad

Table of Contents

1. Introduction

Over the past decade we have seen rapid adoption of the mobile smartphones among users worldwide. It is estimated that the number of smartphones sold in the year 2013 itself was close to 1 billion [1]. It is not only that these devices are affordable but the reason for this evolution of the smartphone business can also be attributed to tremendous advancements in the processor, battery and memory technologies. These devices can handle tasks that were unimaginable few years ago and hence the name Smartphone. The number of applications being built into these mobile devices is ever increasing but so is the computational capability of mobile processors. One of the surveys [2] show that the majority of users purchase smartphones based on its ability to support personal applications. The report also indicates that users are most likely to use their phone for multimedia applications like playing games, photo editing, streaming audio/video content and web surfing. Also more and more intriguing applications like augmented reality is just around the corner and will soon find its way into our lives [3]. Hence, the ability to handle multimedia content is very crucial and this is not true just for smartphones but also for other technologies that are in use today such as tablets, high tech glasses, play stations etc. But, it is well known that processing of multimedia content is highly resource intensive and has a direct impact on the devices` performance.

Of particular relevance in this report is the field of Image analysis and computer vision. Image analysis involves a series mathematical transformations performed on digital images to extract meaningful information. One of the ultimate goals that is sought out to be achieved through Image analysis is to use computers for emulating human vision. A computer needs to be capable of learning from its visual inputs, make inferences and take the necessary action. Computer vision is a field of study that encompasses wide variety of algorithms that are aimed at mimicking functionality of human visual system. Computer vision and Image analysis concepts are applied in numerous areas spanning from a simple photo editor to complex augmented reality applications.

Sophisticated computer vision applications require performing transformations on images at very high frame rates. An image consists of huge chunk of data (pixels) in the orders of hundreds of KBs (Kilobytes) or few MBs (Megabytes) and has direct impact on performance of systems when analyzing them. Running Image analysis algorithms on GPPs (General purpose processors) is often not recommended as they are very inefficient. Especially in Embedded systems with tight constraints on timing, it is difficult to achieve real time performance on resource intensive algorithms by using only GPPs (CPU). Hence different alternatives have been explored in the form of Co-processors, DSPs (Digital signal processors), GPUs (Graphic processing units), SIMD/MIMD (Single instruction multiple data/ Multiple instructions multiple data) processors and hardware accelerators to work in tandem with CPUs to share the workload. GPUs are by far the most preferred choice in the state of the art mobile platforms while it is no surprise that custom accelerators are the most computationally efficient alternative. GPUs offer high degree of flexibility which can be exploited to improve performance of variety of algorithms, but they are power hungry. Accelerators, though very efficient, offer least degree of flexibility and hence are not desirable for varying standards. SIMD processors occupy a sweet spot in this hierarchy of processing units as they provide a better degree of flexibility when compared to accelerators and also they can be computationally efficient when compared to GPUs if the algorithms are vectorizable. In many of the image transformations this is often the case. Since SIMD processors have similar architectures when compared to GPUs, considering SIMD processors as a design alternative in mobile platforms for Image analysis is highly relevant in this context.

In this report we focus on implementation of a widely popular and robust feature extraction algorithm called Scale invariant feature transform (SIFT) [4] that is applied in variety of computer vision applications. Some of the transformations used in SIFT exhibit high degree of data level parallelism and these are mapped on a SIMD processor developed by Ericsson called the Embedded vector Processor (EVP).

The structure of this chapter is as follows: in section.1.1 we briefly discuss basics of a digital image and its characteristics. Section.1.2 gives a conceptual description of the architecture of embedded vector processor (EVP)

from Ericsson with an illustration of a simple image analysis algorithm mapped on EVP. In section.1.3 we formulate the problem statement for our work. In section.1.4 we briefly discuss the related work on SIFT.

1.1. Digital Image

A digital image can be considered as numerical representation of a 2 dimensional array. There are two types of representation of digital images: Raster images and Vector graphics. Normally digital images are associated with raster/bitmap images since most of the imaging devices, displays etc. are raster devices. A raster image consists of finite number of elements called pixels arranged in rows (Height) and columns (Width). Each pixel has a value (integers) assigned to it and indicates the brightness level of a particular color which can be components of RGB (Red, Blue and Green) colors or grayscale.

Color depth/ bit depth of an image is the number of bits required/used to indicate the brightness level of a single pixel in an image. In case of color images a single pixel can be a group of sub pixels for individual color components (RGB) and color depth applies to each individual sub pixels represented as bits per channel (BPC) or bits per sample (BPS) or bits per pixel(BPP). In case of grayscale images, pixels have a single brightness value and the BPP indicates the number of gray levels that can be present in an image varying between and including white and black colors. Color depth of an image specifies the number of distinct colors that can be displayed but does not indicate which colors. There is another parameter called the color gamut. Color gamut specifies the subset of colors that can be represented accurately within a given color space (perceived by human eye) or by certain output device. Figure.1.1 called the CIE (international commission on illumination) diagram explains the concept of the color gamut. The figure shows a full range of colors perceived by the human eye and a subset of colors that a common HDTV can display. Sometimes along with the color components another channel to indicate transparency is also encoded in an image which is called the alpha channel. Typically images used are of 8 BPC color depth with or without alpha channel since the operations on images are computationally simpler and the number of colors are sufficient given the sensitivity of the human eye. Table.1.1 gives different variants of commonly used color systems along with their applications.

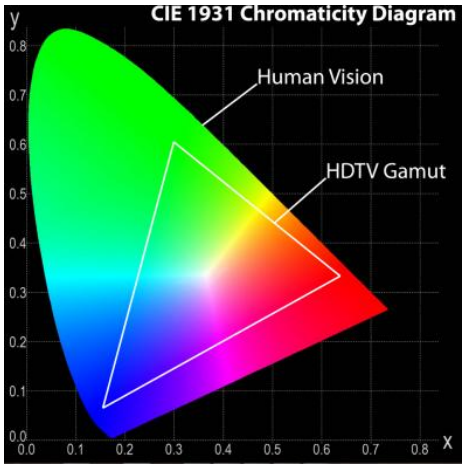

Figure.1.1. Color gamut of a HDTV (CIE diagram) Source [5]

Bitmap/Raster images are often stored in compressed or uncompressed file formats. Once rasterized, these images are transformed into grids of pixels with designated color depth for displaying it on the screen. Compression techniques are required to store images since storing images in raster format needs lot of memory space. The size of an image is directly correlated to the color depth and the number of pixels present in an image. Compression techniques can be either lossy or lossless.

Color systems	Bits per channel (Color depth)	Color gamut	Applications
8 bit true color	3(R,G) and 2(B)	256	Early computers
16 bit direct color	4(RGB) + 4 alpha	4096 colors + 16 level transparency	Mobile phones Small device displays Photographic images
	5(RGB) + 1alpha	32768 colors + 2 level transparency	
	5(R) + 6 (G) + 5(B)	65536 colors	
18 bit color	6 BPC (RGB)	262,144 colors	LCD displays
24 bit true color	**8 BPC (RGB) + Alpha(optional)**	**16,777,216 colors**	**Modern desktops**
Deep color(30/36/48 bit)	10/12/16 BPC	Billions of colors	Graphic cards

Table.1.1. Different coloring systems, their characteristics and applications

Following are the different categories of file formats commonly used to store images.

Uncompressed - Stores images in rasterized format. It is easy to perform Image transformations, as each pixel data can be easily parsed. Examples: *BMP, PGM, PBM.*

Lossless Compression - In lossless compression an exact replica of original raster grid is stored but with reduced file size. The file size depends on the actual graphical complexity of an image. It is often advantageous when storing graphically simpler images (Continuous regions in an image like cartoons or intermediate edited images) as it can achieve better compression. Examples: *PNG, GIF, TIFF.*

Lossy Compression - In lossy compression the image quality is compromised for file size. Often algorithms are designed to exploit the limitation of human eye characteristics and discard unnecessary information. So in actuality the image may appear to be perfectly good even though it is of low quality. They are generally better than lossless compression in achieving lower file size. They are typically used for images that are stored on the internet. Examples: *JPEG, TIFF, EXIF.*

Table.1.2 summarizes different file formats, their characteristics and applications. For most of the image transformations it is required that the images are in uncompressed format. Hence in this report, test images that are chosen are of PGM file format and also they are of 8 bit color depth (Gray Scale) as SIFT is applied on gray images. But this concept can be extended to images with different color depths and color systems.

File formats	Type	Color depth	Compression	Use
JPEG	Lossy	8 BPC (RGB, Gray)	++	Final Distribution, Internet
PNG	Lossless	8/24/48 bit true color With/Without alpha	+	Animation, Intermediate images
GIF	Lossless	8 bit(256 colors)	+	Logos ,cartoons
BMP	uncompressed	1,4,8,16,24,32	--	Cameras, photos, image processing

PGM	**uncompressed**	**8 bit grayscale**	**--**	**Image processing**

Table.1.2. File formats, their characteristics and applications

1.2. Embedded vector Processor

Embedded vector processor (EVP) is a vector DSP developed by Ericsson and is being used in Ericsson modems in baseband processing for mobile standards [6]. Figure.1.2 gives a conceptual overview of the architecture of EVP.

EVP has VLIW architecture and hence has the ability to pack multiple operations in one instruction and executes them in a single clock cycle on parallel Functional Units. It is made up of a scalar data computation unit (SDCU) and a vector data computation unit (VDCU). SDCU can be operate in parallel to VDCU and can perform common scalar operations. The SIMD (vector) width of EVP is 256 bits and the vector operations can be performed in the granularities of 8 bits (limited), 16 bits and 32 bits. Thus the number of elements per vector (P) that can be operated on in parallel can be 32, 16 or 8 depending on the granularity. From the implementation point of view the VALU, VLSU, VMAC, VBCST and VSHU units of the VCDU are of prime importance and will be discussed in more detail below.

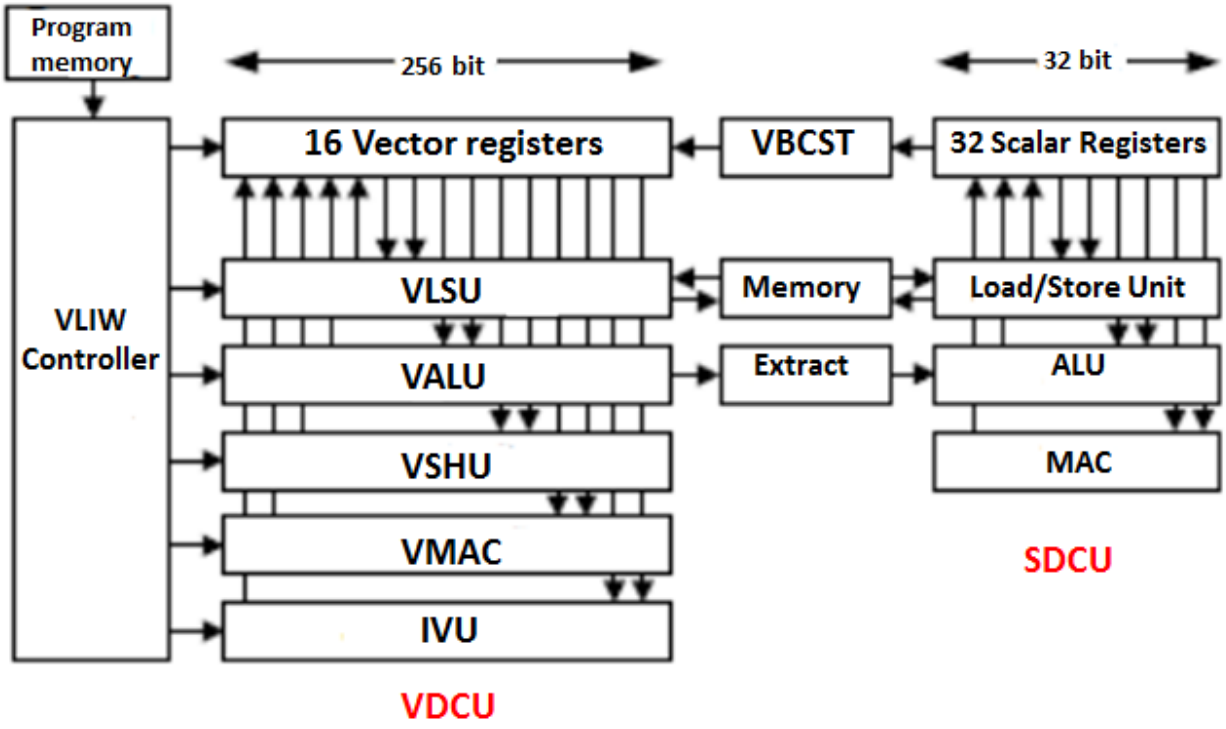

Figure.1.2. Architecture of EVP- Source [6]

- The VDCU unit consists of 16 general purpose vector registers shared among all functional units and each vector is 256 bit wide.
- The vector load and store unit (VLSU) forms the interface between vector units and the data memory. It supports aligned and unaligned loads of vectors from the memory. Aligned load operates on 256 bit boundaries. Unaligned loads can load vectors at 8 bit boundaries. Aligned vector loads have a latency of 3 cycles while unaligned loads take 4 cycles. The initiation interval (Cycles between two operations of given type) for both kinds of loads is 1 cycle. The load and store unit internally maintains a fully associative

cache for unaligned loads and fetches consecutive vectors in a stream. This cache is limited in size and hence often restrictive (only a maximum of three data streams are supported efficiently).

- The vector arithmetic and logic unit (VALU) supports basic ALU operations such as addition, subtraction, comparison etc. and logical bitwise operations on 8/16/32 bit data types. The operations have a latency and initiation interval of 1 cycle.
- The vector multiplication and accumulation unit (VMAC) supports integer/fixed and floating point multiplications. EVP is capable of performing multiplications of only 16 bit arguments with accumulations supported up to 16/32/40 bits. The latency of integer MAC operations is 2 cycles with initiation interval of 1 cycle. There is no support for 8 bit granularity. Capabilities such as saturation and rounding are also included.
- The vector shuffle unit (VSHU) is used to organize, shift/rotate elements in a vector. It uses shuffle patterns (16 patterns supported) that can be used to re-organize values inside a vector. The latency and initiation interval is 1 cycle each.
- Vector Broadcast unit (VBCST) can be used to broadcast a scalar element to all vector data paths.
- Vector mask registers are supported which is used in combination with various operations on EVP to manipulate only parts of vector data. For example, the shuffle operations in conjunction with mask registers can be used to merge two different vectors. EVP also consists of a logical unit called Vector mask arithmetic and logic units (VMALU) for performing logical operations on vectors masks.
- Almost all operations can be used as predicated operations. The predicated operations have an additional parameter of type *bool* which determines whether the operation is performed or not.
- Extra information: An IVU (intra vector unit) unit is present on EVP which supports operations on data elements within a vector. EVP supports bypass so that results of one functional unit can be directly passed to other units without the need for register store in the pipeline stages. An address computation unit (ACU) is also present for performing arithmetic operations on addresses. Specialized program control units that can run in parallel to VDCU and SDCU are included to support hardware loops, calls and branching.
- Software development is done in a language called EVP-C which is a superset of basic ANSI-C language to support specialized functions of EVP.

1.2.1. Implementation of Sobel kernel on EVP

To illustrate mapping of image analysis kernels on EVP let us take an example of a simple edge detection algorithm called the Sobel operator. The Sobel filter consists of two kernels *Gx* and *Gy* of size 3x3 convolved with an image (I) in X and Y direction as shown below.

$$Gx = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I \qquad Gy = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I$$

The two kernels compute the gradient variations in their respective directions. These gradients extract sharp variations in the brightness levels across the image and thus detect edges [6]. Figure1.3. shows the output of a Sobel filter applied on a standard test image of size 512x512.

Convolution is performed by moving each of the kernels across the image, one pixel at a time computed as below. Consider the kernels as a kind of an overlay centered on a pixel of interest. Then the gradient is calculated as the weighted values of pixel of interest and its corresponding 8 neighbors. Figure.1.4 demonstrates the same.

Consider “I11” as the pixel of interest. Then the resulting gradients at I11 can be calculated as follows.

Gx11/Gy11 = G00*I00 + G01*I01 + G02*I02 + G10*I10 + G11*I11 + G12*I12 + G20*I20 + G21*I21 + G22*I22

The coefficients are integer values and are multiplied with pixel values of range [0:255]. After Gx11 and Gy11 are calculated the next pixel which is I12 is made as the pixel of interest and the filters are centered on it. The boundary pixels are not computed on EVP and are instead set to zero while storing back the output image.

Figure.1.3.Sobel edge detection performed on EVP – "lena" standard test image

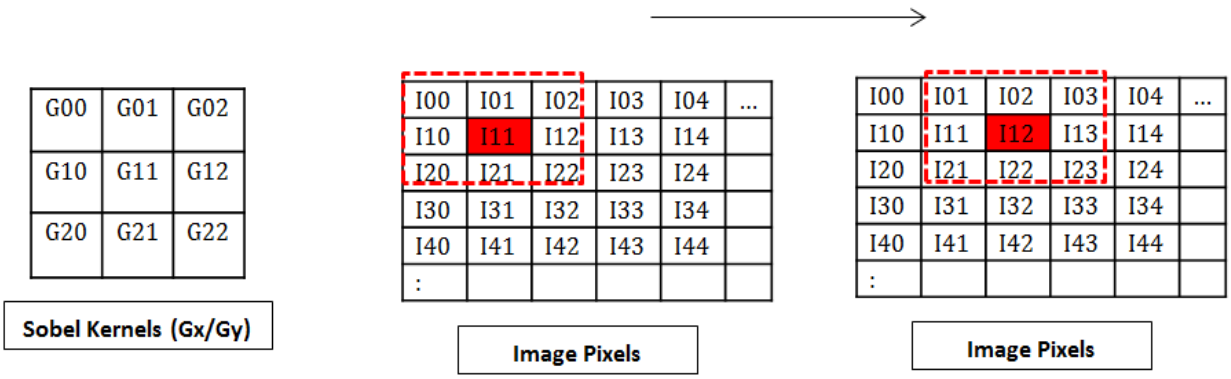

Figure.1.4. Illustration of convolution using Sobel kernels

After calculating the gradients in the two directions the gradient magnitude at each pixel is calculated using the following equation.

$$G = \sqrt{G_X^2 + G_y^2}$$

For simplicity an approximate formula for calculating gradient magnitude is used which is as below.

$$G = \|G_x\| + \|G_y\|$$

The gradient magnitude *G* is stored as the output image which will contain edges. The pixels at the corners are filled with zeroes and are not operated on by the kernels. Although there are 18 coefficients from the two kernels *Gx* and *Gy*, 6 of them are zeroes and hence in total we need to convolve the image with only 12 coefficients.

Let us see how a sequential pseudo code would look like if we were to apply Sobel filters.

```
W -> width of the image
H -> height of the image
I[W][H] -> input image
O[W][H] -> output image

for(i = 1; i < H-1; i++)
{
      for(j = 1; j < W-1; j++)
      {
            X = -1*I[i-1][j-1];
            X += -2*I[i][j-1];
            X += -1*I[i+1][j-1];    // Convolution with Gx kernel
            X += 1*I[i-1][j+1];
            X += 2*I[i][j+1];
            X += 1*I[i+1][j+1];

            Y = -1*I[i-1][j-1];
            Y += 1*I[i+1][j-1];
            y += -2* I[i-1][j];
            y += 2* I[i+1][j];      // Convolution with Gy kernel
            Y += -1*I[i-1][j+1];
            Y += 1*I[i+1][j+1];

            G = abs(X) + abs(Y);
            G = min(G,255);         // Saturate
            O[i][j] = G;
      }
}
```

Now let us consider implementation of Sobel kernels on a SIMD machine which in this case is EVP. We re-represent the whole image as array of vectors with each vector having 16 (P) elements. This is called vectorization. For example if we consider a 512x512 image, then we divide each row into 32 vectors containing 16 pixels each. Figure.1.5 demonstrates an example of how convolution is performed on vectors for *Gx* kernel. In the Y direction three consecutive rows are convolved with the corresponding coefficients. In the X direction, shifted versions of input stream are obtained and convolved and this is illustrated in the figure for row Ri. The same can be applied to other rows Ri+1 and Ri+2. The whole process is repeated in similar fashion for Gy kernel.

The shifted versions of input streams are obtained either by using simple unaligned loads on EVP or by using aligned loads in combination with shuffle and rotate operations. Figure.1.6. shows how this can be achieved. These operations help us in accessing neighbors of each element within a particular vector. Thus when convolved, the output will have the weighted sum of the relevant neighboring pixel values.

There are two possible approaches for performing convolution which is explained in the following sections.

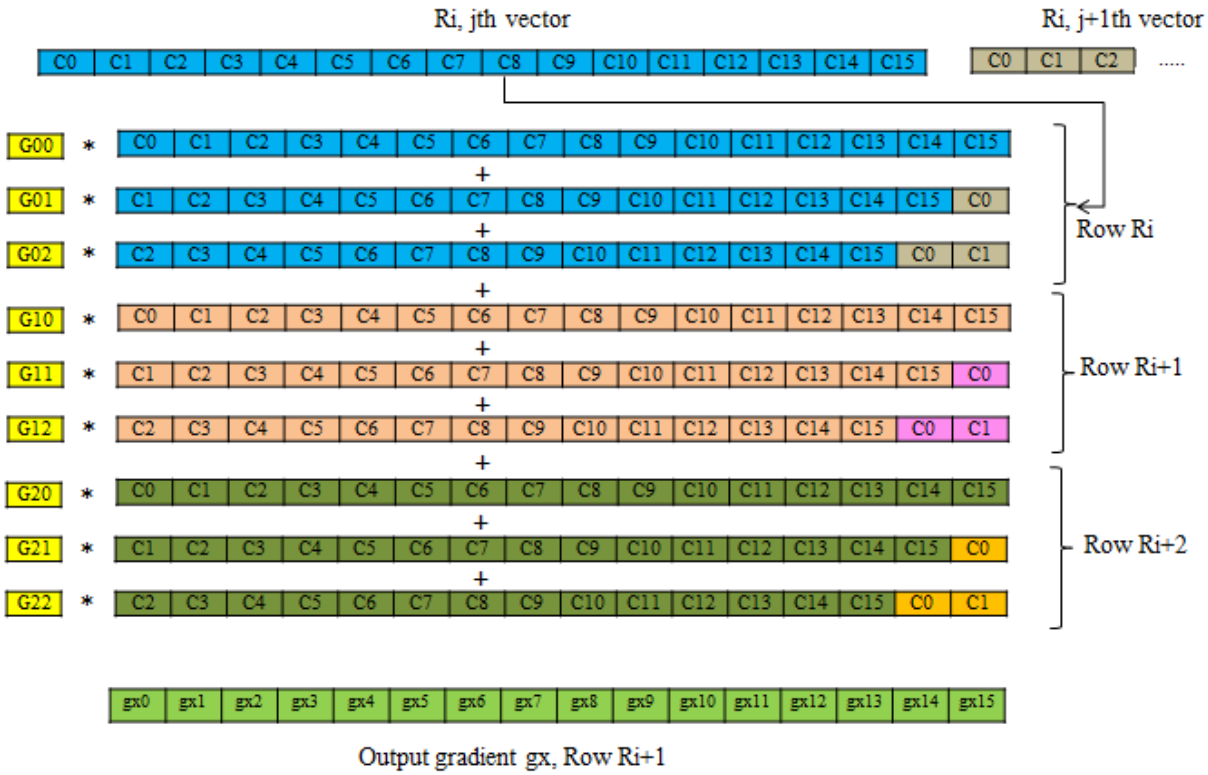

Figure.1.5. Illustration of convolution on EVP

Figure.1.6. Illustration of aligned and unaligned loads

Approach 1

Let us consider implementation on EVP that is similar to the sequential implementation that we discussed in the previous section. The operations are performed on 16 pixels together and the resulting output is for all 16 pixels. Figure.1.5 actually illustrates this approach. Below is the pseudo code for the same.

```
I [H][W/P] -> input vector of elements
for(i = 0; i< H-2; i++)
{
      j = 0;
      I1 = I[i+0][j*P : j*P+P-1];
      I2 = I[i+1][j*P : j*P+P-1];   // Aligned load of first vector in a row
      I3 = I[i+2][j*P : j*P+P-1];

      I11 = I[i+0][(j+1)*P : (j+1)*P+P-1];
      I22 = I[i+1][(j+1)*P : (j+1)*P+P-1]; // Load the next vector in the row
      I33 = I[i+2][(j+1)*P : (j+1)*P+P-1];

      for(j = 0; j < W/P; j++)
      {
            X[ 0 : P-1] = (-1) .* I1;
            X[ 0 : P-1] += (-2) .* I2   // Mac operations with coefficients
            X[ 0 : P-1] +=(-1) .* I3;

            Y[ 0 : P-1] = (-1).* I1;
            Y[ 0 : P-1] += (1).* I3;

            T1 = (I1[1:P-1],I11[0]); //Rotate one position
            T3 = (I3[1:P-1],I33[0]); //Merge and rotate 1 position on VSHU

            Y[ 0 : P-1] += (-2).* T1;
            Y[ 0 : P-1] += (2).* T3;

            T1 = (I1[2:P-1],I11[0],I11[1]);
            T2 = (I2[1:P-1],I22[0],I22[1]);// Merge and Rotate 2 positions
            T3 = (I3[2:P-1],I33[0],I33[1]);

            X[ 0 : P-1] +=  (2) .* T2;
            X[ 0 : P-1] +=  (1) .* T1;
            X[ 0 : P-1] +=  (1) .* T3;

            Y[ 0 : P-1] += (-1).* T1
            Y[ 0 : P-1] += (1).* T3;

            G[0 : P-1] = abs(X[0 : P-1)) + abs(Y[0 : P-1)); // Final gradient

            G[0 : P-1] = min(S[0 : P-1],bcst(255)); // Clipping/Saturation

            O[i+1][j*P :j*P+P-1] =  G[0 : P-1]; //Store

            I1 = I11;
            I2 = I22;    // Use input vector for next iteration
            I3 = I33;

            I11 = I[i+0][(j+2)*P : (j+2)*P+P-1];
            I22 = I[i+1][(j+2)*P : (j+2)*P+P-1]; // Load a new vector
```

```
            I33 = I[i+2][(j+2)*P : (j+2)*P+P-1];
        }
}
```

The implementation generates two bogus values at the end of each row. The schedule generated by compiler for the innermost loop on EVP is as shown in the figure1.7. It takes 12 cycles to calculate output for 16 pixels. The number of cycles in this case is determined by the load on the MAC unit

Cycles	VMAC	VALU	VLSU	VSHU
1	Y[0 : P-1] += (2).* T3;	T3=(I3[2:P-1],I33[0],I33[1]);	I1 = I11;	T1=(I1[2:P1],I11[0],I11[1]);
2	X[0 : P-1] += (2) .* T2;	I3 = I33;	I11 = I[i+0][(j+2)*P : (j+2)*P+P-1];	T3=(I3[2:P-1],I33[0],I33[1]);
3	Y[0 : P-1] += (-1).* T1	G[0 : P-1] = min(S[0 : P-1],bcst(255));	I22 = I[i+1][(j+2)*P : (j+2)*P+P-1];	T2 = (I2[1:P-1],I22[0],I22[1]);
4	X[0 : P-1] += (1) .* T1;	T1 = (I1[1:P-1],I11[0]);	I33 = I[i+2][(j+2)*P : (j+2)*P+P-1];	
5	X[0 : P-1] += (1) .* T3;			
6	Y[0 : P-1] += (1).* T3;			
7	X[0 : P-1] = (-1) .* I1	abs(X[0 : P-1])	T1 = (I1[1:P-1],I11[0]);	X[0 : P-1] = (-1) .* I1
8		abs(Y[0 : P-1])		T1 = (I1[1:P-1],I11[0]);
9	Y[0 : P-1] += (1).* I3;	G[0 : P-1] = abs(X[0 : P-1]) + abs(Y[0 : P-1]);	Y[0 : P-1] = (-1).* I1;	T3 = (I3[1:P-1],I33[0]);
10	X[0 : P-1] += (-2) .* I2;	G[0 : P-1] = min(S[0 : P-1],bcst(255));	T3 = (I3[1:P-1],I33[0]);	I2 = I22;
11	Y[0 : P-1] += (-2).* T1;		T1=(I1[2:P-1],I11[0],I11[1]);	T2 = (I2[1:P-1],I22[0],I22[1]);
12	X[0 : P-1] +=(-1) .* I3;	T1=(I1[2:P-1],I11[0],I11[1]);	O[i+1][j*P :j*P+P-1] = G[0 : P-1];	T3 = (I3[1:P-1],I33[0]);

Figure.1.7. Schedule of innermost loop for approach 1

Approach 2

Now, in the schedule for approach 1 we see that there are 11 MAC operations in 12 cycles and thus VMAC becomes critical resource as it is used 11 out of 12 cycles to compute the output and thus determines the performance of the innermost loop. Since the coefficients of the kernels are simple, many of these MAC operations can be replaced with simple addition and subtraction operations. This way we can exploit the VLIW architecture of EVP to reduce the load on critical resource by sharing loads on different functional units. Below is the pseudo code for this approach.

```
I [H][W/P] -> input vector of elements
for(i = 0; i< H-2; i++)
{
      j = 0;
      I1 = I[i+0][j*P : j*P+P-1];   // aligned load
      I2 = I[i+1][j*P : j*P+P-1];
      I3 = I[i+2][j*P : j*P+P-1];

      I11 = I[i+0][(j+1)*P : (j+1)*P+P-1];
      I22 = I[i+1][(j+1)*P : (j+1)*P+P-1]; // next vector in the row
      I33 = I[i+2][(j+1)*P : (j+1)*P+P-1];

      Xmul1[ 0 : P-1] = (2) .* I2;         // Partial values of convolution
      Ymul1[ 0 : P-1] = (2) .* I1;
      Ymul3[ 0 : P-1] = (2) .* I3;

      X1[ 0 : P-1]  = I1 + I3 + Xmul1;
      Y1[ 0 : P-1]  = I3 + I1;
      Y2[ 0 : P-1]  = Ymul1+ Ymul3;

      for(j = 0; j < W/P; j++)
```

```
        {
                Xmul10[ 0 : P-1] = (2) .* I2;
                Ymul30[ 0 : P-1] = (2) .* I3;
                Ymul10[ 0 : P-1] = (2) .* I1;

                X10[ 0 : P-1]  = I10 + I30 + Xmul10;

                Y10[ 0 : P-1]  = I30 - I10;

                Y20[ 0 : P-1]  = Ymul30 - Ymul10;

                T1 = (X1[2:P-1],X10[0],X10[1]);
                // Merge two vectors, rotate 2 positions

                X[ 0 : P-1] = T1 - X1;

                T2 = (Y2[1:P-1],Y20[0]);  //Merge two vectors, rotate 1 position

                Y[ 0 : P-1] = Y1 + T2;

                T2 = (Y1[1:P-1],Y10[0], Y10[1]);

                Y[ 0 : P-1] = Y + T2;

                G[0 : P-1] = abs(X[0 : P-1)) + abs(Y[0 : P-1));

                G[0 : P-1] = min(G[0 : P-1],bcst(255));


                O[i+1][j*P :j*P+P-1] =  G[0 : P-1]; //Store

                X1 = X10;
                Y1 = Y10;
                Y2 = Y20;

                I11 = I[i+0][(j+2)*P : (j+2)*P+P-1];
                I22 = I[i+1][(j+2)*P : (j+2)*P+P-1];
                I33 = I[i+2][(j+2)*P : (j+2)*P+P-1];
        }
}
```

The schedule generated by compiler for approach 2 is as shown in the figure.1.8. Here since we reduce the number of MAC operations the operations are shared among the other functional units and thus the number of cycles reduces for the innermost loop. This is one of the major advantages of VLIW architectures as we can exploit even the instruction level parallelism. Thus we require only 10 cycles to compute 16 output pixels.

Now consider that we were to run Sobel kernel on a CPU with sequential code as discussed earlier. With 12 coefficients we require at least 12 mac operations and hence at the very least we require 12 cycles to compute one output pixel. Hence for 16 output pixels we require 16x12 = 192 cycles. Let us compare this with EVP implementations to get an idea about the speed up achieved with respect to sequential implementation. This is shown in in table.1.3.

As we can see that with approach 1 we get a speed up of 16(P) which is expected as we perform 12 MAC operations on 16 pixels together in 12 cycles. But by exploiting VLIW nature in approach 2 we can achieve better speed up of ~19x. This speed up does not indicate the overall speed up for the Sobel algorithm itself. The speed up for the

whole algorithm will be lesser than these numbers because the reading of an image from the memory and writing it back to the memory will cost non-negligible amount of cycles. Also the functional overheads and outer loop overheads can increase the overall number of cycles required but, the speed up for the whole algorithm would be still sufficient to improve throughput of Sobel kernel. The idea behind this exercise is to only demonstrate how speed up can be achieved on EVP for image analysis kernels.

```
Cycles   VMAC                          VALU                          VLSU                          VSHU
                                                                                                     T1 = (X1[2:P-
1        X10[ 0 : P-1] + Xmul10;       Y10[ 0 : P-1]  = I30 - I10;   T2 = (Y2[1:P-1],Y20[0]);       1],X10[0],X10[1]);
                                                                          T1 = (X1[2:P-
2                Y2 = Y20;             T2 = (Y1[1:P-1],Y10[0], Y10[1]);   1],X10[0],X10[1]);       T2 = (Y2[1:P-1],Y20[0]);
                                                                       I11 = I[i+0][(j+2)*P :          T1 = (X1[2:P-
3        Y[ 0 : P-1] = Y1 + T2;        T2 = (Y1[1:P-1],Y10[0], Y10[1]);   (j+2)*P+P-1];            1],X10[0],X10[1]);
                                                                       I33 = I[i+2][(j+2)*P :       T2 = (Y1[1:P-1],Y10[0],
4                X1 = X10;                X[ 0 : P-1] = T1 - X1;        (j+2)*P+P-1];                  Y10[1]);
                                                                       I22 = I[i+1][(j+2)*P :
5        Y[ 0 : P-1] = Y + T2;                 abs(X[0 : P-1])          (j+2)*P+P-1];                  Y1 = Y10;
         Ymul30[ 0 : P-1] = (2)
6                .* I3;                      abs(Y[0 : P-1))
         Ymul10[ 0 : P-1] = (2)       G[0 : P-1] = abs(X[0 : P-1)) +
7                .* I1;                      abs(Y[0 : P-1));
         Xmul10[ 0 : P-1] = (2)         G[0 : P-1] = min(G[0 : P-
8                .* I2;                        1],bcst(255));
                                                                       O[i+1][j*P :j*P+P-1] =
9                                      X10[ 0 : P-1]  = I10 + I30;       G[0 : P-1];
         Y20[ 0 : P-1]  = Ymul30
10               - Ymul10;
```

Figure.1.8. Schedule of innermost loop for approach 2

	Sequential	EVP approach 1	Speed up approach 1	EVP approach 2	Speed up approach 2
Cycles to compute 16 pixels	192	12	16	10	19.2

Table.1.3. Vector speed up by exploiting data level parallelism and instruction level parallelism

1.3. Problem Description

In the previous sections we have introduced basic concepts of image analysis and how embedded vector processor can be used to speed up image analysis kernels. In general, computer vision algorithms are far more complex than Sobel operator and mapping them on EVP would be a challenge. One such algorithm is SIFT which we are considering in our work.

Scale invariant feature transform (SIFT) is a sophisticated algorithm applied on images to extract robust features that are highly distinctive and invariant to scale, orientation and illumination of an image. The features extracted can be used in many applications such as object/Scene recognition, motion tracking, stereo correspondence, 3D modelling, image stitching etc. SIFT is composed of various stages and each stage involves a transformation performed on the images. These transformations are computationally very intensive and thus it is necessary to speed them up to improve the overall performance of SIFT. Some of the stages of SIFT exhibit high degree of data level parallelism which can be exploited to improve throughput. Thus the goal of this thesis is:

"To Accelerate SIFT algorithm by off-loading computationally intensive image transformations on to Embedded Vector Processor in a multi-core environment"

The key tasks that need to be performed to achieve this goal are:

- To identify the stages of SIFT that exhibit data level parallelism.
- To define data flow that is suitable for maximizing the performance on EVP.
- Efficient mapping of individual stages of SIFT on EVP.
- Check for functional correctness of SIFT algorithm on EVP.
- To find whether EVP is suitable for performing SIFT.
- Identify shortcomings in the EVP architecture and suggest changes.

1.4. Related work

In order to accelerate SIFT to obtain real time performance research efforts have been made prior to this. Many authors propose custom hardware oriented designs [8, 9, 10] to achieve high performance for SIFT. There are works which propose FPGA based implementations [11, 12, 13, 14, 15] for achieving real time speed up. These custom hardware implementations are mainly focused on applications where SIFT can be applied to compute descriptors at very high frame rates (e.g. 1920x1080 resolution videos at 30 frames/second) in real time [9].

Some of the works also explore the possibility of accelerating SIFT with a heterogeneous multi-core [16, 17, 18, 19] architectures where different kinds of processors are working in tandem. Architectural choices such as symmetric multiprocessor platforms [17], chip multiprocessor platforms (CMP) [17, 18], homogenous multi-core DSP platforms [19] and network on chip multi-core platforms [15] are proposed by authors. One of the implementations [16] show 30 frames/second throughput on 720p video sequences.

Many of these design alternatives are not suitable and practical for mobile platforms. Efforts have been made to speed up SIFT on mobile platforms using general purpose computing on graphic processing units (GPGPU) [20, 21, 22, 23, 24]. In these implementations only parts of the SIFT algorithm that are resource intensive are off-loaded to GPUs. Even with high computational complexity and the memory transfer bottlenecks, these implementations are able to achieve considerable speed up of SIFT (~10x) compared to CPU only implementations [22]. In one of the work [21], authors show an achievement of ~10 fps on execution of SIFT algorithm.

The GPU architectures are similar to SIMD processors although the former provides greater flexibility. The data level parallelism present in some of the stages of SIFT can be exploited by SIMD processors in the same way as GPUs. This is one of the main motivations in considering EVP for running SIFT and it is possible to achieve better efficiency on SIMD processors.

2. Scale invariant Feature Transform – Algorithm description

Image matching is one of the fundamental concepts used in many computer vision applications. SIFT is an algorithm that extracts highly distinctive and stable features from images that can be used to perform reliable matching between different views of an object or a scene [4]. These features are invariant to scale, rotation and illumination of an image. Even if the images are noisy or have substantial affine distortions, or if the 3D viewpoint of an object changes, the features are robust enough to carry out accurate matching [4]. The distinctiveness of the features allows it to be found with good probability even when searching against a large database of features extracted from many other images [4]. SIFT algorithm involves several complex steps which are computationally intensive and time consuming. Hence in order to minimize the cost of performing transformations, the algorithm is implemented with a cascade filtering approach [4] where more intensive operations are performed only at locations of interest which pass the initial test [4]. SIFT algorithm can be divided into four major stages namely:

1.) Scale space extrema detection

2.) Keypoint localization

3.) Orientation assignment

4.) Keypoint descriptor generation

Figure.2.1 gives a pictorial description of the above four stages along with its sub steps. In the following sections we discuss each of the above four stages in detail.

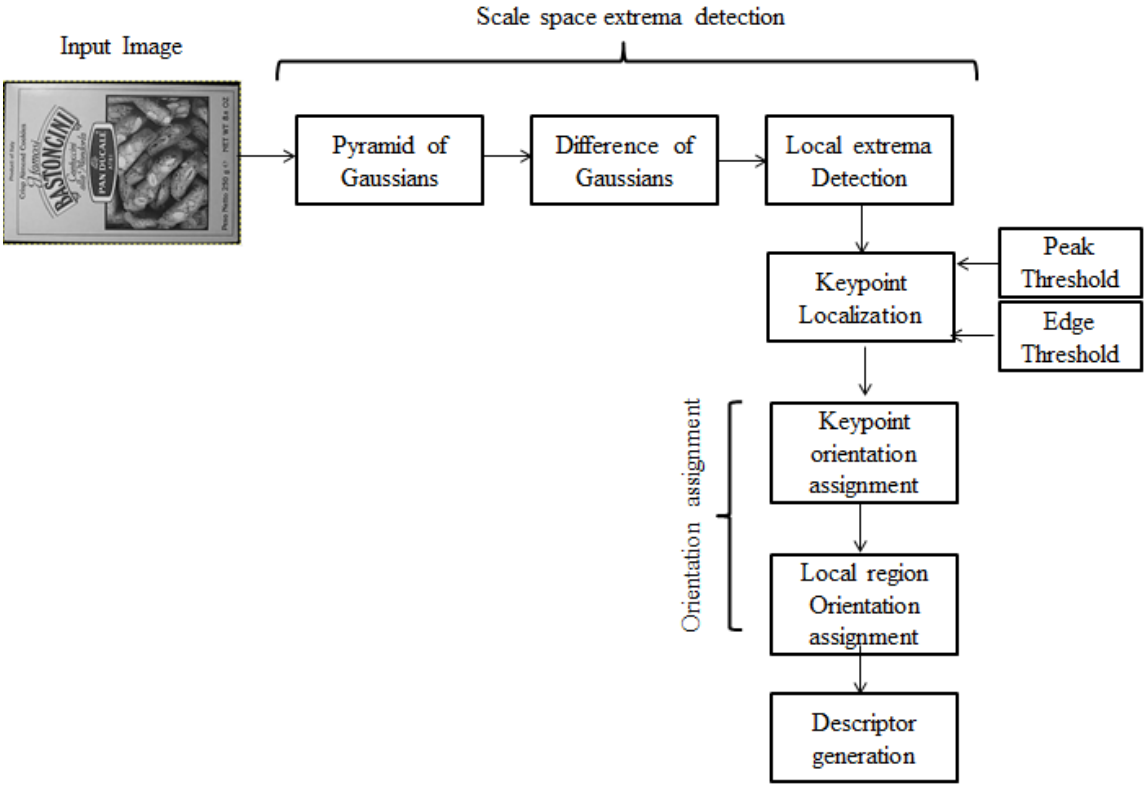

Figure.2.1. Major stages of SIFT algorithm

2.1. Scale space extrema detection

Objects in an image vary a great deal in their sizes. These objects when captured on a camera also appear with different sizes and structures depending on the distance between camera and the object and also the view point. Hence for accurate matching an object or a scene, scale of observation is a very vital component. Unfortunately when processing an unknown image, it is not possible to know the appropriate scale required to analyze interesting structures in an image. Consider for example that we are analyzing an image of a house. It is appropriate to analyze house at a coarser scale but details such as windows require finer scales of observation. Hence, in order to tackle with these unknown scale variations in an image a reasonable approach would be to analyze images at all possible scales. This method of analyzing image structures at different scales is called as Scale-space representation. In this representation a set of smoothed/blurred images are constructed by varying scale factor of the kernel being applied on the image. This concept is key for extracting features that are invariant to scale and this is done in the first stage of SIFT. In the following sections we explain briefly the steps performed in the first stage of SIFT.

2.1.1. Difference of Gaussian

As discussed earlier, potential interest points, also called keypoints, are identified in an image over all possible scales and locations using Scale space representation. One of the widely used scale space kernels is the gaussian function, where the standard deviation parameter σ is used as the scale factor for blurring images. A gaussian scale space, $L(x, y,\sigma)$ is obtained by convolving image $I(x, y)$ with the variable scale (σ) gaussian function. Equation.1 gives the mathematical description of the same.

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

Where * operator is the convolution on x and y, and $G(x, y, \sigma)$ is the gaussian kernel described in the equation.2

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

There are a variety of operators like edge detectors, blob detectors, corner detectors etc. which are used to extract interesting features like edges, rapid intensity fluctuations and structural variations from images that are further used in applications like image recognition, matching, motion tracking etc. Many of these operators can be expressed or modelled using variations of gaussian derivative operators which are also best suited for building scale space framework and has wide applicability in many problems. Consider for example, laplacian-of-gaussian operator, which is applying second derivative on an image in the spatial domain (x, y). Performing laplacian of gaussian on an image detects rapid changes in the intensity in an image and can be used as an edge detector or a blob detector. But features extracted by such operators are not scale invariant. In order to achieve true scale invariance, these operators need to be scale normalized.

Specifically in SIFT; David Lowe [4] proposes to extract potential interest points in an image by finding local extrema points in the scale space. This method is similar to commonly used blob detection algorithms. As mentioned earlier, applying simple blob detectors is not sufficient as features extracted from SIFT need to be scale invariant. Laplacian operator $\nabla^2 G$ is one of the first and the most commonly used blob detectors. Lindeberg [25] in one of his work shows that finding local extrema (maxima and minima) of scale invariant laplacian-of-gaussian function, $\sigma^2\nabla^2 G$**,** convolved with images produce most stable features compared to commonly used operators like Hessian, edge or corner detectors. For achieving scale invariance, laplacian operator needs to be normalized by a factor of σ^2. Instead of applying scale invariant laplacian-of-gaussian operator on images directly, Lowe proposes an alternative operator called difference-of-gaussian function to extract scale-space extrema [4]. The difference-of-gaussian

function convolved with an image, *D(x, y, σ)* can computed by taking the difference of two nearby scales varying by a constant multiplicative factor ***K*.** This can be mathematically represented as equation.3.

$$D(x,y,\sigma) = \big(G(x,y,K\sigma) - G(x,y,\sigma)\big) * I(x,y)$$
$$= L(x,y,K\sigma) - L(x,y,\sigma) \tag{3}$$

Lowe shows that Difference-of-gaussian operator is a very close approximation to applying scale-normalized laplacian of gaussian operator,$\sigma^2\nabla^2 G$ on images [4]. From the heat diffusion equation we have:

$$\frac{\partial G}{\partial \sigma} = \sigma\nabla^2 G \tag{4}$$

Equation.4 can be approximated by taking finite difference of two nearby scales $K\sigma$ and σ. The equation can be rewritten as:

$$\sigma\nabla^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x,y,K\sigma) - G(x,y,\sigma)}{K\sigma - \sigma} \tag{5}$$

Rearranging equation.5 we get

$$G(x,y,K\sigma) - G(x,y,\sigma) \approx (K-1)\sigma^2\,\nabla^2 G \tag{6}$$

Equation.6 indicates that difference-of-gaussian function with scales separated by constant factor can incorporate σ^2 normalizing factor required for laplacian operator to become scale invariant. Difference-of-gaussian operator is particularly more efficient method to follow in SIFT. This is because the results of gaussian blurred images computed during scale space representation are also used in the later stages of SIFT. Instead of again applying laplacian derivative operator on gaussian images it is more efficient to just calculate the difference of gaussian as this is mere subtraction of images with different scales.

Blurring of images by gaussian operators leads to removal of high spatial frequencies in the images. When two differently blurred images are subtracted, only spatial information that lies between ranges of frequencies preserved in both the images are extracted. Hence, Difference-of-gaussian operator can be regarded as a band-pass filter. The images convolved with difference-of-gaussian operator enhances details in the image like edges, intensity fluctuations etc. Unlike other filters, difference-of-gaussian filters are more resilient to noise present in the images and this makes it a better option for extracting details used for further analysis.

Now, let us see how the above concepts are applied to digital images to detect keypoints. Before computing difference of gaussian images *D(x, y, σ)*, an initial base image is incrementally convolved with Gaussian kernels of varying scale factor, σ. The σ value for an image is calculated by multiplying the σ value used in the previous image with a constant multiplicative factor *K*. Thus, a finite set of Gaussian blurred images are produced that are separated by factor of *K* in the scale space and this constitutes an *Octave*. Each octave of scale space is divided into integer number of intervals, *s*, and the factor *K* is calculated as, $K = 2^{1/s}$. To produce a complete octave we need to generate *s*+3 Gaussian blurred images. Once gaussian blurred images are generated, difference-of-gaussian images are obtained by just subtracting adjacent blurred images. After processing one complete octave, the Gaussian blurred

image with twice the initial σ value (2 images from the top of the octave) is down sampled to half the size by taking every second pixel of each row and column. This down sampled image forms the base image for processing the next octave in a similar fashion as discussed above. Thus a pyramid of Gaussian blurred images is built.

It has been empirically determined [4] by Lowe that five octaves and six scales per octave are effective for determining stable keypoints with maximum repeatability. Hence in this report we use the same parameters where $s = 3$ and $K = 2^{1/3}$. Also the initial σ value of 1.6 is used as suggested by Lowe to minimize the performance costs while achieving optimum repeatability [4]. It is also assumed that original image has a blur of σ value 0.5 and hence this image is up sampled (σ value 1) by linear interpolation technique to double the size before processing the first octave. The reason behind doubling of the image size is that it increases the number of stable keypoints by a factor of 4 [4].

Figure.2.2 gives a pictorial description of the whole process of constructing gaussian pyramid. There are five octaves and in each octave there are six scales which are repeatedly convolved by gaussian kernels separated in scale space by constant factor K. Figure.2.3 shows how difference-of-Gaussian images are constructed for the first octave and this can be extended to other octaves. In this figure we can also observe how details in the Difference of gaussian images become coarser as we go towards higher scales.

2.1.2. Local Extrema detection

In this step, maxima and minima points in the Difference-of-gaussian images, $D(x, y, \sigma)$, are found. This is done by comparing each sample point (Pixel) in $D(x, y, \sigma)$ with eight of its neighbors and nine neighbors present in the scales above and below. The sample point that is compared should be either greater than or lesser than all of its neighbors. Then such a sample point is considered as a potential keypoint candidate and its location is used for further processing. As stated earlier this step is a variation of commonly used blob detectors. Figure.2.4 depicts this step.

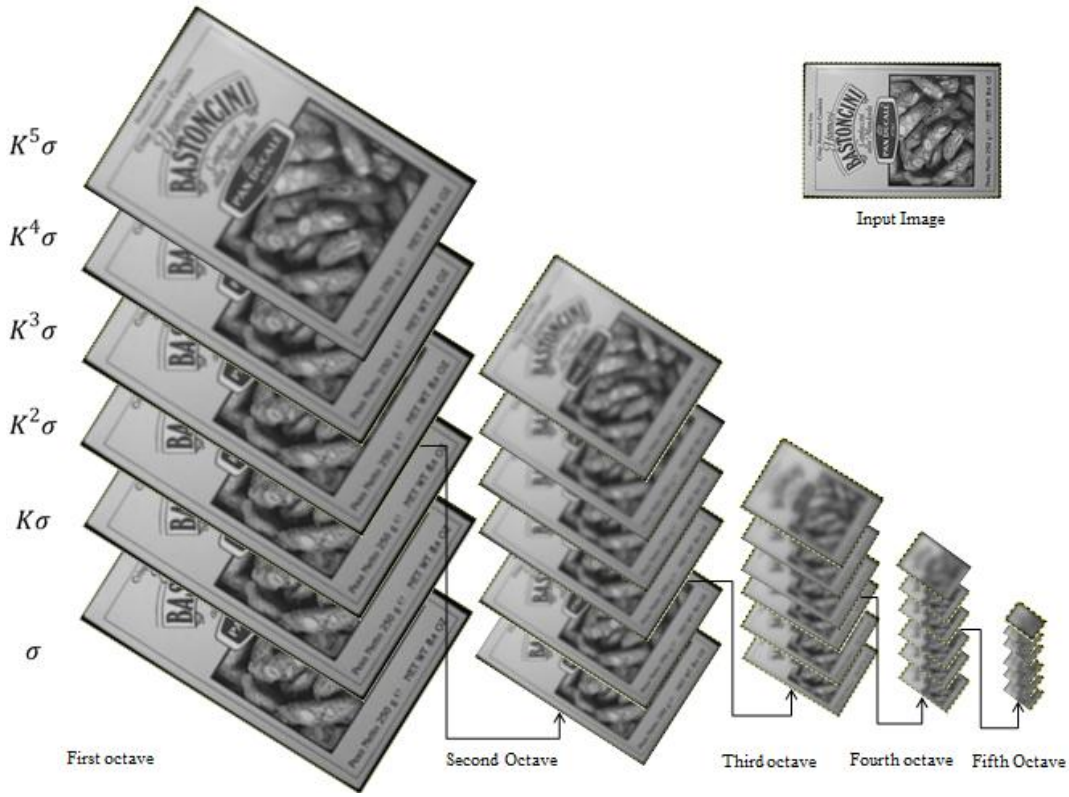

Figure.2.2. Construction of gaussian pyramid

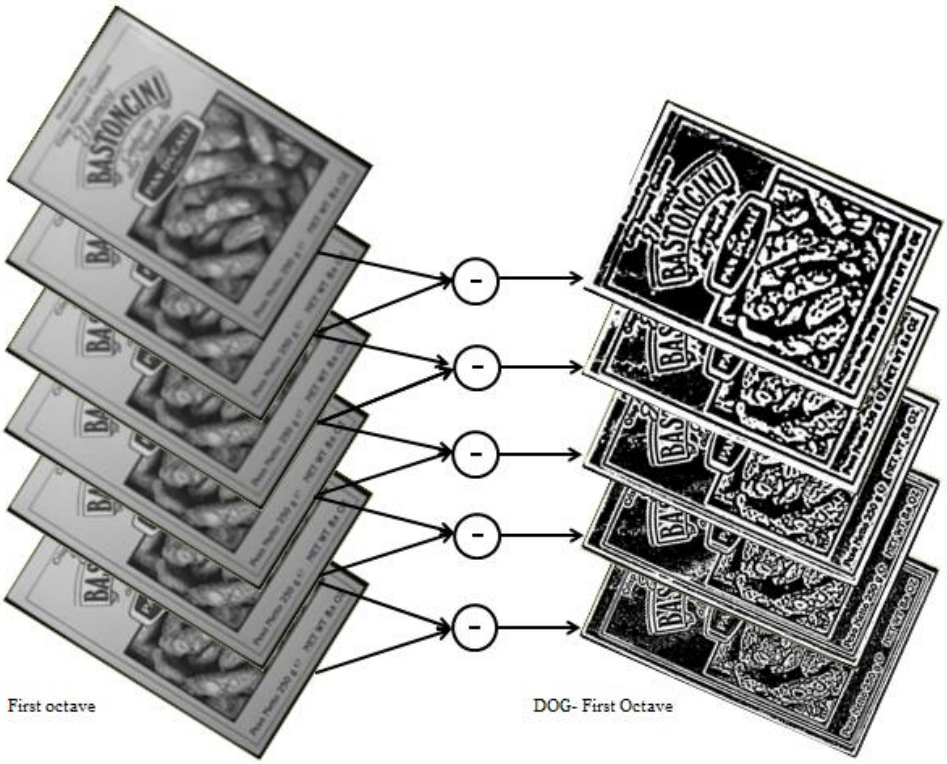

Figure.2.3. Calculating Difference of Gaussian

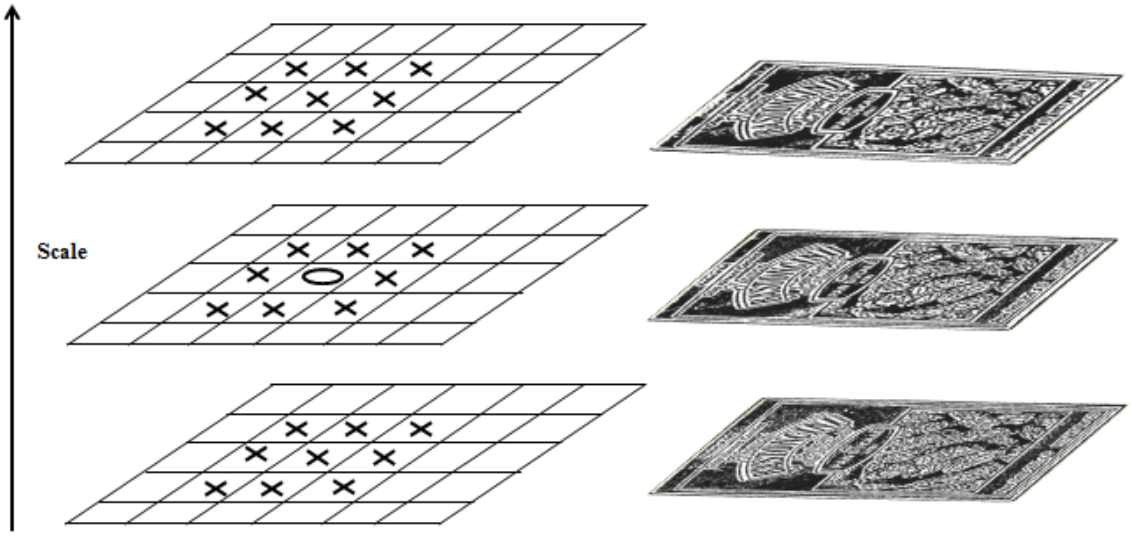

Figure.2.4. Local extreme detection

2.2. Accurate keypoint localization

This is the second stage of SIFT where the potential keypoints detected are further refined for their location and scale to get the interpolated subpixel value of its coordinates in the scale space. Along with this, a detailed analysis of the neighbors in the scale space is made to eliminate those candidates that are non-distinctive with very low contrast values or those which are poorly localized along the edges [4].

2.2.1. Peak/Contrast Threshold

From the extrema detection step we get the location and scale of the potential keypoints which are discrete. By treating scale-space as a continuous function, these keypoint locations can be refined further to get accuracy at sub-pixel level. This will improve the correctness of matching and leads to more stable keypoints [4]. To find the interpolated location of the extremum a 3D quadratic function is applied to the local sample points and its closest neighbors. In Lowe`s approach, Taylor expansion of second order is applied to scale-space function, *D(x, y, σ)* with the sample point as the origin.

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}}\mathbf{x} + \frac{1}{2}\mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x} \tag{7}$$

Here, D and its derivatives are calculated at the sample point where $\mathbf{x}$ = (x, y, σ) is the offset from it. The interpolated extrema offset denoted as, $\hat{\mathbf{x}}$, from sample point is calculated by taking derivative of equation.7 with respect to $\mathbf{x}$ and setting it to zero. Thus we get.

$$\hat{\mathbf{x}} = -\frac{\partial^2 D^{-1}}{\partial \mathbf{x}^2}\frac{\partial D}{\partial \mathbf{x}} \tag{8}$$

The derivatives are approximated to differences of neighboring sample points and this result in a 3x3 linear equation for computing offset of the extremum, $\hat{\mathbf{x}}$. If the offset of the extremum is greater than 0.5 in any of the three dimensions then it means that the extrema point lies closer to a different sample point. Hence the location of the sample point is changed and the interpolation is performed with new sample point as the origin. Finally after the iteration is complete, the offset , $\hat{\mathbf{x}}$ is added to the actual discrete location from the previous step to get the interpolated sub-pixel extrema location.

After calculating the offset, $\hat{\mathbf{x}}$, the value is substituted back in equation.7. Then the absolute value of D($\hat{\mathbf{x}}$) is compared with a set threshold to discard those keypoints with low contrast values. In the paper, Lowe [4] suggests to use a threshold value of 0.03 assuming values of pixels are in the range of [0, 1].

$$D(\hat{\mathbf{x}}) = D + \frac{1}{2}\frac{\partial D^T}{\partial \mathbf{x}}\hat{\mathbf{x}} \tag{9}$$

Figure.2.5 shows how contrast(Peak) threshold is applied to a test image. As we can see, with a lesser threshold keypoints with low contrast values are retained. As threshold increases the number of interest points decrease. Figure.2.6 shows an image of a house on which extrema detection is applied. The original image (a) is of dimension 233x189 pixel. After extrema detection, 832 keypoints are detected shown in the image (b). A minimum contrast threshold of 0.03 (pixel in range of 0-1) is applied and this reduces the number to keypoints to 729 shown in the image (c). The arrows in the figure indicates are the interest points extracted at different scales.

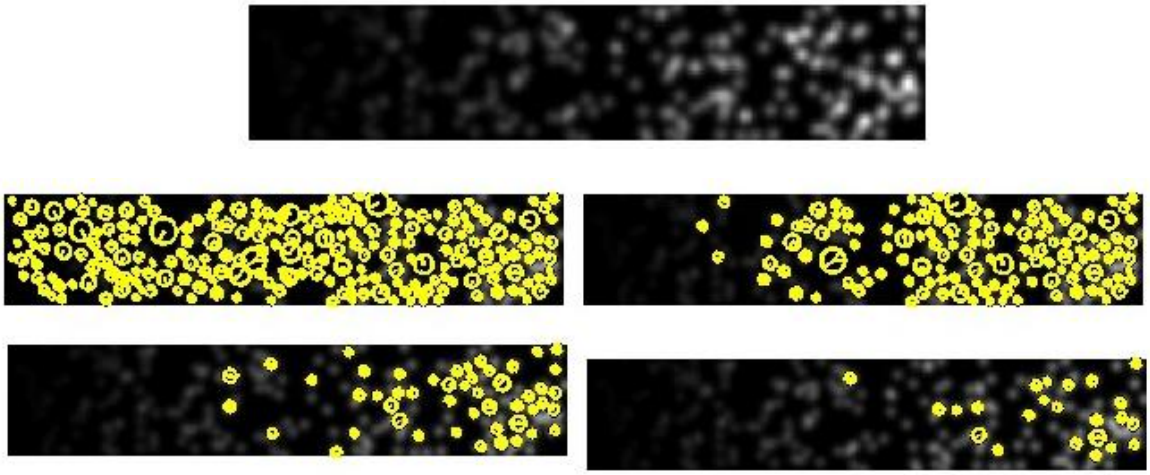

Figure.2.5 Contrast(Peak) threshold applied on a test image(top most) – Source[26]
Images [From top left] with applied Peak threshold = {0, 10, 20, 30}

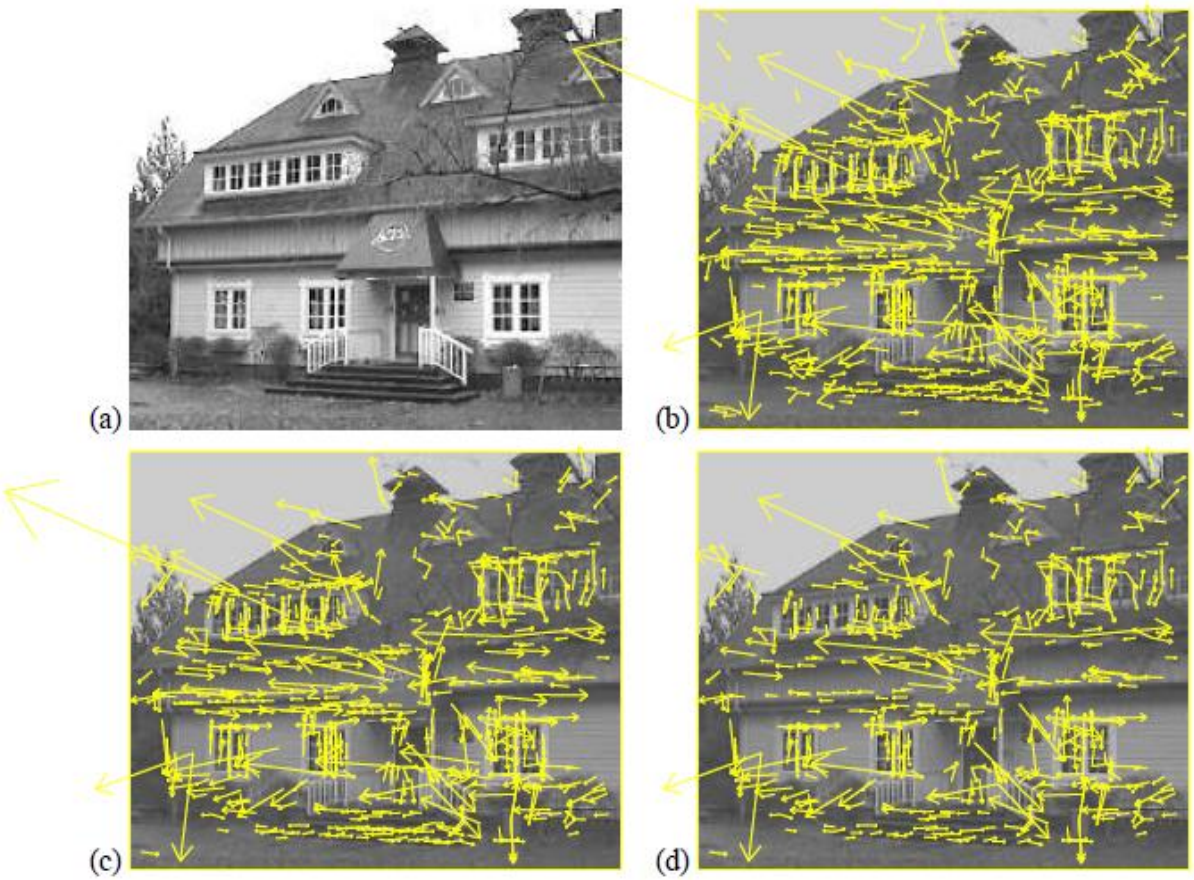

Figure.2.6 – Keypoint refinement – Applying contrast threshold and edge threshold –Source [4].

2.2.2. Edge Threshold

Lowe is interested in detecting blob like structures in SIFT as these structures are easier to match over varying scales and are best suited for applications like object recognition/tracking. Difference of gaussian operator gives strong responses to both blobs as well as edges in the image. Some of these edge responses may be poorly defined and thus have very small curvatures in one of the directions. Such sample points are often unstable as they are easily prone to noise and can yield to badly localized frames. A poorly localized edge will have large principal curvature across the edge compared to the curvature along the edge.

One of the most commonly used functions to determine blob like structures is the hessian operator. Hessian kernel is a 2x2 matrix, as shown below, where *Dxx* and *Dyy* are the partial second order derivatives in x and y directions respectively while *Dxy* is the mixed partial derivative in both x and y directions. The key here is to choose those points that maximize the determinant of hessian matrix as it eliminates points with small second derivatives in a single direction. The partial derivatives are calculated by taking the differences of neighboring sample points. The hessian operator is already normalized for scale since this operation is performed on difference of gaussian images.

$$\mathbf{H} = \begin{bmatrix} Dxx & Dxy \\ Dxy & Dyy \end{bmatrix}$$

We know that Eigen values of **H** are proportional to principal curvatures. The idea here is to minimize the ratio of principal curvatures. Let α and β be the Eigen values of H. Consider α as the larger Eigen value (larger curvature) such that, $\alpha = r\beta$. The ratio r decreases as the two Eigen values get closer and is at its minimum when both Eigen values are equal. This would mean that ratio of principal curvatures is minimum. Without computing Eigen values, sample points at edges can be tested for this property by computing determinant and trace of the hessian matrix. The determinant and trace of hessian matrix can be written as follows.

$$\mathrm{Tr}(\mathbf{H}) = Dxx + Dyy = \alpha + \beta$$

$$\mathrm{Det}(\mathbf{H}) = DxxDyy - (Dxy)^2 = \alpha\beta$$

(10)

Substituting $\alpha = r\beta$ in the equation.10 we get,

$$\frac{\mathrm{Tr}(\mathbf{H})^2}{\mathrm{Det}(\mathbf{H})} = \frac{(\alpha + \beta)^2}{\alpha\beta} = \frac{(r\beta + \beta)^2}{r\beta^2} = \frac{(r+1)^2}{r}$$

(11)

The ratio, $\frac{(r+1)^2}{r}$ is minimal when r is minimal and this happens when two principal curvatures are close to each other. Hence to keep a check on the ratio of principal curvatures, we can just use the following condition.

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

(12)

D.Lowe uses value r = 10 in the experiments [4] conducted in his research. Thus, the sample points whose ratio of principal curvatures is greater than 10 are eliminated. Figure.2.7. shows how blob detection varies when different edge thresholds are applied. We can see that for smaller values of r, the hessian operator detects blobs which has close to perfect curvatures and discards other features. In Figure.2.6 the number of keypoints reduces to 536 after applying edge threshold which is shown in the image (d).

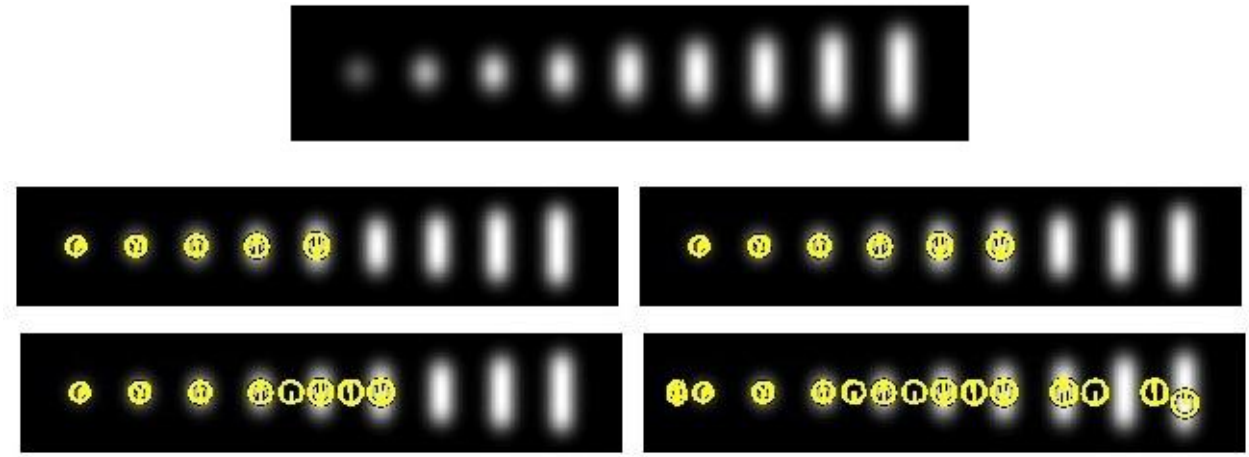

Figure.2.7- Detection of frames of increasing edge threshold- Source [26]
Input image [Topmost], Images [From top left] with applied Edge threshold r = {3.5, 5, 7.5, 10}

2.3. Orientation assignment

Now we have detected and refined keypoints in the image with scale and location assigned to it. The next step is to study the local region around the keypoint to assign orientation to it such that descriptors evaluated are relative to this orientation and hence invariant to rotation of images. Gaussian blurred images with the closest scale to the Keypoint`s interpolated scale is chosen to assign keypoint orientation in a scale-invariant manner. Let us Consider *L(x, y)* as the sample point of a gaussian blurred image at a given scale. The gradient magnitude and the direction of all the sample points in the keypoint region are pre-computed using the following equation.

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \tan^{-1}((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)))$$

(13)

Where *L(x+1, y) – L(x-1, y)* and *L(x, y+1) – L(x, y-1)* are simple neighboring pixel differences in the discrete domain. Using the gradient magnitude and orientations of samples within a region around keypoint an orientation histogram is constructed. This orientation histogram is divided into 36 bins covering 360 degrees which means that each bin corresponds to 10 degree of rotation. The local orientations are weighted by their gradient magnitude and also by a gaussian-weighted window that has σ which is 1.5 times the scale of the keypoint in order to accommodate for the distance of the samples in the region from the keypoint. These weighted values contribute to the bins in the histogram in their corresponding directions.

The highest peak in the orientation histogram corresponds to the dominant direction of the local gradients. There can be multiple dominant directions to a keypoint. Other Peaks in the histogram that have magnitudes greater than or equal to 80% of the highest peak are also regarded as dominant directions and a separate descriptors is created. Thus a particular keypoint may have multiple descriptors at the same sample location and D.Lowe points out that this improves stability of matching significantly [4]. The discrete position of the peak orientation is also interpolated by fitting a quadratic (Parabola) function to three of its closest peaks to get a more accurate peak position. Figure.2.8 shows how dominant orientation is calculated from magnitude and gradient values of the local sample points. The values are weighted by a gaussian circular window which is indicated by an overlaid circle in the figure. The blue line indicates the peak orientation and the green line indicates the second peak

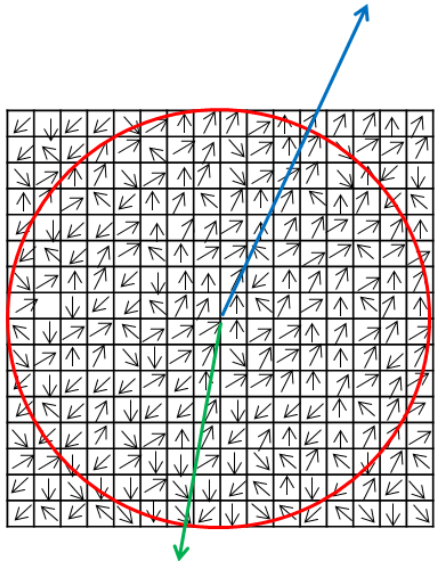

Figure.2.8. Keypoint orientation assignment

2.4. Keypoint Descriptor generation

Currently we have calculated the scale, spatial location and orientation of the keypoints. The next stage is to capture local image properties around the keypoint since these local features add distinctiveness to the descriptors evaluated and improves accuracy of matching. The local features also need to be evaluated in an invariant manner like rest of the parameters to be robust.

Lowe [4] proposes to build local histograms in the keypoint region in a similar fashion as the previous step. Again here the gradient magnitude and orientation of samples around keypoint are pre-computed on the gaussian image with closest scale to the keypoint`s interpolated scale. The gradient magnitudes are weighted by a gaussian weighing function with σ equal to 1.5 times the width of the sampling window so that weights assigned decrease smoothly as the distance of samples increase from the center. But here the region around the keypoint is selected differently. To achieve rotational invariance, the co-ordinates that are sampled are rotated relative to the dominant orientation of the keypoint and gradient orientations are calculated relative to this orientation.

A 16x16 array (Empirically established by D.Lowe) of samples are selected around keypoint rotated in the direction of keypoint orientation. This 16x16 array is subdivided into 4x4 sample regions which form the descriptors. Within each of these sub-blocks orientation histograms are computed with 8 bins each separated by 45 degree of rotation. These 8 histogram values from each 4x4 regions form the 128 element feature vector used to build descriptors. This feature vector is normalized to unit length in order to make it invariant to illumination changes in different images. Thresholds are applied to feature vectors to tackle large changes in relative gradient magnitudes caused by non-linear illumination changes, hence laying more emphasis on distribution of orientations rather than matching magnitudes of large gradients. Figure.2.9. below demonstrates how keypoint descriptors are generated.

2.5. Object recognition

In this section we very briefly discuss how scale invariant features extracted from SIFT can be used in applications like Image recognition. First, features are extracted from training images and are stored in a database. Then features calculated in the test image are individually matched with the features in the database. To match features, second nearest neighbor (2-NN) algorithm is used. Nearest neighbor is identified as the keypoint that has minimum Euclidean distance for the feature vector. A second nearest neighbor is identified and the distances of both these neighbors are compared.

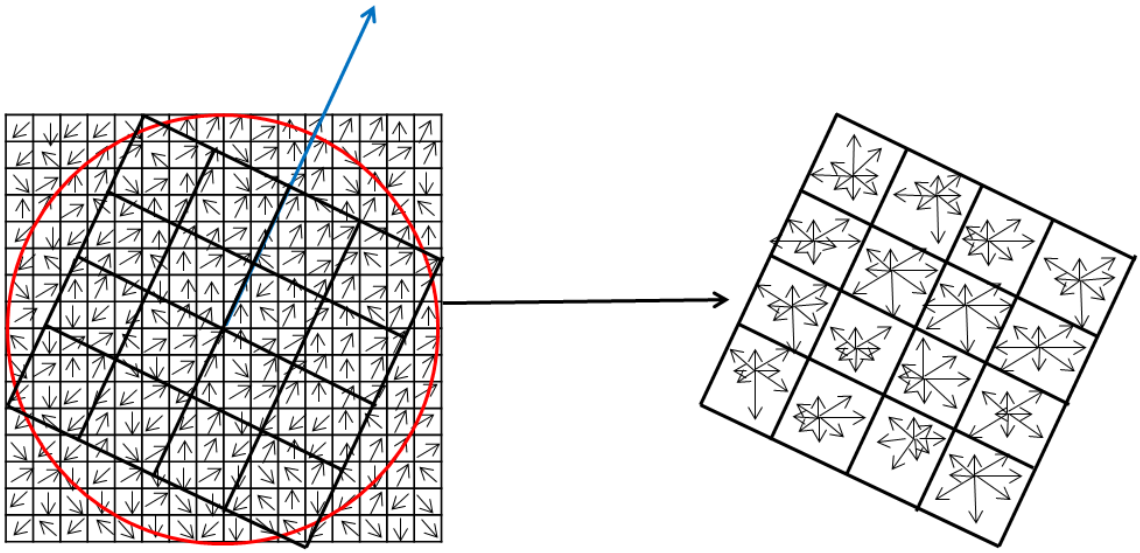

Figure.2.9. Descriptor generation – 128 vector elements

The ratio of the distance of the closest neighbor to the second closest should be significantly high (> 0.8) to reject incorrect matching. A less expensive approach called the Best Bin First (BBF) is used to find nearest neighbors based on Euclidian distance. These steps discard many of the false matches that occur because of background clutter. When identifying an object, multiple features that belong to same object should match. This is realized by using a technique called Hough transform where clusters of features vote for the same pose of an object if there is a correct match. If there are at least 3 votes accumulated for the same pose of an object from different clusters of features then the probability that it is a correct match is high and will be considered as a potential candidate for object matching. Then each of these clusters is evaluated by a geometric verification procedure where least-square solutions for the best estimated affine projection parameters relating the training image to the test image is obtained. Each match should agree within half the error range that was used for parameters in the Hough transform bins. Any other image features that agree with pose of image are kept while outliers are discarded. After discarding outliers if less than 3 points remain then the match is rejected. Figure.2.10. shows an example of object recognition application performed using SIFT.

Figure.2.10. Object recognition using SIFT – Source [4]

3. Architecture

In this section we are going to briefly discuss the experimental setup used for implementing SIFT on EVP. Also we propose a dataflow for SIFT if we were to implement it in a heterogeneous environment.

3.1. Experimental setup

The implementation of SIFT on EVP is based on a popular implementation provided by VLFEAT [26]. The SIFT implementation from VLFEAT is nearly accurate [26] and is largely compatible with Lowe`s implementation. Hence in our experiments all the results are evaluated keeping VLFEAT as the reference.

The whole implementation of SIFT is done in EVP-C programming language in the eclipse SDK with EVP libraries. There are two setups for simulating EVP implementation. In one of the setup (VCOM) the compiler generates EVP instructions that can run on actual EVP hardware. In the other setup (VECOM) the compiler converts EVP intrinsics to host machine instruction set and thus the code can be run on the host machine which in this case is X86.

In our experiments we use the VECOM setup to check the functional correctness of SIFT algorithm since the simulations can run faster and also we can conceptually simulate multi-core scenario. The code on VECOM can be easily ported to VCOM setup to run the algorithms on EVP hardware. The VCOM setup is used in our experiments to get compiler schedules and also to calculate the number of cycles taken to run computationally intensive algorithms.

Memory configuration

The data memory on EVP is very limited. This is because EVP is designed such that they are suitable for modem applications which do not require huge data memory. While simulating in VECOM we need not worry about this. In the implementation, the input image, gaussian pyramid, difference-of-gaussian pyramid, gradient pyramid, keypoint locations etc. are stored as static arrays in the data memory. The size of these arrays depends on the image size which is configurable during the compilation time. These arrays are essentially 2 dimensional arrays of vector elements with each vector containing 16 pixels (explained in detail in section 4.1). For the SIFT implementation to be able to run on actual EVP hardware the data memory should at the very least be able to store all the values for the first octave (biggest). Then the same memory can be re-used for other octaves. The implementation can also be changed to dynamic memory allocation scheme but the memory allocation should be aligned to vector boundaries.

Data type conversion

In our experiments "box" test image is used provided by Lowe [27] which is of dimension 324x223 and 8BPC grayscale image. Experiments were done to determine the necessary precision required for data-types. In our implementation we convert the input data-type (8 bit *unsigned char*) into either 16 bit floating-point data-type or 16 bit fixed point data-type. In case of floating point implementation we use two different standards for the experiments. One is the 8.7 representation supported in the first generation of EVP and the other is the 10.5 representation supported in the 2nd generation and both are as per IEEE standard 754-1985. In the floating point implementation the input data is represented in the range of [0:255]. EVP provides intrinsics that can change integer numbers to floating point numbers.

In case of fixed point representation we experiment with two formats which are explained below.

- First Format: 14 bits for input data, 1 bit for sign, 1 bit for overflow. Shift left input by 6 bits.
- Second Format: 15 bits for input data, 1 bit for overflow. Shift left input by 7 bits.

For fixed point implementation, the input data is represented in the range of [0:1] and is scaled (shifting bits) accordingly to convert it into one of the formats. For second format the assumption is that the input data is always positive (pixel values). Hence, instead of performing signed fixed point operations we can perform unsigned integer

operations to get the same result. The advantage in the second format is that we get 1 bit extra to represent input data and thus can improve accuracy of results.

16/32 general purpose registers

Also in our experiments we consider 2 different EVP architectures; one with 16 general purpose registers (1st generation) and the other with 32 general purpose registers (2nd generation). The experiment is done to find the performance of algorithms and how it is affected by register count.

3.2. Data flow

The SIFT implementation on EVP is broken down in to two modules: one is the application module and the other is the kernel module (explained in more detail later in section 4.1). This is done to essentially partition those algorithms (kernels) that exhibit data level parallelism from the sequential algorithms. In a multi-core environment, the kernels can be off-loaded to EVP to exploit this data level parallelism and improve performance. Figure.3.1. gives a pictorial description of partitioning the algorithm for heterogeneous implementation. The data flow proposed is similar to some of the proposals made previously for GPU implementation [20, 21].

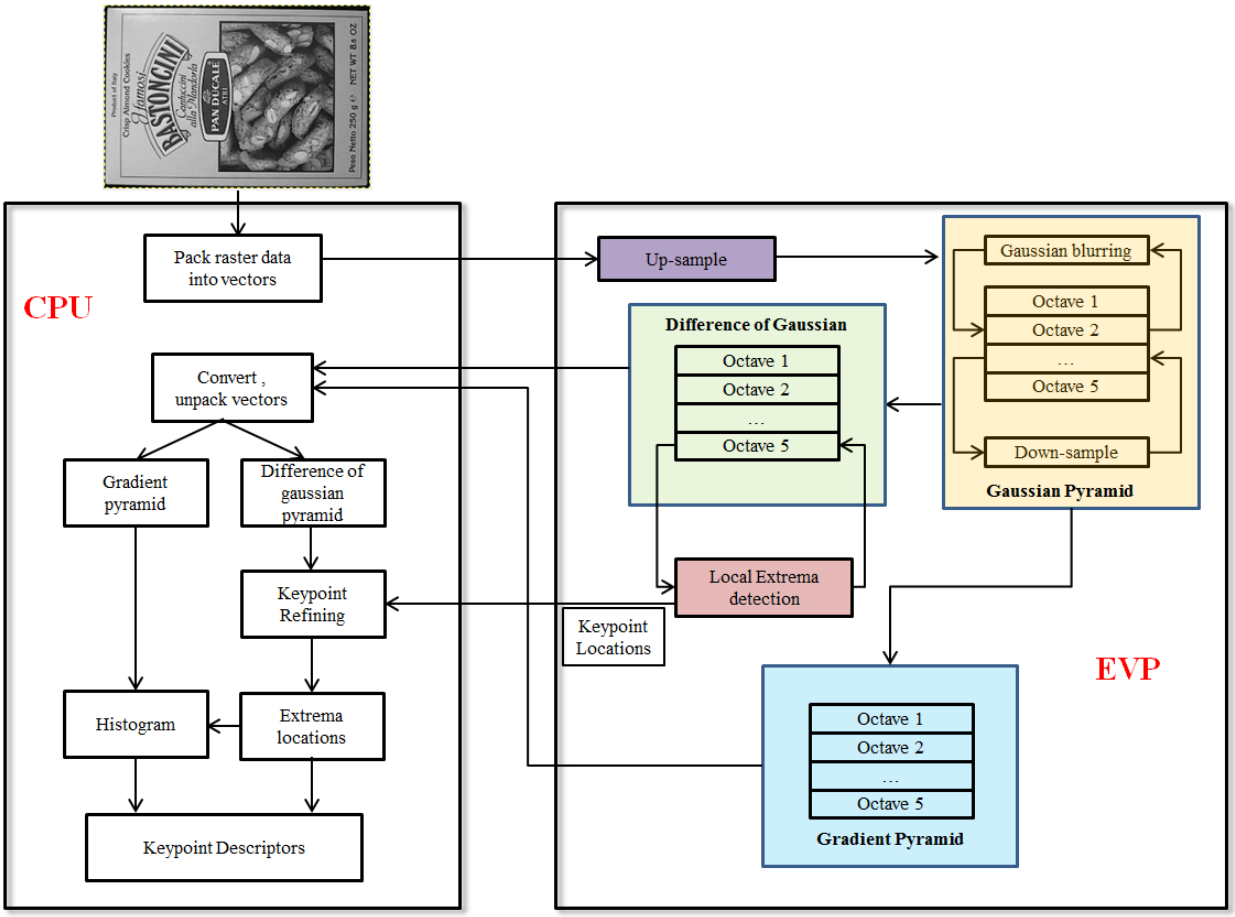

Figure.3.1 Proposed algorithm partitioning for SIFT in multi-core environment

High degree of data level parallelism can be observed in the stages of SIFT such as up-sampling, down-sampling, gaussian blurring, gradient pyramid construction and extrema detection. Hence these algorithms are implemented on EVP. Once the potential extrema points are extracted from the difference of gaussian images the algorithm essentially becomes sequential. The keypoint refinement, histogram construction and descriptor generation are

performed only at locations of interest which are random. Vectorizing these operations is challenging and can be very inefficient on EVP. One of the main differences compared to previous GPU implementations [20] [21] is that we also implement local extrema detection stage on EVP.

The main bottleneck in this implementation is the memory read-back. From the figure.3.1 we can see that difference of gaussian pyramid and the gradient pyramid needs to be read back from EVP memory. This can cost performance significantly. But for GPU implementations it is shown previously [21] that read back overhead together with GPU processing time in less than CPU processing time for stages such as gaussian pyramid and gradient pyramid. This is one the main motivations for the proposed data flow.

In this thesis we implement SIFT only till the keypoint refinement stage. The keypoint refinement stage itself is implemented on host machine(X86) while the rest of the stages are implemented for EVP as in the figure.3.1.

4. Mapping SIFT on EVP – Problems and Approaches

The implementation of SIFT on EVP can be broken down into two main modules as introduced in section.3.2. One is the application module and the other is the kernel module. Following sections give a detailed description of the two modules and how they encompass different stages of SIFT.

4.1. Application

Application module is nothing but the main routine of the SIFT algorithm. The application routine is responsible for implementing the whole functionality of SIFT. The routine internally calls many subroutines from the kernel module which implements computationally intensive code of SIFT. The application module can be run on a host processor in a multi-core environment while the kernel routines can be off loaded on to EVP to achieve speed up.

The first main task in the application module is to read pixel data from an input image into EVP data memory. For performing SIFT we use grayscale images with PGM file format. The color depth of the images used in the experiments is 8 bits. Before extracting raster data from PGM images the header needs to be parsed to get information of the image parameters like width (W), Height (H) and the maximum value of the pixel (same as BPP). The most commonly used PGM format consists of 4 lines of header followed by the raster data stored as binary values (*unsigned Char* type). In our experiments we use grayscale images of maximum pixel value 255 (8BPP), width 324 and height 223. Figure.4.1 shows how a PGM file looks like. The raster data is represented in ASCII only for the purpose of understanding here since text editors cannot represent binary raster values. The first line in the file consists of a "magic number" which identifies the file type and is made up of two characters 'P5'. An optional comment may follow after the first line. In the third line we can parse the width and height of the image separated by a whitespace. The fourth line provides information about the maximum value of each pixel. After this the raster data follows which are pixel values (binary) starting from top left of the image to bottom right traversed row by row.

```
P5
# CREATOR: GIMP PNM FilterVersion 1.1
324 223
255
21 18 25 22 22 21 23 23 22 26 25 24 28
28 25 26 23 21 23 25 28 24 23
24 26 25 24 28 28 25 26 23
26 25 24 28 28 25 26 23 26 24 30 25 29
28 25 27 28 27 27 27 29 255
255 33 255...........
.....................................
.....................................
```

Figure.4.1. Example PGM file format for grayscale image of size 324x223 pixels

Using the above information we parse the header to get the required information about the parameters and then start storing raster values on EVP memory. Although the pixel values are discrete ranging from 0-255 integer values, the transformations performed on pixels in SIFT are not necessarily integer operations. Most of the computations performed in SIFT generate values that have a fractional part and thus we require changing the input data type to floating/fixed point data types on the host machine. Hence more bits are needed to represent the values and in our implementation we convert 8 bit integer data types of the input to 16 bit data types before starting the transformations. The conversion of data type is performed in the up-sampling stage which is explained later.

Let us consider that we have an input image of size 324x223 pixels. The image can be regarded as a 2D array of pixels having rows and columns. Now, after parsing the PGM file header, each pixel data of 8 bits are read from a file row wise and are converted to 16 bit data types. With a 256 bit SIMD width on EVP we can store sixteen pixels in one vector. Figure.4.2 shows how the 2D array of image pixels is mapped on to EVP memory.

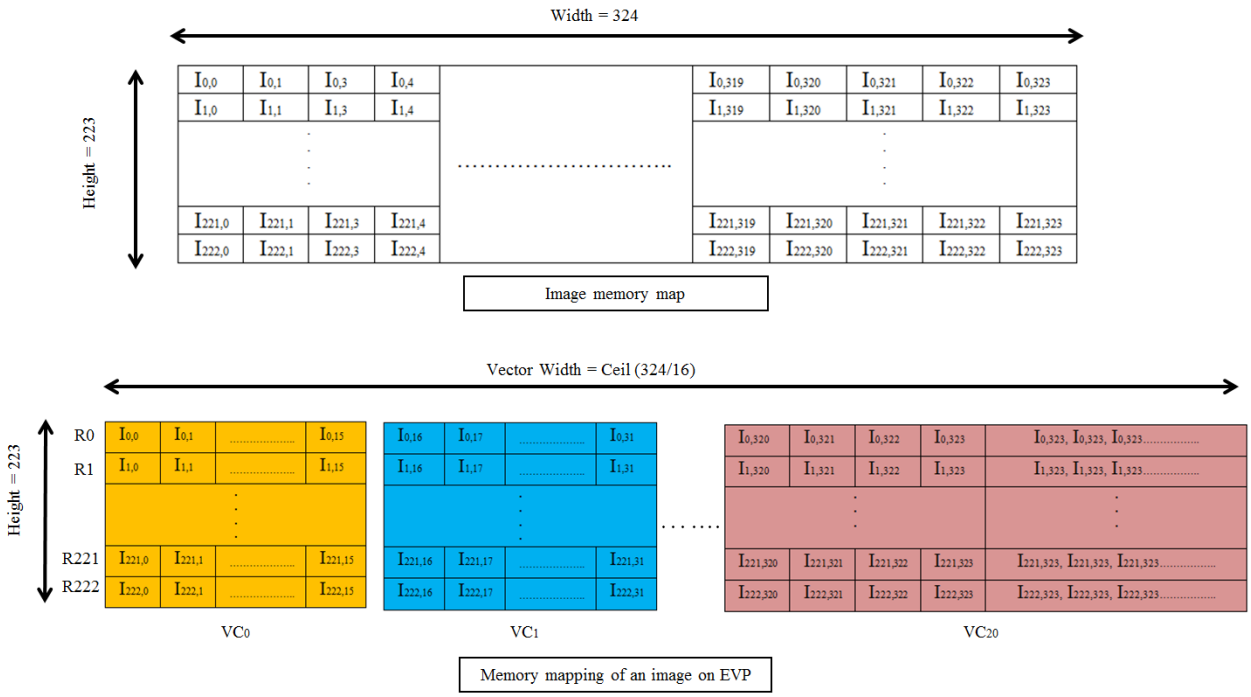

Figure.4.2. Storing an image on EVP memory

A 2D array of pixels is re-represented as a 2D array of vector elements with each vector having 16 pixels. From Figure.4.2 let us take an example of the row R0. The row R0 will contain pixels from $I_{0,0}$:$I_{0,323}$. However on EVP, R0 is represented by 21 vectors VC_0:VC_{20} where VC_0 is made up of pixels $I_{0,0}$:$I_{0,15}$, VC_1 is made up of pixels $I_{0,16}$:$I_{0,31}$ and so on. Here one important point to be noticed is that the image width 324 is not a multiple of 16. Thus the row does not fit exactly in to vector boundaries. Although 21 vectors can accommodate 336 pixels we only have 324 pixels per row from the input image. Hence we need to pad the rest of the 12 elements in the vector VC_{20} with the last row element $I_{i,323}$. This will help in avoiding any erroneous output caused due to junk values that are uninitialized beyond row boundary. Likewise all the other rows are stored in EVP memory in a 2D array of vectors. After storing the image in the EVP memory, we start performing transformations on it.

4.2. Kernels

Most of the image transformations are implemented as routines in the kernel module. The routines in the kernel module are designed such that they have very little control logic and can exploit data level parallelism to the fullest. Also various optimization techniques are employed in each routine to obtain significant throughput. We observe the possibility of exploiting data level parallelism in transformations such as up-sampling, down-sampling, difference-of-gaussian, extrema detection, and gaussian and gradient pyramid constructions as discussed earlier in section.3.2. In this section we discuss implementation of each of these operations on EVP in detail.

4.2.1. Up-sampling

According to the theory explained in section 2.1.1, we need to up-sample the input image to double its size in order to create the base image for the first octave. This is achieved by applying a simple linear interpolation technique where a pixel is created between two consecutive pixels in X and Y directions by taking the average of their pixel

values. For understanding let us consider an image of size 3x3. Figure 4.3 shows how the corresponding up-sampled image of size 6x6 is obtained using linear interpolation. Note that before we perform linear interpolation the data types of the input image are converted to either floating/fixed point data types as discussed in section.3.1. EVP provides intrinsics to convert integer data type to floating point data types. For converting to fixed point data types we may scale the input data by shifting the bits to left by 6/7 bits. The up-sampling is done only once on the input image, hence we have an opportunity to merge data conversion in the same step.

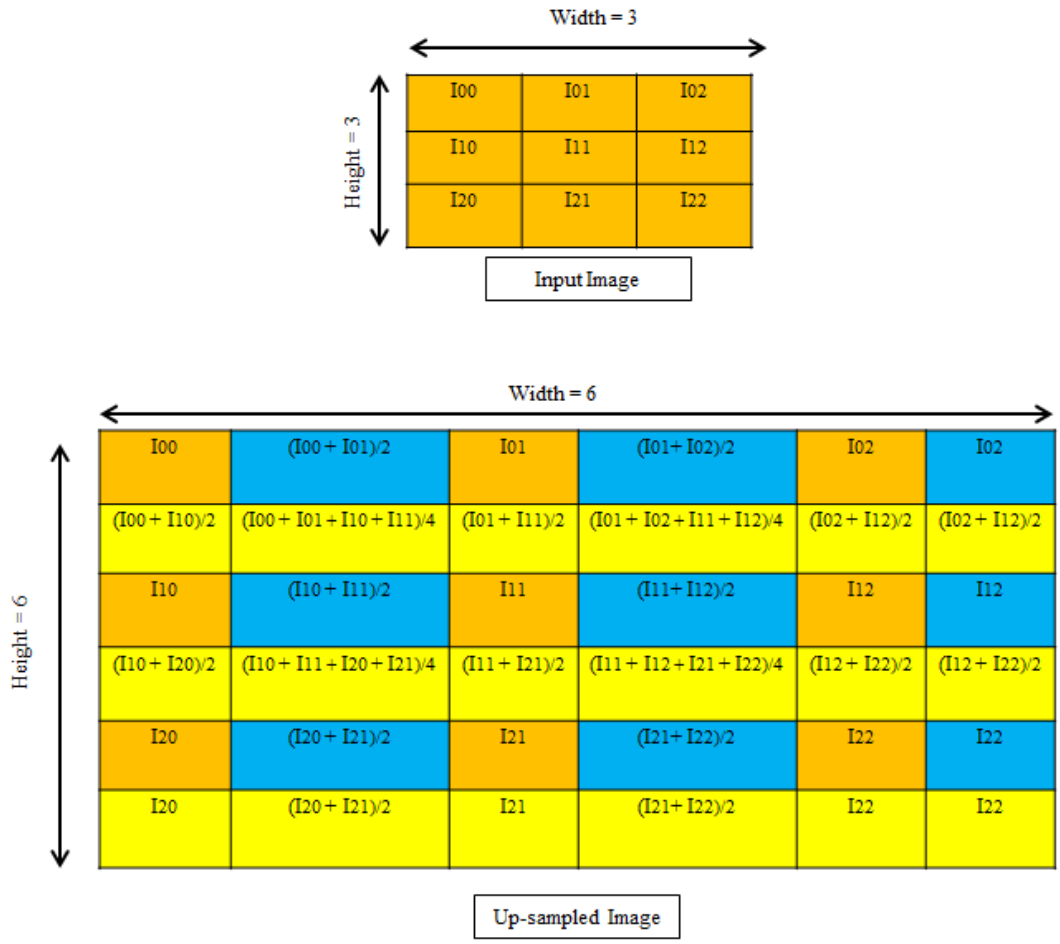

Figure.4.3. Linear interpolation to up-sample an image of size 3x3

Interpolation is performed sequentially first in X direction and then in Y direction or vice versa. The pixels in blue and yellow colors in the figure.4.3 are the interpolated values in X and Y directions respectively. Notice here that the pixels in the last two columns and the last two rows are repeated. This is because of the fact that there are no pixels beyond boundaries in both directions. Thus the nearest pixel values are repeated at the image boundaries.

Now let us see how linear interpolation is performed on EVP. The image used in our experiments is of size 324x223. On up-sampling this image we need to have an image of size 648x446. Currently we have represented the input image as an array of vectors and we needed 21 vectors to represent each row. In case of up-sampled image we need not necessarily require 42 vectors for each row. A row of size 648 can be still stored as 41 vectors (Ceil(648/16)). Hence up-sampling does not necessarily mean doubling the number of vectors in a row on EVP.

Now let us consider a column vector VC_j on which up-sampling is performed. Figure.4.4 illustrates how this can be performed on EVP. First the interpolation is done in X direction. A vector of 16 elements results in 32 elements in

an up-sampled image. First the vector under consideration VC_j is multiplied by half. Then a shifted version of the same vector with one pixel offset in the X direction is obtained by taking first element of next column vector VC_{j+1} and inserting it to the end of vector VC_j. This can be achieved either by using unaligned access supported by EVP or through mask and shuffle operations to merge the two vectors. This new vector is again multiplied by half and then it is added with VC_j to get the interpolated values X1: X16. Thus we have 16 elements of actual input vector and another vector of 16 interpolated values. Using two shuffle operations in combination with masks we can merge these two vectors to get two up-sampled output vectors.

Interpolation in Y direction is straight forward. The average of rows R_i and R_{i+1} are taken to compute vector of interpolated values in Y direction. This is repeated for all rows one after the other. The vectors in blue and yellow are the interpolated vectors in X and Y directions respectively.

NOTE: The implementation may vary for fixed point implementations. In this case we use an intrinsic supported on EVP that averages two input vectors in a single instruction.

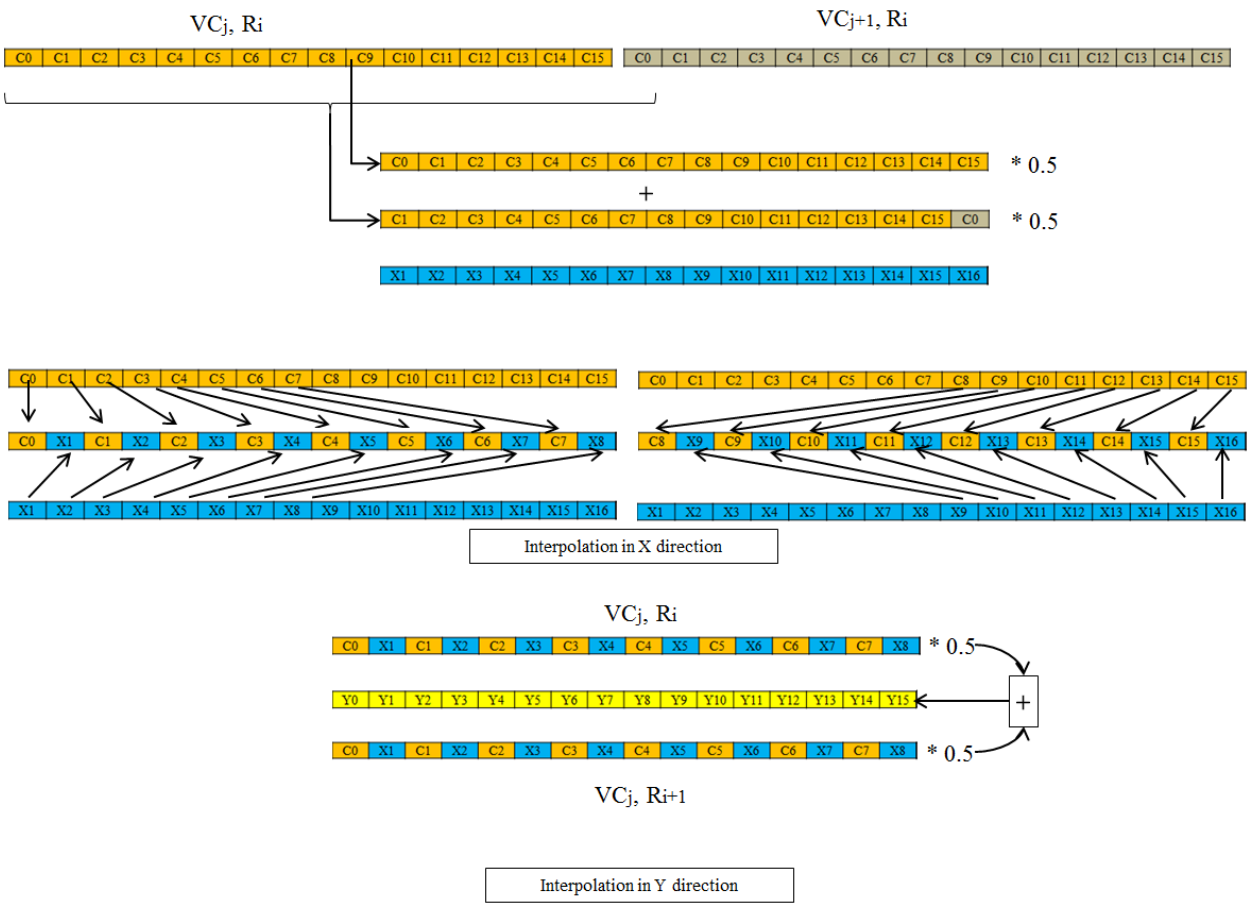

Figure.4.4. Linear interpolation on EVP

Here, the rightmost column vectors and the vectors of the last row need to be handled differently. The rightmost column vector is up-sampled separately from the rest of the column vectors. Figure.4.5 describes how rightmost column vector is handled. In the section.4.1 we had seen how the last column vector was stored when the row size does not fit in to the vector boundary by padding the rest of the uninitialized elements with the last row pixel. This will help us in up-sampling. Figure.4.5 shows the last column vector for image of width 324. In the vector VC_{20} we have only four elements C0:C3 and the rest are set to C3. Now we interpolate this vector with a zero padded vector

in place of VC_{j+1} in a similar fashion as discussed earlier. The output image has a width of 648 and each row will have 41 (Ceil (648/16)) column vectors. Also we require just 8 elements of the interpolated output to be stored in the up-sampled image. Thus we discard unnecessary samples as shown in the figure 4.5. Also in the up-sampled image we selectively pad the last column vector with zeroes wherever the elements are uninitialized. This is useful when we perform gaussian blurring.

Note that, if the image size perfectly fits into vector boundaries then we do not require padding of the last column vector. To get the last row pixel in this case we use a vector consisting of only last row element in place of vector of zeroes (VC_{j+1}) as in figure.4.5 and then perform interpolation. In this case we store all 32 elements.

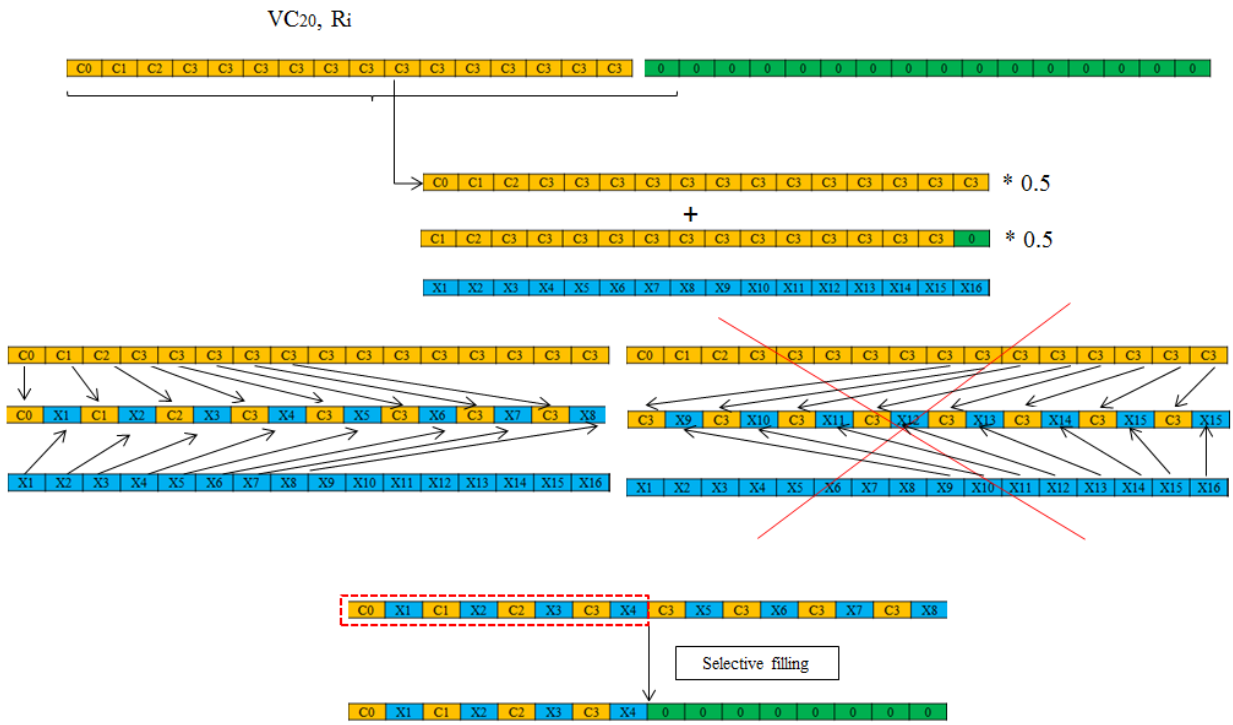

Figure.4.5. Handling edge vectors of a row during up-sampling

Handling the last row vectors (R_{H-1}) is simple. The vectors of penultimate row (R_{H-2}) are copied and stored as last vector (R_{H-1}). The pseudo code for this approach can be seen below.

```
vw = ceil(W/P);    //Number of vectors in a row for input image
ow = ceil(2*W/P);  //Number of vectors in a row for output image
h = H;             //Height of the input image
P = 16;            //Number of elements per vector

/** X direction */
for(i = 0; i < h; i++)
{
      C1 = I[i][j*P : j*P + P - 1]; // load a vector from input image
      C1 = convert(C1) // Convert to fixed or floating point data type
      T1 = 0.5 .* C1;

      for(j = 0; j < vw; j++)
      {
            C2 = (j < vw -1) ? I[i][(j+1)*P : (j+1)*P + P - 1] : vbcst(0);
```

```
            C2 = convert(C2);

            T2 = 0.5 .* C2;
            T3 = (T1[1:15],T2[0]);
            T3 = T1 + T3;

            U1 = (C1[0],T3[0],C1[1],T3[1], C1[2],T3[2], C1[3],T3[3],
                  C1[4],T3[4],C1[5],T3[5], C1[6],T3[6], C1[7],T3[7]);
            U2 = (C1[8],T3[8],C1[9],T3[9], C1[10],T3[10], C1[11],T3[11],
                  C1[12],T3[12],C1[13],T3[13], C1[14],T3[14], C1[15],T3[15]);

            C1 = C2;
            T1 = T2;

            //Store up-sampled rows to output vector
            O[2*i][(2*j)*P : (2*j)*P + P - 1] = U1;
            O[2*i][(2*j+1)*P:(2*j+1)*P + P - 1] = U2;
      }
}

/** Y direction */
for(i = 0 ; i < h; i++)
{
      for(j=0; j < ow; j++)
      {
            R1 = O[2*i][j*P : j*P + P - 1];
            R2 = (i < h-1)? O[2*(i+1)][j*P : j*P + P - 1] : R1;

            T1 = R1 .* 0.5;
            T2 = R2 .* 0.5;
            T3 = T1 + T2;

            O[2*i+1][j*P : j*P + P - 1] = T3;
      }
}
```

4.2.2. Gaussian blurring

In this section we discuss how Gaussian convolution algorithm is mapped on EVP. We decided to use 1.6 as the initial σ value for building the gaussian pyramid as suggested by Lowe in his paper [4]. In section 2.1 we have seen how each octave is built. The blurring factor for a particular layer (except first) in an octave is calculated by multiplying the σ value used in the previous layer by a constant multiplicative factor *K*. Based on the σ value, the gaussian kernel requires $\lceil 6\sigma \rceil + 1$ coefficients to be convolved with images, which is a width of 3σ on either side of the peak in the distribution. Note here that we define kernel width as the number coefficients present on any one of the sides of the central coefficient. As we can notice, with increase in σ value the width of the gaussian kernel increases. This means that for higher levels of blurring the width of gaussian kernel is higher. This kernel width can get to very large values for higher scales and convolution of images with kernels of large widths can impact the performance greatly. Instead of calculating each layer of the octave from the base image we use a technique called recursive gaussian blurring method which is more efficient on EVP. The idea behind recursive method is, gaussian blur with a factor σ_a is equivalent to applying gaussian blur with factors σ_b and σ_c sequentially if $\sigma_a^2 = \sigma_b^2 + \sigma_c^2$ [20]. Using this logic, consider we needed to build a particular layer in an octave with a scale factor σ_1 and the scale factor used in the previous layer was σ_0. The gaussian blurred image of scale $\sigma_1$can be obtained by applying an additional amount of blur to the previous blurred image of scale σ_0. This additional amount of blur can be calculated as $\Delta\sigma = \sqrt{\sigma_1^2 - \sigma_0^2}$. The value of $\Delta\sigma$ is relatively lower compared to individual scale factors. Hence going by the

thumb rule $\lceil 6\sigma \rceil + 1$,the kernel width required to apply additional blur of $\Delta\sigma$ is relatively less. This improves efficiency as the number of memory accesses and computations reduce when applying blurring. Also, the calculated $\Delta\sigma$ value for a particular level is same for all octaves with respect to their base images. Hence calculating these $\Delta\sigma$ values and their corresponding kernel coefficients for the first octave is enough and can be re-used for other octaves. In our implementation we use a width of 4σ on either side of the gaussian peak as implemented by VLFEAT [25]. This can improve accuracy although if performance is the criteria we may use 3σ.

A gaussian kernel calculated to convolve with an image is a 2D array of coefficients which are symmetrical. Blurred images are obtained by moving the kernel across the image, 1 pixel at time. Consider the kernels as an overlay on the image centered on the pixel of interest (in red) as shown in figure 4.6. The calculated value of the output pixel is the weighted sum of values of the center pixel and its neighbors in the input image.

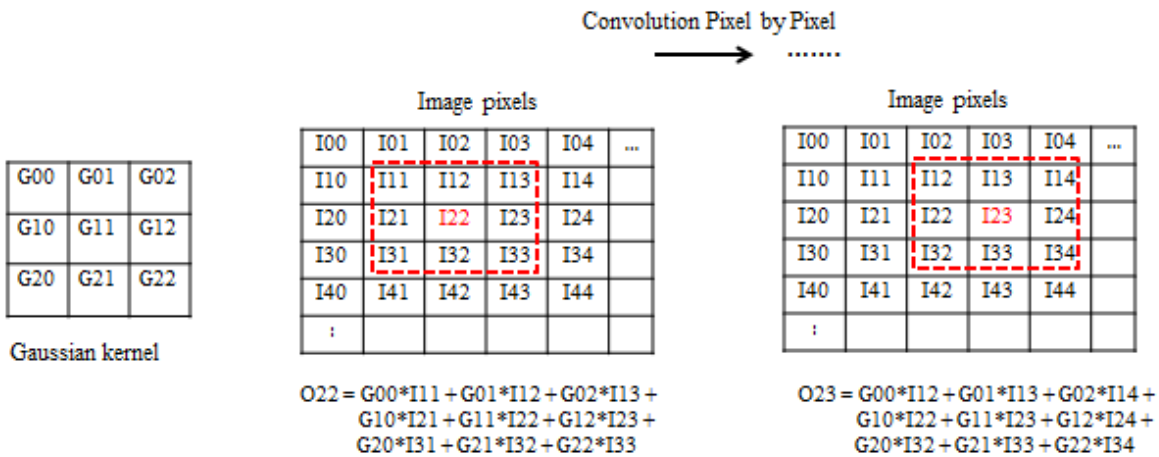

Figure.4.6. Convolution of gaussian kernel with an image

The coefficients of the gaussian kernel are symmetric in nature (like its distribution) and this can be exploited to improve the performance of convolution with kernels of smaller width. We know that 2D gaussian kernels are a combination of kernels in x and y directions. We can split 2D gaussian kernels into two 1D kernels and convolve sequentially in x and y directions with individual kernels. Figure4.7. shows an example 2D gaussian kernel of width = 2 split into 1D kernels for each direction. There are many advantages from doing this. Firstly, lesser number of coefficients needs to be stored and the coefficients are same for both directions. Also, since individual kernels are symmetric we can try to exploit symmetry in each direction with lesser states to maintain. For the example kernel in figure.4.7 we would need to store only values {1, 4, 6}. The coefficients of the kernel are normalized to range [0:1] before applying it on an image. This ensures that the output values stay within the pixel value range [0:255] for an 8 BPP image.

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} = \underbrace{\frac{1}{256}\begin{bmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{bmatrix}}_{\mathbf{Gy}} * \underbrace{\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix}}_{\mathbf{Gx}}$$

Figure.4.7. 2D gaussian kernel to 1D kernel

Since we know the initial σ value, the width of the kernels required for each layer can be pre computed using the $\Delta\sigma$ value. As stated earlier, these $\Delta\sigma$ values are same for all octaves for respective layers; hence we would require only a finite number of gaussian kernels of known widths that are used throughout. This is a very important observation that is used for constructing gaussian pyramid on EVP. Implementing a generic gaussian convolution function on EVP that can handle variable kernel widths is inefficient because of the control logic (Conditionals, branches, dynamic allocations) that needs to be built in. Instead we decided to have separate implementations for each kernel width. The advantage behind this approach is that we can design and optimize each kernel on need basis. For example, in our implementation we use gaussian kernels of width 5, 7,8,10 and 13. A kernel of width 5 means 11 coefficients each in a 1D kernel of x and y direction or an 11x11 matrix of coefficients in a 2D kernel.

In the following sections, we are going to discuss how gaussian kernels are implemented on EVP. Like mentioned earlier, we decided to perform blurring by applying convolution in x and y directions one after the other. This will help us in exploiting symmetry and improving performance.

4.2.2.1. Convolution in Y direction

In this section we discuss two different approaches by considering an example kernel of width 5. The logic behind these approaches can be easily extended to other kernel widths and hence will not be discussed.

Approach 1

Consider a 5x gaussian kernel with coefficients {h5:h0} as shown in the Figure.4.8. A straight forward method of convolving in Y direction is by loading 11 input samples from each row centered on the sample (R_i) for which the output is computed. The samples in this case are a vector of pixel elements. Each of these samples is multiplied with the corresponding coefficients of the gaussian kernel and is accumulated to get the output for the center row. Next, another input vector is loaded as required to do the convolution for the next row (R_{i+1}). This process is repeated till the outputs for all rows are calculated. Then we traverse in X direction with one vector offset (P) and start processing all the row samples in a similar fashion. Figure.4.8 illustrates convolution in Y direction.

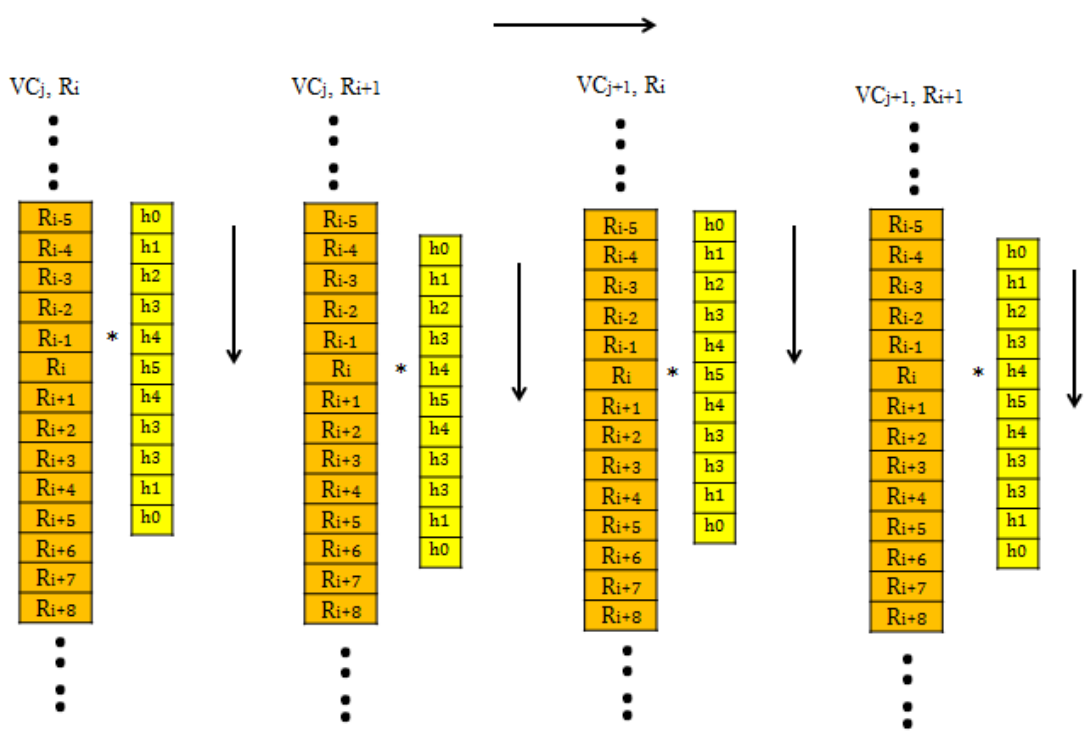

Figure.4.8. Convolution in the Y direction

The pseudo code for this approach is as follows

```
for(j = 0; j < CEIL(W/P); j++)
{
      R0[0:P-1] = vbcst(0);   //Load zeroes if no pixels are available
      R1[0:P-1] = vbcst(0);
      R2[0:P-1] = vbcst(0);
      R3[0:P-1] = vbcst(0);
      R4[0:P-1] = vbcst(0);

      R5[0:P-1]  = I[0][j*P : j*P+ P-1];
      R6[0:P-1]  = I[1][j*P : j*P+ P-1];
      R7[0:P-1]  = I[2][j*P : j*P+ P-1];
      R8[0:P-1]  = I[3][j*P : j*P+ P-1];
      R9[0:P-1]  = I[4][j*P : j*P+ P-1];
      R10[0:P-1] = I[5][j*P : j*P+ P-1];
      // Load input samples from consecutive rows

      for(i = 0; i < H ; i++)
      {
            ACC_Y = R0 .* h0;
            ACC_Y += R1 .* h1;
            ACC_Y += R2 .* h2;
            ACC_Y += R3 .* h3;
            ACC_Y += R4 .* h4;
            ACC_Y += R5 .* h5;
            ACC_Y += R6 .* h4;
            ACC_Y += R7 .* h3;
            ACC_Y += R8 .* h2;
            ACC_Y += R9 .* h1;
            ACC_Y += R10 .* h0;

            O[i][j*P : j*P+ P-1]  = ACC_Y;   // Store back the output

            R0 = R1;
            R1 = R2;
            R2 = R3;
            R3 = R4;
            R4 = R5;          // Use the loaded values for next iteration
            R5 = R6;
            R6 = R7;
            R7 = R8;
            R8 = R9;
            R9 = R10;

            R10 = (i < h-6)? I[i+1][j*P : j*P + P-1]: vbcst(0);
            // Load zeroes if no pixels are available
      }
}
```

From pseudo code we see that for each output to be computed eleven row samples (vectors) R0:R10 are required and eleven MAC operations are performed. Out of these eleven loaded row samples ten of them can be reused for the next iteration as we only traverse by one row offset in the Y direction. Thus in each iteration we require to load just one new row sample. This method can be efficient only if there are enough registers to store row samples for the next iteration. If there are no enough registers then we require loading of all eleven samples for each iteration and this can be inefficient.

NOTE: While performing convolution in Y direction special care needs to be taken when computing output for the edge pixels. This is because of the fact that there are no pixels beyond boundaries to access. Thus for 5x kernel we require to handle first and last five rows differently. This is evident from the pseudo code. The samples that are not present are replaced with zeroes and the output is computed. The first five rows are loaded with zeroes and are replaced eventually with actual row samples as we traverse in the Y direction. The last rows are conditionally loaded with the actual samples if present or zeroes depending on the distance we have traversed in the Y direction. This is also illustrated in figure.4.9.

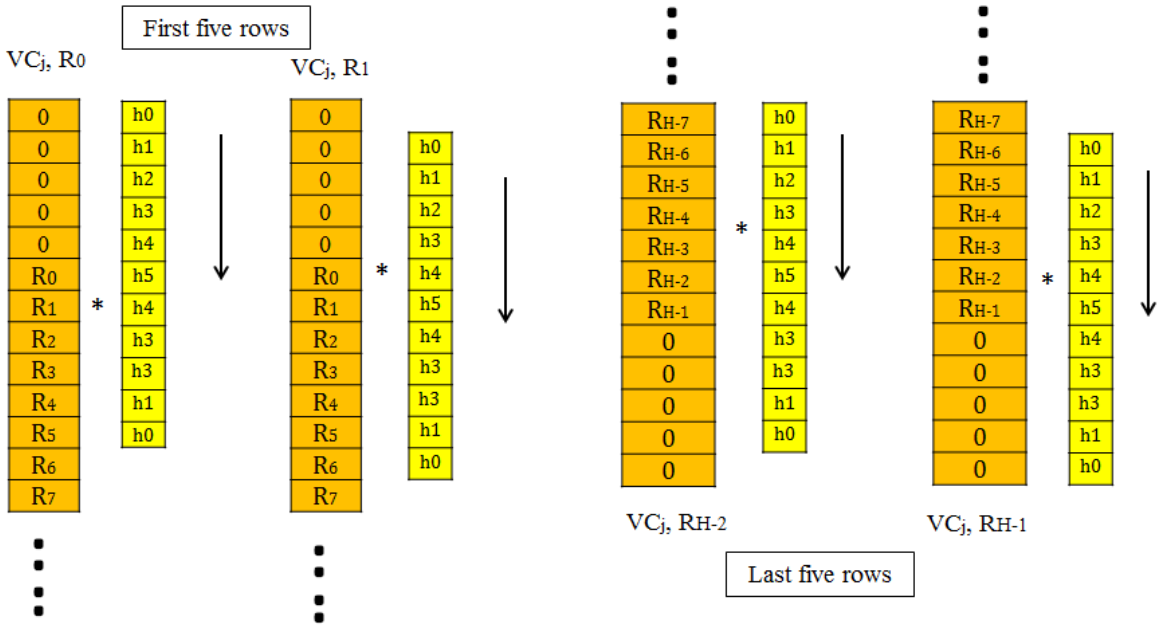

Figure.4.9. Handling edge pixels using zero padding

Approach 2

From previous approach we saw that for each output computed, 11 MAC operations were required. The load on the MAC functional unit of EVP in this case determines the performance of the kernel. Instead of multiplying the samples with 11 coefficients we devise a more optimized method where we can use the symmetry in the kernel coefficients. The rows with same offsets from central sample are multiplied with the same coefficients and accumulated. Thus we can add these rows with same offset from the center and then multiply the sum with the corresponding coefficient. For example, let us consider the first convolution on column 1 in the Figure.4.8 where rows R_{i-5} and R_{i+5} can be added and the sum can be multiplied with coefficient h0 before accumulating the result. The same applies to R_{i-4} and R_{i+4}, R_{i-3} and R_{i+3} and so on. The center sample is multiplied with the highest coefficient. This way the load on MAC unit is reduced and ALU unit can be used for addition. Sharing loads between different functional units helps in improving throughput as it reduces bottlenecks caused by a particular functional unit. Also these functional units can operate in parallel hence instruction level parallelism is also exploited. Figure.4.10. illustrates this approach.

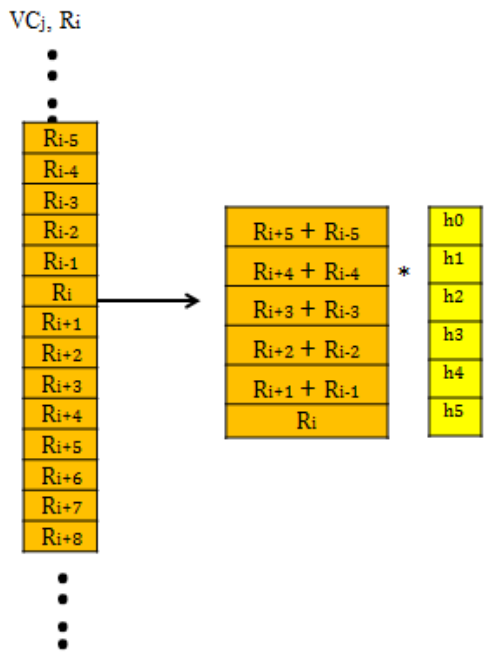

Figure.4.10. Exploiting symmetry of the kernel coefficients in Y direction

The pseudo code for this approach is as follows

```
for(j=0; j< CEIL(W/P);; j++)
{
      R0[0:P-1] = vbcst(0);   //Load zeroes if no pixels are available
      R1[0:P-1] = vbcst(0);
      R2[0:P-1] = vbcst(0);
      R3[0:P-1] = vbcst(0);
      R4[0:P-1] = vbcst(0);

      R5[0:P-1]  = I[0][j*P : j*P+ P-1];
      R6[0:P-1]  = I[1][j*P : j*P+ P-1];
      R7[0:P-1]  = I[2][j*P : j*P+ P-1];
      R8[0:P-1]  = I[3][j*P : j*P+ P-1];
      R9[0:P-1]  = I[4][j*P : j*P+ P-1];
      R10[0:P-1] = I[5][j*P : j*P+ P-1];
      // Load input samples from consecutive rows

      for(i=0; i<H ; i++)
      {
            T5 = R4 + R6;
            T4 = R3 + R7;
            T3 = R2 + R8;
            T2 = R1 + R9;
            T1 = R0 + R10;
            // Add samples that are symmetrically place

            ACC_Y = R5 .* h5;
```

```
            ACC_Y += T5 .* h4;
            ACC_Y += T4 .* h3;
            ACC_Y += T3 .* h2;
            ACC_Y += T2 .* h1;
            ACC_Y += T1 .* h0;

            O[i][j*P : j*P+ P-1]  = ACC_Y;// Store back the output

            R0 = R1;
            R1 = R2;
            R2 = R3;
            R3 = R4;
            R4 = R5;          // Use the loaded values for next iteration
            R5 = R6;
            R6 = R7;
            R7 = R8;
            R8 = R9;
            R9 = R10;

            R10 = (i < h-6)? I[i+1][j*P : j*P+ P-1]: vbcst(0);
            // Load zeroes if no pixels are available

        }
}
```

From pseudo code we see that intermediate values T1:T5 are computed from samples that are equidistant from sample R5. When compared to eleven MAC operations required in approach 1, here we need just six MAC operations. The rest is similar to the first approach.

Figure.4.11 shows how the schedules as obtained from the compiler for the innermost loop vary between the two approaches for a 5x kernel on EVP. As we can observe, the number of cycles needed for the inner loop in the first approach is determined by the number of MAC operations performed. In the second approach since we utilize the VALU, the load is shared leading to lesser MAC operations and thus the bottleneck caused by the MAC functional unit is avoided.

NOTE: Although this seems to be very promising, the second approach itself requires more registers to be active in the inner loop to store the intermediate values T1:T5. The above schedules are obtained by considering an architecture that has 32 general purpose registers. But, the first generation EVP architectures have only 16 registers. At any point in time if there are no sufficient vector registers to work with, then the values are stored back and accessed from the data memory. The access to the data memory is very costly and will hamper the performance of kernels greatly. This is called as register spilling.

Approach 1

Cycles	VLSU	VMAC	VALU	VSHU
1		ACC_Y - R1 .* h1;	R0 - R1;	
2		ACC_Y +- R2 .* h2;	R1 - R2;	R2 - R3;
3		ACC_Y +- R3 .* h3;	R3 - R4;	
4		ACC_Y +- R4 .* h4;	R4 - R5;	
5		ACC_Y +- R5 .* h5;	R5 - R6;	R6 - R7;
6		ACC_Y +- R6 .* h4;	R7 - R8;	R8 - R9;
7	R10 - (j < h-6)? I[++i][j*p:j*P+P-1]:vbcst(0);	ACC_Y +- R7 .* h3;	R9 - R10;	
8		ACC_Y +- R8 .* h2;		
9		ACC_Y +- R8 .* h1;		
10		ACC_Y +- R10 .* h0;		
11				
12	O[i][j*p:j*P+P-1] - ACC_Y;	ACC_Y +- R0 .* h0;		

Approach 2

Cycles	VLSU	VMAC	VALU	VSHU
1	R4 - R5;	ACC_Y +- T5 .* h4	R3 - R4;	R5 - R6;
2	R8 - R9;	ACC_Y +- T4 .* h3	R6 - R7;	R7 - R8;
3	R10 - (j < h-6)? I[++i][j*p:j*P+P-1]:vbcst(0);	ACC_Y +- T3 .* h2	R9 - R10;	
4		ACC_Y +- T2 .* h1	T3 - R2 + R8;	
5		ACC_Y +- T1 .* h0	T2 - R1 + R9;	
6	R0 - R1;		T1 - R0 + R10;	R1 - R2;
7	O[i][j*p:j*P+P-1] - ACC_Y;	R2 - R3;	T5 - R4 + R6;	
8		ACC_Y - R5 .* h5	T4 - R3 + R7;	

Figure.4.11. Schedules for approach 1 and approach 2- 32 registers architecture

Table.4.1 gives a comparison of the number of cycles required in the inner loop for 16 and 32 register architectures considering the second approach. From the table we can observe that the number of cycles required for computing one iteration of the inner loop increases sharply for larger kernel widths. This can be attributed to the increase in number of values that we carry over to the next iteration and also the number of intermediate values we compute in the current iteration for larger kernel sizes. Since these values cannot be kept active in the limited number of available registers there are more load/store operation performed from/to external memory. A better approach when working with 16 registers would be to reduce the number of intermediate values computed. Hence, for smaller kernel widths, the first approach can be a better option for architectures with 16 registers. For larger kernel widths, a more simplistic approach could be followed where all row samples are loaded each time instead of keeping it active in the registers for the consecutive iterations. Thus we need to also take into account architectural limitations to design efficient algorithms. In our experiments we follow the second approach for kernels of widths 5, 7, 8 and 10. The second approach becomes inefficient for kernel width 13. This is again because of register spilling. Thus for 13x gaussian kernel we follow the first approach.

For the rest of the document we implement algorithms considering architectures with 32 general purpose registers.

	Kernels	16 Register architecture	32 register architecture
Second approach	5x Gaussian	9	8
	7x Gaussian	36	11
	8x Gaussian	52	13
	10x gaussian	82	16
First approach	13x gaussian	66	28

Table.4.1. Cycle count for inner most loop - 16 versus 32 register architectures – Y direction convolution

4.2.2.2. Convolution in X direction

In this section we are going to discuss the design of an example kernel of width 5 for convolution in X direction whose logic can be extended to other kernel widths.

Figure.4.12 shows an example 5x gaussian kernel {h0:h5} where convolution is applied on a particular a row (R_i) in X direction. As we have seen earlier, each row is divided into vectors of pixel elements. Consider one such vector with 16 elements. Figure.4.12 shows how convolution needs to be performed in X direction to get outputs for samples C5 and C6. Instead of traversing one sample each in X direction it is possible to compute the output for the whole vector efficiently on EVP using a technique we call partial accumulation. Figure.4.13 demonstrates this technique. Convolution is performed on shifted versions of input samples from a particular row. To obtain shifted versions of input samples we use unaligned accesses supported by EVP since vectors are stored sequentially in the memory. But one can observe that the output samples obtained are from O5:O20. The outputs O0:O4 depend on the previous vector samples. The positions of the output samples relative to input should not be altered while performing convolution. This problem can be tackled by starting the convolution with a vector that has samples from the previous vector using unaligned access. Figure.4.14 illustrates this approach where outputs for column vector VC_j, O0:O15, are calculated using samples in the column vectors VC_{j-1} and VC_{j+1}. The outputs for VC_{j+1} are also computed in similar fashion by using samples from VC_j and VC_{j+2}.

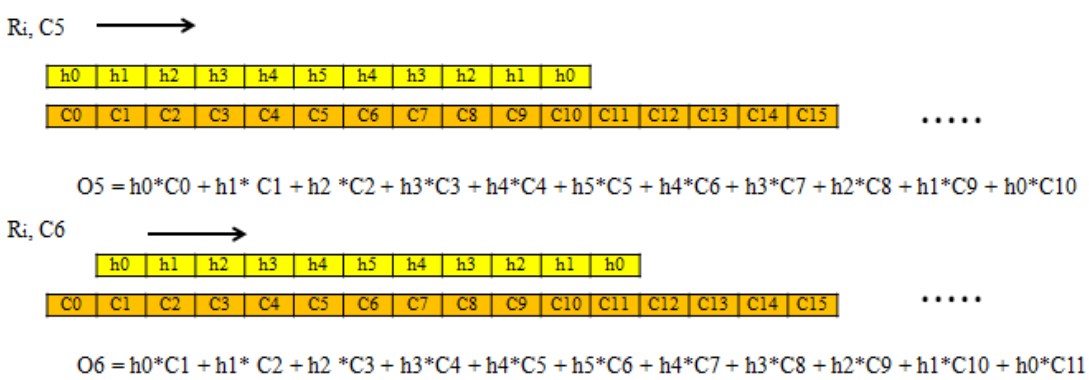

Figure.4.12. Convolution in X direction

Again in X direction there is a possibility of exploiting symmetry but doing so does not yield any benefit. This is because; the samples that are loaded during a particular iteration cannot be reused for the consecutive loop iterations when traversing in the X direction. Taking the example in Figure.4.14 the inputs required to compute outputs for VC_j and VC_{j+1} are not the same. This means that for each loop iteration we require new 11 set of input samples loaded from the memory. Hence in this case even if we try to reduce MAC operations in the inner loop, the load/store unit creates the bottleneck. Thus we decide to not exploit symmetry in X direction.

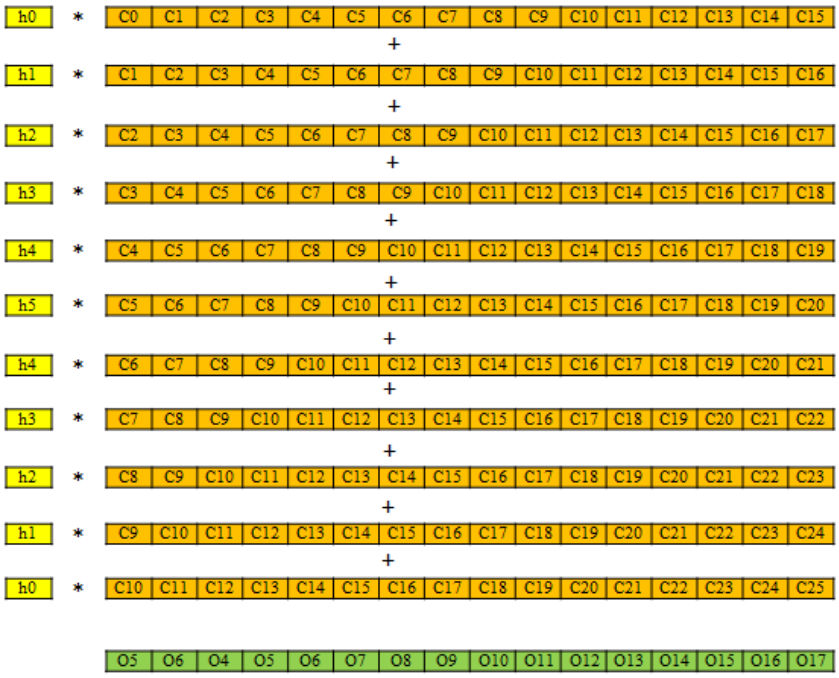

Figure.4.13. Partial accumulation technique

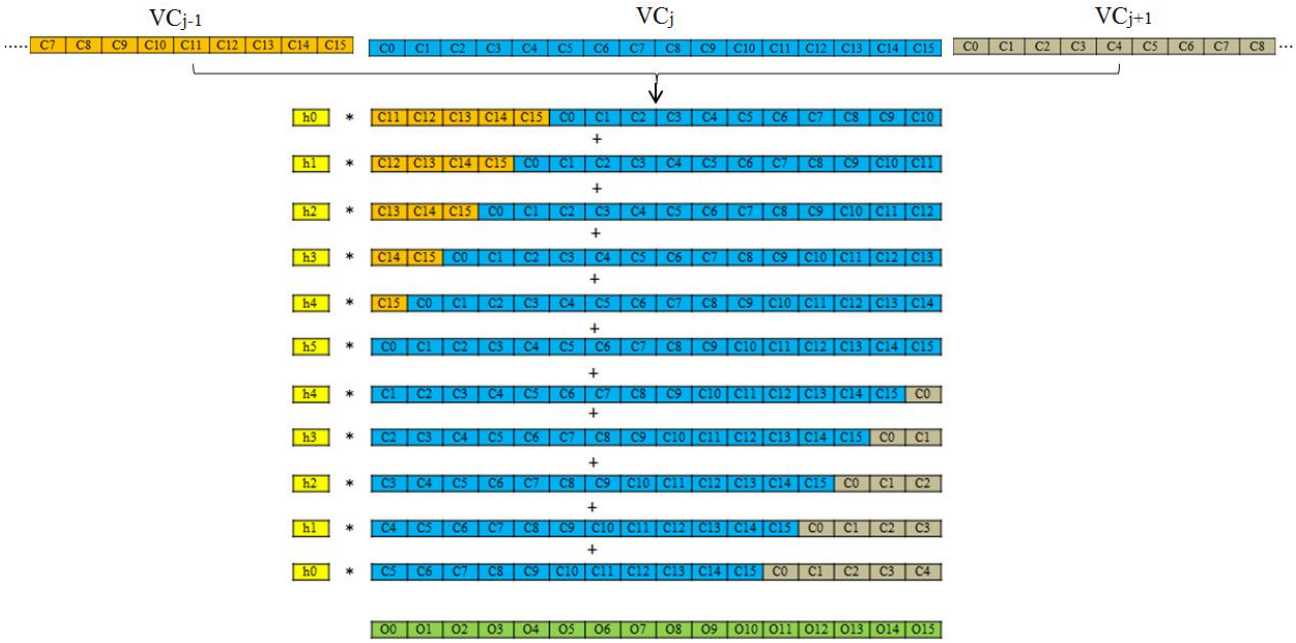

Figure.4.14. Convolution in X direction

As usual, the edge pixels need to be handled differently even when convoluting in X direction. In the each row, convolution on first and the last vectors are performed separately. Again here we use zero padding technique where values of pixels beyond boundaries are considered as zero. There are two possibilities here. One is to pad the input image with vector of zeroes at the start and end of each row. The other method is by using masked operations to ensure the relevant input samples contribute to accumulated results for edge pixels. We use the latter method.

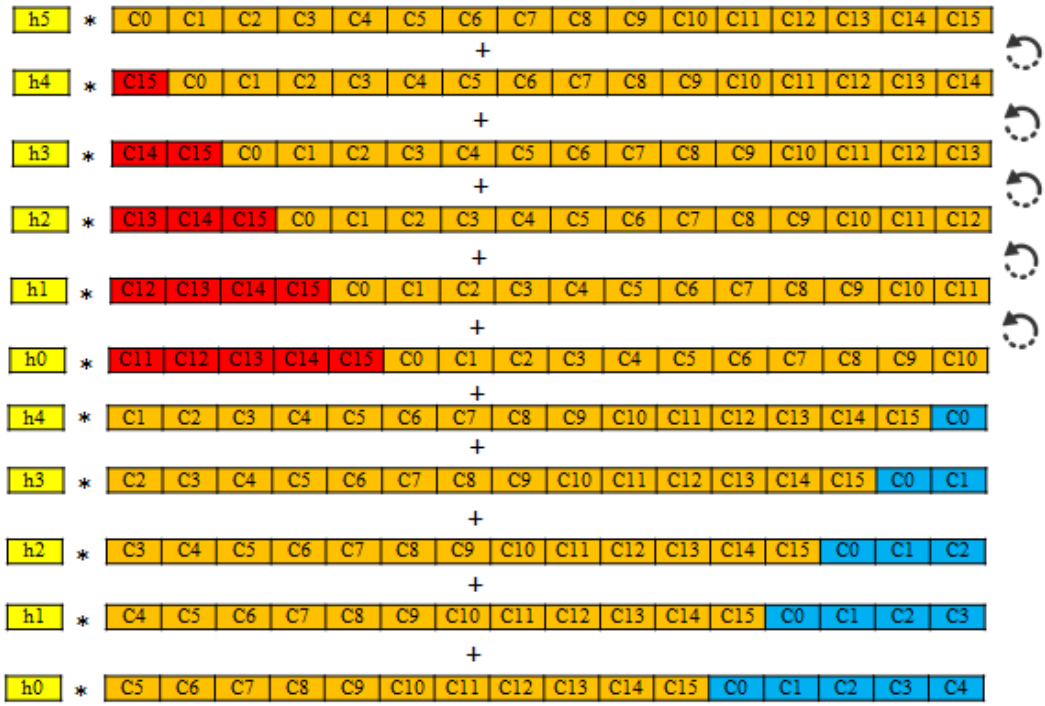

Figure.4.15. Convolution on first column vector, X direction

Figure.4.15 illustrates how the first column vector in a particular row is convolved. The samples shown in red are masked and are not added to the final output. The samples in blue are the samples from the next vector. The samples C0:C4 depends on pixels outside boundaries which are essentially zeroes and do not add value to output. This is instead achieved by using masked accumulation of only those weighted components that should be present in output. For example, output for sample C0 can be obtained as below.

$$O0 = 0*h0 + 0*h1 + 0*h2 + 0*h3 + 0*h4 + C0*h5 + C1*h4 + C2*h3 + C3*h2 + C4*h1 + C5*h0$$

From figure.4.15. we can see only weighted samples of C0:C5 are added to get output for C0 using masks. These masks can be set before hand or can be calculated on the fly. In our implementations we use preset masks but again here the number of mask registers are limited and can cause register spilling. Hence it is better to calculate masks on the fly for bigger gaussian kernels. This is left for future work.

The last column vector in the row is also operated in similar fashion as shown in Figure 4.16. In this case, masked accumulation is performed at the other end of the vector. If the last vector has lesser than 16 samples then only the required samples are selectively filled while the rest are set to zeroes.

NOTE: The other method that we could have followed where each row can be literally padded with vector of zeroes could lead to more simplistic design. In this case we would have processed the whole row in the same fashion without worrying about the edges. The approach could have saved some complexity in handling the edges and improved performance a bit more. But this would have increased the amount of memory required to store images.

Thus it is a tradeoff between memory and performance. In retrospect, we believe that it is better to pad the whole input image with a vector of zeroes while performing convolution if the performance is the only concern.

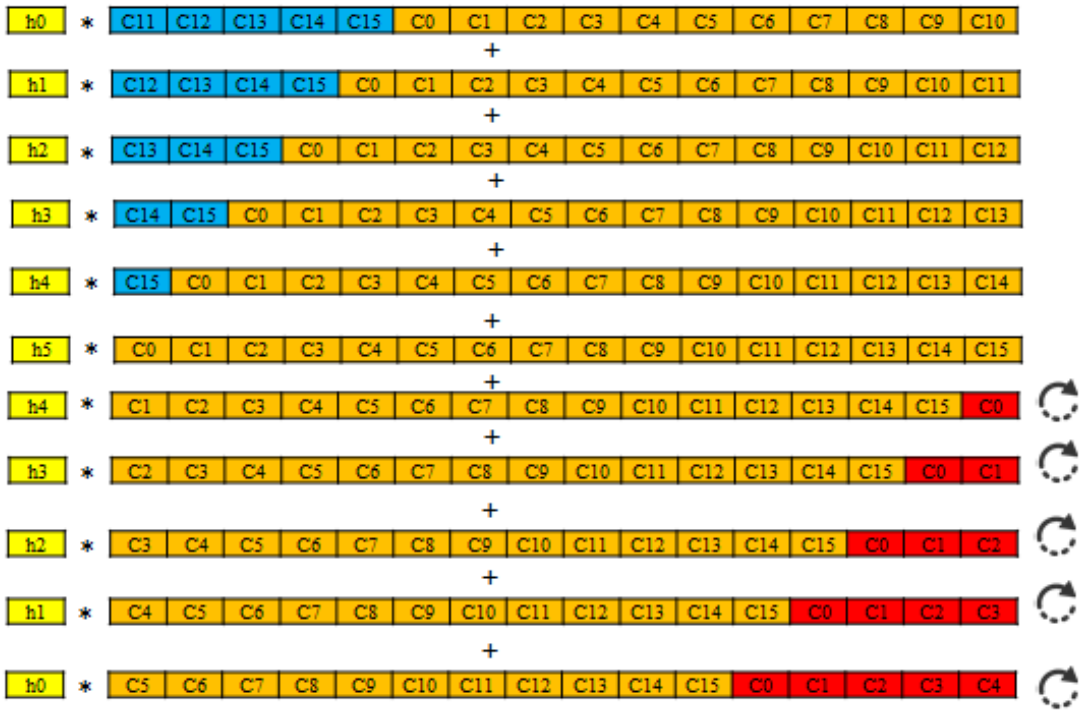

Figure.4.16. Convolution on last column vector, X direction

The pseudo code for convolution is X direction is as below.

```
vw = Ceil(W/P);
l = W - P*floor(W/P);
for (i = 0; i < H; i++)
{
      j = 0;
      /** First column vector */
      VCj[0:P-1] = I[i][j*P : j*P + P -1]; //aligned load
      C0 = (VCj[11:15], VCj[0:10]);
      C1 = (VCj[12:15], VCj[0:11]);    //Rotate
      C2 = (VCj[13:15], VCj[0:12]);
      C3 = (VCj[14:15], VCj[0:13]);
      C4 = (VCj[15], VCj[0:14]);
      C5 = VCj[0:15];
      C6 = (VCj[1:14], VCj+1[0]);      //unaligned load
      C7 = (VCj[2:14], VCj+1[0:1]);
      C8 = (VCj[3:14], VCj+1[0:2]);
      C9 = (VCj[4:14], VCj+1[0:3]);
      C10 = (VCj[5:14],VCj+1[0:4]);


      Acc_X  = C5 .* h5;
```

```
Acc_X += (C4 .* h4) & 0x0111111111111111;
Acc_X += (C3 .* h3) & 0x0011111111111111;
Acc_X += (C2 .* h2) & 0b0001111111111111;  // Masked MAC operations
Acc_X += (C1 .* h1) & 0b0000111111111111;
Acc_X += (C0 .* h0) & 0b0000011111111111;
Acc_X += C6 .* h4;
Acc_X += C7 .* h3;
Acc_X += C8 .* h2;
Acc_X += C9 .* h1;
Acc_X += C10 .* h0;

O[i][(j)*P : (j)*P + P-1] = Acc_X;

// Software pipelining
C0 = (VCj[11:15], VCj+1[0:10]);
C1 = (VCj[12:15], VCj+1[0:11]);
C2 = (VCj[13:15], VCj+1[0:12]);
C3 = (VCj[14:15], VCj+1[0:13]);
C4 = (VCj[15], VCj+1[0:14]);

for(j = 0; j < vw -2; j++)
{
      Acc_X  = (C4 .* h4);
      Acc_X += (C3 .* h3);
      Acc_X += (C2 .* h2);
      Acc_X += (C1 .* h1);
      Acc_X += (C0 .* h0);

      C5 = VCj+1[0:15];
      C6 = (VCj+1[1:14], VCj+2[0]);    //unaligned load
      C7 = (VCj+1[2:14], VCj+2[0:1]);
      C8 = (VCj+1[3:14], VCj+2[0:2]);
      C9 = (VCj+1[4:14], VCj+2[0:3]);
      C10 = (VCj+1[5:14],VCj+2[0:4]);

      Acc_X += C5 .* h5;
      Acc_X += C6 .* h4;
      Acc_X += C7 .* h3;
      Acc_X += C8 .* h2;
      Acc_X += C9 .* h1;
      Acc_X += C10 .* h0;

      O[i][(j+1)*P : (j+1)*P + P-1] = Acc_X;

      C0 = (VCj+1[11:15], VCj+2[0:10]);
      C1 = (VCj+1[12:15], VCj+2[0:11]);
      C2 = (VCj+1[13:15], VCj+2[0:12]);
      C3 = (VCj+1[14:15], VCj+2[0:13]);
      C4 = (VCj+1[15], VCj+2[0:14]);
```

```
        }

        /** Last column vector */
        Acc_X  = C4 .* h4;
        Acc_X += C3 .* h3;
        Acc_X += C2 .* h2;
        Acc_X += C1 .* h1;
        Acc_X += C0 .* h0;

        C5 = I[i][j*P : j*P + P -1];    //aligned load
        C6 = (VCj[1:15], VCj[0]);
        C7 = (VCj[2:15], VCj[0:1]);
        C8 = (VCj[3:15], VCj[0:2]);
        C9 = (VCj[4:15], VCj[0:3]);     // Rotate
        C10 = (VCj[5:14],VCj[0:4]);

        Acc_X += C6 .* h4 & 0b1111111111111110;
        Acc_X += C7 .* h3 & 0b1111111111111100;
        Acc_X += C8 .* h2 & 0b1111111111111000;
        Acc_X += C9 .* h1 & 0b1111111111110000;
        Acc_X += C10 .* h0& 0b1111111111100000;

        O[i][(j)*P : (j)*P + P-1] = l > 0 ? (Acc_X[0:l-1],0..0) : Acc_X;
        // Selectively fill only required number of samples for last vector
}
```

In the pseudo code we also illustrate how software pipelining is performed by repeating the same instructions before start and end of the innermost loop. This helps the compiler to build optimized schedules for the loops.

The schedule for the innermost loop obtained from the compiler is as shown in the Figure.4.17. We can see that the VLSU and VMAC unit determine the critical resources to compute the output pixels. Table.4.2 gives the number of cycles for gaussian kernels of width 5, 7, 8, 10 and 13.

Cycles	VLSU	VMAC
1	C4 = (VCj+1[15], VCj+2[0:14]);	Acc_X += C3 .* h3;
2	C5 = VCj+1[0:15];	Acc_X += C2 .* h2;
3	C6 = (VCj+1[1:14], VCj+2[0]);	Acc_X += C1 .* h1;
4	C7 = (VCj+1[2:14], VCj+2[0:1]);	
5	C8 = (VCj+1[3:14], VCj+2[0:2]);	Acc_X += C5 .* h5;
6	C9 = (VCj+1[4:14], VCj+2[0:3]);	Acc_X += C0 .* h0;
7	C10 = (VCj+1[5:14],VCj+2[0:4]);	Acc_X += C6 .* h4;
8	C0 = (VCj+1[11:15], VCj+2[0:10]);	Acc_X += C7 .* h3;
9		Acc_X += C8 .* h2;
10	C1 = (VCj+1[12:15], VCj+2[0:11]);	Acc_X += C9 .* h1;
11	C2 = (VCj+1[13:15], VCj+2[0:12]);	Acc_X += C10 .* h0;
12	C3 = (VCj+1[14:15], VCj+2[0:13]);	
13	O[i][(j+1)*P : (j+1)*P + P-1] = Acc_X;	Acc_X = C4 .* h4;

Figure.4.17. Schedule of the innermost loop – convolution in X direction

	Kernels	16 Register architecture	32 register architecture
Convolution in X direction	5x Gaussian	13	13
	7x Gaussian	19	18
	8x Gaussian	44	20
	10x gaussian	64	23
	13x gaussian	68	30

Table.4.2. Cycle count for the innermost loop – Convolution in X direction

The input image is recursively convolved with gaussian kernels of width 5, 7, 8, 10 and 13 to build a particular octave. This applies to all the octaves.

4.2.3. Difference of Gaussian

After constructing an octave the next step is to apply difference of gaussian operator on scales of the octave. The difference of gaussian function is nothing but performing a simple subtraction of adjacent blurred images. On EVP this can be done very efficiently. Vectors with same spatial positions in the adjacent scales are obtained and the lower scales` vector ($R_i$, $VC_j$, s) is subtracted from higher scales` vector (R_i, VC_j, s+1) as shown in figure.4.18. This can be done sequentially since vectors are stored sequentially in the memory and thus we need not worry about X and Y directions. For 6 scales in an octave 5 difference-of-gaussian images are computed.

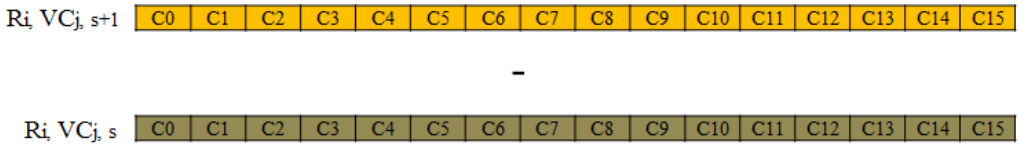

Figure.4.18. Schedule of the innermost loop – convolution in X direction

Pseudo code for performing difference of gaussian is as follows.

```
for(k = 0; k < s; k++)
{
        for(i = 0; i < H; i++)
        {
                for(j = 0; j < ceil(W/P); j++)
                {
                        I1 = GAUSS[k][i][j*P : j*P + P -1];
                        I2 = GAUSS[k+1][i][j*P : j*P + P -1];

                        D1 = I2 - I1;
                        DOG[k][i][j*P : j*P + P -1] = D1;
                }
        }
}
```

In the actual code we essentially combine i and j iterations.

4.2.4. Down-sampling

After building a particular octave, the base image for constructing the next octave is obtained by down-sampling one of the blurred images (Second from top of an octave) from the current octave. The image is down-sampled to half the original size by taking every second pixel in the X and Y direction. For example consider an example 5x5 image that needs to be down-sampled as shown in the Figure.4.19.

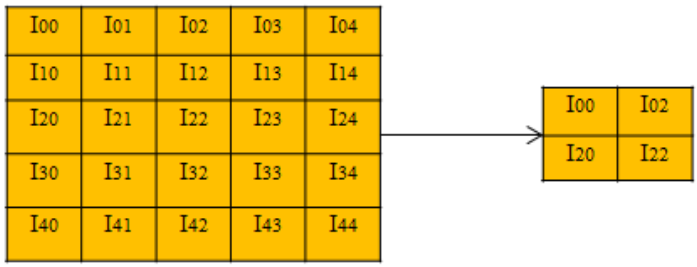

Figure.4.19. Down-sampling on an example 5x5 image

On EVP down-sampling an image is pretty straight forward. In X direction two consecutive vectors VC_j and VC_{j+1} are taken and are merged to a single vector using masked shuffle operations. In the Y direction every alternate row is discarded. Figure.4.20 illustrates the same.

NOTE: If there are odd number of column vectors then the last column vector is processed separately but in a same manner. We then need to apply shuffle and mask operation with the last column vector VC_j and a zero vector (VC_{j+1}) to set the rest of the uninitialized elements to zero.

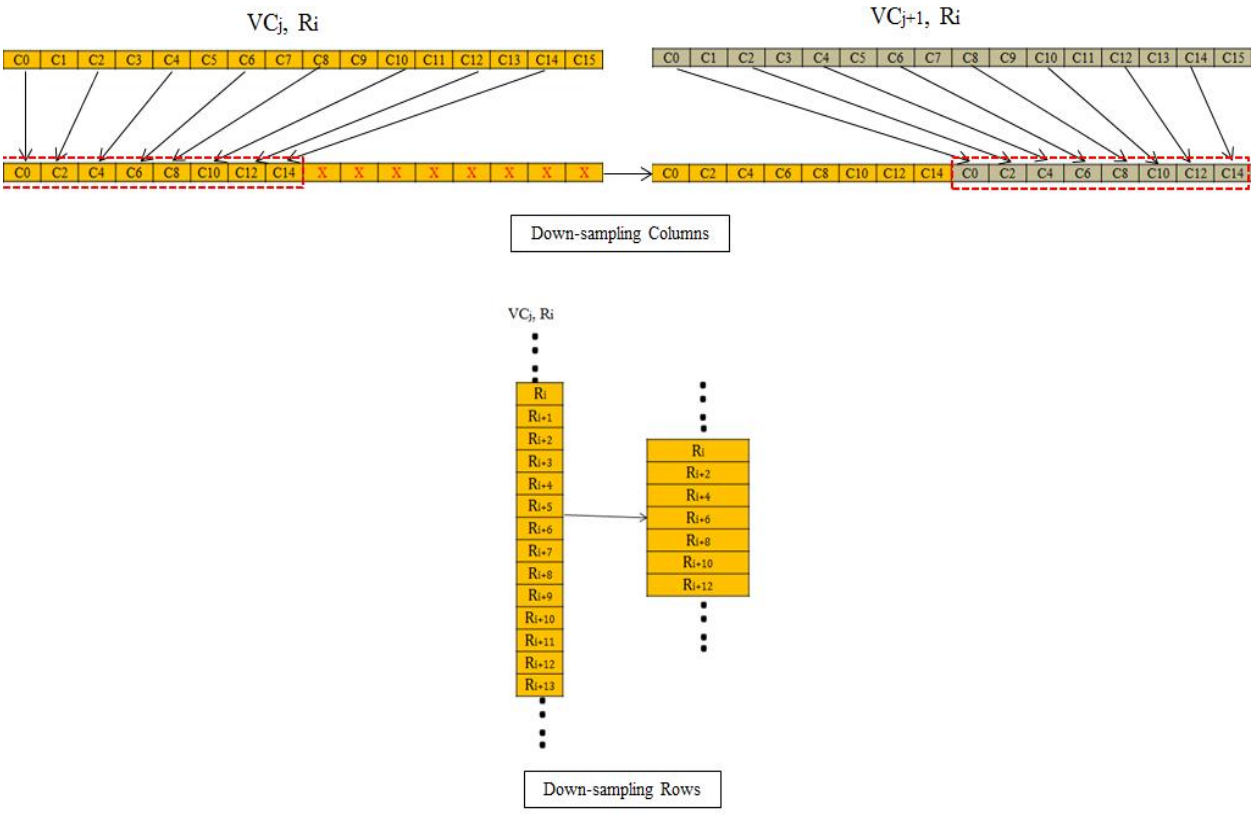

Figure.4.20. Down-sampling on EVP

The pseudo for down-sampling the image is as follows.

```
ow = CEIL(W/2/P)/2;
odd = CEIL(W/2/P)%2 //The width is an odd number

for(i = 0; i < H/2 ; i++)
{
      for(j = 0; j < ow; j+=2)
      {
            C1[0:P-1] = I[i*2][j*P: j*P + P-1];
            C2[0:P-1] = I[i*2][(j+1)*P:(j+1)*P + P-1];

            D[0:P-1] =
            (C1[0],C1[2],C1[4],C1[6],C1[8],C1[10],C1[12],C1[14],
            C2[0],C2[2],C2[4],C2[6],C2[8],C2[10],C2[12],C2[14]);
            // Shuffle with mask

            O[i][j*P :j*P + P-1] = D;
      }

      C1[0:P-1] = odd ? I[i*2][j*P : j*P + P-1] : C1[0:P-1];
      O[i][j*P :j*P + P-1] = odd ?
      (C1[0],C1[2],C1[4],C1[6],C1[8],C1[10],C1[12],C1[14],0,0,0,0,0,0,0,0):
      O[i][j*P :j*P + P-1];
}
```

4.2.5. Extrema detection

After difference-of-gaussian images for a particular octave are built, local maxima and minima points in these difference-of-gaussian scales are obtained. We have 5 difference-of-gaussian scales (s0-s4) and from these 3 intervals are used of extrema detection (s1-s3). As discussed in section 2.1.2 we take each pixel in a particular scale and compare it with 26 of its neighbors in the current and adjacent scales. This operation is performed on all the samples present in a scale and hence can be easily vectorizable on EVP. Let us consider vectors from three consecutive rows in scales s0, s1 and s2 as shown in the figure.4.21. Now we are interested in finding whether samples of row R_i in scale s1 is greater than or lesser than its 26 neighbors. In the figure, 26 neighbors of sample C1 from (R_i, s1) are shown by a red dotted window. When we move this window by one sample offsets in X direction we get 26 neighbors for samples C2, C3 and so on.

Comparing each sample with 26 neighbors is very inefficient. Instead we devise a method where maxima/minima of all 26 neighbors are computed first and then this result is compared with the sample of interest. In the example shown in Figure.4.21 the maxima/minima of all 26 neighbors of sample C1 (R_i,s1) are first computed and then C1 is compared with the result. As a first step, let us see how maxima/minima values for a particular scale are computed.

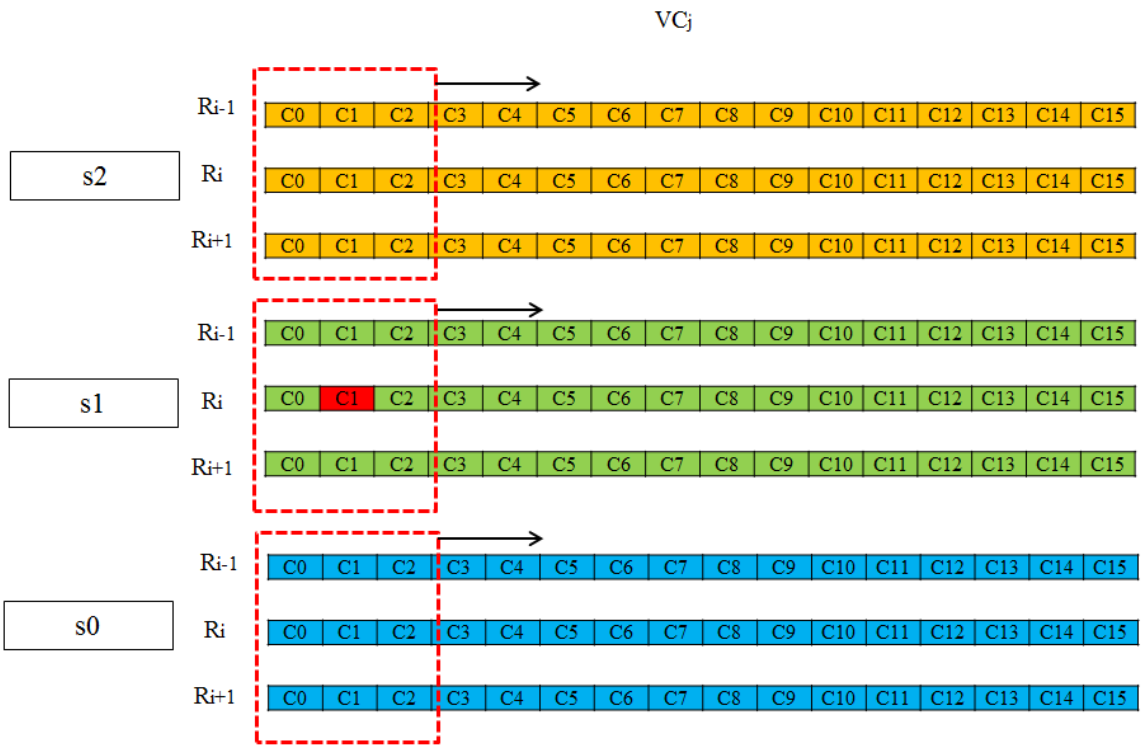

Figure.4.21. Extrema detection on EVP

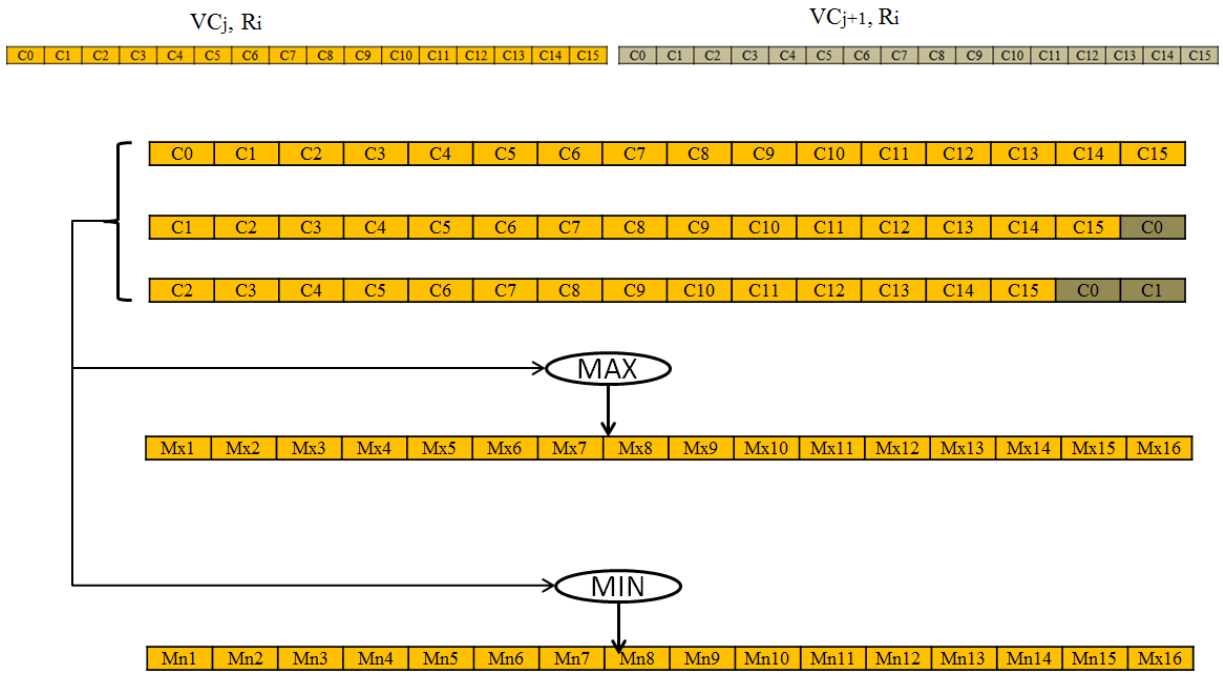

Figure.4.22. Maxima and minima detection in X direction

Let us consider scale s2 where we have vectors from three rows R_{i-1}, R_i, R_{i+1}. The process of finding maxima/minima is done in two steps; first in the X direction and then in the Y direction. In the X direction we must find maxima of three consecutive samples in a vector. By combination of unaligned load operations and max/min

operations it is possible to find maxima/minima for three successive samples. This is illustrated in Figure.4.22. As we can see Mx1 and Mn1 are the maxima and minima of samples C0, C1 and C2. The next outputs Mx2 and Mn2 are for samples C1, C2 and C3 and so on. Without having to traverse sample by sample we compute maxima and minima for the whole vector for all possible 3 consecutive samples. Now this step is performed on vectors from all three rows R_{i-1}, R_i, R_{i+1}.

The resulting 3 maxima and minima vectors for rows R_{i-1}, R_i, R_{i+1} are again subject to max and min operations respectively to obtain the combined maxima and minima values for a particular scale. This result is actually the maxima/minima values of all possible 3x3 windows centered on each sample of the vector from row R_i.

The method is repeated for scale s0 in a similar fashion to get maxima/minima values for this scale. Now for scale s1 the maxima/minima values for rows R_{i+1} and R_{i-1} are computed first. Then in case of row R_i only the maxima/minima of two adjacent neighbors is obtained excluding the sample of interest since we need to compare the same with rest of the values.

Thus we have maxima/minima values of 18 neighbors in scale s0 and s2, 6 neighbors from rows R_{i+1} and R_{i-1} in scale s1 and two adjacent samples of the interest point. Now we compare all the samples of vector (R_i,s0) with these maxima/minima vectors to determine whether the samples are greater than all the maxima values or lesser than all the minima values. This is essentially computing maxima and minima of all possible 3x3x3 cubes. The result of the comparison is stored as a bit map which is can be used to extract the co-ordinates of the sample point later.

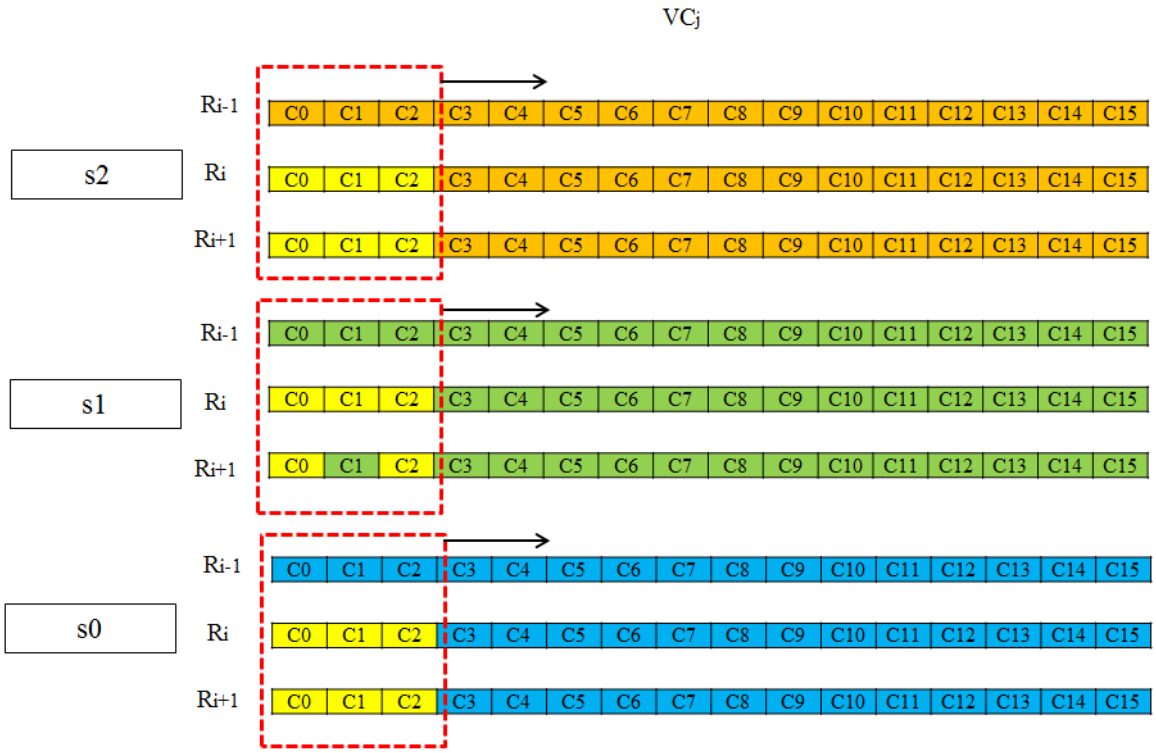

Figure.4.23. Re-using maxima values for consecutive iterations

Note: The process is repeated by traversing row by row. Thus in the next step samples are centered on row R_{i+1} in scale s1. Since we traverse by one row offset many of the maxima/minima values computed in the previous iteration can be re-used. Maxima and minima of rows R_i and R_{i+1} from scales s0, s1 and s2 can be re-used in the next iteration. Thus we can reduce many max and min operations in each loop iteration and reduce the bottleneck created

by the VALU unit in performing these operations. Figure.4.23 shows those samples (in yellow) that can be re-used for the next iteration on rows R_i, R_{i+1}, R_{i+2}. The window is just for illustrating one 3x3 window but it applies to the whole vector and all possible 3x3 windows. Below is a pseudo code for finding maximum for scale 0 which can be extended to other scales. The sample of interest in scale 1 is compared with maxima/minima values of all scales. Here the value maxscale1 is computed from only 8 neighboring pixels excluding the sample of interest.

```
..........

for(j = 0; j < Ceil(W/P); j++)
{
      /** For scale 0 */
      VCj = I[s][i][j*P : j*P + P -1];
      C0 = VCj[0:15];
      C1 = (VCj[1:15], VCj+1[0]);
      C2 = (VCj[2:15], VCj+1[0], VCj+1[1]);

      Rmax0 = Max(C0,C1,C2); // Row 0
      Rmin0 = Min(C0,C1,C2);

      VCj = I[s][i+1][j*P : j*P + P -1];
      C0 = VCj[0:15];
      C1 = (VCj[1:15], VCj+1[0]);
      C2 = (VCj[2:15], VCj+1[0], VCj+1[1]);

      Rmax1 = Max(C0,C1,C2); // Row 1
      Rmin1 = Min(C0,C1,C2);

      VCj = I[s][i+2][j*P : j*P + P -1];
      C0 = VCj[0:15];
      C1 = (VCj[1:15], VCj+1[0]);
      C2 = (VCj[2:15], VCj+1[0], VCj+1[1]);

      Rmax2 = Max(C0,C1,C2); // Row 2
      Rmin2 = Min(C0,C1,C2);

      maxscale0 = Max(Rmax0, Rmax1, Rmax2);
      minscale0 = Max(Rmin0, Rmin1, Rmin2);

      ..........

      for(i = 0; i < H - 2; I++)
      {
            maxall = Max(maxscale0, maxscale1, maxscale2);
            minall = Min(minscale0, minscale1, minscale2);

            VCj = I[s+1][i+1][j*P : j*P + P -1];

            S = (VCj[1:15], VCj+1[0]);  // Samples of interest

            bMap[s+1][i+1] ][j*P : j*P + P -1] = (S > maxall)| (S < minall);
            // Bit map

            ..........
```

4.2.6. Gradient and orientation assignment

Once the location of the keypoint is extracted, the gradient magnitude and orientation of samples present in the keypoint region are computed as explained in the section 2.3. As a first step we shall compute *gx* and *gy* which are gradients in X and Y directions respectively. The gradients *gx* and *gy* at certain sample point is a simple difference of adjacent samples in X and Y direction as shown below.

$$gx = I_{x+1,y} - I_{x-1,y}$$

$$gy = I_{x,y+1} - I_{x,y-1}$$

Figure.4.24 shows how gradients *gx* and *gy* are obtained for the example 5x5 image. Again here special attention should be given to pixels at the image boundaries. Whenever we need to compute gradient which requires accessing pixels beyond boundaries, the nearest pixel is used instead. This is evident in the figure.4.24. In case of gx, the first and the last column, and in case of gy, the first and the last row are handled differently as shown.

I00	I01	I02	I03	I04
I10	I11	I12	I13	I14
I20	I21	I22	I23	I24
I30	I31	I32	I33	I34
I40	I41	I42	I43	I44

Input Image

I01 - I00	I02 - I00	I03 - I01	I04 - I02	I04 - I03
I11 – I10	I12 - I10	I13 – I11	I14 – I12	I14 – I13
I21 – I20	I22 - I20	I23 – I21	I24 – I22	I24 – I23
I31 – I30	I32 - I30	I33 – I31	I34 – I32	I34 – I33
I41 – I40	I42 - I40	I43 – I41	I44 – I42	I44 – I43

Gradient in X direction (gx)

I10 – I00	I11 - I01	I12 - I02	I13 - I03	I14 - I04
I20 - I00	I21 - I01	I22 - I02	I23 - I03	I24 - I04
I30 – I10	I31 – I11	I32 – I12	I33 – I13	I34 – I14
I40 – I20	I41 – I21	I42 – I22	I43 – I23	I44 – I24
I40 – I30	I41 – I31	I42 – I32	I43 – I33	I44 – I34

Gradient in Y direction (gy)

Figure.4.24. Gradient in X and Y dimensions

Computing *gx* and *gy* is straight forward on EVP. In the X direction a simple subtraction of the original vector from its unaligned shifted version by 2 positions gives gx. In Y direction a simple subtraction of vectors from adjacent rows gives *gy*. Figure.4.25 depicts this step. Again special care should be taken while computing boundary pixels (first and last vectors in a row). Using *gx* and *gy* we can now compute gradient magnitude and gradient using the following equations.

$$mag(x,y) = \sqrt{gx^2 + gy^2}$$

$$\theta(x,y) = \tan^{-1}\frac{gy}{gx}$$

The support for square root and arctan functions are not there on EVP and thus we leave our implementation by calculating only partial results like $gx^2 + gy^2$ and $\frac{gy}{gx}$.

Although the gradient magnitude and orientation are required for only small area of keypoint region it is more efficient to compute these values for all the pixels [21]. And we know that all the keypoints are obtained from scales 1 to 3 of an octave. Hence pre-computing gradient magnitude and orientations for these scales can improve performance greatly.

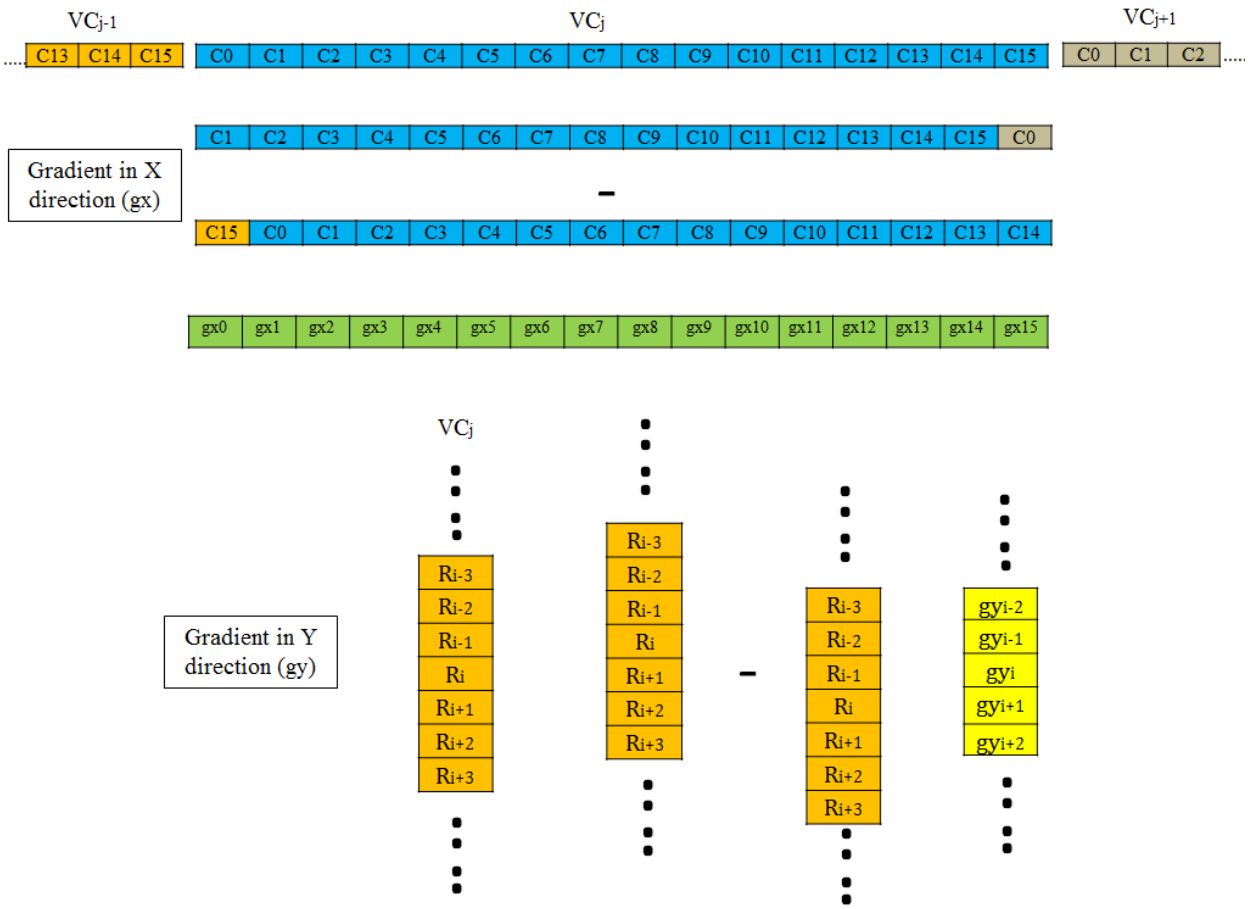

Figure.4.25. Gradient calculation on EVP in X and Y dimensions

The pseudo code for gradient and orientation assignment is as follows

```
for(i = 0; i < H; i++)
{
        /* First column vector */
        VCj = I[i][j*P : j*P + P -1];
        C0 = VCj[0:15];
        C1 = (VCj[1:15], VCj+1[0]);
        C2 = (VCj[2:15], VCj+1[0], VCj+1[1]);
        T1 = C1 - C0;
        T2 = C2 - C0;
        GradX[0: P-1] = (T1[0],T2[0:14]);
        SqGradX = GradX * GradX;

        R0 = (i > 0) ? I[i-1][j*P : j*P + P -1] : I[i][j*P : j*P + P -1];
        R1 = (i < H-1)? I[i+1][j*P : j*P + P -1]:I[i][j*P : j*P + P -1];
        GradY[0: P-1] = R1 - R0;
        SqGradY = GradY * GradY;
        GradMag[i][j*P : j*P + P -1] = sqrt(SqGradX + SqGradY);
```

```
        GradOrnt[i][j*P : j*P + P -1] = atan2(SqGradY/SqGradX);

        /* Middle column vectors */
        for(j = 1; j < Ceil(W/P) - 1; i++)
        {
              /* First column vector */
              VCj = I[i][j*P : j*P + P -1];
              C0 = (VCj-1[15], VCj[0:14]);
              C1 = (VCj[1:15], VCj+1[0]);

              GradX[0: P-1] = C1 – C0;
              SqGradX = GradX * GradX;

              R0 = (i > 0) ? I[i-1][j*P : j*P + P -1] :I[i][j*P : j*P + P -1];
              R1 = (i < H-1)? I[i+1][j*P : j*P + P -1]:I[i][j*P : j*P + P -1];

              GradY[0: P-1] = R1 – R0;
              SqGradY = GradY * GradY;
              GradMag[i][j*P : j*P + P -1] = sqrt(SqGradX + SqGradY);
              GradOrnt[i][j*P : j*P + P -1] = atan2(SqGradY/SqGradX);
        }
        /** Last column vector */
        VCj = I[i][j*P : j*P + P -1];
        C0 = VCj[0:15];
        C1 = (VCj-1[15], VCj[0:14]);
        C2 = (VCj[1:15], VCj[15]);
        T1 = C2 – C1;
        T2 = C0 – C1;
        GradX[0: P-1] = (T1[0 : 14],T2[15]);
        SqGradX = GradX * GradX;

        R0 = (i > 0) ? I[i-1][j*P : j*P + P -1]: I[i][j*P : j*P + P -1];
        R1 = (i < H -1) ? I[i+1][j*P : j*P + P -1] : I[i][j*P : j*P + P -1];
        GradY[0: P-1] = R1 – R0;
        SqGradY = GradY * GradY;
        GradMag[i][j*P : j*P + P -1] = sqrt(SqGradX + SqGradY);
        GradOrnt[i][j*P : j*P + P -1] = atan2(SqGradY/SqGradX);
}
```

4.2.7. Keypoint refining

After getting the locations (coordinates) of maxima and minima from the difference-of-gaussian images we refine these keypoints to get the interpolated subpixel value of coordinates as explained in section 2.2.1. To do so we find the offset that needs to be added to the actual discrete coordinates to get interpolated location. This is done using equation (8) from section 2.2.1 which is:

$$\hat{\mathbf{x}} = -\frac{\partial^2 D^{-1}}{\partial \mathbf{x}^2}\frac{\partial D}{\partial \mathbf{x}}$$

The solution for offset $\hat{\mathbf{x}}$ is obtained by solving a linear equation as shown below.

$$\begin{bmatrix} Dxx & Dxy & Dxs \\ Dxx & Dyy & Dys \\ Dxs & Dys & Dss \end{bmatrix} \hat{\mathbf{x}} = \begin{bmatrix} -Dx \\ -Dy \\ -Ds \end{bmatrix}$$

If D(x,y,s) is a pixel located at the position x,y in scale s of the pyramid of difference-of-gaussian images then:

- *Dx* = (D(x+1,y,s) – D(x-1,y,s)) / 2
- *Dxx* = D(x+1,y,s) + D(x-1,y,s) - 2*D(x,y,s)
- *Dxy* = (D(x+1,y+1,s) + D(x-1,y-1,s) - D(x-1,y+1,s) - D(x+1,y-1,s)) / 2
- Similarly we can compute *Dy, Dyy, Ds, Dss, Dys, Dxs*

Once we calculate the offset $\hat{\mathbf{x}}$ we check whether these offsets exceed 0.5 in x, y or s direction. If the sample points have offset greater than 0.5 in any dimension then it means that actual extrema point is close to a different position. Thus we pivot to the new location and re-iterate in the same way. The pivoting is done until the iteration stops. Once the new location is obtained, the value of the pixel at the new interpolated location is compared with peak threshold. If the value of the pixel is less than the peak threshold then the keypoint is rejected.

Once the keypoint passes the peak threshold test we apply edge threshold. To pass edge threshold stage the keypoint should satisfy the below condition.

$$\frac{(Dxx + Dyy)^2}{(Dxx * Dyy - Dxy * Dxy)} < \frac{(r+1)^2}{r}$$

If the condition is satisfied then the keypoint is accepted for descriptor generation. Here r is the edge threshold and the value used is equal to 10.

On EVP, we do not implement this stage. Keypoint refining is essentially sequential in nature and it is difficult to vectorize the steps performed since locations of keypoints are random. The keypoint refining is implemented on the host machine in our experiments to find the locations. To do this we convert the difference-of-gaussian values to floating point numbers on the host machine and apply the same routine as used in VLFEAT implementation [26].

5. Results and Observations

In this chapter, we are going to evaluate the implementation of SIFT on EVP. As stated earlier, we used VLFEAT`s reference implementation to compare results of each of the kernels in our experiments. The reference implementation is run on an X86 machine with 32 bit floating point data types.

5.1 Data-type exploration for precision

The first design choice that had to be made was to whether use floating point or fixed point data representation for results obtained in SIFT. We explore both the options as stated in the section.3.1. All the experiments (unless specified) are performed on "box" [27] image of size 324x223 provided by D.Lowe in his reference implementation.

Implementation 1 – 8.7 floating point representation

In this implementation 8.7 floating point data types is considered. We find the local extrema points from the first three Difference-of-Gaussian scales of the first octave and then try to see how many of these keypoint locations match with the reference implementation of VLFEAT. These scales extract the highest number of keypoints in SIFT algorithm and hence are vital for accuracy of matching. The keypoints that are extracted in the local extrema detection stage and the refining stage are compared with the corresponding keypoints from reference implementation. Table.5.1 shows the results of the same. The keypoints are matched for the exact co-ordinates in x, y and s dimensions.

Octave 1, Only first three difference-of-Gaussian images	VLFEAT, Number of keypoints before refining	VLFEAT, Number of keypoints after refining	EVP (8.7), Number of keypoints before refining	EVP (8.7), Number of keypoints after refining	Number of keypoints that match before refining	% matches before refining w.r.t VLFEAT	Number of keypoints that match after refining	% matches after refining w.r.t VLFEAT
	384	170	1002	936	115	**29.9**	50	**29.4**

Table.5.1. Matching keypoints – Implementation 1

From the table.5.1 we can clearly observe that the number of keypoints that actually matched with the reference implementation is very poor. Matching keypoints are those whose co-ordinates x, y and s are exactly the same. Only close to 30% of keypoints present in the reference implementation were also found in the EVP implementation. Obviously the choice of 8.7 floating point representation is a no go. The main reason for this inaccuracy is the loss in precision that is inherent in floating point representation. The Gaussian blurred images require sufficient level of accuracy even after the decimal point. With only 8 bits for mantissa it is not possible to represent values of blurred images with good accuracy in the fractional part. Although the floating point representation can provide huge dynamic range, the pixel values operated on in an image lies within a range of values [0:255]. The fractional part is rounded off as the value of the samples grows and this leads to a quantization error. In the extrema detection stage we will be comparing a sample with 26 of its nearest neighbors which have very close values and accuracy in the fractional part can impact the result greatly. With quantization error, this step can lead to many false positives and false negatives and hence there are so many keypoints that are detected in the extrema detection and keypoint refining step. We observe that many of these maxima and minima points are decided based on the accuracy in the fractional part of their values. If the input pixel value is large and requires more bits to be represented then we lose precision in the fractional part. This is the main reason for in accuracy.

The result gets even worse as we move to the next scales. This is because we follow recursive blurring method. Blurred image for the current layer is obtained by applying convolution on the previous layer. This leads to error propagation and further loss in precision as we go to higher layers. Figure.5.1 gives a histogram plot of the error in the gaussian values computed on EVP with respect to VLFEAT implementation for the first four Gaussian images in the first octave. The deviation seen from the reference values is significant which leads to loss in accuracy of keypoint matching. Also we can observe how the histograms tend to widen as we go to higher levels of gaussian blurred images. The error becomes larger because of the propagation of quantization error.

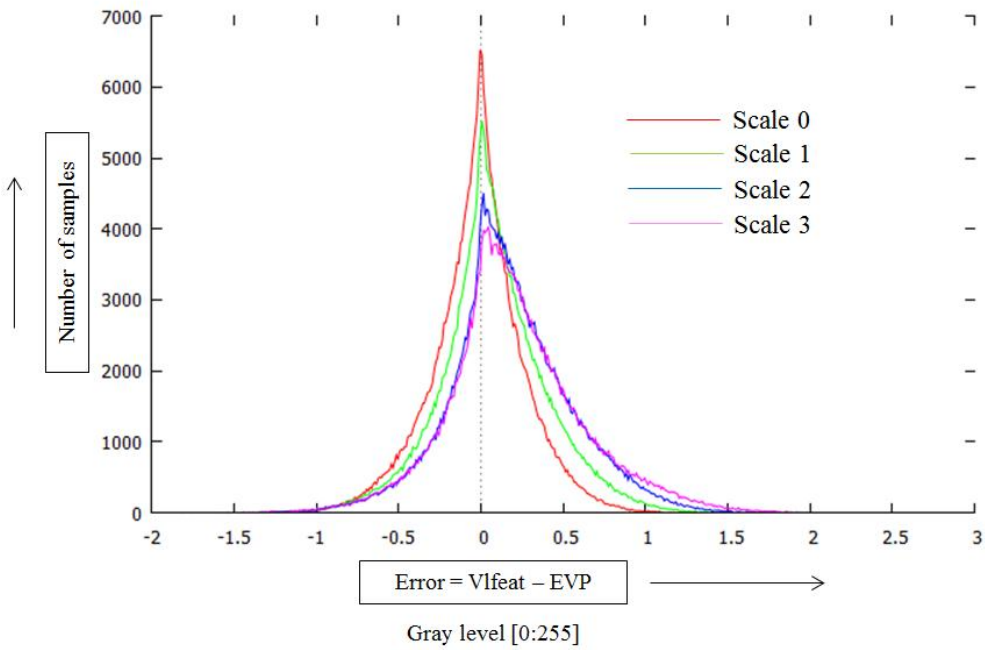

Figure.5.1 Histogram plot of error in the Gaussian blurred images - Implementation 1

Implementation 2 – 10.5 floating point representation

In this implementation we repeat the same steps as performed in implementation 1 but the only difference here is that we represent floating point numbers using 10.5 format. Thus we have 2 more bits of mantissa compared to the previous implementation. Table.5.2 gives the results obtained for this experiment.

Already we can see that there is an improvement in the number of keypoints that match with the reference implementation. Approximately 52% of the keypoints after refining stage match with the reference implementation. The histogram plot of error in gaussian values is shown in the figure.5.2. We can notice that the range of error is lower when compared to previous implementation. Also the number of samples that are close to zero error increases. Thus it can be established that the precision provided by mantissa bits are crucial for accuracy in SIFT. But still 52% accuracy in keypoint matching is not acceptable and is not good for implementing SIFT. This inaccuracy can affect matching of features greatly and thus even this implementation is not recommended and obviously we require more bits to represent even fractional part with good accuracy.

Octave 1, Only first three difference-of-gaussian images	VLFEAT, Number of keypoints before refining	VLFEAT, Number of keypoints after refining	EVP (10.5), Number of keypoints before refining	EVP (10.5), Number of keypoints after refining	Number of keypoints that match before refining	% matches before refining w.r.t VLFEAT	Number of keypoints that match after refining	% matches after refining w.r.t VLFEAT
	384	170	682	532	204	**53.1**	88	**51.7**

Table.5.2. Matching keypoints – Implementation 2

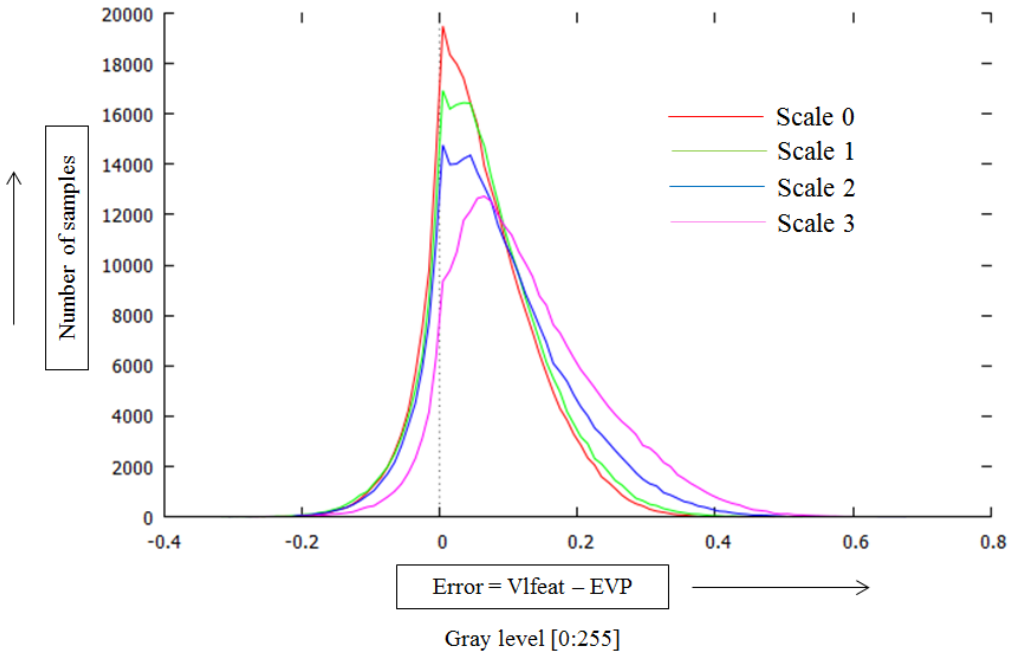

Figure.5.2 Histogram plot of error in the Gaussian blurred images - Implementation 2

The above implementations are performed only on the first three difference-of-gaussian images of the first octave as there is no point in extending it to the whole octave and also further octaves. The first octave gives the most number of SIFT features and having better accuracies at the start is far more important than higher scales.

From the above two experiments it was concluded that it is not suitable to perform SIFT with the floating point data types supported by EVP. Next we explore fixed point implementations.

Implementation 3 – 16 bit fixed point implementation

EVP supports 16 bit fixed point representation where 1 bit is used as the sign bit and rest of the 15 bits for data. The input image is first scaled to a range of [0:1] and then the operations are performed. In this experiment we use 14 bits to represent input data with 1 sign bit and 1 overflow bit. The same steps from the previous experiments are repeated and we can notice further improvements in the number of keypoints that match. The histogram plot for error in gaussian values in this case is as shown in figure.5.3

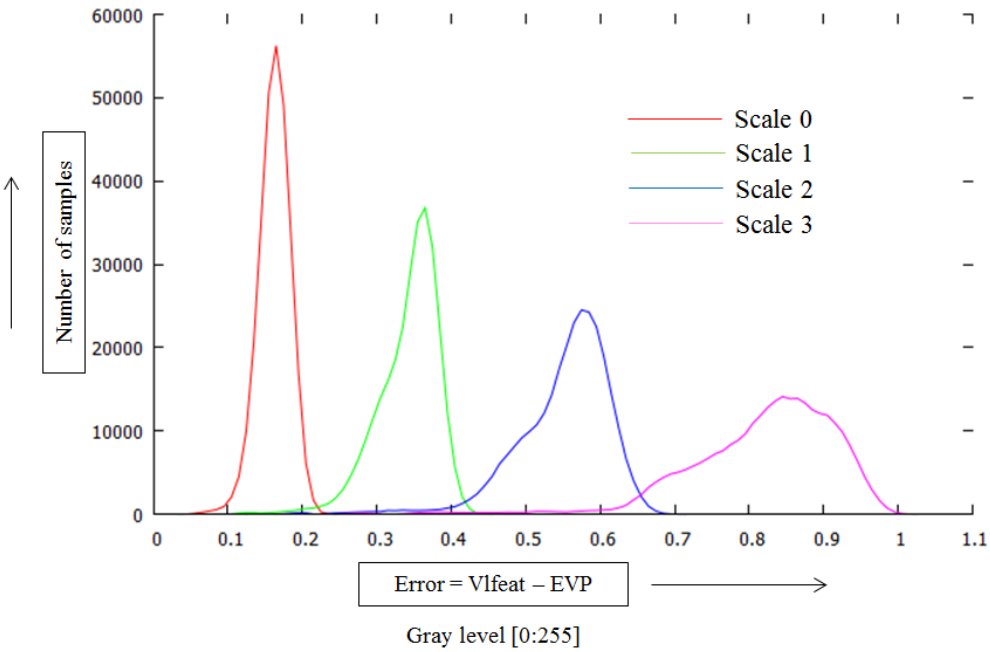

Figure.5.3 Histogram plot of error in the Gaussian blurred images - Implementation 3

From the plot we can observe that error range for the scale0 is lesser than previous implementations and this leads to better matching. Although we observe that the error range widens a lot more compared to 10.5 for higher scales, we observe that the percentage of keypoints that match are still higher. The precision matters lesser and lesser as we move to higher scales. The quantization error is also present in case of fixed point implementation because of rounding off. This error again propagates to higher scales. But in fixed point implementation the precision loss does not depend on the value of the input pixels as in case of floating point. Thus we observe better matching.

We can see from the table.5.3 that approximately 70% of keypoints match for the first octave. This is better than results for 10.5 floating point implementation. Overall we see approximately 73% of keypoints present in the reference implementation matching with the keypoints extracted on EVP. Thus fixed point implementation gives better results for SIFT compared to floating point implementation.

Octaves, all scales	VLFEAT, Number of keypoints before refining	VLFEAT, Number of keypoints after refining	EVP (Fixed), Number of keypoints before refining	EVP (Fixed), Number of keypoints after refining	Number of keypoints that match before refining	% matches before refining w.r.t VLFEAT	Number of keypoints that match after refining	% matches after refining w.r.t VLFEAT
1st	767	392	777	455	564	73.53	278	70.91
2nd	252	126	248	142	170	67.46	89	70.63
3rd	77	58	87	62	64	83.11	54	93.10

4th	14	11	14	11	12	85.71	9	81.81
5th	2	2	2	1	1	50	1	50
Total	1112	589	1128	671	811	**72.9**	431	**73.2**

Table.5.3. Matching keypoints – Implementation 3

Implementation 4 – 16 bit fixed point with 32 bit accumulation

From the previous experiments we observed that rounding off in fixed point implementation was the main cause for inaccuracies. In the gaussian convolution step the MAC operations performed are essentially recursive and the partial results are truncated to 16 bit values in every MAC operation. This affects the least significant bits greatly. Instead of accumulating result in 16 bits we can accumulate them as 32 bit results and discard the lower 16 bits which gets affected by rounding. Using this technique the rounding off error can be contained and better accuracy can be achieved. We can already see its effect in the histogram plot as shown in figure.5.4 compared to previous implementation.

The error range reduces significantly compared to previous fixed point implementation and the number of matches also improves considerably because of this. Table 5.4 shows the results for this experiment. We can observe close to 87% of keypoints matching in the first octave and overall approximately 88 percent of keypoints match.

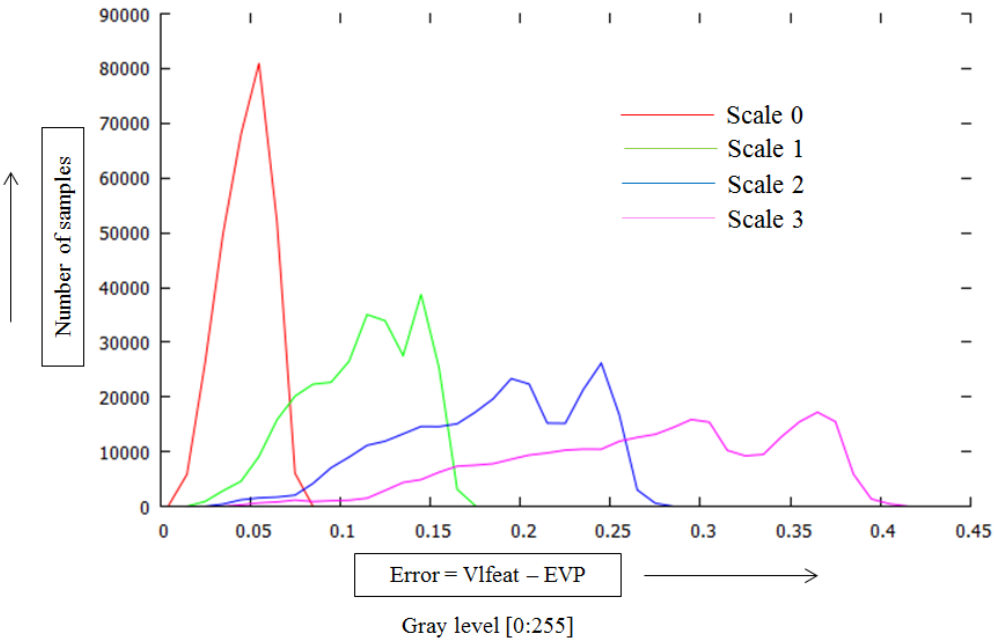

Figure.5.4 Histogram plot of error in the Gaussian blurred images - Implementation 4

Octaves, all scales	VLFEAT, Number of keypoints before refining	VLFEAT, Number of keypoints after refining	EVP (Fixed), Number of keypoints before refining	EVP (Fixed), Number of keypoints after refining	Number of keypoints that match before refining	% matches before refining w.r.t VLFEAT	Number of keypoints that match after refining	% matches after refining w.r.t VLFEAT
1st	767	392	666	368	646	84.2	340	86.7
2nd	252	126	212	126	205	81.3	114	90.5
3rd	77	58	72	57	66	85.7	55	94.8
4th	14	11	12	10	11	78.6	8	72.7
5th	2	2	2	2	2	100	2	100
Total	1112	589	964	563	930	**83.6**	519	**88.1**

Table.5.4. Matching keypoints – Implementation 4

Implementation 5 - 16 bit integer with 32 bit accumulation

We know that the values of gaussian blurred images lies in the range of [0:255] and there is no need to have a signed bit to represent these values. Also there is essentially no difference between fixed point operations and integer operations. In the previous implementations we used 14 bit to represent data with 1 sign bit and 1 overflow bit. Instead we can discard sign bit and use 15 bits to represent data and 1 bit for tackling overflow. The result of any operation will also have value represented in 15 bits. The overflow bit is required only because for few operations the value can go beyond 15 bits (For e.g. Gaussian convolution in Y direction). Also the bit used for overflow can be used as a sign bit whenever we require performing signed operations (E.g. difference-of-Gaussian). This way we gain one extra bit for representing input data and this can improve precision even further. Again here we use 32 bit accumulation and throw away 16 least significant bits to contain error due to rounding. The histogram plot of the error for this experiment is shown in figure.5.5.

The range of error reduces even further and we get close to 90% of keypoints matching overall. Predominantly the mismatch happens at the first few octave where precision is still a concern. But for further octaves we achieve close to 100% matching. The images in first octave will have more detail and high precision in the first octave is necessary. But of course in these experiments we are comparing our 16 bit implementations with 32 bit floating point reference implementation on X86.

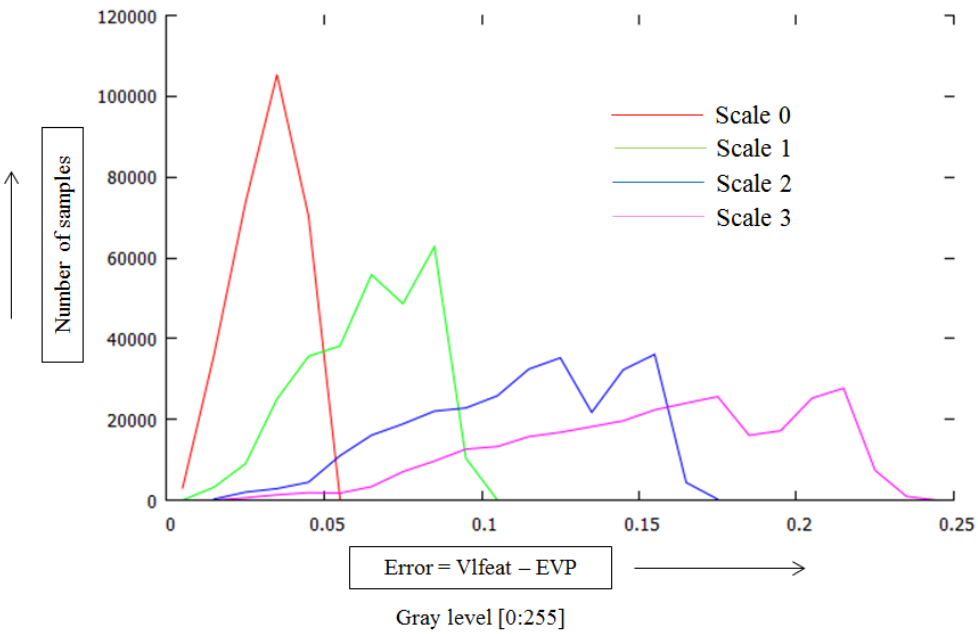

Figure.5.5 Histogram plot of error in the Gaussian blurred images - Implementation 5

Octaves, all scales	VLFEAT, Number of keypoints before refining	VLFEAT, Number of keypoints after refining	EVP (integer), Number of keypoints before refining	EVP (integer), Number of keypoints after refining	Number of keypoints that match before refining	% matches before refining w.r.t VLFEAT	Number of keypoints that match after refining	% matches after refining w.r.t VLFEAT
1st	767	392	710	372	693	90.4	346	88.3
2nd	252	126	229	120	224	89.9	116	92
3rd	77	58	74	57	67	87	55	94.8
4th	14	11	15	12	14	100	11	100
5th	2	2	2	2	2	100	2	100
Total	1112	589	1030	563	1000	**89.9**	530	**90**

Table.5.5. Matching keypoints – Implementation 5

Note: Along with false positives and false negatives in extracting keypoints we also observe certain keypoints displaced by 1-2 pixel position in x or y direction. This is again because of error and rounding. In the future we can also try to find keypoints whose locations fall within a certain preset radius from center of the location that we are searching for. This way the number of keypoints that matches can go further up. Also the keypoints we match here are discrete x and y co-ordinates. In the keypoint refining stage we interpolate on these locations to get sub pixel values for the locations. Here we will lose accuracy since we use 16 bits to represent difference of gaussian values. The accuracy loss of interpolated keypoint locations need to be tested and is left for future work.

In GPU implementations [20] [21] done previously we see that the average number of keypoints extracted are far lesser compared to the number of keypoints extracted on EVP. Some of these keypoints are essentially dropped to get real time performance on SIFT. This is one of the reasons for us to believe that 90% accuracy is sufficient compared to VLFEAT implementation. But it is yet to be seen how 90% accuracy affects image matching.

5.2 Performance of gaussian pyramid

In this section we compare the number of cycles taken for each of the implementations discussed in the previous section for building gaussian pyramid. Table.5.6 gives the number of cycles taken by each of the kernels. We can see from the results that the implementation for architectures having 16 registers is very in-efficient because of register spilling. The number of cycles taken is more than double than that of 32 register architectures. Hence number of vector registers is very crucial for the design.

The number of cycles taken for fixed point implementations is fewer compared to floating point implementation and this is because there are more fixed point resources (fixed point addition on MAC unit) present on EVP than floating point. We find that the accumulation with 16 bit elements for fixed point implementation takes lesser cycles compared to 32 bit accumulation. This is obvious because in 32 bit accumulation we need one extra register and register spilling may cost few extra cycles in this case. The gaussian kernel of width 5 is performed twice in the first octave and that is the reason why the number of cycles is more compared to kernels of width 7 and 8. Clearly we can establish from these results that there is no point in continuing with floating point implementation as the performance does not vary much and we have already shown that it yields poor accuracy in extracting keypoint locations.

For the rest of the document we focus on the fifth implementation that we discussed in the previous section. All the results are based on this implementation unless specifically stated.

Algorithms	Implementation1 and 2		Implementation 3	Implementation 4	Implementation 5
	Floating point – 10.5 or 8.7 with 16 registers	Floating point – 10.5 or 8.7 with 32 registers	Fixed point – 14 bit input 16 bit accumulation with 32 registers	Fixed point – 14 bit input 32 bit accumulation with 32 registers	Unsigned integer – 15 bit input 32 bit accumulation with 32 registers
Up-sample	75,642	75,642	75,601	75,601	75,601
Gaussian-5x	959,828	916,683	915,375	915,375	915,375
Gaussian-7x	1,370,113	728,549	726,825	728,549	728,549
Gaussian-8x	2,353,853	829,631	805,030	805,030	805,030
Gaussian-10x	3,576,913	1,009,092	1,008,230	960,916	960,916
Gaussian-13x	3,338,599	1,464,992	1,464,992	1,514,035	1,514,035
Down-sample	31,260	31,260	31,260	31,260	31,260
Total Gaussian pyramid	**11,706,208**	**5,055,849**	**5,027,313**	**5,030,766**	**5,030,766**

Table.5.6. Number of cycles taken to build gaussian pyramid for “box” image

In table.5.7 we present the number of cycles required to compute one output pixel on EVP for different octaves. If we were to run these algorithms sequentially on a GPP then the cycle counts to compute one output pixel would be greater. We observe that as the size of the image reduces the number of cycles taken per pixel increases. This is because in our implementation the outer loop in-efficiencies and function overhead starts to become more and more prominent as the inner loop iterations reduce. We have optimized only the innermost loops in our experiments and there is room for improvement in these figures by optimizing outer loops as well.

Algorithms	**Octave 1**	**Octave 2**	**Octave 3**	**Octave 4**	**Octave 5**	**Average**
Up-sample	0.26	-	-	-	-	0.26
Gaussian-5x	1.3	1.4	1.5	1.7	1.9	1.6
Gaussian-7x	1.9	1.9	2.1	2.4	2.6	2.2
Gaussian-8x	2.1	2.1	2.3	2.6	3.0	2.4
Gaussian-10x	2.5	2.6	2.8	3.2	3.6	2.9
Gaussian-13x	3.9	4.0	4.3	5.0	5.7	4.6
Down-sample	-	0.3	0.4	0.5	0.8	0.5

Table.5.7. Cycles per output pixel (Implementation 5) for gaussian kernels

We know that the gaussian kernels are the most resource intensive algorithms in SIFT and thus we use EVP to exploit data level parallelism. Since we use 16 elements in a vector, ideally we should be able to achieve a speed up of 16x using EVP. In table 5.8 we present speed up achieved on EVP for each octave. An, assumption that we make is that the number cycles that is needed to compute an output pixel will be determined by the number of mac operations in the sequential code. Using this assumption we compute speed up by dividing the number of cycles per pixel required in each octave with the minimum number of mac operations. This gives a fairly good idea about the speed up achieved on EVP.

Algorithms	Minimum MAC operations/ Minimum cycles when run sequentially	Octave 1 Vector Speed up	Octave 2 Vector Speed up	Octave 3 Vector Speed up	Octave 4 Vector Speed up	Octave 5 Vector Speed up	Average Vector Speed up
Gaussian-5x	22	16.3	15.8	14.7	12.9	11.7	**14.3**
Gaussian-7x	30	16.1	15.5	14.4	12.6	11.3	**14.0**
Gaussian-8x	34	16.5	15.9	14.8	12.9	11.5	**14.3**
Gaussian-10x	42	17.1	16.4	15.2	13.1	11.5	**14.7**
Gaussian-13x	54	14.0	13.4	12.4	10.8	9.5	**12.0**

Table.5.8. Speed up by exploiting both data parallelism and instruction level parallelism for gaussian kernels

As we can observe the average speed up achieved are close enough if not equal to 16. For bigger images the speed up gained is higher and this is very useful as it can improve performance greatly. We can achieve a speed up greater than 16 in few cases. This can be attributed to the convolution in Y direction as we also exploit instruction level parallelism by utilizing symmetry of the kernel coefficients. For the gaussian kernel of width 13 we cannot efficiently exploit symmetry and thus we have poorer performance compared to others. While convolving in X direction optimizing the outermost loop (iteration on height) can yield significant performance gain. Using zero padding at the start and end of a row it is possible to reduce complexity in the outer loop. This can be considered as one of the possibilities for future work.

Even if we do not consider the algorithmic optimizations to exploit instruction level parallelism we can still achieve an average ~11x speed up for all kernels as shown in table.5.9.

Algorithms	Minimum Operations after algorithmic optimization	Octave 1 Vector Speed up	Octave 2 Vector Speed up	Octave 3 Vector Speed up	Octave 4 Vector Speed up	Octave 5 Vector Speed up	Average Vector Speed up
Gaussian-5x	17	12.6	12.2	11.4	10.0	9.0	**11.0**
Gaussian-7x	23	12.4	11.9	11.1	9.7	8.7	**10.7**
Gaussian-8x	26	12.6	12.2	11.3	9.9	8.8	**11.0**
Gaussian-10x	32	13.1	12.5	11.6	10.0	8.8	**11.2**
Gaussian-13x	54	14.0	13.4	12.4	10.8	9.5	**12.0**

Table.5.9. Speed up by without taking into account instruction level parallelism for gaussian kernels

Similarly for up-sampling and down-sampling we may obtain numbers for speed-up as shown in Table 5.10. Note that in all these cases we take a very idealistic scenario of the critical operations when running code sequentially. This may not be the case as the number of cycles required to run the code sequentially can be more. For example in case of down-sampling we see that speedup is very poor. But we consider that the schedule is determined by only a load and a store operation. The latency of loads is more than just 1 cycle. Thus in a real scenario we will observe better speed-up. We take a more pessimistic approach to determine these numbers but in actuality the speedup can be better. The speed-up for up-sampling is on the lower side because we do not factor in 2 operations which convert input data type.

Algorithms	Minimum operations/ Minimum cycles when run sequentially	Octave 1 Vector Speed up	Octave 2 Vector Speed up	Octave 3 Vector Speed up	Octave 4 Vector Speed up	Octave 5 Vector Speed up	Average Vector Speed up
Up-sample	2 Mul + 1 Add	11.5	-	-	-	-	**11.4**
Down-sample	1Load+1 Store	-	6.6	5.57	4.23	2.6	**4.2**

Table.5.10. Speed up by exploiting data parallelism on up-sampling and down-sampling

5.3 Performance of difference of gaussian, extrema detection and gradient pyramid

Difference of Gaussian

Table.5.11. gives the number of cycles taken to perform difference-of-gaussian operation for different implementations. The difference of gaussian performs better with fixed point implementation since there are more resources on EVP to handle fixed point operations. The vector speed up obtained for each octave can be obtained as shown in Table.5.12. In the difference of gaussian the loads are the dominant operations along with subtraction and hence determine the sequence of critical operations. Thus we have minimum of 3 cycles in an ideal scenario.

Algorithm	Implementation1 and 2		Implementation 3	Implementation 4	Implementation 5
	Floating point – 10.5 or 8.7 with 16 registers	Floating point – 10.5 or 8.7 with 32 registers	Fixed point – 14 bit input 16 bit accumulation with 32 registers	Fixed point – 14 bit input 32 bit accumulation with 32 registers	Unsigned integer – 15 bit input 32 bit accumulation with 32 registers
Difference of gaussian	492,385	492,385	430,915	430,915	430,915

Table.5.11. Total number of cycles for difference of gaussian

Algorithm	**Octave 1**	**Octave 2**	**Octave 3**	**Octave 4**	**Octave 5**	**Average**
Difference of gaussian	0.22	0.23	0.24	0.26	0.28	0.25
Vector speed up. 2loads + 1 sub	**13.5**	**13.2**	**12.6**	**11.4**	**10.8**	**12.3**

Table.5.12. Cycles per pixel and vector speed up for difference of gaussian

Extrema detection

The results for extrema detection are obtained for all octaves considering implementation 5. Table.5.13 gives the number of cycles required for each octave and the total number of cycles required in SIFT for detecting local extrema. The numbers are obtained for implementation where we extract bit map. If we were to perform rest of the algorithm in SIFT on EVP bit map would be sufficient. If the locations of extrema points were to be sent back to the host processor then this bit map needs to be parsed to get the exact x, y and s co-ordinates. This can add more cycles to the implementation as shown in the table.5.13. Currently we assume that the bit map would be sufficient and if necessary can be sent back to the host. Table.5.14 gives the speed up and the cycles per pixel required for detecting extrema points for generating bit map.

Algorithms	**Octave 1**	**Octave 2**	**Octave 3**	**Octave 4**	**Octave 5**	**Total**
Extrema detection	2,027,316	518,583	134,901	36,300	8,841	2,725,941
Extracting locations	712,558	194,130	55,625	14,166	3,285	979,764

Table.5.13. Total number of cycles for extrema detection

Algorithms	**Octave 1**	**Octave 2**	**Octave 3**	**Octave 4**	**Octave 5**	**Average**
Cycles per pixel	2.34	2.39	2.50	2.72	2.73	2.54
Vector speedup. Minimum of 52 comparisons/Pixel	**22.2**	**21.7**	**20.8**	**19.2**	**19.0**	**20.5**
Vector speed up Without algorithmic optimization (32 comparisons/pixel)	**13.7**	**13.4**	**12.8**	**11.8**	**11.7**	**12.6**

Table.5.14. Cycles per pixel and speed up for extrema detection

Gradient and orientation assignment

Gradient pyramid update consists of computationally intensive operations like square root, division and *arctan2*. Currently there is no support for square root and arctan operations on EVP but we can estimate the number of cycles it may take when the feature is added on EVP. The square root and arctan functions are supported in EVP libraries for experimental purposes and can be used to count the number of cycles each function takes. The gradient pyramid is built for only 3 scales (1-3) in each octave. This is because all the keypoints are present at these levels and pre-computing these using EVP can speedup this stage. Table.5.15. gives a fair estimate of the number of cycles required for computing gradient pyramid for three scales per octave and table 5.16 gives cycles/pixel for computing each octave. These numbers need to be verified once the square root and arctan functions are available on EVP.

Algorithms	**Octave 1**	**Octave 2**	**Octave 3**	**Octave 4**	**Octave 5**	**Total**
Gradient pyramid	1,025,372	262,935	68,727	18,663	4,635	4,140,996

Table.5.15. Total number of cycles for gradient pyramid construction

Algorithms	**Octave 1**	**Octave 2**	**Octave 3**	**Octave 4**	**Octave 5**	**Average**
Cycles per pixel	3.5	3.6	3.8	4.2	4.3	3.9

Table.5.16. Estimate of the cycles/pixel for gradient pyramid construction

5.4 Total execution time

The execution time for different kernels on EVP for "box" image (320x223) is presented in the Table 5.17 assuming that EVP is clocked at 416 Mhz. The memory transfer and read back are not factored in here and importance is given to performance of the kernels. The gaussian pyramid, difference of gaussian, local extrema detection and keypoint refining have been tested completely with the VLFEAT reference implementation. For gradient pyramid we obtain cycle count conceptually and can be considered as a fair estimate. Due to time limitations we do not go further to build the actual descriptors. As we have seen, the observed matching in keypoint position is close to 90% with reference to VLFEAT implementation.

Algorithms	In cycles	In milliseconds (@416Mhz)
	EVP	EVP
Gaussian pyramid	5,030,766	12.1
Difference of gaussian	430,915	1.0
Local extrema	2,725,941	6.6
Extract locations	979,764	2.4
Gradient pyramid	4,140,996	10.0
Total	**12,328,618**	**32.0**

Table.5.17. Execution time in cycles and milliseconds for "box" image

5.5 Benchmarking

In this section we compare the performance of SIFT on EVP with a GPU implementation done previously [21]. The authors of the paper benchmark SIFT with a dataset of 48 images on a mobile device with a Qualcomm Snapdragon S4 Pro APQ8064 chipset clocked at 1.5 GHz and an Adreno320 GPU clocked at 325 MHz. The images in the data set have widths of 320 and height varying from 214 to 256. The average number of keypoints detected per image is 95. They benchmark this data set with CPUs of ARM-v5 and ARM-v7a instruction set architectures. Then they suggest a heterogeneous implementation that includes ARM-v7a CPU and a GPU (Adreno320) and benchmark the same.

Table.5.18 gives a comparison of the performance of EVP with respect to these benchmarks. We assume that the average size of images is 320x235 and then compute the processing time for EVP. In case of EVP the cycle counts are obtained from the simulator and the time taken in milliseconds for performing various stages is calculated by assuming clock frequency of 416 MHz. In the GPU implementation the authors provide processing time for executing different stages in milliseconds and we can compute back the approximate number of cycles taken for each stage. Also for heterogeneous implementation ARMv7a+Adreno320, the numbers shown in bold red are executed on ARM.

In the paper, the authors use GPU for offloading stages like gaussian pyramid construction, gradient pyramid construction and descriptor generation. Stages such as difference of gaussian pyramid construction, extrema detection, keypoint refinement, orientation assignment and descriptor generation are performed on ARM. Whereas in our implementation we perform both difference of gaussian and extrema detection stage on EVP.

Implementations	EVP	ARMv7a	ARMv5	ARMv7a + Adreno320
Architecture	SIMD, 256 bit wide	Thumb2 instruction set, superscalar, Neon extension	Thumb instruction set, superscalar	SIMD, 16 ALUs, 8 Flops/Alu
Frequency, Mhz	416,(28nm)	1500	1500	325,(28nm)
Data type	16 bit Fixed point	32 bit floating point	32 bit floating point	32 bit floating point
Frame size	320x235	320x235	320x235	320x235
Gaussian pyramid (cycles)	5,129,430	122,670,000	1,066,440,000	13,048,750
Difference of gaussian (cycles)	438,755	22,950,000	45,420,000	**18,840,000**
Local extrema detection (cycles)	4,011,625	NA	NA	NA
Gradient pyramid (cycles)	4,216,347	118,215,000	348,555,000	5,271,500
Gaussian pyramid (milliseconds)	12.3	81.78	710.96	40.15
Difference of gaussian (milliseconds)	1.1	15.3	30.28	**12.56**
Local extrema detection (milliseconds)	9.6	NA	NA	NA
Gradient pyramid (milliseconds)	10.1	78.81	232.37	16.22
Gaussian pyramid (cycles/pixel)	68	1631	14181	174
Difference of gaussian (cycles/pixel)	6	305	604	**251**
Local extrema detection (cycles/pixel)	53	NA	NA	NA
Gradient pyramid (cycles/pixel)	56	1572	4635	70

Table.5.18. Comparison of processing times of EVP with Adreno320 GPU implementation [21]. Numbers shown in bold red are performed on ARM in heterogeneous implementation

Following observations can be made from table.5.18:

- ARMv7a generates much efficient code compared to ARMv5 as mentioned in the paper and thus there is significant improvement in performance for ARMv7a architectures. Hence we shall compare performance with ARMv7a.
- For gaussian pyramid construction we can achieve a speed up of ~24x on EVP compared to ARMv7a CPU. The speed up achieved when compared to Adreno320 GPU is ~2.6x on EVP.
- The authors suggest that the difference of gaussian be performed on ARMv7a. We observe that difference of gaussian pyramid construction provides ~41-50x speed up on EVP.
- For the gradient pyramid we observe a speed up of ~28x compared to ARMv7a CPU and ~1.3x compared to adreno320 GPU.
- Here for local extrema detection the cycle per pixel actually depends on the number of keypoints generated. This number is an over estimate as we extract lot more keypoints than just 95.

The speed up in each case is calculated using the cycle/pixel values calculated since the frequencies and technologies used may vary.

NOTE: The authors in the paper also use optimization techniques in constructing the gaussian pyramids. Here the scales from 0-2 of the octaves (except first octave) are computed directly from down-sampling scales 3-5 of the previous octaves. We do not take into consideration this optimization in our implementation. Else the speed up observed on EVP will be more than 2.6x.

The authors provide profiling data for processing steps involved in SIFT for "graf" image of dimensions 320x256 which has the highest number of detail hence can be considered as worst case scenario in the dataset. Here they compute processing time for extrema detection step and the difference of gaussian pyramid construction. We can use this to estimate the speed up achieved by EVP compared to Adreno320 GPU which is shown in table 5.19. The numbers shown in bold red are the workload partitioning choices made by the authors.

Implementations	EVP (416Mhz)	ARMv7a(1.5Ghz)	Adreno320 GPU (325Mhz)	Read-Back
Gaussian pyramid (cycles)	5,590,430	147,750,000	16,903,250	-
Difference of gaussian (cycles)	478,130	24,120,000	7,371,000	-
Local extrema detection (cycles)	4,261,264	21,195,000	7,637,500	-
Gradient pyramid (cycles)	4,594,926	121,095,000	6,825,000	-
Gaussian pyramid (ms)	13.44	98.5	**52.01**	**14.38**
Difference of gaussian (ms)	1.15	**16.08**	22.68	16.01

Local extrema detection (ms)	10.24	**14.13**	23.5	
Gradient pyramid (ms)	11.05	80.73	**21**	**10.65**
Gaussian pyramid (cycles/pixel)	68	1804	206	-
Difference of gaussian (cycles/pixel)	6	294	90	-
Local extrema detection (cycles/pixel)	52	259	93	-
Gradient pyramid (cycles/pixel)	56	1478	83	-

Table.5.19. Comparison of processing times of EVP, Adreno320 GPU and ARMv7a, standalone implementations on "graf" image [21]. Numbers in bold red are partitioning choices made by authors

Following observations can be made from table.5.19:

- For gaussian pyramid construction we can achieve a speed up of ~26x on EVP compared to ARMv7a CPU. The speed up achieved when compared to Adreno320 GPU is ~3x on EVP. Here they do not consider optimizations on gaussian pyramid construction.
- We observe that difference of gaussian pyramid construction provides ~49x speed up compared to ARMv7a and speed up of 15x compared to Adreno320 GPU.
- For extrema detection stage we observe a speed up of ~5x compared to ARVv7a and a speed up of ~1.8x compared to adreno320 GPU. This is one of the main motivations for performing this stage on EVP.
- For the gradient pyramid we observe a speed up of ~26x compared to ARMv7a CPU and ~1.5x compared to adreno320 GPU

The above results confirm with the previous table and also give an insight about the speed up that is achievable for difference of gaussian and local extrema detection stages compared to stand-alone GPU implementation. The reason for the authors to consider performing the difference of gaussian construction on ARM is because the operations involved are relatively simpler and the read back to ARM costs significant amount of processing time. Instead they read back gaussian pyramid images from GPU and perform difference of gaussian on ARM. On EVP the difference of gaussian pyramid can be constructed very efficiently as shown. So instead of reading back the gaussian pyramid the difference of gaussian pyramid can be read back directly. The gaussian pyramid values are only used in gradient pyramid construction and this is anyways done on EVP. Compared to reading back 30 gaussian blurred images it is quicker and more optimal to read 25 difference of gaussian images directly.

Next we are going to see how EVP can perform if we were to implement the whole the algorithm with heterogeneous implementation similar to what the authors suggest. For this we again use the average processing times benchmarked for 48 image data set used in the paper for different stages of SIFT. For the stages that are not implemented on EVP we re-use the numbers taken on ARM-v7a making the same assumption as the authors that an average of 95 keypoints are extracted per image.

Table.5.20. gives an estimate of the processing time taken for each stage of SIFT on EVP+ARMv7a heterogeneous implementation. Here we re-use the processing times for stages such as orientation assignment, keypoint refinement and description generation of the ARM implementation for the estimate. In the table.5.19 we observed that the processing time for local extrema detection on ARM and on EVP are very close. Hence we can subtract (shown in bold red) the processing time for local extrema detection from the total time taken for all three stages (extrema detection + refining + orientation) on ARM. Thus we can obtain approximate processing time only for keypoint refinement and orientation assignment. Since we are performing extrema detection on EVP the keypoint locations need to be read back to the ARM but this is negligible as the number of keypoints on an average is 95 per image. The processing time for read back of gaussian in case of Adreno320+ARMv7a implementation can be reused to estimate the read back of difference of gaussian pyramid from EVP. This processing time would be lesser since we read back only 25 difference of gaussian images compared to 30 gaussian images. Also the authors mention that this read back also includes processing time for unpacking of gaussian values. Hence it is fair to assume that the estimate for reading back difference of gaussian images will be lesser. Also we reuse the read back processing time for gradient pyramid from Adreno320+ARMv7a implementation.

NOTE: The authors perform descriptor generation on GPU whereas in EVP we have not explored this possibility as yet and this is a key candidate for future work. Hence we cannot estimate the numbers for keypoint generation if it were to be implemented on EVP.

Algorithms	ARMv7a in milliseconds	EVP + ARMv7a partitioning	EVP + ARMv7a in milliseconds	**EVP + ARMv7a estimate(ms)**	Adreno320 + ARMv7a partitioning	Adreno320 + ARMv7a in milliseconds
Gaussian pyramid	81.78	EVP	12.3	**12.3**	GPU	40.15
Difference of gaussian	15.3	EVP	1.1	**1.1**	ARM	12.56
Gradient Pyramid	78.81	EVP	10.1	**10.1**	GPU	16.22
Read back gaussian	-	-	8.32	**8.32**	-	8.32
Read back gradient	-	-	16.75	**16.75**	-	16.75
Local Extrema detection	21.40	EVP	9.6	**9.6**	ARM	20.37
Keypoint refining + Orientation assignment		ARM	21.40	**11.8**		
Descriptor generation	79.74	ARM	79.74	**79.74**	GPU	52.52

Table.5.20. Estimate of Total processing time for heterogeneous EVP implementation. Stages performed on ARM (shown in Orange), EVP (shown in Yellow), Adreno320 GPU (shown in Green) and Memory read-back (Shown in Blue)

In the section.3.1 we proposed a possible data flow for implementing SIFT in a heterogeneous architecture. Keeping this in mind we summarize the estimated processing times in table.5.21. Following are the conclusions that can be draw from table.5.21.

- We achieve 14.3 fps on keypoint detection when compared to 8.5 fps that the authors achieve in GPU implementation. This is a speed up of ~1.7x using EVP.
- For descriptor generation we get 12.5 fps when compared to 19 fps on GPU. This is because we have not implemented descriptor generation on EVP. Where as in GPU implementation the descriptor generation is performed on GPU.

Algorithms	EVP + ARMv7a estimate in milliseconds	Adreno320 + ARMv7a in milliseconds	ARMv7a in milliseconds
Gaussian pyramid	12.3	40.15	81.78
Difference of gaussian	1.1	12.56	15.30
Gradient Pyramid	10.1	16.22	78.81
Read back gaussian	8.32	8.32	NA
Read back gradient	16.75	16.75	NA
Local Extrema detection	9.6	20.37	21.40
Keypoint refining + Orientation assignment	11.8		
Keypoint detection total	**69.97**	**117.02**	**197.43**
Descriptor generation total	**79.74**	**52.52**	**79.74**
Keypoint detection (fps)	**14.3**	**8.5**	**5.0**
Descriptor generation (fps)	**12.5**	**19**	**12.5**
Total EVP/Adreno320	**33.1**	**108.9**	**NA**
Total ARM	**91.54**	**32.9**	**277.17**
Total Read back	**25.07**	**25.07**	**NA**
Maximum FPS	**11**	**9.2**	**3.6**

Table.5.21. Comparison of execution times for EVP and Adreno320 heterogeneous implementations. Stages performed on ARM (shown in Orange), EVP (shown in Yellow), Adreno320 GPU (shown in Green) and Memory read-back (Shown in Blue)

- If we assume that the SIFT extraction is fully pipelined such that both ARM and EVP can perform tasks in parallel then the maximum frames that can be processed is determined by the maximum of the execution times on ARM and EVP (shown in bold red). The read back here is a bottleneck but if we assume that a DMA controller is present on EVP that can asynchronously write values back to ARM then the DMA, EVP and ARM can all run in parallel. Figure.5.6 gives a pictorial description of this pipelining approach.
- In case of ARM only implementation the maximum fps is calculated on the total time taken for both keypoint detection and descriptor generation.

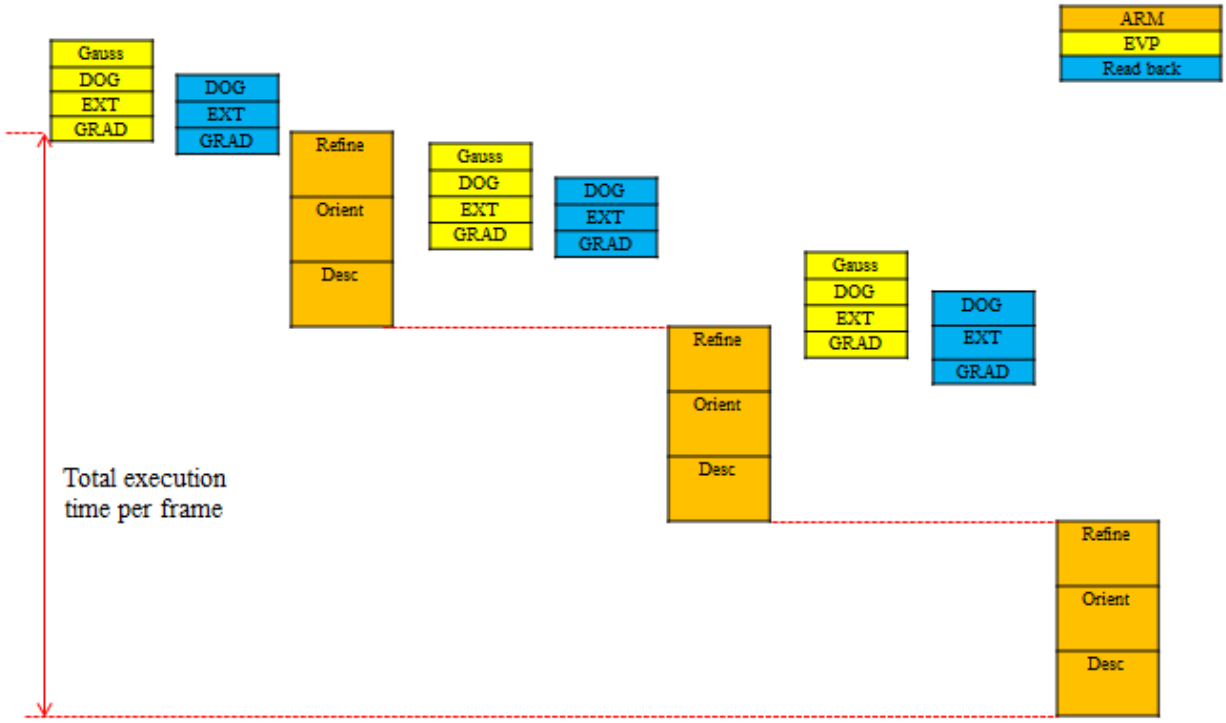

Figure.5.6. Pipelining the SIFT on EVP-ARM heterogeneous implementation

- Approximately ~11 fps can be achieved with EVP+ARM implementation where as in case of GPU implementation we get ~9 fps. This is again without us implementing descriptor generation on EVP. In case of EVP+ARM implementation the ARM processor would be a bottleneck as it takes more processing time. If descriptor generation were to be implemented on EVP then the performance of EVP+ARM will improve further.
- In case of GPU the maximum processing time is taken by GPU and hence the number of frames per second is determined by the GPU processing time.

We see that there is load imbalance when we consider EVP+ARM heterogeneous implementation. The load ratio of EVP to ARM is approximately 1:3. Hence more stages of SIFT need to offloaded on EVP to get maximum performance from heterogeneous implementation. Descriptor generation stage is the most intensive step and hence the possibility of mapping it on EVP needs to be explored since on GPU it has been implemented.

6. Conclusions and Future work

6.1 Conclusions

In this report we have shown how parts of SIFT algorithm can be efficiently mapped on SIMD processors like EVP to exploit data level parallelism. Some of the main conclusions that we can draw from this thesis work are:

- The floating point representations 8.7 and 10.5 formats supported by EVP are not suitable for performing SIFT algorithm as the precision is very important for extracting SIFT keypoints.
- With 16 bit fixed point representation it is possible to extract keypoint locations with 90% accuracy.
- Performance of fixed point implementation is better than that of floating point implementation in terms of the processing time on EVP.
- The EVP architecture with 32 general purpose registers gives better performance when compared to 16 register architectures. More than 50% of cycles are reduced using 32 register architecture.
- An average vector speed up of at least ~11.2x for gaussian pyramid construction, ~12.3x for difference of gaussian construction and ~12.6x for extrema detection can be theoretically achieved using EVP compared to sequential implementation. Practically we achieve better speed up than this.
- Real time performance of SIFT on EVP can be achieved with approximately 11 frames/second in heterogeneous architecture such as ARMv7a+EVP.
- The ratio of EVP processing time to that of ARM is 1:3 and there is a load imbalance in this implementation. Thus more stages of SIFT need to be offloaded on to EVP to achieve better performance.
- For SIFT to perform efficiently on EVP it is necessary that there is sufficient amount of data memory. The amount of data memory should at the very least be able to simultaneously store gaussian pyramid images, difference of gaussian pyramid images, gradient pyramid images for the first octave.

6.2 Future work

The mapping of SIFT on EVP is not complete and hence there is lot of room for exploring possibility of implementing rest of the stages. Some of the key candidates for future work are:

- The implementation of gaussian kernels which are the most intensive algorithms can be further optimized to give better speedup. Calculating masks on the fly and padding input images with vector of zeroes can reduce the complexity of the outer loops in the implementation and can further improve performance.
- The keypoints extracted in refining stage are discrete coordinates. These are used in experiments to check for matching. It is highly important to also check the accuracy of interpolated sub pixel values of these keypoint locations. This will actually determine the accuracy of the final descriptor generated.
- Descriptor generation stage is implemented in some GPU implementations. This can be used as a reference for implementing this stage on EVP.
- The difference of gaussian pyramid needs to be read back from ARM in order to refine keypoints. We can avoid this read back by implementing refining on EVP. The algorithm is essentially sequential and involves control logic but it can be implemented on scalar computation unit of EVP.
- In a heterogeneous implementation we see that EVP processing time is approximately 1/3rd of ARM processing. Hence more stages have to be offloaded to EVP from ARM to achieve higher frame rates.
- In the current implementation we match exact keypoint locations. Instead a preset window of radius 1-2 pixels in x and y dimension can be used to find those keypoints that are very close from the center. This way more keypoints could be matched which are affected by precision loss.
- Exploring the possibility of reducing error in the gaussian values by using techniques like rounding instead of truncation for fixed point implementation.
- Energy consumption needs to be measured and benchmarked for performing SIFT algorithm on EVP.

7. References

[1] *Global Smartphone Shipments Reach a Record 990 Million Units in 2013*, Strategy Analytics, retrieved from http://blogs.strategyanalytics.com/WSS/post/2014/01/27/Global-Smartphone-Shipments-Reach-a-Record-990-Million-Units-in-2013.aspx - on Jan 27, 2014.

[2] Y. Mahfoufi, "Bringing Mobile Multimedia to Best-In-Class Smartphones", Strategic white paper, Alcatel Lucent, 2010.

[3] *How to Leverage Augmented Reality for Your Business*, A Metaio AR White Paper, retrieved from www.metaio.com – 2014.

[4] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," Int. J. Computer Vision, vol. 60, no. 2, pp. 91–110, 2004.

[5] *Are tablets up to the task of accurate color testing?*, dot color, retrieved from http://dot-color.com/category/terminology/ - on December 19, 2012

[6] S.L.M.Goossens, "An SIMD register file with support for dual-phase decimation and transposition", Master thesis, TU/e, June 6 2010.

[7] Szeliski. R, "Computer vision: algorithms and applications. Springer", ISBN 978-1-84882-934-3, 2011

[8] F.-C. Huang, S.-Y. Huang, J.-W. Ker, and Y.-C. Chen, "High-performance SIFT hardware accelerator for real-time image feature extraction," IEEE Trans. Circuits and Syst. for Video Technol., vol. 22, no. 3, pp. 340–351, 2012

[9] K. Mizuno, H. Noguchi, G. He, Y. Terachi, T. Kamino, T. Fujinaga, S. Izumi, Y. Ariki, H. Kawaguchi, and M. Yoshimoto, "A low-power real-time SIFT descriptor generation engine for full-HDTV video recognition," *IEICE Trans. Electron.*, vol. 94, no. 4, pp. 448–457, Apr. 2011.

[10] J. Jiang, X. Li and G. Zhang, "SIFT Hardware Implementation for Real-Time Image Feature Extraction," 2013

[11] S. Zhong, J. Wang, L. Yan, L. Kang, and Z. Cao, "A real-time embedded architecture for SIFT," *J. Syst. Arch.*, vol. 59, no. 1, pp. 16–29, Jan. 2013.

[12] J. Wang, Z. Sheng, L. Yan, and Z. Cao, "An embedded system-on-a-chip architecture for real-time visual detection and matching," IEEE Trans. Circuits Syst. Video Technol., vol. 24, no. 3, pp. 525–538, Mar. 2014.

[13] V. Bonato, "A parallel hardware architecture for scale and rotation invariant feature detection," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 18, no. 12, pp. 1703–1712, Dec. 2008.

[14] L. Chang and J. Hernández-Palancar, "A hardware architecture for SIFT candidate keypoints detection," in *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*. New York, NY, USA: Springer-Verlag, 2009, pp. 95–102.

[15] K. Mizuno, H. Noguchi, and G. L. He, "Fast and low-memorybandwidth architecture of SIFT descriptor generation with scalability on speed and accuracy for VGA video," in *Proc. Int. Conf. FPLA*, Sep. 2010, pp. 608–611.

[16] J. Oh, G. Kim, I. Hong, J. Park, S. Lee, J.-Y. Kim, J.-H. Woo, and H.-J. Yoo, "Low-power, real-time object-recognition processors for mobile vision systems," Micro, IEEE, vol. 32, no. 6, pp. 38–50, 2012.

[17] Zhang Q, Chen Y, Zhang Y, et al. *SIFT implementation and optimization for multi-core systems[C]*.Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on. IEEE, 2008: 1-8.

[18] Xin L, Wenjie C, Tao M, et al. *Real-time algorithm for sift based on distributed shared memory architecture with homogeneous multi-core dsp[C]*.Intelligent Control and Information Processing (ICICIP), 2011 2nd International Conference on. IEEE, 2011, 2: 839-843.

[19] Hao Feng; Li, E.; Yurong Chen; Yimin Zhang, "Parallelization and characterization of SIFT on multi-core systems," *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on* , vol., no., pp.14,23, 14-16 Sept. 2008

[20] Rister, B.; Guohui Wang; Wu, M.; Cavallaro, J.R., "A fast and efficient sift detector using the mobile GPU," *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on* , vol., no., pp.2674,2678, 26-31 May 2013

[21] Wang, G., Rister, B., & Cavallaro, J. R. (2013). Workload analysis and efficient OpenCL-based implementation of SIFT algorithm on a smartphone. In*Proceedings in IEEE global conference signal and information processing (GlobalSIP)* (pp. 759-762).

[22] S. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, “Feature tracking and matching in video using programmable graphics hardware,” Machine Vision and Applications, vol. 22, pp. 207–217, 2011.

[23] S. Warn, W. Emeneker, J. Cothren, and A. Apon, “Accelerating SIFT on parallel architectures,” in IEEE International Conference on Cluster Computing and Workshops, Sept. 2009, pp. 1–4.

[24] H. Xie, K. Gao, Y. Zhang, J. Li, and Y. Liu, “GPU-based fast scale invariant interest point detector,” in IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP), March 2010, pp. 2494 –2497.

[25] Lindeberg, T. 1994. Scale-space theory: A basic tool for analysing structures at different scales. Journal of Applied Statistics, 21(2):224-270.

[26] A. Vedaldi and B. Fulkerson, “VLFeat: An open and portable library of computer vision algorithms,” http://www.vlfeat.org/.

[27] Demo Software: SIFT Keypoint Detector, Version 4, July 2005, David Lowe, and Retrieved from http://www.cs.ubc.ca/~lowe/keypoints/siftDemoV4.zip