SIFT Feature Extraction on a Smartphone GPU Using OpenGL ES2.0

by

Guy-Richard Kayombya

S.B. C.S, M.I.T., 2009

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May, 2010
[June 2010]

Author_____
Department of Electrical Engineering and Computer Science
May 17, 2010

Certified by_____
Chad Sweet, Engineer, Sr Staff/Mgr Qualcomm
VI-A Company Thesis Supervisor

Certified by_____
Prof. Seth Teller, EECS
M.I.T. Thesis Supervisor

Accepted by_____
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

SIFT Feature Extraction on a Smartphone GPU
Using OpenGL ES2.0
by
Guy-Richard Kayombya

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 2010

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

SIFT describes local features in image used for object recognition in a vast array of application, such as augmented reality, panorama stitching. These applications are becoming very popular on Smartphones but also require considerable amount of computing power. GPUs offer a significant amount of untapped computing power that can help increase performance and improve user experience. We explore the feasibility of parallel heterogeneous computing on current generation of Smartphone. We show that the CPU and GPU can work in tandem to solve complex problems. However the mobile platform remains very restrictive requires a lot of effort from the programmer but does not achieve the same performance gains as observed on the PC.

Thesis Supervisor: Seth Teller
Title: Prof of Computer Science and Electrical Engineering

Acknowledgements

It is a pleasure to thank those who made this thesis possible. I am extremely grateful to my manager at Qualcomm Chad Sweet for providing me with the opportunity to work on this project. He has provided constant support, very insightful feedback and his help was critical in making this thesis a success.

I am extremely indebted to Ming-Chang Tsai. Not only was he a great mentor from whom I learned a lot but he was also a great team mate and working with him was a remarkable experience. His technical knowledge, his experience, the quality of his advice and his sense of humor were integral to a wholesome work environment.

I would like to thank my thesis supervisor Prof Teller. He gracefully accepted to supervise my thesis work, was always available to answer my questions; he offered unwavering help with all the administrative entanglements related to the thesis.

I would also like to thank Serafin Diaz, Slawek Grzechnik, Meetu Gupta, Meghana Haridev, Srinivas Pillappakkam, Parixit Aghera and the entire Blur team for the personal and technical help they provided. I couldn't have asked for a more supportive team; it would not have been a smooth ride without them.

Lastly thanks go to my parents, siblings and friends at MIT and Qualcomm for constantly pushing me to finish and providing company and moral support.

Table of Contents

Table of Figures

1. Introduction

The computational power of the Graphics Processing Units (GPU) in current Smartphone is rapidly increasing to support more demanding graphical applications. At the same time GPU vendors are delivering platforms that give more control to the programmers which which enable non-graphics or general purpose computing. With this increase in programmability, the GPUs on these mobile devices are ultimately becoming coprocessors to which computational work can be offloaded in order to free up the CPU and increase applications performances through parallel processing.

The Scale Invariant Feature Transform algorithm detects extracts, localizes and describes distinctive, repeatable local features in images. The SIFT features can be used for object recognition. They describe the appearance of the object at particular areas of interest and are supposed to be invariant to scale, rotation and illumination. The SIFT algorithm is computation intensive and data parallel which makes it a prime candidate for the multi-heterogeneous-core processing on CPU and GPU.

The objective of this thesis is to implement the SIFT algorithm on the Z400 family of GPU developed by Qualcomm and identify in the process the limitations of the current generation of GPUs. The results of the research will help discover areas of potential improvement and offer recommendation for the support of general purpose computing for the next generation of GPUs.

2. Background and Motivation

The Qualcomm Research Center (QRC) is working on a project whose objective is to develop Augmented Reality (AR) enablers to incorporate into future Qualcomm products and services. Current AR applications designed for Smartphones mainly rely on sensors such as GPS, digital compasses and accelerometer to overlay relevant information on top of the live camera view. The data read from these sensors can be noisy and lead to incorrect or unstable augmentation, thus negatively impacting the user experience. Object recognition and tracking is used to supplement the sensors and precisely identify the position of objects that are suitable for augmentation. Naturally, this strategy eliminates the noise problem mentioned earlier but also introduces other challenges. The project team determined SIFT was a good representation of the type of computer vision algorithms to be implemented on Smartphones in the near future. Nonetheless, SIFT is a very computation intensive algorithm and it is imperative to take advantage of all the computational power available on the target platforms: QSD8x50 and MSM7x30 chipsets. The QSD8x50 features a 1GHZ Scorpion ARM processor, a 133 MHZ Z430 GPU. The MSM7x30 has an 800MHz ARM processor and 192 MHz Z460 GPU.

It has been proven on the PC that GPU implementation of SIFT can be 10x faster that it's optimized counter parts on CPU[3]. This is the main motivation behind the research proposed in this thesis; we aim to investigate whether a similar performance gain can be attained on Smartphones.

3. The SIFT Algorithm

The SIFT algorithm published by D.Lowe[1] takes an image as input and outputs a set of distinct local feature vectors. The algorithm is partitioned in four stages.

- Scale Space Extrema Detection

This first step is where the SIFT keypoints are detected. This is accomplished by convolving the input image,$I(x,y)$ with Gaussian filters of varying widths,$G(x,y,k_i\sigma)$ and taking the difference of Gaussian-blurred images,$L(x,y,k_i\sigma)$. This creates a Difference of Gaussians "scale space" function defined as follows:

$$D(x,y,k_{i)} = \big(G(x,y,k_{i+1}) - G(x,y,k_i)\big) * I(x,y) = L(x,y,k_{i+1}) - L(x,y,k_{i+1})$$

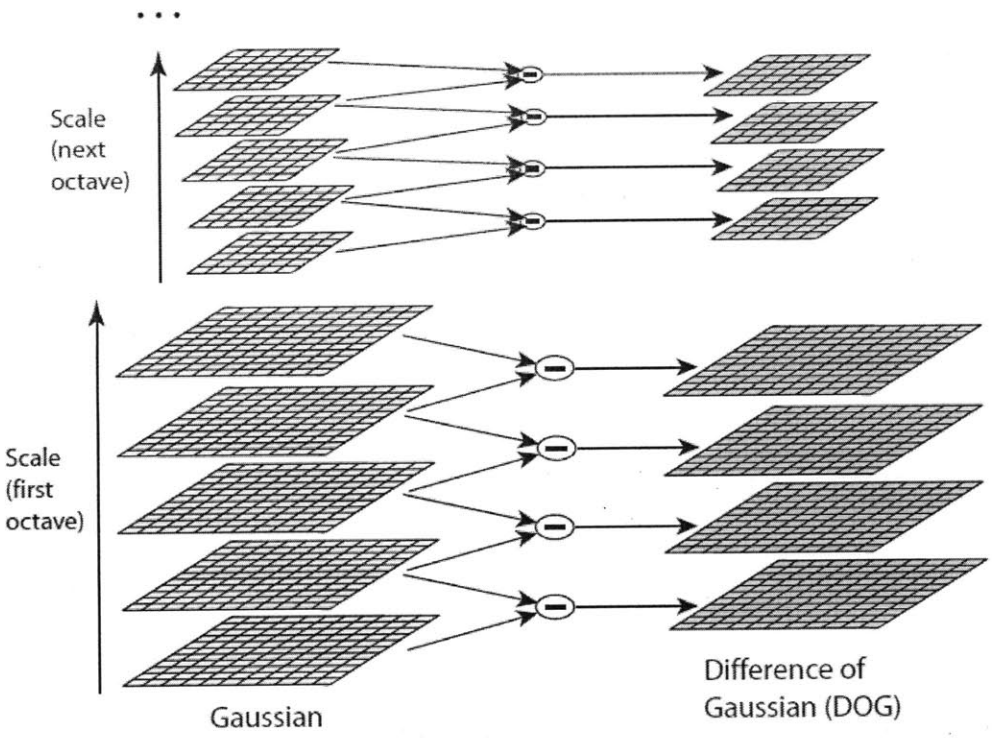


Figure 1: For each octave of scale space, the initial image is repeatedly convolved with Gaussians to produce the set of scale space images shown on the left. Adjacent Gaussian images are subtracted to produce the difference-of-Gaussian images on the right. After each octave, the Gaussian image is down-sampled by a factor of 2, and the process repeated.

Figure 1:Gaussian and DoG pyramids, (Lowe)

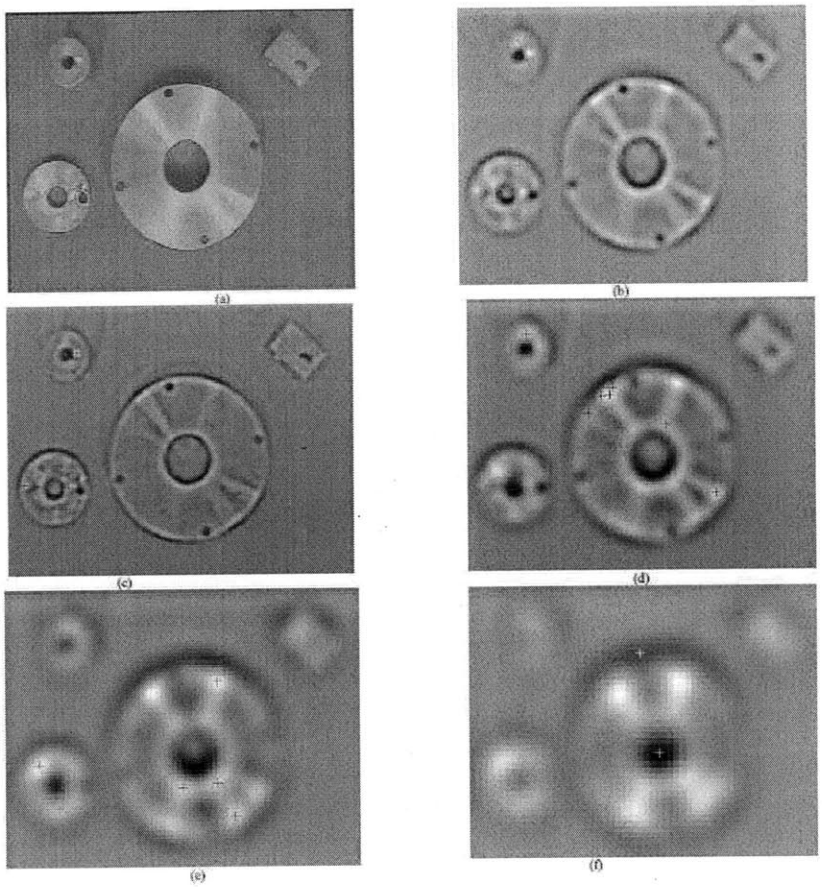


Fig.1 this figure shows Extremum in Difference-of-Gaussian Images.(a) The original image.(b) extrema in layer 1 of octave 2 image, shows with green color. (c) extrema with layer 2 of octave 2 image, shows with red color. (d) extrema in layer 3 of octave 2 image, shows with blue color. (e) extrema in layer 4 of octave 2 image, shows with purple color. (f) extrema in layer 5 of octave 2 image, shows with azury color.

Figure 2: This figure shows local extrema in difference of Gaussians images (Xiaohua and Weiping)

Keypoints are the local maxima/minima of the DoG function. This is done by comparing each pixel to its 26 immediate neighbors (8 on the same scale and 9 corresponding neighbors on each of the 2 neighboring scales). A pixel is selected as a candidate pixel if it is strictly greater or strictly smaller than all its 26 neighbors.

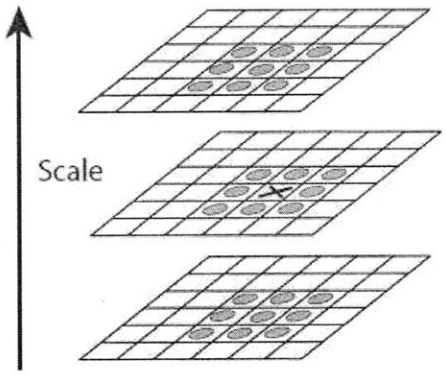


Figure 2: Maxima and minima of the difference-of-Gaussian images are detected by comparing a pixel (marked with X) to its 26 neighbors in 3x3 regions at the current and adjacent scales (marked with circles).

Figure 3: Local extrema extraction (Lowe)

- Keypoint Refinement:

The scale-pace extrema detection generates too many keypoints that are potentially unstable. First, this stage refines the location of the keypoints by improving the localization to a sub-pixel accuracy using a Taylor series expansion of the DoG function.

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

where D and its derivatives are evaluated at the position of the candidate keypoint. The interpolated location of the new extremum is given as

$$\boldsymbol{z} = -\left(\frac{\partial^2 \boldsymbol{D}}{\partial \boldsymbol{x}^2}\right)^{-1}\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{x}}$$

Second, this stage rejects the keypoints that have low contrast (hence sensitive to noise). The value of keypoint at the refined location is given as

$$\boldsymbol{D}(z) = \boldsymbol{D} + \frac{1}{2}\frac{\partial \boldsymbol{D}^{-1}}{\partial \boldsymbol{x}}\boldsymbol{z}$$

If $|\boldsymbol{D}(\boldsymbol{z})|$ is smaller than a certain threshold the keypoint is discarded. Finally, the keypoints that are poorly located on edges are excluded. In these cases the principal of curvature across the edge would be significantly larger than the curvature in the perpendicular direction. The curvature is retrieved from the Eigen-values of the second-order Hessian Matrix H. The ratio of principal curvature is directly related to the ratio of the trace and determinant. If the latter is above a certain threshold the keypoint is deemed poorly located and rejected.

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

$$if \ \frac{(D_{xx} + D_{yy})^2}{D_{xx}D_{yy} - (D_{xy})^2} > \frac{(r+1)^2}{r}, reject\ keypoint$$

- Orientation Assignment:

In this step, each refined keypoint is assigned one or more orientations based on the gradient values over its neighboring region. This generates keypoints that are invariant to rotation as they can be described relative to this orientation. First, for a given scale of Gaussian-blurred image the gradient magnitude $m(x,y)$ and orientation $\theta(x,y)$ is precomputed.

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

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

Then an orientation histogram with 36 bins, each covering 10 degrees is formed. Each sample from the window of neighboring pixels is added to the corresponding orientation bin and weighted by its gradient magnitude and a by a Gaussian-weighted circular window. Finally, the orientation with the highest peak and local peaks that are within 80 % of the highest peak are assigned to the keypoint.

- Keypoint Description

This stage computes a descriptor vector for each keypoint such that the descriptor is highly distinctive and invariant to variations such as illumination, 3D viewpoint, etc. We first sample the image gradients in a 16X16 region around the keypoint. The magnitudes of the gradients vectors are then weighted by a Gaussian function with σ equal to 1.5 times of the scale of the keypoint. The orientation of the gradient is rotated relative to the keypoint orientation in order to achieve invariance to rotations. A 4x4 array of histogram with 8 bins each is computed from the values of the gradients magnitude and orientation in a 4x4 subsection of the initially sampled neighborhood. Each bin represents a cardinal direction and its value corresponds to the sum of the gradients magnitude near that direction. The descriptor then becomes a vector of all the values of these histograms. So a histogram contains 16x16x8=128 elements.

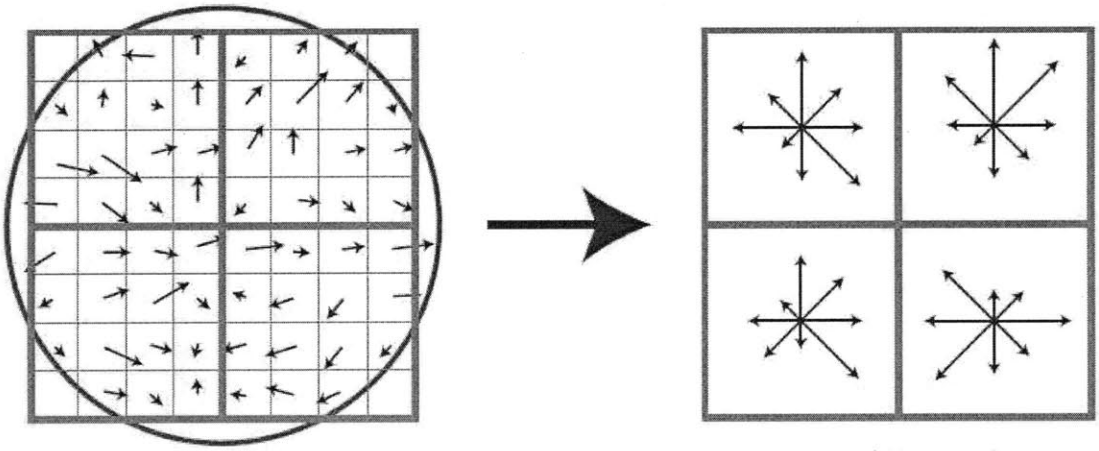


Figure 7: A keypoint descriptor is created by first computing the gradient magnitude and orientation at each image sample point in a region around the keypoint location, as shown on the left. These are weighted by a Gaussian window, indicated by the overlaid circle. These samples are then accumulated into orientation histograms summarizing the contents over 4x4 subregions, as shown on the right, with the length of each arrow corresponding to the sum of the gradient magnitudes near that direction within the region. This figure shows a 2x2 descriptor array computed from an 8x8 set of samples, whereas the experiments in this paper use 4x4 descriptors computed from a 16x16 sample array.

Figure 4:Keypoint Descriptor (Lowe)

4. General Purpose Computing on GPU (GPGPU)

4.1 What algorithms are most suitable for GPU optimization?

Computer graphics involves highly parallel computations that transform input streams of independent vertices and pixels to output streams of color pixels. To accelerate computer graphics, modern GPUs come with many programmable processors that apply a kernel computation to stream elements in parallel. Essentially, the same computation is applied to streams of many input elements but each element has no dependencies on other elements. Therefore algorithms that are best suited for GPGPU have to share two principal characteristic: data parallelism and independence. These two attributes can be combined into a single concept known as *arithmetic intensity*, which is the ratio of computation to bandwidth. SIFT is an ideal candidate for GPU optimization. It essentially consists of specialized kernel operations on large streams of independent pixels and keypoints.

4.2 Mapping traditional computational concepts to the GPU

Programming for the GPU involves a set of tricks to map general purpose programming concepts from the CPU to the graphics specific nomenclature and framework on the GPU. The following sections will described the tricks in detail

4.2.1 Graphics pipeline Overview

There are two types of programmable processors on current GPUs: the vertex processor and fragment processor. The vertex processors process streams of vertices that define 3D objects in computer graphics. They apply vertex programs (aka. Vertex shaders) to transform the 3D virtual position of each vertex to its projected 2D coordinates in the screen. For more complex 3D effects vertex processors can manipulate attributes such as such color, normal, texture coordinates. The output of the vertex processor is passed on to the next stage of the pipeline

where each set of 3 transformed vertices is used to compute a 2D triangle. The triangles are then rasterized to generate streams of fragments. The final pixels in the frame-buffer are generated from each fragment. A fragment contains the information needed to compute the color of pixel in the final image, including color, depth, and (x,y) position in the frame-buffer. The fragment processors apply fragment programs (aka fragment shaders) to each fragment in the stream to output the final color pixel. In computer graphics fragment shaders typically perform operation such as lighting, shadows, translucency and other phenomena.

4.2.2 Available Computational resources

The computational resources available on the Z400 series of GPUs that we can take advantage of for general purpose computing are the following:

- Programmable parallel processors:

The basic primitives in Computer graphics are 4 elements vectors in homogenous space (x,y,z,w) and 4-component colors (red, blue, green, opacity). For this reason vertex and fragment processors have hardware to processes 4-components vectors. As mentioned earlier, the device has multiple basic processing units which apply the same kernel instructions to a large stream of elements. For the Z400 family, the distinction between the vertex and fragment processor is only functional because the same hardware resources are used for both. Most of the SIFT computation will take place in the fragment programs because it's the last stage of the pipeline and produces a direct output to the frame-buffer. The vertex program will primarily be used to setup variables to be linearly interpolated by the rasterizer.

- Rasterizer:

Each set of 3 vertices transformed by the vertex processor is used to generate triangles in the form of edge equations. The rasterizer's role is to generate stream of fragments from this

triangles. The rasterizer also performs linear interpolation of per-vertex attributes. As such we can think of the rasterizer as an address interpolator. Below we show how memory addresses are represented as texture coordinates.

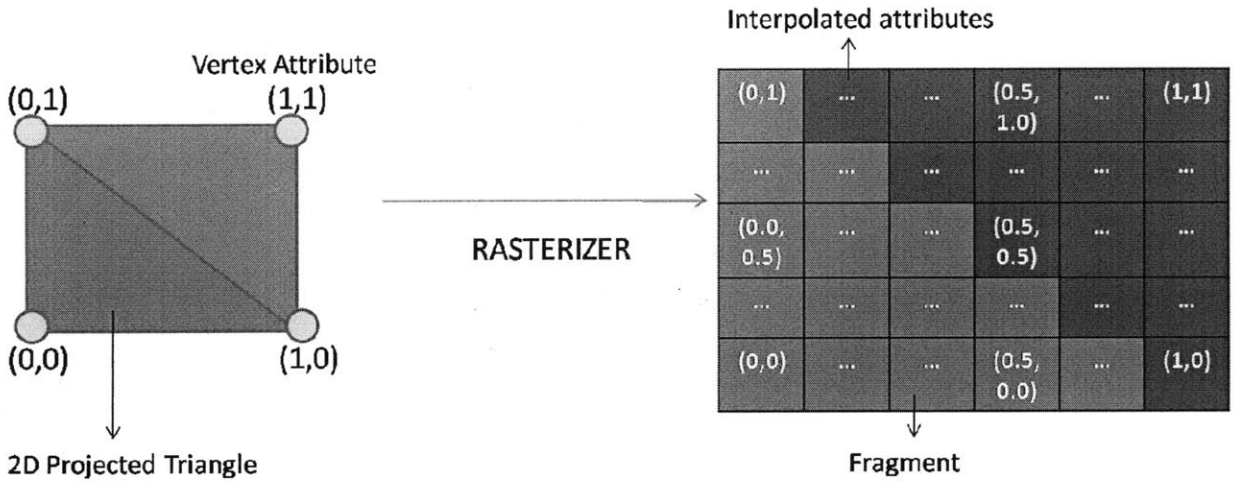


Figure 5: The rasterizer generates fragments from the transformed geometry and interpolates per-vertex attributes

- Texture-Unit:

Shader processors can access memory in the form of textures. We can think of the texture unit as a read-only memory interface.

- Render-To-Texture:

In computer graphics when an image is generated by the GPU, it is typically written to the frame-buffer memory to be displayed. However some 3D effects require the image to be written to texture memory. This render-to-texture feature is essential for GPGPU, because it allows direct feedback of GPU output to input without involving host processor-GPU interaction.

4.2.3 From CPU to GPU concepts

- Streams: GPU Textures = CPU Arrays

Textures and vertex arrays are the fundamental data structures on GPUs. As mentioned earlier fragment processors are more adequate for GPGPU than vertex vertex processors. Therefore data arrays on the CPU can be represented as textures on the GPU.

- Kernels: GPU Fragment Programs = CPU "Inner Loops"

To apply a computation kernel to a stream of elements on the CPU, we store the elements in an array and iterate over them using a loop. The instructions inside the loop are the kernel. On the GPU, the same instructions are written inside the fragment program. There are two levels of parallelism: one is afforded by the many parallel processors and the other one is enabled by the 4-vector structure of GPU arithmetic. Each color, RGBA, is a vector.

- Render-to-Texture = Feedback

Complex algorithms in GPGPU can have inter-dependencies between stream elements. Such kernels can have low arithmetic intensity. However the computation can be broken into independent kernels executed one after the other. In this case a kernel must process an entire stream before the next kernel can proceed. Thanks to the unified memory model on the CPU, feedback is trivial to implement: memory can be read or written anywhere in a program. Feedback is more difficult to achieve on the GPU, we must use render-to-texture to write the results of a fragment program to memory so they can then be used as input to future programs.

- Geometry Rasterization = Computation Invocation

To run a program in GPGPU,we need to draw geometry. The vertex processor will transform the geometry, the rasterizer will determine which pixel in the output buffer it covers and generate a fragment for each one and finally the kernel described in the fragment program will be executed. The kind of geometry to draw depends on the application. In GPGU we are typically processing elements of a 2D rectangular stream. Therefore the most common invocation is a quadrilateral.

- Texture Coordinates = Computational Domain, Vertex Coordinates = Computational Range

Any computation has an input domain and an output range. On the CPU , for example, the simple sub-sampling of two 2D dimensional array of width W and height H by a factor of k will be defined as follows.

```
for (i = 0; i < W/k; i++)
      for(j = 0; j < H/k;i++)
            a[i][j] = A[i*k][j*k]
```

The number of nested loops defines the dimensionality and the limits of the for-loop define the range while the domain is controlled by the variable indexing. GPUs provide a simple way to deal with this, in the form of texture coordinates and vertex coordinates. As mentioned earlier, the rasterizer generates a fragment for each pixel that will be covered by geometry and each fragment is processed to create the final pixel. The vertex coordinates determined what region of the output buffer will be covered and thus determine the number of fragments that will be generated which is essentially the range of the computation. For the CPU example mentioned

above the four vertices will have the following coordinates (-W/2,H/2)*C, (-W/2,-H/2)*C, (W/2,H/2)*C and (W/2,-H/2)*C where C is a constant that depends on the various factor. Texture coordinates are defined by the programmer and assigned to each vertex during geometry creation. The rasterizer linearly interpolates the coordinates at each vertex to generate a set of coordinates for each fragment. The interpolated coordinates are passed as input to the fragment processor. In computer graphics, these coordinates are used as indices for texture fetches. For GPGPU, we can think of them as array indices, and we can use them to control the domain of the computation. As mentioned previously, the basic geometry for 2D GPGPU is a quadrilateral. The four texture coordinates are usually (0,0), (0,Y),(X,0),(X,Y) which are respectively attached to the top left, top right, bottom left and bottom right vertices. The rasterizer will generate coordinates sampled at an interval of X/W on the horizontal dimension and Y/H in the vertical dimension. For the 2D scenario presented above X=Y=k.

4.3 Z400 specification

The GPUs in the Z400 family are designed by Qualcomm and featured on the Snapdragon and MSM7x30 platforms. They provide the computation power to run state of the art 3D application such as games and complex user interface on mobile devices. The Z430 has one stream processor clocked at 133 MHz that can run up to 32 software threads at a time which can each process 4 stream elements in parallel. The Z460 comes with one steam processor clocked at 192 MHz that can run up to 32 software thread at a time which can each process 16 stream elements in parallel.

4.4 OpenGL ES 2.0

The Z400 family of GPU supports the OpenGL ES 2.0 open standard. The standard defines a graphics pipeline with two programmable stages; the vertex and fragment shaders separated by a rasterizer as shown in the figure below. The depth/stencil test, colour buffer blending, and dithering stages of the pipeline are inactive when rendering to framebuffer holding floating point data. This is an unfortunate limitation considering that these stages provide usefull hardwares ressources for accumlation, averaging, and flow control operations.

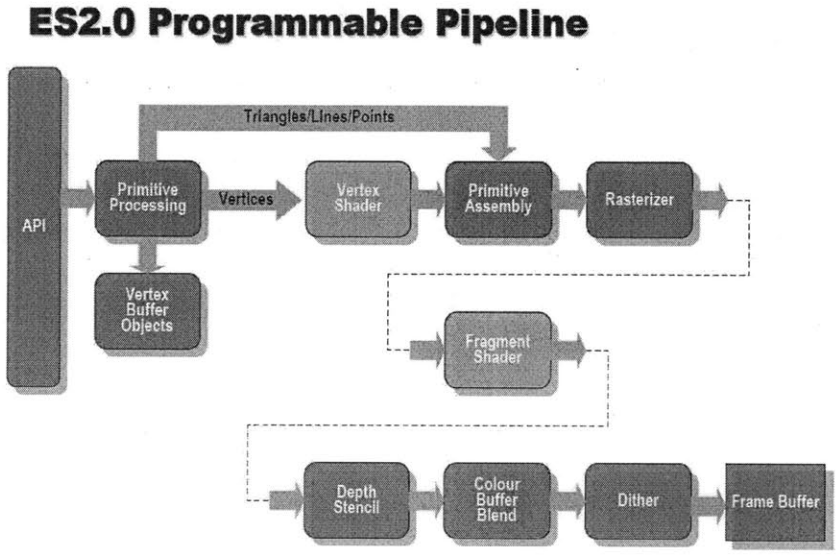


Figure 6: OpenGLES 2.0 introduces a programmable graphics pipeline that enables GPGPU on mobile devices for the first time.

The vertex shader defined by OpenGL E.S 2.0 takes as input 8 vertex specific attributes These variables store 4 elements as floating point vectors such as the position, texture coordinates, etc. The host machine can pass constant to the vertex shader through the uniform variables. The vertex shader can output up to 8 varying variables that will be interpolated by the rasterizer. The predefined variable gl_Position stores the new 3D virtual position of the vertex being processed.

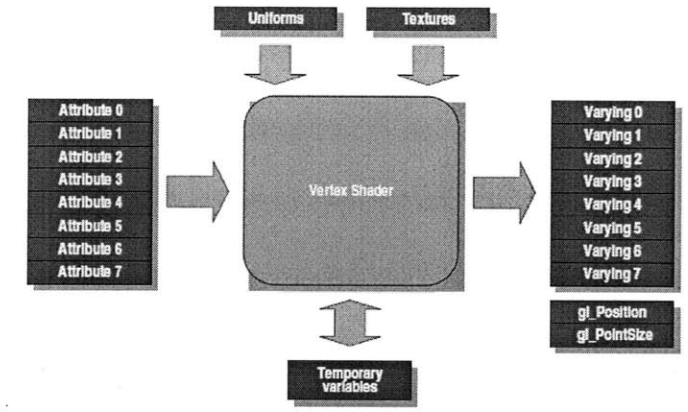


Figure 7: Vertex Shader

The fragment shader defined by OpenGL ES 2.0 takes in input 8 variying 4-vectors interpolated by the rasterizer. The gl_FragCoord predefined input variable stores the coordinates in 2D of the fragment to process. The output of the fragment shader is stored in the variable gl_FragColor.

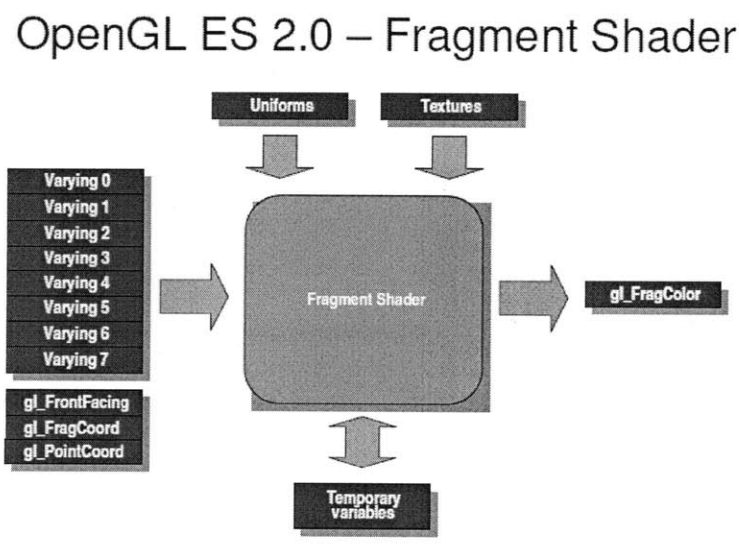


Figure 8:Fragment Shader

OpenGL ES 2.0 also introduces the concept of frambebuffer objects (FBO) which allow for simpler and more efficient render-to-texture. FBOs allow the programmer to easily switch

out the texture which the GPU renders to. They eliminate any overhead associated with alternative methods such as context switching or data readback from the GPU's memory to the host CPU. Although FBOs provide more flexibilty, the OpenGL ES 2.0 API still has some serious shortcomings that severly undermine GPGPU on the Z400. For instance, floating point framebuffers are not supported. This limitation basically renders any GPGPU that requires floating point outputs impossible on the Z400. Fortunatly the team in charge of the OpenGL E.S 2.0 driver at Qualcomm was able to come up with an internal extension to circumvent this limitation. Another shortcoming of the ES 2.0 standard is the maximum number of render target supported by the framebuffer. The framebuffer only supports one render target, which means that one can only output 4 values per pixel i.e the 4 elements of gl_Fragcolor. This limitation will heavily influence the design of the GLES-SIFT proposed in the next section.

5. GLES-SIFT architecture

The SIFT algorithm described earlier can be functionally divided into three stages. The first one is Pyramid building which consist of image doubling, Gaussian filtering and Difference-of-Gaussian generation. The second one is keypoints determination which consists of extrema detection and keypoint refinement. The third one is features generation which consists of orientation generation and descriptor generation. These operations can be further categorized into two types of processing. One that we will call "pixel-rate processing" which consists of per-pixel operations and another that we will call "keypoint-rate processing" which consists of per-keypoint operations. In pixel-rate processing the input is two-dimensional, the arithmetic intensity is low and the kernels execute uniformly. In keypoint-rate processing the

input size is considerably smaller, the arithmetic intensity is very high and the kernels are not uniform.

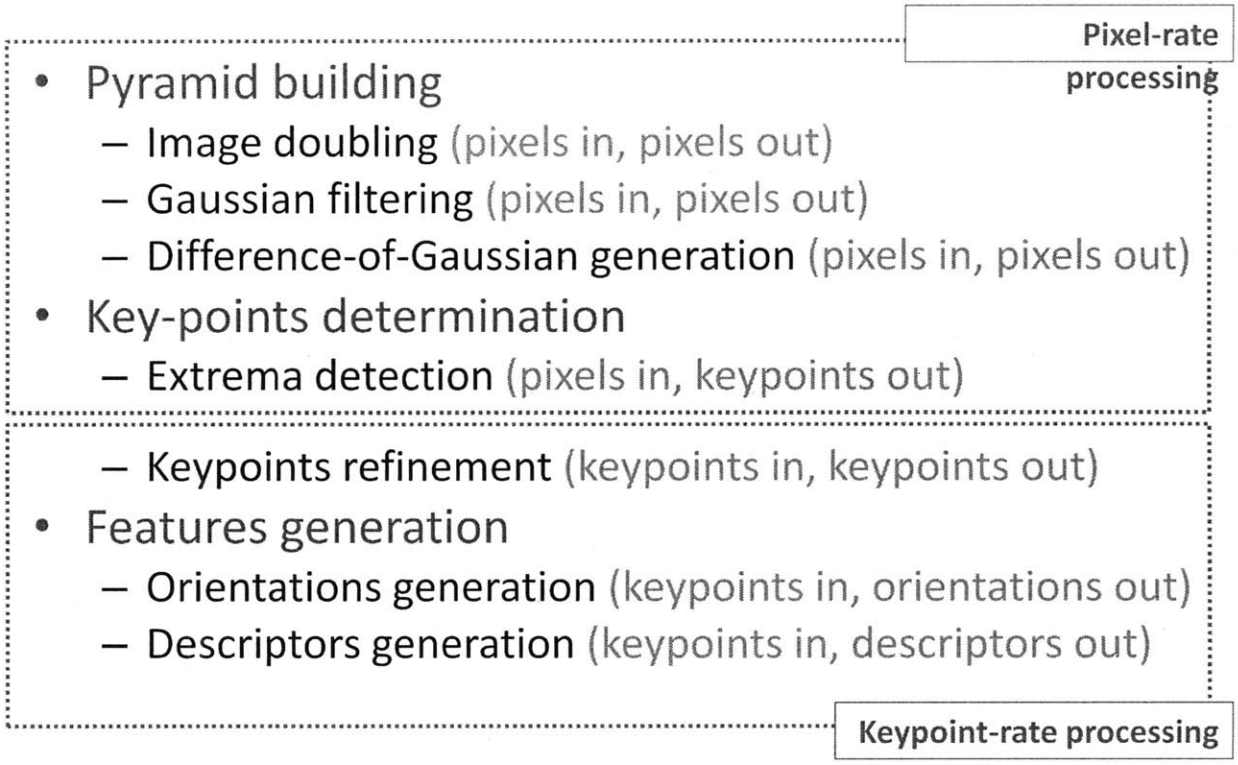


Figure 9: High-Level functional overview of GLES-SIFT

5.1 Data Flow

The image from the camera buffer is first up sampled then smoothed to form the base of the Gaussian pyramid which is referred to I(-1,-1) in the figure below. The rest of the images in octave -1 are obtained by smoothing I(-1,1) with Gaussian function of sigma 1.5×2^{scale}. The difference of Gausian images D(o,s) are obtained by taking the subtracting I(o,s-1) from I(o,s) where o and s respectively refer to the octave and scale and s > -1. For the remaining octaves, the first three images of the Gaussian pyramid and the Difference of Gaussian function i.e I(o>-1,s<2) and D(o>-1,s<2) are obtained by sub sampling the corresponding last three images from the previous octave i.e. I(o-1,s+3) and D(o-1,s+3). The rest of the images I(o,s>1) and D(o,s>1) are computed just like the images in octave -1.

For each octave we then compute the maximum and minimum for each pixel of DOG image across all the scales to obtain images M(o,-) and M(o,+). Then we compare the three middle DOG images D(o,1), D(o,2), D(o,3) to the M(o,-) and M(o,+) to determine the coarse and sparse keypoint 2D map, K(o). Finally the coarse keypoint map is compacted to form the 1D compact keypoint map o(o) which in turn is used to produce the refined keypoint map.

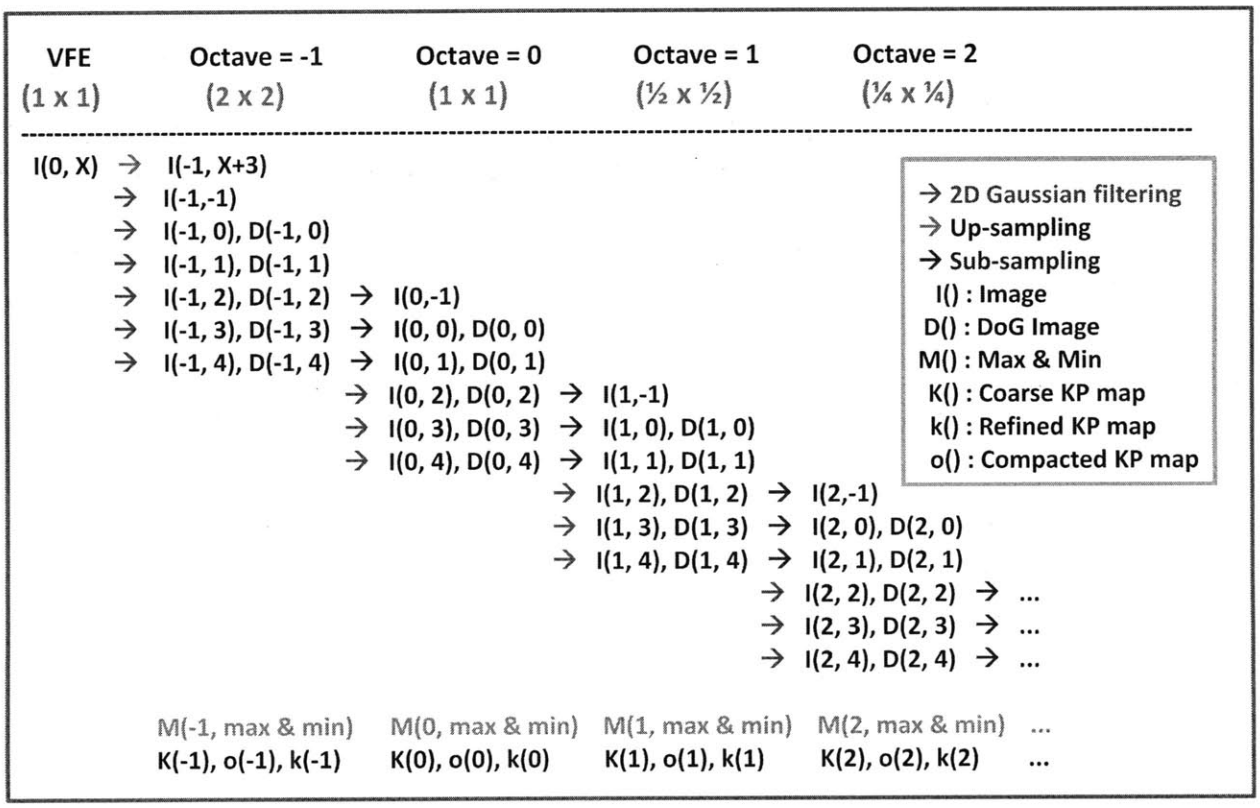


Figure 10:Data flow analysis of GLES-SIFT

5.2 Multi-Channel processing for higher arithmetic intensity

Textures in OpenGL ES 2.0 can store up to 4 floating point values per pixel. In computer graphics these values usually hold the 4 components that describe the color and transparency of a surface (Red,Green,Blue,Alpha). Additionally the fragment shader can output 4 values per fragment/pixel corresponding to each channel. Considering that SIFT uses grayscale images we can pack four images in one texture and perform "multi-channel processing". For instance we

can perform up to 4 Gaussian filters, or 4 sub sampling or 4 image subtraction operation concurrently. Multi-channel processing allows speedup of the pixel-rate processing by cutting down the number of inefficient memory access and increasing arithmetic intensity. The channel grouping that we chose is illustrated below. The current grouping is to some extent arbitrary as we didn't experiment with any other. It was mainly chosen for the implementation simplicity. The images that are computed concurrently are :

- I(-1,-1), I(-1,-0), I(-1,1) by multi-channel smoothing of up sampled input image
- D(-1,0),D(-1,1) by multi-channel processing of (-1,-1), I(-1,-0), I(-1,1)
- I(o,2), I(o,3), I(o,4) by multi-channel smoothing of I(o,-1),for o in [-1;N] and N is number of octaves
- D(o,2), D(o,3), D(o,4) by multi-channel processing of I(o,1), I(o,2), I(o,3), I(o,4)
- I(o,-1),I(o,0),I(o,1) by multi-channel sub sampling of I(o-1,2), I(o-1,3), I(o-1,4) , for o in [0,N] and N is the number of octaves
- D(o,-1),D(o,0),D(o,1) by multi-channel sub sampling of D(o-1,2), D(o-1,3), D(o-1,4) , for o in [0,N] and N is the number of octaves

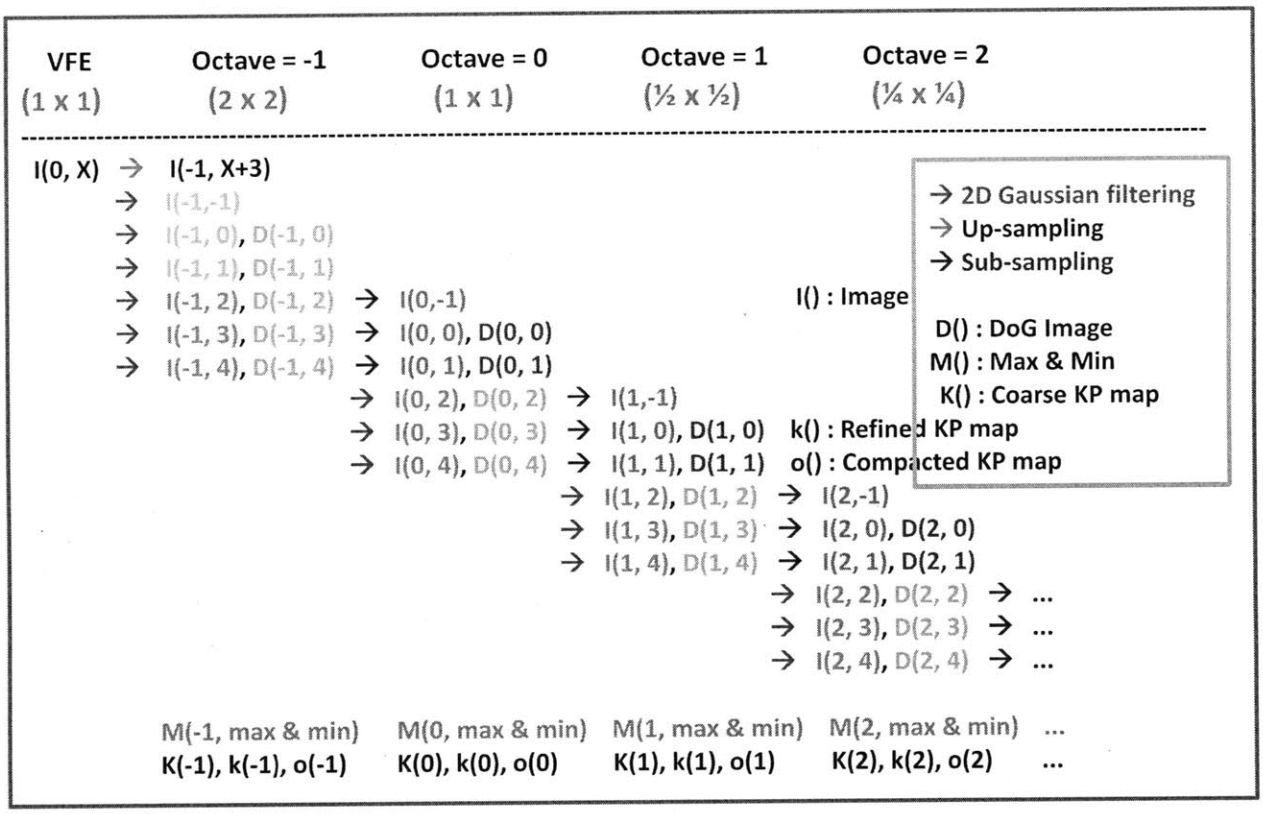


Figure 11: GLES-SIFT data flow with multi-channel processing. For each octave, the images with the same color are computed at the same time

5.3 Data compaction for keypoint-rate efficiency

The number of keypoints detected in an image is considerably smaller than the number of pixels in the framebuffer resulting in a very sparse coarse keypoint map. Such a map could be used as input to the the keypoint refinement stage, however this design would require a check for each pixel to determine if it is an extremum. If-then-else statements severely impact the performance of a shader because they force the hardware threads to execute different code paths by increasing the instruction fetch latency. In order to avoid the if-then-else the coarse sparse keypoint map is uploaded to the CPU where it is compacted. The compaction process loops over the sparse 2D image and logs the x,y position of the pixels marked as extrema and adds them to a compact 1D array. We chose a 1D dimensional array for ease of implementation but the question

still remains whether different dimensions would result in better performance considering that OpenGLES is optimized for handling 2D textures/arrays. During this process the GPU stays idle, however this stage allows for considerable speedups later.

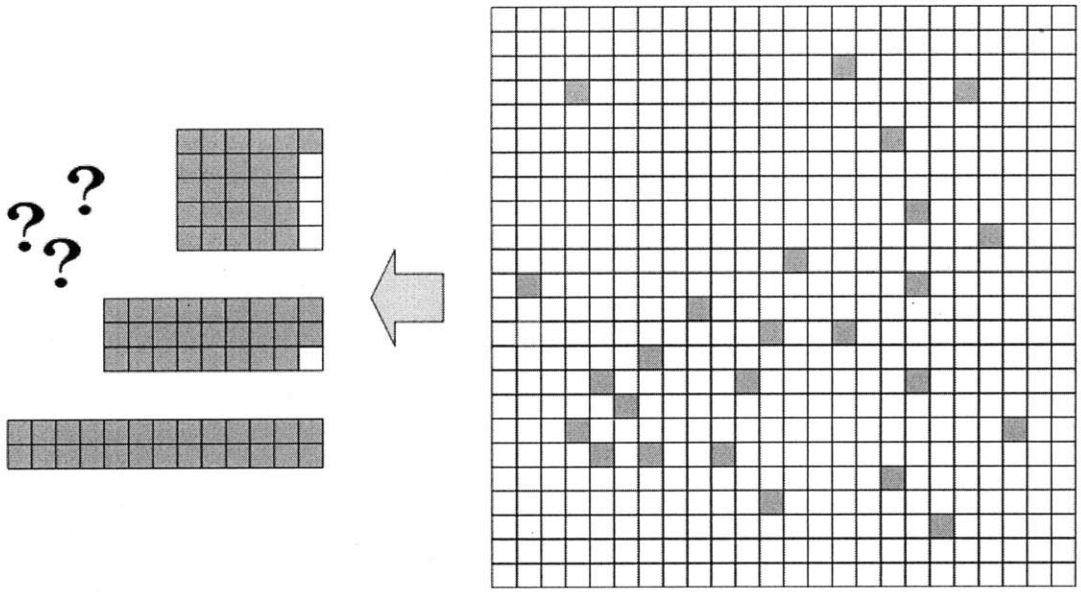


Figure 12: Data compaction consolidates a sparse map into a dense array. The gain in performance justify the cost of the operation .

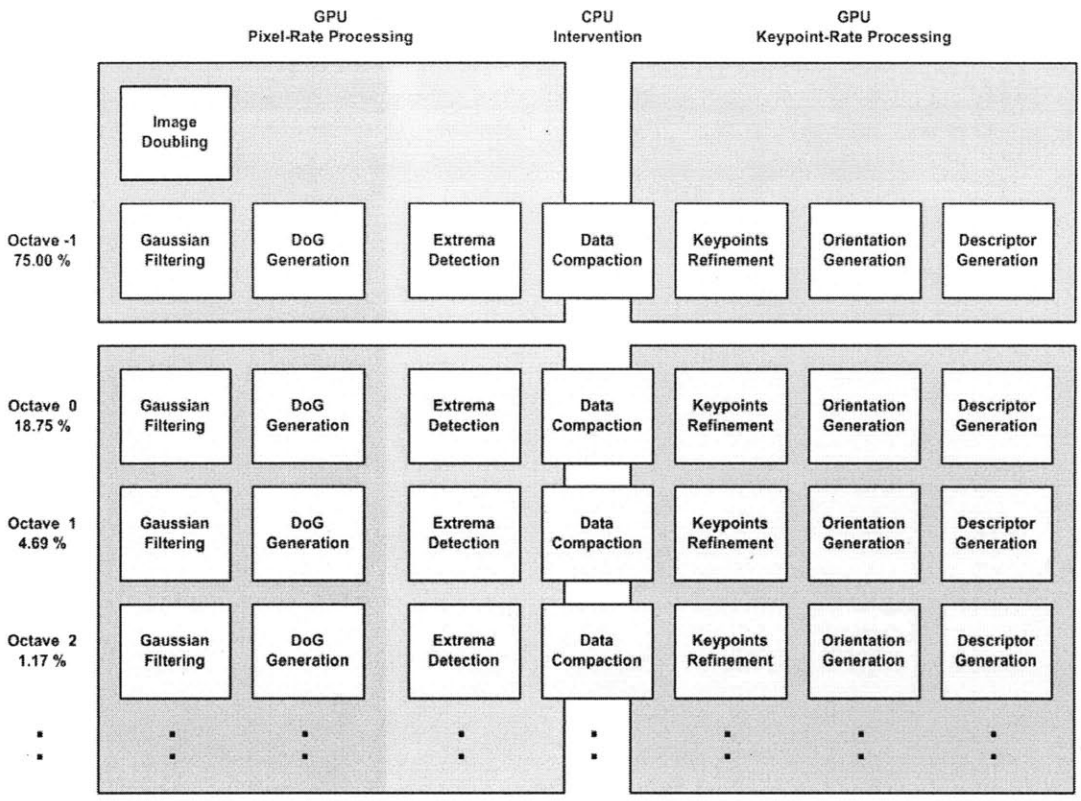


Figure 13:GLES-SIFT data flow with CPU data compaction.

5.4 Descriptor generation prohibitively difficult on current Smartphone GPU's to implement on current GPUs

The SIFT feature is a vector of 128 elements. For each keypoint the descriptor generation process outputs 128 values. However, the fragment shader in OpenGL ES 2.0 can only output a maximum of 4 keypoint. A naïve solution would execute the entire process 32 times and output parts of the vector 4 elements at a time. This is obviously a very suboptimal solution that we can discard without further exploration. At this point in time there is no elegant GPU-only solution. Future work should explore the possibility of using assembly instructions to bypass the output limitation.

6. GLES-SIFT implementation

6.1 GLES-SIFT GPU Driver

This section describes the implementation of the CPU-side application that controls the GPU's operation. The interaction between the CPU application and the GPU is facilitated by the OpenGL ES 2.0 Application Programming Interface. The latter provides numerous functions for initialization of the GPU resources, compilation of vertex and fragment programs, geometry generation, texture management, rendering,etc…

6.1.1 GPU initialization

This step is executed once at startup. It creates a OpenGL ES 2.0 context for the application. The context will store, at the kernel-level in the GPU driver, pointers to all the OpenGL objects such textures, frame-buffer objects, compiled shader programs. OpenGL ES also needs the operating system to allocate a region of the frame-buffer device to the calling application. This operation is not useful to our GPGPU purpose but cannot be avoided because OpenGL was ultimately designed for computer graphics and drawing objects on the display.

6.1.2 Memory/Texture Allocation

As mentioned earlier, textures are the fundamental memory objects in GPGPU. For GLES-SIFT, the memory requirements depend on the size of the input image. At startup the memory is allocated for a default image size of 200x200. However if the size of the input image is different, memory is reallocated before starting the SIFT computation. For each octave, if the number of scales in the image pyramid is N the following is the breakdown of the texture allocation.

- N textures for Gaussian smoothed images
- N-1 textures for DoG images

- 2 temporary textures to hold intermediate data for operations that require multiple rendering passes
- 1 texture to store nature of a pixel location.
- 1 texture to store the refined position of keypoints.

The height and width of an octave are obtained by scaling the height and width of the input image by a factor of 2^{-k} where k is the number of the octave.

All the textures in one octave will have the same height and width except for the keypoint texture whose size is fixed to 1x2048. We assume that the number of keypoints per octave will rarely exceed that limit and are aware that this is not an optimal allocation and some memory space is wasted. In addition, all the textures store 4 floating point values, one for each color component, per pixel. Therefore the total amount of memory allocated for an image of size WxH can be calculated as follows:

$$\sum_{k=-1}^{M} (2N+2)\ images * (2^{-k}W)(2^{-k}H)\frac{pixels}{image} * (4)\frac{componets}{pixels} * (4)\frac{bytes}{component} + 2\,048pixels * (4)\frac{componets}{pixel} * (4)\frac{bytes}{component}$$

For an image of 200x200, M = 7, the total allocated size is approximately 41 MB

The Z400 is allocated 32MB of contiguous memory (PMEM) by the operating system by default. In order to accommodate that amount of memory requirement a change in the kernel code was required.

6.1.3 Vertex/Fragment program management

The following are the computation kernels that are implemented for execution on the GPU. Each kernel corresponds to a pair of vertex program and fragment program.

- Vertical Gaussian filtering

- Horizontal Gaussian filtering
- Subsampling
- Substraction
- Extrema detection
- Keypoint refinement
- Orientation generation

The compilation and execution of a shader program involves a tedious amount of repetitive code. Moreover OpenGL ES creates different pointer for each uniform variable used by a shader program. Considering that there are seven different types of shaders and each of them uses several uniform variables, a `GlesSiftProgram` class was created to help manage the complexity. Each instance of `GlesSiftProgram` is initialized with the source code of the vertex and fragment programs. It also keeps tracks in its state variables of the OpenGL ES pointers to the respective program objects and uniform value objects. `GlesSiftProgram` class also provides a method to execute the shader which is described in the next section.

6.1.4 Executing a rendering pass

The execution of a rendering pass is the GPGPU equivalent to a function call or computation invocation on the CPU. In the OpenGL ES 2.0 framework the process follows the following steps:

- Create 3D geometry
- Activate the textures that should be made available to the shader processors by attaching them to a texture unit
- Select the output texture to render
- Select which combination of fragment/vertex program to use

- Set the values of the uniform variables
- Clear the current framebuffer
- Draw

For our GPGPU purposes, we create a screen aligned quad that covers the entire area of render target. In OpenGL ES 2.0 the quad needs lie on the Z=0 plane and extend from -1 to 1 in the vertical and horizontal dimensions of the render target. GLES-SIFT uses the top left corner of a texture as the origin. In order to match this referencing style in the fragment programs the top left vertex is initialized with texture coordinate attribute (0,0) while the bottom right ones is initialized with texture coordinate (1,1).

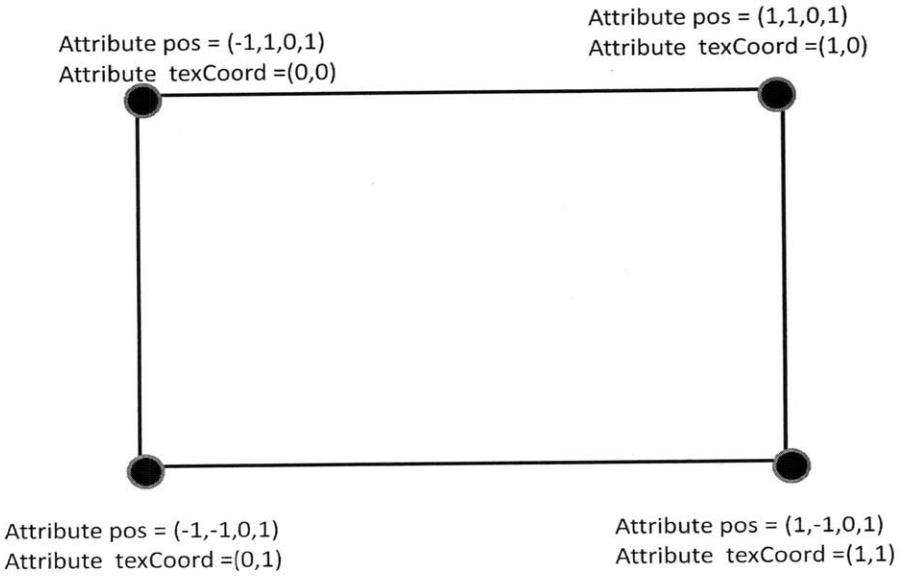
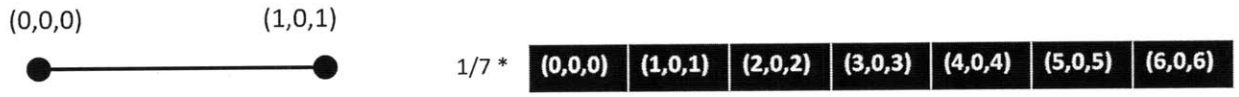


Figure 14: Position and Texture coordinates for the 4 points of the rendered quad

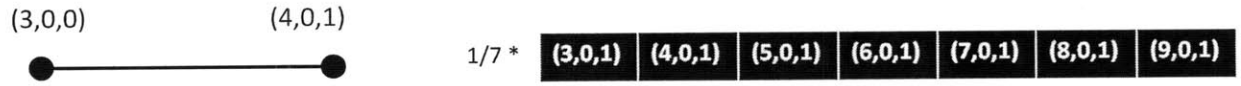
6.2 Gaussian Filtering

The Gaussian kernel is separable, meaning that the 2D convolution can be separated in two successive 1D convolution. The number of operation is reduced from MNK^2 to $MN(2K)$ where the (M,N) is the size of the image and K is the size of the filter kernel. Therefore GLES-

SIFT uses two shader programs; one for horizontal filtering and one for vertical filtering. The filtering operation requires each fragment to sample the pixels values of its neighbors. Alas, calculating the location of the neighbors introduces overhead computation that wastes precious clock cycles. However, we can use the varying variables to move the computation of the coordinates to the rasterizer. This does not reduce the number of total instructions but the workload on the different stages of the pipeline is more evenly shared, the fragment shader becomes less of a bottleneck and we achieve speedup through better parallelism. To understand the process, let's first consider the rasterization of a 7 pixel long line and the interpolation a single varying variable as shown in figure below .



If we shift the first component of the vertex varying variables by 3, the varying variables for each fragment will be also shifted by 3 after interpolation.



Now let's consider the case of a horizontal kernel **k** of length 5. We need to encode the location of 5 taps, so the vertex program for this filter generates two varying vectors `v_vTexCoord1` and `v_texCoord2` for a total of eight interpolation values. **k** being a horizontal filter means that the vertical component of the each tap is invariant. This value is stored in `v_vTexCoord1.y` and `v_vTexCoord2.y`. The rest of the 6 available values are used to store the horizontal components of the 5 taps.

```
//In the vertex program
attribute texCoord; //Texture coordinates
varying v_vTexCoord1;
varying v_vTexCoord2;

v_vTexCoord1.xy = texCoord.xy;
v_vTexCoord1.zw = texCoord.xx + vec2(-1,1);
v_vTexCoord2.xy = texCoord.xy + vec2(-2,0);
v_vTexCoord2.z =  texCoord.x + 2;
```

In the fragment shader the sampling coordinates are retrieved using the sample sizzling operator. For instance the right most tap would be `v_vTexCoord2.zy`. The sizzling operator is compiled into a single register read instruction which executes in one cycle in . Unfortunately this trick only allows the definition of up to 24 sets coordinates while the largest kernel is need to accommodate 31. Therefore the horizontal and vertical pass of the Gaussian filter are each divided into two intermediate pass. Taking advantage of the symmetry of the kernels about the central tap we group together the taps that have the same coefficient.

6.3 Extrema Detection

The algorithm proposed by Lowe for extrema detection is not suitable for the GPU. Lowe[1] proposes to compare the value at every pixel location, in each of the three inner scales of an octave of the DoG pyramid, to the values of its 26 neighbors as illustrated in the figure below. On the GPU fragments cannot share any state so the result of previous pairwise comparison cannot be used to discard irrelevant pixel locations in advance. On the GPU this method would require (3x3x3-1)x2x3xWxH = 156xWxH pair-wise comparison, where W and H are the width and height of the image.

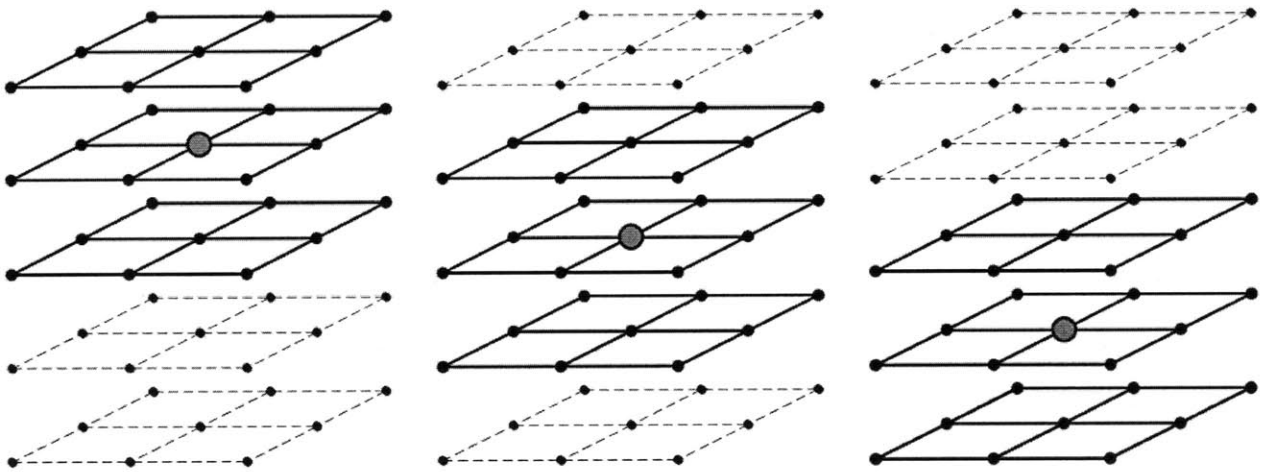


Figure 15:Extrema detection method initially proposed by Lowe is not suited for GPU computation because results from previous pair-wise comparisons are not reusable

GLES-SIFT implements an alternative method to reduce computational complexity . The method is illustrated in the figure below .. For each pixel location, we first calculate the maximum and minimum across all scales and obtain two images of size MxN one storing maxima and other storing minima. Then for each block of dimension 3x3 we check if the value of the center pixel is a minimum in the minima image or a maximum in the maxima image. If so, we retrieve the scale of the extremum by searching for its value across the 3 inner scales of DoG function at the same pixel location. The method requires (4+8+3)x2xWxH = 30xWxH operations, 20% the complexity of the previous method. However it does not produce the exact same keypoints. Simulation data shows that approximately 90-95 % keypoints are identical.

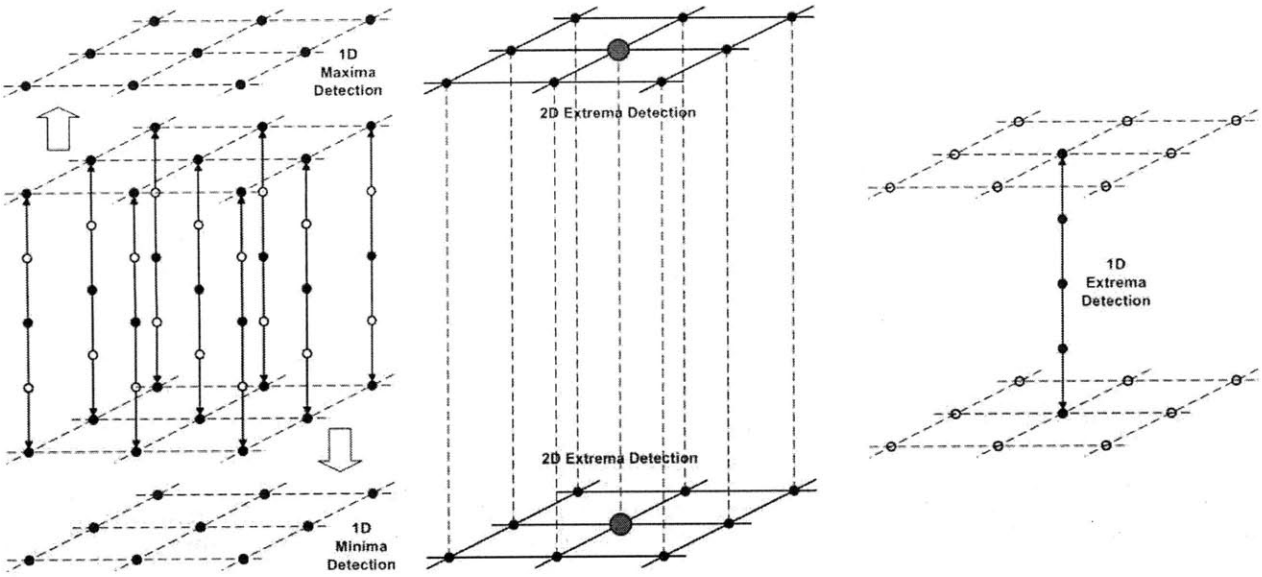


Figure 16: Alternative method for extrema detection that is better suited for GPU computation but produces slightly different keypoints.

Extrema Detection is implemented with two rendering pass. The first one builds the two images of minima and maxima.

```
//assume that dog[1..5] contain the pixel value for scales 1..5 of
//the DoG at the current fragment position
Return variables are in capital letters
//load the pixel values of the first 4 scales into one vec4
vec4 dogvec4 = vec4(dog1,dog2,dog3,dog4);
vec4 pivotvec4 = vec4(dog5,dog5,dog5,dog5);

//figuring out maxima among all
//perform a component-wise max operation. For each pair of component
//returns the max value
resulvec4  = max(pivotvec4, dogvec4);
resulvec2  = max( resulvec4.xy, resulvec4.zw);
MAXIMUM = max( resulvec2.x, resulvec2.y );

// figuring out minima among all
resulvec4  = min(pivotvec4, dogvec4);
resulvec2  = min( resulvec4.xy , resulvec4.zw);
MINIMUM = min( resulvec2.x, resulvec2.y );
```

The second pass determines if each pixel location in the extrema images corresponds to local maximum or local minimum.

```
//assume that maxi/mini[dx,dy] stores the pixel value of the neighbor
pixel at dx,dy in the maxmima and minima images respectively . Return
variables are in capital letters

//3x3 maxima detection
//sample 8-neighbors and store them in 2 vec4
vec4 neighbors1 = vec4(maxi[0,1],maxi[0,-1],
                       maxi[1,0],maxi[-1,0]);

vec4 neighbors2 = vec4(maxi[-1,-1],maxi[-1,1],
                       maxi[1,-1], maxi[1,1]);

//sample the values of the center pixel and copy it 4 times for vector
operation
vec4 centerpix  = vec4(maxi[0,0], maxi[0,0],
                       maxi[0,0], maxi[0,0]);

//compute the max value among all the neighbors
tempvec4  = max(max(centerpix, neighbors1), neighbors2));
tempvec2  = max(tempvec4.xy,tempvec4.zw);

if maxi[0,0] == max(tempvec2.x,tempvec2.y) then
   IS_MAX = true; // the center pixel is a maximum
   SCALE  = dot(vec3(2,3,4),equal(vec3(dog2,dog3,dog4),vec3(max[0,0]))

//repeat the same for minimum
```

6.4 *Keypoint Refinement*

To refine the position of a keypoint to a sub-pixel resolution, we interpolate the location of the extremum using a quadratic Taylor expansion of the discrete DoG function. The new location is obtained by taking the derivative of the Taylor expansion function and solving for the roots. The position of the interpolated extremum is given by

$$\boldsymbol{z} = -\left(\frac{\partial^2 \boldsymbol{D}}{\partial \boldsymbol{x}^2}\right)^{-1} \frac{\partial \boldsymbol{D}}{\partial \boldsymbol{x}}$$

where the derivatives are evaluated at the current keypoint. **z** is the solution to the linear system

$$Hessian * x = -Jacobian$$

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

Let f(x,y,s) be the value of the pixel located at the x,y position and scale s the pyramid of DoG images. The derivatives are calculated from image gradients using the following formulas

- Dx = (df/dx) = (f(x+1, y,s) - f(x-1, y, s)) / 2
- Dxx = ($d^2f/(dx)^2$) = ((f(x+1, y, s) - f(x, y, s)) - (f(x, y,s) - f(x-1, y, s)))
- Dxy = ($d^2f/(dx)(dy)$) = ((f(x+1, y+1, s) - f(x-1, y+1, s)) - (f(x+1, y-1, s) - f(x-1, y-1, s))) / 4
- Similarly for Dy, Ds, Dyy, Dss, Dys, Dxs

We use Gaussian elimination to solve the linear system above. The pseudo code is fairly straight forward.

```
for( col = 0 ; col < 3 ; col++ ){
      pivot.z = -1.0;
      tmpv3   = vec3( h[0][col], h[1][col], h[2][col] );
      for( row = col ; row < 3 ; row++ ){
            tmpf = abs( tmpv3[row] );
            if( tmpf > pivot.z ){
                  pivot.y   = tmpv3[row];
                  pivot.z   = tmpf;
                  pivot_row = row;
            }
      }
      singular          = ( pivot.z < 1e-10 );
      tmpv4             = h[pivot_row];
      h[pivot_row] = h[col];
      h[col]              = tmpv4 / pivot.y;
      tmpv3             = vec3( h[0][col], h[1][col], h[2][col] );
      for( row = col + 1 ; row < 3 ; row++ ) h[row] = h[row] - h[col] *
tmpv3[row];
}
```

However GLSL compiler does not support variable looping—highlited in blue in the figure above. Therefore we need to unroll the loops, but the complexity of the code increases exponentially as shown below.

```
// col = 0;
tmpf = abs( h[0][0] ); { pivot.y = h[0][0]; pivot.z = tmpf; pivot_row = 0;}
tmpf = abs( h[1][0] ); if( tmpf > pivot.z ) { pivot.y = h[1][0]; pivot.z =
tmpf; pivot_row = 1; }
tmpf = abs( h[2][0] ); if( tmpf > pivot.z ) { pivot.y = h[2][0]; pivot.z =
tmpf; pivot_row = 2; }
singular = ( pivot.z < 1e-10 );
if( pivot_row == 0 ) { tmpv4 = h[0];}
if( pivot_row == 1 ) { tmpv4 = h[1]; h[1] = h[0]; }
if( pivot_row == 2 ) { tmpv4 = h[2]; h[2] = h[0]; }
h[0] = tmpv4 / pivot.y;
tmpf = h[1][0]; h[1] = h[1] - h[0] * tmpf;
tmpf = h[2][0]; h[2] = h[2] - h[0] * tmpf;
// col = 1;
tmpf = abs( h[1][1] );{ pivot.y = h[1][1]; pivot.z = tmpf; pivot_row = 1; }
tmpf = abs( h[2][1] ); if( tmpf > pivot.z ) { pivot.y = h[2][1]; pivot.z =
tmpf; pivot_row = 2; }
singular = any( bvec2( singular, ( pivot.z < 1e-10 ) ) );
if( pivot_row == 1 ) { tmpv4 = h[1];                    }
if( pivot_row == 2 ) { tmpv4 = h[2]; h[2] = h[1]; }
h[1] = tmpv4 / pivot.y;
tmpf = h[2][1]; h[2] = h[2] - h[1] * tmpf;
// col = 2;
tmpf     = abs( h[2][2] ); { pivot.y = h[2][2]; pivot.z = tmpf; }
singular = any( bvec2( singular, ( pivot.z < 1e-10 ) ) );
h[2]     = h[2] / pivot.y;
```

6.5 Orientation Assignment

The main implementation challenge for orientation assignment is the representation of the orientation histogram. A 1D array of 36 elements would be conceptually convenient for repetitive filtering and peak detection loops. However dynamic array indexing is not supported by the compiler and all the loops have to be unrolled. We opted to represent the histogram as an array of nine 4-vectors. This representation saves on the number of registers used and allows for parallel vector operations. Another limitation of the GLSL compiler is the lack of variable array

indexing. Operation such as "for(i=0;i<36;i++) **hist[i]=vec4(0.0)**" are not possible. The only alternative is to unroll the loop.

```
hist[0] = vec4( 0.0 );
hist[1] = hist[0];
hist[2] = hist[0];
hist[3] = hist[0];
hist[4] = hist[0];
hist[5] = hist[0];
```

The lack variable indexing becomes more obstructive for more complex operation like peak detections. A simple expression like "peak = hist[0]; index = 0; for(i = 1 ; i < 24; i++) if(hist[i] > peak){ peak = hist[i]; index = I; }" becomes very complex

```
tmpv3.x = hist[0].x; tmpv3.y = hist[0].y; tmpv3.z = hist[0].z; pik =  0; tmpf = tmpv3.y;
if( hist[0].z > tmpf ) { tmpv3.x = hist[0].y; tmpv3.y = hist[0].z; tmpv3.z = hist[0].w; pik =
1; tmpf = tmpv3.y; }
if( hist[0].w > tmpf ) { tmpv3.x = hist[0].z; tmpv3.y = hist[0].w; tmpv3.z = hist[1].x; pik =
2; tmpf = tmpv3.y; }
if( hist[1].x > tmpf ) { tmpv3.x = hist[0].w; tmpv3.y = hist[1].x; tmpv3.z = hist[1].y; pik =
3; tmpf = tmpv3.y; }
if( hist[1].y > tmpf ) { tmpv3.x = hist[1].x; tmpv3.y = hist[1].y; tmpv3.z = hist[1].z; pik =
4; tmpf = tmpv3.y; }
if( hist[1].z > tmpf ) { tmpv3.x = hist[1].y; tmpv3.y = hist[1].z; tmpv3.z = hist[1].w; pik =
5; tmpf = tmpv3.y; }
if( hist[1].w > tmpf ) { tmpv3.x = hist[1].z; tmpv3.y = hist[1].w; tmpv3.z = hist[2].x; pik =
6; tmpf = tmpv3.y; }
if( hist[2].x > tmpf ) { tmpv3.x = hist[1].w; tmpv3.y = hist[2].x; tmpv3.z = hist[2].y; pik =
7; tmpf = tmpv3.y; }
if( hist[2].y > tmpf ) { tmpv3.x = hist[2].x; tmpv3.y = hist[2].y; tmpv3.z = hist[2].z; pik =
8; tmpf = tmpv3.y; }
if( hist[2].z > tmpf ) { tmpv3.x = hist[2].y; tmpv3.y = hist[2].z; tmpv3.z = hist[2].w; pik =
9; tmpf = tmpv3.y; }
if( hist[2].w > tmpf ) { tmpv3.x = hist[2].z; tmpv3.y = hist[2].w; tmpv3.z = hist[3].x; pik =
10; tmpf = tmpv3.y; }
if( hist[3].x > tmpf ) { tmpv3.x = hist[2].w; tmpv3.y = hist[3].x; tmpv3.z = hist[3].y; pik =
11; tmpf = tmpv3.y; }
if( hist[3].y > tmpf ) { tmpv3.x = hist[3].x; tmpv3.y = hist[3].y; tmpv3.z = hist[3].z; pik =
12; tmpf = tmpv3.y; }
if( hist[3].z > tmpf ) { tmpv3.x = hist[3].y; tmpv3.y = hist[3].z; tmpv3.z = hist[3].w; pik =
13; tmpf = tmpv3.y; }
if( hist[3].w > tmpf ) { tmpv3.x = hist[3].z; tmpv3.y = hist[3].w; tmpv3.z = hist[4].x; pik =
14; tmpf = tmpv3.y; }
if( hist[4].x > tmpf ) { tmpv3.x = hist[3].w; tmpv3.y = hist[4].x; tmpv3.z = hist[4].y; pik =
15; tmpf = tmpv3.y; }
if( hist[4].y > tmpf ) { tmpv3.x = hist[4].x; tmpv3.y = hist[4].y; tmpv3.z = hist[4].z; pik =
16; tmpf = tmpv3.y; }
if( hist[4].z > tmpf ) { tmpv3.x = hist[4].y; tmpv3.y = hist[4].z; tmpv3.z = hist[4].w; pik =
17; tmpf = tmpv3.y; }
if( hist[4].w > tmpf ) { tmpv3.x = hist[4].z; tmpv3.y = hist[4].w; tmpv3.z = hist[5].x; pik =
18; tmpf = tmpv3.y; }
if( hist[5].x > tmpf ) { tmpv3.x = hist[4].w; tmpv3.y = hist[5].x; tmpv3.z = hist[5].y; pik =
19; tmpf = tmpv3.y; }
if( hist[5].y > tmpf ) { tmpv3.x = hist[5].x; tmpv3.y = hist[5].y; tmpv3.z = hist[5].z; pik =
20; tmpf = tmpv3.y; }
if( hist[5].z > tmpf ) { tmpv3.x = hist[5].y; tmpv3.y = hist[5].z; tmpv3.z = hist[5].w; pik =
21; tmpf = tmpv3.y; }
if( hist[5].w > tmpf ) { tmpv3.x = hist[5].z; tmpv3.y = hist[5].w; tmpv3.z = hist[6].x; pik =
22; tmpf = tmpv3.y; }
```

7. Performance Analysis

GLES-SIFT was tested on Qualcomm Form Factor Accurate (FFA) with QSD8250 and MSM7x30 platforms running an Eclair build of Android (version 2.0). The test image is an image of size 200x200. The correctness of the algorithm was tested against the Matlab

implementation of SIFT by Andrea Vedaldi[6]. Due to time limitation we were not able finish testing of the orientation assignment. However, all the other stages including Gaussian and DoG pyramid building, extrema detection and keypoint refinement were fully tested. The refined keypoint positions generated by GLES-SIFT match 100% the keypoint positions generated by the Matlab implementation.

7.1 Timing

The execution times for each stage are broken down below

Stage	Execution time(ms)
Gaussian Pyramid	530
• Horizontal Filtering	185
• Vertical Filtering	345
DoG Pyramid	53
Extrema Detection	124
Pyramid readback	126
Keypoint refinement (including compacting keypoint list on CPU)	111
Keypoint list readback	8
Total	952

The difference observed between the execution time of the horizontal and vertical filter fragment programs is due to the type of input image. In the case of the horizontal filtering the input is a grayscale image with 8 bits per pixel. For the vertical filtering the input is multiplexing

of three grayscale images each with 32 bits per pixel. The amount of data accessed by the vertical filter is significantly higher, resulting in lower arithmetic intensity.

The pyramid read-back is a necessary evil if we want the CPU and GPU to collaborate. Both processors are assigned independent region of the system memory and sharing data has to involve a data copy. Unfortunately the memory bus clock speeds is relatively low and the amount of data to transfer is high. Also the OpenGL E2.0 call for read-back, glReadPixels(), is a blocking operation so the transfer time cannot be hidden by concurrent computation.

8. Conclusion and Future Work

This thesis allows us to confirm that the current generation of GPUs in the Z400 family is capable of GPGPU. However these platforms are very restrictive and are not yet mature for optimizing complex algorithm such as SIFT through parallel heterogeneous computation. We believe that simpler algorithms that have the following characteristics are more suitable candidates

- high arithmetic intensity
- infrequent data readback from GPU memory
- not using floating point data
- using simple data structures

While we wait for the next generation of GPUs that support OpenCL which will deliver real GPGPU capabilities, the current OpenGL ES 2.0 standard and Z400 software should be amended to allow drawing to floating point frame-buffer memory and read-back without copying data. Also at the same time future work should be invested in creating a GPGPU framework for OpenGL E.S 2.0 to abstract out all the graphics terminology.

9. References

[1]Lowe, D.G. "Distinctive Image Features from Scale-Invariant Keypoints." IJCV. 2004. Vol(60)2,pp.91-110.

[2]Pharr, Matt and Randima Fernando. Gpu Gems 2. Boston: Addison-Wesley, 2005.

[3]Sudipta, N.Sinha, Frahm Jan-Michael and Pollefeys Marc. GPU-based Video Feature Tracking and Matching. Technical Report. Chapel Hill, NC: Department of Computer Science, UNC Chapel Hill, 2006.

[4] Xiaohua, Wang and Fu Weiping. "Optimized SIFT Image Matching Algorithm." International Conference on Automation and Logistics. Qingdao, China, 2008. 843-847.

[5] Munshi, Aaftab. «OpenGL ES 1.1+ and ES 2.0.» 24 March 2006 r. AMD Developer Central. 17 May 2010 r. <http://developer.amd.com/media/gpu_assets/GDC06-GLES_Tutorial_Day-Munshi-OpenGLES_Overview.pdf>.

[6] Vedaldi, Andrea. SIFT for Matlab. 01 April 2010 r. 17 May 2010 r. <http://www.vlfeat.org/~vedaldi/code/sift.html>.