TAMPERE UNIVERSITY OF TECHNOLOGY

Sumeet Sen

Implementation of Depth Map Filtering on GPU

Master of Science Thesis

Examiner: Prof. Atanas Gotchev

Examiner and topic approved in
The Department of Signal Processing
meeting on 09.04.2014

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
Sumeet Sen : Implementation of Depth Map Filtering on GPU
Master of Science Thesis, 54 pages, 7 Appendix pages
May 2016
Major: Signal processing
Examiner: Prof. Atanas Gotchev
Keywords: Depth map, Bilateral Filter, General Purpose Graphics Processing Unit (GPGPU), OpenCL

The thesis work was part of the Mobile 3DTV project which studied the capture, coding and transmission of 3D video representation formats in mobile delivery scenarios. The main focus of study was to determine if it was practical to transmit and view 3D videos on mobile devices. The chosen approach for virtual view synthesis was Depth Image Based Rendering (DIBR).

The depth computed is often inaccurate, noisy, low in resolution or even inconsistent over a video sequence. Therefore, the sensed depth map has to be post-processed and refined through proper filtering. Bilateral filter was used for the iterative refinement process, using the information from one of the associated high quality texture (color) image (left or right view).

The primary objective of this thesis was to perform the filtering operation in real-time. Therefore, we ported the algorithm to a GPU. As for the programming platform we chose OpenCL from the Khronos Group. The reason was that the platform is capable of programming on heterogeneous parallel computing environments, which means it is platform, vendor, or hardware independent.

It was observed that the filtering algorithm was suitable for GPU implementation. This was because, even though every pixel used the information from its neighborhood window, processing for one pixel has no dependency on the results from its surrounding pixels. Thus, once the data for the neighborhood was loaded into the local memory of the multiprocessor, simultaneous processing for several pixels could be carried out by the device.

The results obtained from our experiments were quite encouraging. We executed the MEX implementation on a Core2Duo CPU with 2 GB of RAM. On the other hand we used NVIDIA GeForce 240 as the GPU device, which comes with 96 cores, graphics clock of 550 MHz, processor clock of 1340 MHz and 512 MB memory.

The processing speed improved significantly and the quality of the depth maps was at par with the same algorithm running on a CPU. In order to test the effect of our filtering algorithm on degraded depth map, we introduced artifacts by compressing it using H.264 encoder. The level of degradation was controlled by varying the quantization parameter. The blocky depth map was filtered separately using our implementation on GPU and then on CPU. The results showed improvement in speed up to 30 times, while obtaining refined depth maps with similar quality measure as the ones processed using the CPU implementation.

PREFACE

I would like to express my sincere thanks to my supervisor Prof. Atanas Gotchev for his support, guidance and encouragement throughout the span of my thesis. His insights on how to analyze and present a research work was of great help. Also his suggestions on technical reading materials during the review meetings was valuable.
My gratitude to the Department of Signal Processing and Tampere University of Technology for giving me the opportunity to work and study at their facility. I would also like to appreciate the help I received from my colleagues at the Department of Signal Processing, especially Smirnov Sergey, during my thesis work.
My final thanks goes to my family and friends, who stands by me in every walk of life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS AND NOTATIONS

GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
CPU	Central Processing Unit
OpenCL	Open Computer Language
API	Application Program Interface
2D	Two Dimension
3D	Three Dimension
DIBR	Depth Image Based Rendering
CV	Computer Vision
HVS	Human Visual System
JIT	Just In Time
QP	Quantization Parameter
SAD	Sum of Absolute Difference

1. INTRODUCTION

The Graphics Processing Units, better known as GPUs came into existence to satisfy the ever increasing demand for computational power. They have proved to be especially effective in the field of image processing, computer vision and computer graphics, where the work load is parallel in nature and same operations are performed on large set of data. One related field of study is stereoscopic 3D, which creates or enhances the perception of 3-dimensional space providing binocular 3D cues.

In humans, the perception of 3-dimensional space is a result of capturing two different views of the same scene by a pair of eyes. The views are then fused in the brain to create simple clear image and to perceive the depth of the scene.

For digital representation, the three most important components for 3D viewing are the left view image, the right view image and the scene depth map. The left and right views are color images that are captured through two different cameras separated by a small distance. Depth map is a gray-scale image which represents the distance of color pixels from the corresponding scene.

The acquisition of depth maps for a scene can be done by active or passive means. The passive approaches are for example depth-from-stereo [32] or depth-from-multi-view [34]; basically the depth map obtained from multiple images of the same scene. On the other hand, active methods rely on depth estimated using range sensors and active illumination source [3]. Even though these technologies are promising, there are common limitations associated with them. The depth computed is often inaccurate, noisy, low in resolution or even inconsistent over a video sequence. Therefore, the sensed depth map has to be post-processed and refined through proper filtering.

In spite of such drawbacks in depth sensing, synthesizing views using the depth image has significant advantages when the transmission of 3D content is taken into account. That is because, since depth map is a gray scale image representing piece-wise smooth scene, it can be compressed more effectively compared to a color image (left or right view) [20].

1.1 Problem Formulation

It is to be noted that sub-sampling and compression adds its own characteristic artifacts (like blockiness) to the depth image. Thus, it can be seen that on the whole, the depth image suffers from inaccuracies, noise and compression artifacts [30]. In case, a view is rendered from such suboptimal depth map, it will be crippled with severe distortions. Hence, the depth image needs to be rectified or filtered before it can be used for further processing.

This brings us to the problem that this thesis will try to address. The pre-processing of depth images has to be performed fast enough such that the 3D viewing can be done in real-time.

1.2 Objective

The purpose of this work was to develop efficient techniques for enhancing the quality of depth maps at real time using GPUs. In order to achieve this, the first step was to identify the algorithm which not only smooths an image but also preserves the edges. Edges in our case represent the depth discontinuities which have great significance for view synthesis.

Secondly, the filtering algorithm should be such that it can be easily parallelized or certain modifications can be made to the algorithm such that it can be implemented on a GPU. In the case of image processing, parallel processing is achieved when the computations done on an individual pixel is independent on the results obtained for it's neighboring pixels.

The bilateral filter was successfully used by Smirnov *et al.* in [14]. The same filter was used as the starting point of this study. The next tasks were to figure out whether

- The algorithm can attain desired speed-ups when implemented on GPU.
- The ease of implementation on GPU platform using open specifications like the OpenCL.
- Finally, the possible boost in computation efficiency that can be attained for the OpenCL implementation over more traditional C code running on CPU.

1.3 Contribution

The contributions of this thesis are as follows:

- Attempts were made to put forward the recent techniques to utilize the computing power for systems which might contain single or multiple CPUs and GPUs. For this purpose, the device agnostic and open source platform called OpenCL was chosen.
- It provides the reader a basic tutorial to start working with OpenCL.
- It also contains some of the tools which one might find useful while using OpenCL

1.4 Structure of this thesis

The thesis is organized as follows: Chapter 2 gives a brief introduction to stereoscopy; it provides an insight on the depth cues that create the impression of the three dimensional space and the techniques for recreating them on digital displays. Chapter 3 describes some of the fundamental elements for GPU programming using OpenCL. Chapter 4 explains the filtering algorithm and how it can be implemented for the GPU platform. Chapter 5 provides the results obtained from the filtering operations and finally, Chapter 6 summarizes the findings in conclusion.

2. A BRIEF INTRODUCTION TO STEREOSCOPY

Humans are gifted with a pair of eyes, which simultaneously captures rays of light in two views and sends the information to the brain. The brain after some complex processing fuses them to create a three dimensional (3D) map of the visible space. The sensation of sight in humans comprises of a single clear color image and a perspective disparity map of the scene.

The photograph captured by a film or digital camera has one fundamental difference with the ones formed in the brain. The former is represented in two dimensions (2D) i.e. height and width. It lacks the depth information, which in the field of computer vision (CV) is regarded as the third dimension.

2.1 Depth cues

Since, 3D viewing requires at least two image sensors (eyes or cameras), it is also regarded as stereoscopy. However, study made by Rolfe *et al.* in [31] suggests, along with stereo vision the brain might use other cues to determine the relative distances and depth in a scene. These cues are gathered by only one eye and hence known as monocular (or extrastereoscopic) cues.

2.1.1 Monocular depth cue

Monocular depth cues play a significant role in perception of depth. To an extent, the significance depends on the distance between the observer and the scene. Some of the important ones are listed below.

- *Accommodation of the eye:* Human eye can change its optical power in order to focus on the object of interest. This is achieved by changing the form of the elastic lens. Even though this is mostly a reflex action but can be controlled to some extent.

- *Relative size:* Objects which creates a bigger image on the retina is perceived to be closer.
- *Relative height:* Objects placed higher in the scene generally tend to be perceived as further away.
- *Overlapping of objects:* If one object is occluded by another, it is obvious that the occluding object is closer. This is also known as interposition.
- *Linear perspective:* The image of two parallel lines or edges, formed on the retina, seem to converge in such a way that they will eventually meet at infinity.
- *Blur and De-saturation:* When the eyes focus on an object it forms a sharp and color rich image on the retina. The part of the scene that lies outside the focus plane appears to be blur. Converting a part of a scene to gray-scale also gives the effect of out-of-focus region. This is known as de-saturation.
- *Haze, De-saturation, and a shift to bluishness:* In the real world, the light traveling from distant objects has to pass through dust or water vapor present in the air giving rise to such effects.
- *Other cues:* Light, cast shadow and textured pattern are some of the cues commonly used to replicate 3D scene.

2.1.2 Binocular depth cue

The most important aspect here is that the eyes are separated from each other by a certain distance (about 64 mm in adults) called the *inter-ocular* distance. This helps to observe two different perspectives of the same scene. From this, the brain derives two important cues called *vergence* and *retinal disparity* shown in *Figure* 2.1.

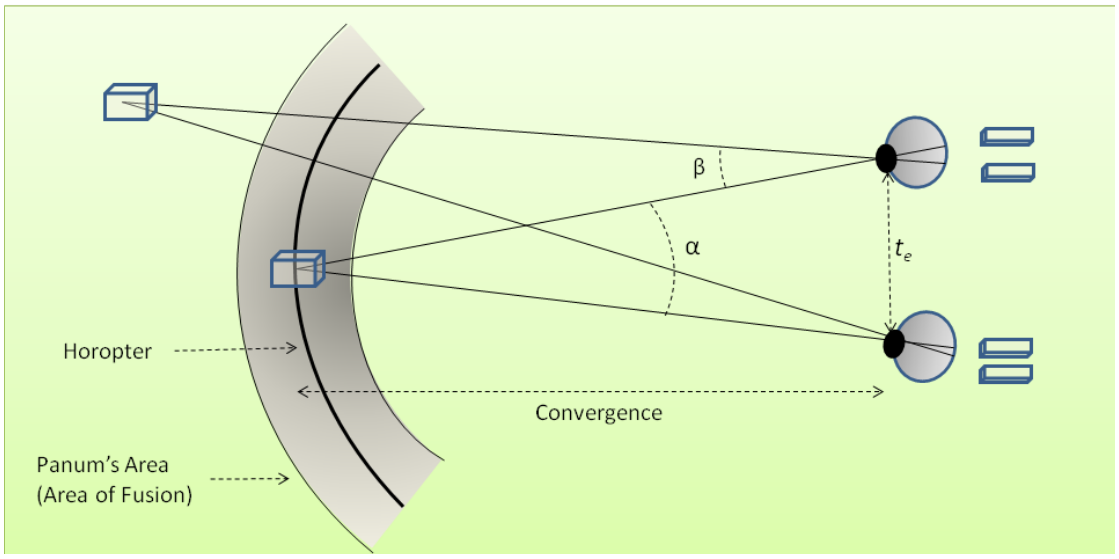


Figure 2.1: *Convergence and retinal disparity. (Adapted from [28])* .

Vergence: When the eyes tries to focus on an object it rotates the eye-ball such that the image is formed at the center of the retina. The angle of rotation is called the vergence angle β, as shown in *Figure* 2.1. For near objects the eyes rotate towards each other and is known as convergence. The angle formed between the two optical axis is called convergence angle α. They are related as $\beta = \alpha/2$. Vergence assists the HVS to estimate the distance of the object to the eye.

Retinal disparity: The object lying on the horopter forms an image on the center of the retina for both eyes. However, the images formed for objects lying outside the horopter has certain amount of disparity compared to each other (as shown in *Figure* 2.1). This is called retinal disparity or binocular parallax. Along with the angle of vergence, this gives an important cue for the brain to construct 3D image of the object and the environment surrounding it.

2.1.3 Depth cues for 3D Displays

Most commercial 3D displays available today are planostereoscopic devices. Such devices provide the viewer with a pair of images; one for each eye, on the same plane of the screen. These two images are exact similar given that each point in one view has a corresponding point in the other view which is slightly displaced. This displacement or disparity provides cue for the brain to create an illusion of 3D. There are in fact three kinds of parallaxes (as shown in *Figure* 2.2).

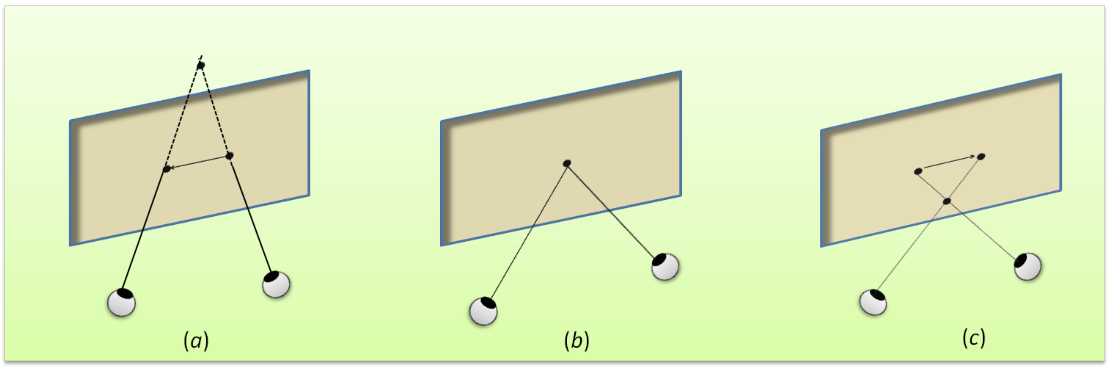


Figure 2.2: *Disparity or Parallax for 3D viewing. (a) Positive disparity (b) Zero disparity (c) Negative disparity. (Adapted from [28])* .

Positive disparity: In this case, the point in the the right view lies on the right of it's corresponding point in the left view (see *Figure* 2.2(a)). Hence, the viewing rays seem to converge on the plane behind the screen. This gives an impression that the object lies in the screen space (for screen space see *Figure* 2.3). To display an object located at infinity the displacement has to be equal to the inter-ocular distance; which means, it is the maximum possible disparity that can be produced on the screen.

Zero disparity: In this case, the left and right view points are located at the same point. Thus the viewing rays seem to converge on the screen (see *Figure* 2.2(b)).

Negative disparity: Finally, for negative disparity, the point in the right view lies on the left of it's corresponding point in the left-view (see *Figure* 2.2(c)). Thus the viewing rays seem to converge at a point in front of the screen giving the impression that the object lies in the theater space (for theater space see *Figure* 2.3).

2.2 Issues related to Binocular Disparity

It is to be noted that, over use of the disparity to create the 3D effect might lead to uncomfortable viewing experience. The comfort zone for 3D viewing lies near the screen (see *Figure* 2.3) i.e. when the disparity is small. Larger disparity places the 3D object further away from the screen, however they are difficult for the brain to simulate. The retinal rivalry areas should be avoided and only visited when absolutely necessary. Otherwise, improper 3D production will stress the viewer and lead to bad viewing experience.

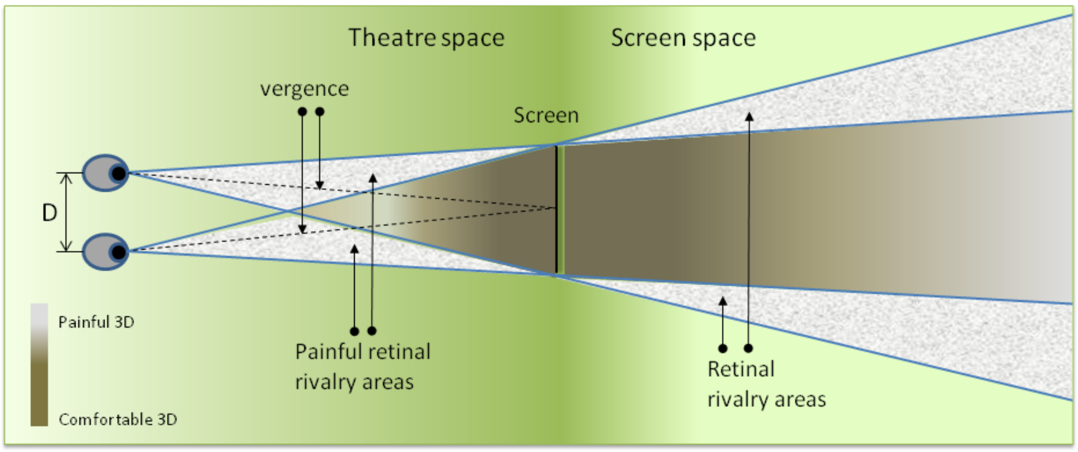


Figure 2.3: *Stereoscopic comfort zone (Adapted from [29])* .

The amount of 3D experience varies also with the distance from the viewing plane, as shown in *Figure* 2.4. Same amount of disparity will simulate different depth depending on the position of the viewer with respect to the screen. It is important this into account, since the 3D content generated are viewed on multiple screen types; like the movie theater, television and even hand-held devices like phones and tablets.

2.3 Stereoscopic 3D

This thesis deals with stereoscopic (left and right view) images which also has the third dimension i.e. depth. Such images are captured with two cameras which

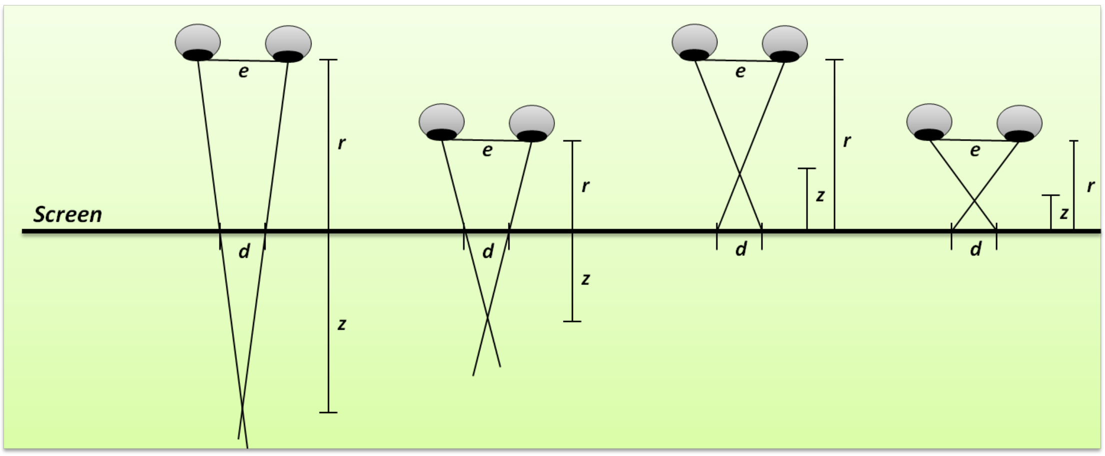


Figure 2.4: *depth Vs viewing distance (Adapted from [29])* .

are calibrated, rectified and separated by a certain distance called base-line. In simple words, the setup tries to mimic the human eye.

Disparity refers to how the coordinates of a particular feature vary from one view to another. Each point on the disparity plane may be considered as a two dimensional vector; which maps the corresponding points in the left view, to the right view of a stereo pair. This is more commonly known as stereo matching or stereo correspondence and is discussed by D. Scharstein and R. Szeliski in [32]. An example of disparity image (from Middlebury dataset) is found in *Figure* 2.5. The image shows left and right views in the top row and its corresponding disparities in the bottom row. The disparity values are encoded in gray scale between 0 to 255. The value of zero represents unknown disparity.

Depth information can be obtained in different ways. High resolution depth information can be computed from a pair of intensity images using stereo correspondence or stereo matching algorithms. This is known as passive methods for generating depth. Another popular setup is by using special illumination and sensors pair, like the time-of-flight cameras. These are called active methods and is explained in details by Lange and Seitz in [3]. However, the sensor based methods needs some post-processing owing to their low resolution, inaccurate estimation for larger distances and also due to the fact that the image sensor and the depth sensor are placed at slightly shifted location [25].

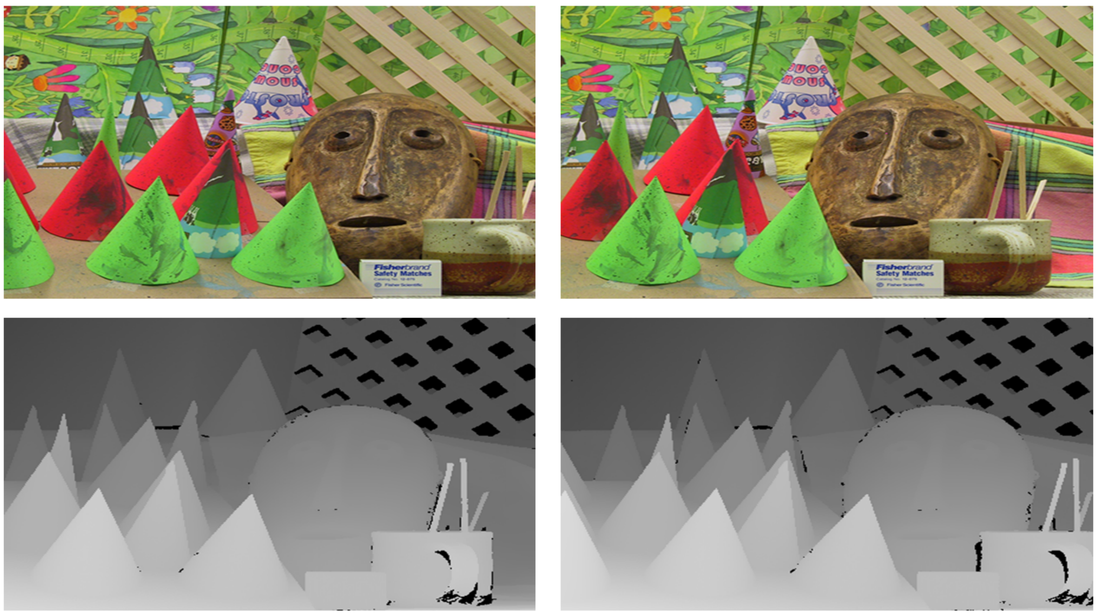


Figure 2.5: *Top left and top right are the two views of stereo pair. Lower left shows the corresponding disparities [38] .*

There are two basic steps involved in stereo matching: first use a matching function like sum of absolute difference or cross-correlation to identify similar patches in the image pair try to estimate the best match using techniques like graph cut [34], plane sweeping [36], belief propagation [35] or a mix of several different approaches.

Using triangulation, a depth map is computed which indicates the relative distance of any point in the scene from the camera plane. Considering the disparity as d, base offset i.e. separation between the two cameras as b and focal length to be f, the depth D can be calculated as

$$D = \frac{b \cdot f}{d} \tag{2.1}$$

The techniques discussed above needs two distinct devices to work as a pair i.e. two cameras or a transmitter and a receiver. This setup is challenging in a way that the devices have to be perfectly synced and calibrated. However, there are some other ways to gather depth information with just one camera. Some of those methods are: depth from defocus explained by Subbarao *et at.* in [4], the concept of coded aperture using a conventional camera [5], and a combination of these techniques [6].

2.4 Depth Image Based Rendering

Depth Image Based Rendering or DIRB is the technique of creating multiple views for 3D rendering from a single 2D image and it's corresponding depth information [8]. There are few advantages of DIBR over two camera setup to capture, transmit and render 3D scenes to viewers. Some of them are mentioned by Solh *et al.* in [7].

- Since both views are generated from the same image, photometric asymmetries between the views becomes irrelevant.
- A narrow transmission bandwidth is needed; considering the 8 bit gray scale depth map contains much less data than a colored image.
- The user has control over the amount of 3D to be rendered depending on views preference.

However, DIBR does not always provide the best and desired 3D rendering outcome. It has its own challenges. Solh *et al.* explains in [7] that final outcome is very sensitive to

- The quality of the depth map
- The compression artifacts
- The warping techniques used
- Data loss and errors that might occur during transmission
- The algorithms used in the 3D displays

The three major steps involved in DIBR are:

- Pre-processing

- Image warping
- Filtering of disoccluded holes

Pre-processing of depth map: Depth map represents the 3D geometry related to the retinal disparities. (see *Figure* 2.2) and therefore it is one of the critical element in 3D warping. The base plane is called the Zero-Parallax Z_c and is calculated as

$$Z_c = \frac{(Z_{far} - Z_{near})}{2} \tag{2.2}$$

where Z_{far} is the farthest clipping plane and Z_{near} is the nearest clipping plane of the depth map. The zero-parallax is chosen somewhere at the midpoint so that similar depth levels can be achieved both in front and back of that plane. It is important that all measured depth values are normalized within the clipping range, otherwise it will generate wrong disparity values during image warping.

Next, the depth map needs to be filtered. This is because any errors during depth estimation or the presence of noise will lead to contrast fluctuations in the reconstructed synthetic views. Some of the simple techniques are the use of Gaussian mask or plain averaging filter. However, it smooths the discontinuities and hence is not suitable for depth images. Some advanced edge preserving techniques are discussed by Yang *et al.* in [16] and Smirnov *et al.* in [14]. They utilize bilateral filtering proposed by Tomasi *et al.* in [1] is used, which preserves the edges and object boundaries. For the case of depth map refinement, the bilateral filter is modified to a cross-modality version [16], [14].

3D Image warping: Using the depth map, each point in the image is re-projected into the 3D space; and then they are projected back to the image sensor plane of the virtual camera [8]. This process is known as 3D image warping in the world of Computer Vision.

Disocclusion and Hole filling: Image warping reveals new regions in the synthesized novel views which were previously invisible in the original image. These newly formed areas are devoid of any texture or depth information. This phenomenon is known as disocclusion and it creates holes or "void" regions in the synthesized images. If these regions are not treated, they will produce annoying visible artifacts for the viewer.

Some advanced hole filling techniques are described bu Solh *et al.* in [11] and by Po *et al.* in [10]. However, a more efficient way might be to treat the depth maps at the pre-processing step so that the effect of disocclusion can be reduced. This is described by Fehn in [8] and Zhang *et al.* in [9].

2.5 Coding of depth map

Depth map has subtle difference to video data which has to be taken into consideration while encoding it. In case of video, there are lots of high frequency components which need to be preserved. Otherwise, loss of high frequencies will lead to blurriness and hence low visual perception.

On the other hand, depth data mostly contains homogeneous regions with only few sharp variations at object boundaries. Thus, it can be encoded very efficiently with high compression ratio. In most cased 8 bits of data is sufficient for coding depth maps, which means there can be only (2^8) 256 depth levels. It is usually an inverted range image; 0 refers to the farthest point whereas 255 refers to the nearest. Since the range map is inverted the nearby objects have higher depth resolution.

However, it is absolutely necessary to preserve those variations which mostly occur at the foreground-background object boundaries. Failure to encode the variations correctly will lead to reconstruction error and thus visual artifacts in the synthesized view. Some examples of coding schemes for DIBR kind of setups are proposed by Maitre *et al.* in [19] using shape adaptive wavelet based codec and Kim *et al.* in [21] using rate distortion optimization for depth encoding.

3. OPENCL: THE OPEN SOURCE SPECIFICATION FOR GPU PROGRAMMING

OpenCL stands for Open Computing Language initially proposed by Apple in 2008 and are currently maintained by Khronos Group. The other big companies that are involved in developing the OpenCL specification include NVIDIA, AMD and Intel among others.

OpenCL is a specification, which means, programmers have to write their own implementations which are compliant to the specification. Thus OpenCL is a programming interface which offers a framework to build applications on top of it. This framework allows the user to take advantage of all the system resources (CPU, GPU, DSP chip).

OpenCL is designed to support general purpose parallel computation. It can be used for variety of tasks which involves heavy computation and calculation like in computer graphics, digital signal processing, scientific calculations, analysis of financial data, and many more.

This chapter focuses on those specific details which are in the scope of this thesis. Hence, there might be several features which are left out or not explained in detail. For detailed information, please refer to OpenCL specifications from the Khronos Group [37].

3.1 The OpenCL Architecture

Most devices available today comprises of a heterogeneous collection of CPUs, GPUs and several other processing elements that work in coordination. OpenCL provides the framework which provides language for parallel programming, APIs, libraries and run-time system needed for software development on such heterogeneous platforms.

This framework can be described using four models which will be explained briefly in the sections below.

- Platform Model
- Memory Model
- Execution Model
- Programming Model

3.1.1 Platform Model

The platform model consists of a **host** connected to one or more OpenCL devices. These OpenCL devices in turn contain **compute units** (CUs), which are a collection of **processing units** (PEs). The PEs are responsible for all the computations happening in the device. The model is shown in *Figure 3.1.* The developer submits the commands to be executed on the device through the host. The host is responsible for setting up the OpenCL devices. The device splits up the instructions and data on its processing units; then they are executed as Single Instruction Multiple Data (SIMD) or Single Program Multiple Data (SPMD).

3.1.2 Execution model

Programming in OpenCL has two basic parts: the **host code** that executes on the CPU (the host), and the **kernel codes** that execute on one or more OpenCL devices. The way in which the kernel executes on the work units (or CUs) of the device defines the execution model in OpenCL.

The basic element of work unit is called a **work item** (or PEs) which are grouped to form **work groups**. All the work groups are put together to form the **NDRange** i.e. n-Dimensional Range. The total number of the executable units on the GPU is called the **global size** which is the size of the NDRange. On the

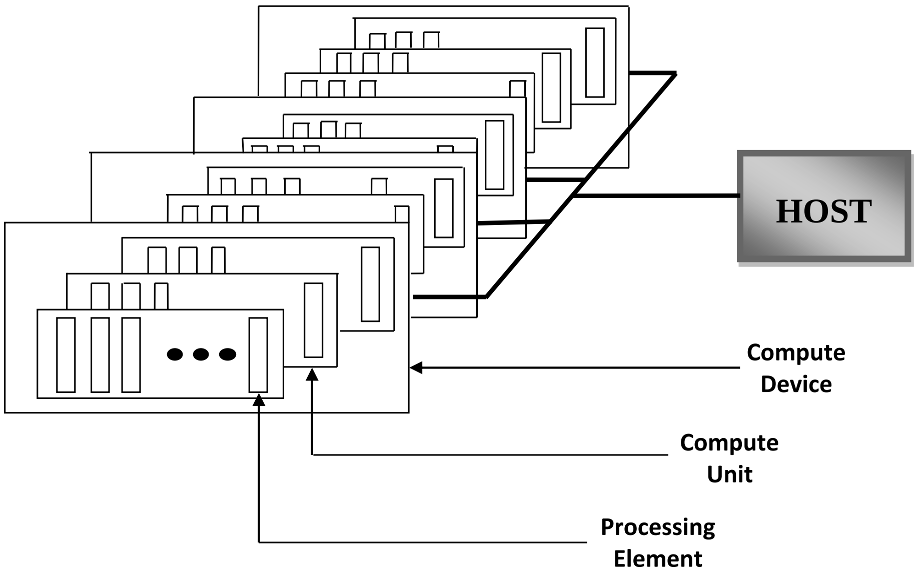


Figure 3.1: *Platform Model: One host, together with one or more OpenCL devices, each with one or more compute units. Each CU has one or more processing elements. (Adapted from [37, p. 23]).*

other hand, the size of a work group is called the **local size**. It is to be noted that on a GPU, the global size needs to divide evenly into local size.

One of the powerful features of OpenCL, is the ability to index the global (NDRange) and local work spaces in one, two and three dimensions, using the inbuilt functionality provided by the specification. This concept is well explained in [37, p. 24-25] and is also mentioned below.

Consider the 2-dimensional index space as shown in *Figure 3.2.* The NDRange is in 2-dimension, which means that the index space for the work items are represented as (G_x, G_y), the size of each work group is (S_x, S_y) and the global ID offset (F_x, F_y). Since the global size G_x and G_y; the total number of work items is $G_x{*}G_y$.

The local indices for the work items are represented in S_x by S_y index space; therefore the number of work items in a single work group is $S_x * S_y$. Given (S_x, S_y) and (G_x, G_y) the number of work groups can be computed.

A 2-dimensional index space is needed to uniquely identify a work group. Either the global ID (g_x, g_y) can be used or a combination of the work group ID (w_x, w_y), work group size (S_x, S_y) and the local ID (s_x, s_y) inside the work group as shown by the equation below.

$$(g_x, g_y) = (w_x \times S_x + s_x + F_x, \quad w_y \times S_y + s_y + F_y) \tag{3.1}$$

The total count for the work groups is given as:

$$(W_x, W_y) = (G_x/S_x, \quad G_y/S_y) \tag{3.2}$$

The work group ID for a work item is derived from global ID and the work group size as:

$$(w_x, w_y) = ((g_x - s_x - F_x)/S_x), \quad (g_y - s_y - F_y)/S_y) \tag{3.3}$$

3.1.3 Memory Model

The memory model can be classified into four address spaces as shown in *Figure 3.3*. More details on the allocation and usage of the memory available in the device can be found from the *Table 3.1*.

- **__global**: It refers to the **global memory** which allows read and write access to all work items in the work group.
- **__constant**: It refers to the **constant memory** which is a part of the Compute Device Memory. However it remains constant during the execution of the kernel. The initialization and the allocation of the memory objects for the constant memory is done by the host.
- **__local**: It refers to the **local memory** of the work group which can be shared by all the work items belonging to that work group. They are much faster compared to the global memory and their availability is quite limited.
- **__private**: It refers to the **private memory** of the work item. Private

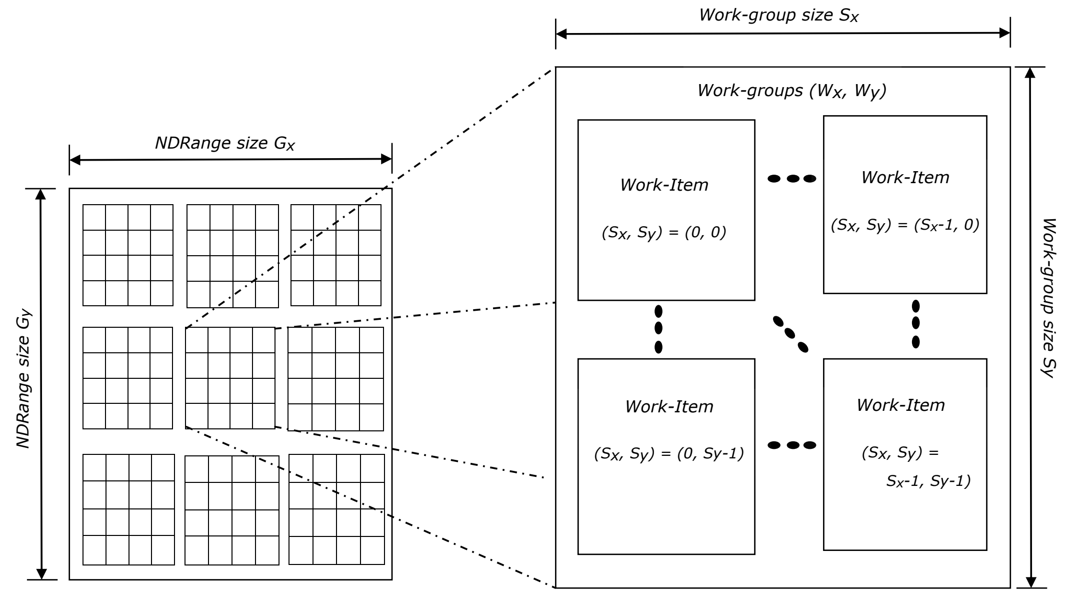


Figure 3.2: *The execution model: Shows work items, work groups, global IDs, local IDs and and how they are indexed over 2-Dimension. (Adapted from [37, p. 25]).*

memory of one work item is unavailable to other work items. They act mostly like the resistors and hence are extremely fast.

Memory Objects

In order to reference the global memory, OpenCL provide handles called Memory Objects which are of type `cl_mem`. They can be categorized as follows:

Buffer Objects: These form one-dimensional contiguous collection of elements which can have scalar (e.g. int, float), vector or even user-defined data types. Importantly, they can be accessed directly by a kernel using pointers.

Image Objects: OpenCL has special buffers to hold texture and images. Each element of this buffer is a 4-component vector of type float or signed/unsigned integer. They cannot be directly accessed using pointers, hence built-in functions have to be used in order to handle them.

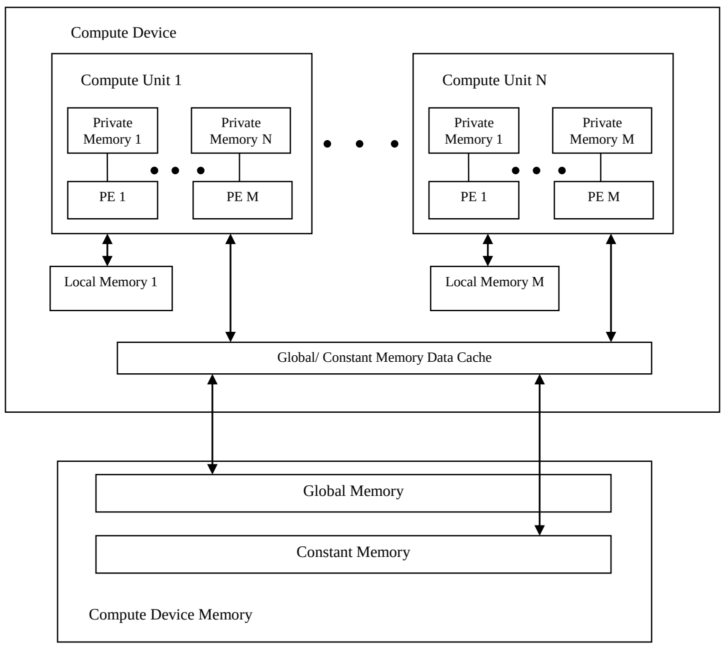


Figure 3.3: *Conceptual OpenCL device architecture with processing elements (PE), compute units and devices. (Adapted from [37, p. 28]).*

3.1.4 Programming Model

The Programming model for OpenCL can be **Data Parallel** or **Task Parallel**. In fact, it can also be a combination of the two. The programmer needs to implement a synchronization method for proper execution.

Data Parallel scheme implies that an individual or a set of instructions are applied to a chunk of data within a set of data structure. In OpenCL terms, the work items are first defined, and then the data is mapped onto those work items. The work items can be programmed with same instructions which are to be performed on data mapped onto it.

Task Parallel scheme implies that the instructions are arranged as multiple concurrent tasks which will then run on a single or multiple command-queues. In OpenCL, this can be obtained as using data types like vectors or enqueue multiple tasks in the form of kernels.

	Global	**Constant**	**Local**	**Private**
Host	Dynamic allocation Read/Write access	Dynamic allocation Read/Write access	Dynamic allocation No access	No allocation No access
Kernel	No allocation Read/Write access	Static allocation Read-only access	Static allocation Read/Write access	Static allocation Read/Write access

Table 3.1: *Memory Region - Allocation and Memory Access Capabilities [37, p. 28]*

Synchronization can be achieved in three ways.

- **Work group barriers** can be used to synchronize between work items in a single work group. The execution can continue only when all the work items of the work group has executed until the barrier.
- **Command-queue barrier** can be used to to synchronize the commands enqueued to the command-queue(s) within a single context. It ensures that all expected operations pertaining to the previously queued tasks are executed before it starts executing the next command.
- **Waiting for an event** from an Application Program Interface (API). The events identifies the task and the memory object that it updates.

3.2 The OpenCL Framework

The most important characteristic of OpenCL is it's capability to combine a host and one or more compatible devices into a single heterogeneous parallel computing system. The OpenCL framework takes care of this aspect with three of its components:

- OpenCL Platform Layer
- OpenCL Runtime
- OpenCL Compiler

All application has to first implement this framework in order to setup the computing device. The subsequent paragraphs will explain the steps needed to setup the OpenCL framework. An overview of the OpenCL APIs are provided below. Chapter 4 explains the OpenCL Specifications in details [37].

The first step is to know the list of available platforms. This is done by using the function **clGetPlatformIDs**. It returns the number of available platforms and a list of them.

```
cl_int clGetPlatformIDs (cl_uint num_entries,
                         cl_platform_id *platforms,
                         cl_uint *num_platforms)
```

To get specific information about the OpenCL platform specified by the `cl_platform_id` obtained from `clGetPlatformIDs`, the function **clGetPlatformInfo** is used.

```
cl_int clGetPlatformInfo (cl_platform_id platform,
                          cl_platform_info param_name,
                          size_t param_value_size,
                          void *param_value,
                          size_t *param_value_size_ret)
```

Similarly, to get the list of devices available for a particular platform OpenCL has the API **clGetDeviceIDs** and for specific information about a device there is the API **clGetDeviceInfo**.

```
cl_int clGetDeviceIDs (cl_platform_id platform,
                       cl_device_type device_type,
                       cl_uint num_entries,
                       cl_device_id *devices,
                       cl_uint *num_devices)
```

```
cl_int clGetDeviceInfo (cl_device_id device,
                        cl_device_info param_name,
                        size_t param_value_size,
                        void *param_value,
                        size_t *param_value_size_ret)
```

The context is one of the critical element in an OpenCL host code, which is later used by the runtime to keep track of command-queues, memory, compiling and executing kernels on specified devices. It can be created for one or more devices. The programmer can specify the `cl_context_properties`, list of devices and also a callback function *pfn_ notify* that can be registered by an application.

```
cl_context clCreateContext (const cl_context_properties *properties,
                            cl_uint num_devices,
                            const cl_device_id *devices,
                            void (CL_CALLBACK *pfn_notify)(const char
                                 *errinfo,const void *private_info,
                                 size_t cb,void *user_data),
                            void *user_data,
                            cl_int *errcode_ret)
```

3.3 The OpenCL Runtime

Runtime refers to the section of a program which creates command-queues, memory objects, program objects and sets up the kernel code for execution.

3.3.1 Command Queues

The operations on objects such as memory, program and kernel objects can be set up in an orderly manner using command queue. Multiple command-queues can be established by the application for different operations as long as they are independent.

```
cl_command_queue clCreateCommandQueue (
                        cl_context context,
                        cl_device_id device,
                        cl_command_queue_properties properties,
                        cl_int *errcode_ret)
```

3.3.2 Buffer Objects

A **buffer** is 1-Dimensional data storage which can be a scalar data type (e.g. int, float), a vector or a user defined structure. Given below are some built-in APIs for creating, reading and writing buffer objects. For an extended list of APIs please refer to Chapter 5 of [37].

API for creating a buffer object: **clCreateBuffer**

```
cl_mem clCreateBuffer (cl_context context,
                       cl_mem_flags flags,
                       size_t size,
                       void *host_ptr,
                       cl_int *err_ret)
```

where,
context: Valid OpenCL context created for the buffer object using the API clCreateContext.
flags: Specifies how and which memory will be used and allocated. The values for the flags might be:

```
CL_MEM_READ_WRITE,
CL_MEM_WRITE_ONLY,
CL_MEM_READ_ONLY,
CL_MEM_USE_HOST_PTR,
CL_MEM_ALLOC_HOST_PTR,
CL_MEM_COPY_HOST_PTR,
CL_MEM_HOST_WRITE_ONLY,
CL_MEM_HOST_READ_ONLY,
CL_MEM_HOST_NO_ACCESS
```

size: Size in bytes for the buffer memory that needs to be allocated.
host_ptr: Pointer to the buffer data that is to be allocated.
err_ret: Error code that will be returned.

API to read from a buffer object to host memory: **clEnqueueReadBuffer**

```
cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
                            cl_mem buffer,
                            cl_bool blocking_read,
                            size_t offset,
                            size_t size,
                            void *ptr,
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event)
```

API to write to a buffer object from host memory: **clEnqueueWriteBuffer**

```
cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_write,
                             size_t offset,
                             size_t size,
                             void *ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)
```

where,
command_queue: The host command queue where the read or write command will be queued.
buffer: Valid buffer object.
blocking_write: Specifies whether the reading or writing operation is blocking or non-blocking.
offset: Offset in bytes in the buffer object while reading or writing.

ptr: Points to the buffer in the host memory from where data will be read or written to.
num_events_in_wait_list and *event_wait_list*: List of events that needs to be completed before the read or write command can be executed.
event: These are return values which identifies a particular read or write operation. Mostly they are used to query if a particular operation is complete.

3.3.3 Image Objects

Similar to the APIs for handling buffer objects, there are also APIs for working with images and textures. The image objects can have one, two or three dimensional frame buffers. Some of the methods for using these objects are listed below. For more details refer to Chapter 5 of [37].

```
cl_mem clCreateImage (cl_context context,
                      cl_mem_flags flags,
                      const cl_image_format *image_format,
                      const cl_image_desc *image_desc,
                      void *host_ptr,
                      cl_int *errcode_ret)
```

Notice that the image channel order and image data type can be specified using the pointer *image_format*. One simple and commonly used format is shown is *Figure 3.4* where the order is `CL_RGBA` and the data type is `CL_UNORM_INT8` and `CL_UNORM_INT16`.

The image descriptor *image_desc* specifies the image type, width, height, depth, number of images in the image array, scan-line(row) pitch, slice pitch and a valid reference to a memory object.

There are also APIs defined for reading and writing images:

```
cl_int clEnqueueReadImage (cl_command_queue command_queue,
                           cl_mem image,
```

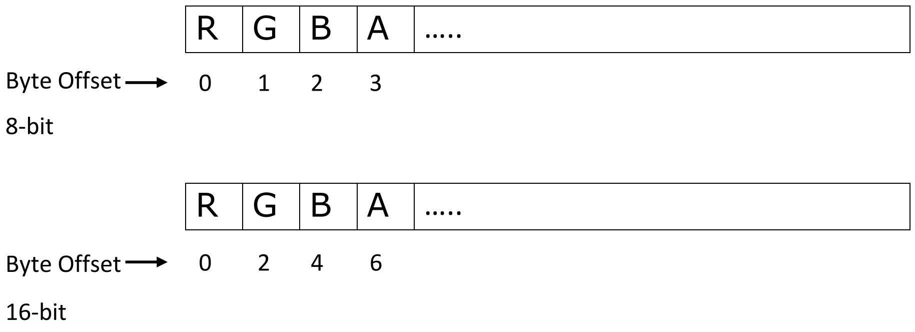


Figure 3.4: *Image buffer channel. (Recreated from [37]).*

```
                              cl_bool blocking_read,
                              const size_t *origin,
                              const size_t *region,
                              size_t row_pitch,
                              size_t slice_pitch,
                              void *ptr,
                              cl_uint num_events_in_wait_list,
                              const cl_event *event_wait_list,
                              cl_event *event)
```

```
cl_int clEnqueueWriteImage (cl_command_queue command_queue,
                             cl_mem image,
                             cl_bool blocking_write,
                             const size_t *origin,
                             const size_t *region,
                             size_t row_pitch,
                             size_t slice_pitch,
                             void *ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)
```

3.3.4 Program Objects

The OpenCL kernel code is in-fact an array of characters or a *string*, identified by the qualifier *__kernel*. The program can consist of a set of kernels, other auxiliary functions and constant data which the kernel functions can use. The code is compiled with a Just-in-Time (JIT) compiler of OpenCL runtime to generate a program object. The program object encapsulates the context, binary program source, the latest built executable and the number of kernel objects.

The following API is used to create a program object from the source code specified in the *string* and then associate it with a context.

```
cl_program clCreateProgramWithSource (cl_context context,
                                      cl_uint count,
                                      const char **strings,
                                      const size_t *lengths,
                                      cl_int *errcode_ret)
```

To create a program object from the binary code specified by the *binaries* array and then associate it with a context, the following API used.

```
cl_program clCreateProgramWithBinary (cl_context context,
                                      cl_uint num_devices,
                                      const cl_device_id *device_list,
                                      const size_t *lengths,
                                      const unsigned char **binaries,
                                      cl_int *binary_status,
                                      cl_int *errcode_ret)
```

3.3.5 Kernel Objects

Creating Kernels The OpenCL gives an option to create kernel objects individually for one kernel at a time using the API **clCreateKernel**, or for all the kernels for the program object at the same time using **clCreateKernelsInProgram**.

```
cl_kernel clCreateKernel (cl_program program,
                          const char *kernel_name,
                          cl_int *errcode_ret)
```

Setting Kernel Arguments In order to execute a kernel code, the arguments need to set first. This can be done using the API **clSetKernelArg**

```
cl_int clSetKernelArg (cl_kernel kernel,
                       cl_uint arg_index,
                       size_t arg_size,
                       const void *arg_value)
```

Executing Kernels Before the kernels are built it has to be enqueued for execution on a device using the *command_ queue* and *kernel* object. At this time the programmer can specify the *global_ work_ offset*, *global_ work_ size*, *local_ work_ size* and also a list of events that need to completed before this particular command could be executed. It returns *event* object that identifies the kernel instance.

```
cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
                               cl_kernel kernel,
                               cl_uint kernel,
                               const size_t *global_work_offset,
                               const size_t *global_work_size,
                               const size_t *local_work_size,
                               cl_uint num_events_in_wait_list,
                               const cl_event *event_wait_list,
                                   cl_event *event)
```

Finally the **clBuildProgram** program builds (i.e. compiles and links) a program executable for all the devices from the source or the binary associated with the *program* which was created using **clCreateProgram** or **clCreateProgramWithBinary**.

```
cl_int clBuildProgram (cl_program program,
                       cl_uint num_devices,
                       const cl_device_id *device_list,
                       const char *options,
                       void (CL_CALLBACK *pfn_notify)
                          (cl_program program, void *user_data),
                       void *user_data)
```

3.3.6 Conclusion

These are a few of the essential APIs provided by the OpenCL platform considered for this thesis. There are several other ways to control how the events are managed or synchronized for optimal utilization of the available processing units. Furthermore, there are even ways to partition the memory resources and allocate them for most efficient usage. These investigations were not performed for this project. This work barely utilizes the full potential of OpenCL and there is lot of scope for future studies on this subject.

4. ALGORITHM AND IMPLEMENTATION

This chapter explains in detail the filtering technique used to restore a compressed or low quality depth map; such that it can be used for 3D view synthesis. Further, it provides techniques to run the algorithm on a device with multiple processors like the CPU and GPU.

In order to create 3D content one needs to capture two slightly shifted views with a camera pair. Later when the left view is projected to the left eye and right view to the right eye, the brain will create the 3D experience out of it.

In the Mobile 3DTV project scenario where the 3D video data needs to be transmitted to a mobile device, it is necessary to use a format where the amount of data is kept minimal and transmission bit rate remains relatively constant. Hence only one view (color image) accompanied by a depth image is sent. The depth image is mostly of low resolution and compressed 8-bit gray scale image. It contains only a fraction of data compared to the color image.

The additional views are reconstructed using 3D rendering techniques at the receiver end before viewing. For stereoscopic displays which need 3D glasses, one additional view is enough. Whereas, for autostereoscopic displays, the number of views generated targets multiple users [30]. Since the depth map is of low quality, it has to be carefully restored in order to have a high quality rendering. The subsequent sections contain detailed explanation of the restoration technique used for this thesis work.

4.1 Filtering of depth map

Depth maps inherently undergo several degradations caused by up-scaling, quantization, coding errors or just false estimations. Its is critical that these artifacts are smoothened out so that they have minimal impact on the 3D view rendering. The quality of rendering has major impact on the 3D viewing experience and several studies have been conducted on stereo image quality [18] [22] [23] [24].

Smoothing is easy to achieve with low-pass Gaussian filter as explained by Paris *et al.* in [15]. Using simple variations of this techniques, Zhang *et al.* [9] showed improvements in depth quality from perspective of view synthesis. In the simplest form, the idea is to develop a weighting kernel which is applied over the whole image. As a result each pixel in the output image will be a weighted average of it's neighbors.

On the other hand, edges at the depth transitions are of great importance and need to be preserved. Thus the image has to be smoothened leaving the edges unaffected. This is where bilateral filtering described in [1] by Tomasi and Manduchi becomes relevant.

Bilateral filtering (shown in *Figure 4.2*) applies weighting similar to weighted Gaussian averaging. It is only that the influence of individual pixels on one another not only depend on it's proximity, but also similarity in their intensity values. At the edge, the intensity of pixels on one side of the edge varies significantly from the other. Thus, while smoothing, the pixels on different sides of the edge are treated differently. This prevents the blurring of edges.

4.1.1 Initial Inspiration for using Bilateral Filter on Depth Images

The depth maps obtained from the range sensors or in this case down-sampled for transmission are limited in resolution. This means that one initial depth pixel covers the area of several color pixels in the view plus depth 3D representation. Aligning of the depth map and its corresponding color image has to be performed in order to obtain the two modalities with the same size.

Now, the assumption is made that discontinuities in the depth image correlates directly with the edges in the texture image. Also the low frequency homogeneous regions in the color image are at about same distance from the sensor [17].

Considering the above assumption, a higher resolution depth image can be synthesized from a lower resolution depth image by taking assistance from the corresponding color image. In one method discussed by Yang *et al.* in [16] they consider the information in the depth map as not exact depth values but as probabilistic distribution from depth. Then the depth map was quantized into several layers to create a 3D volume of depth probability. This in computer vision literature is known as cost volume. The cost volume is then filtered iteratively using bilateral filter to find out the best match.

As discussed by Yang *et al.* in [16], using one or two registered color images as reference, the algorithm can iteratively refine the depth map, in terms of both its spatial resolution and depth precision. This whole process is illustrated in *Figure 4.1*. The low resolution depth image is first up-sampled to the size of the intensity image. This serves as the initial hypothesis from which the cost volumes are built. Using the information from the intensity image bilateral filter is applied to the cost volume. The best cost volume is selected as the refined depth hypothesis and and is used for the next iteration.

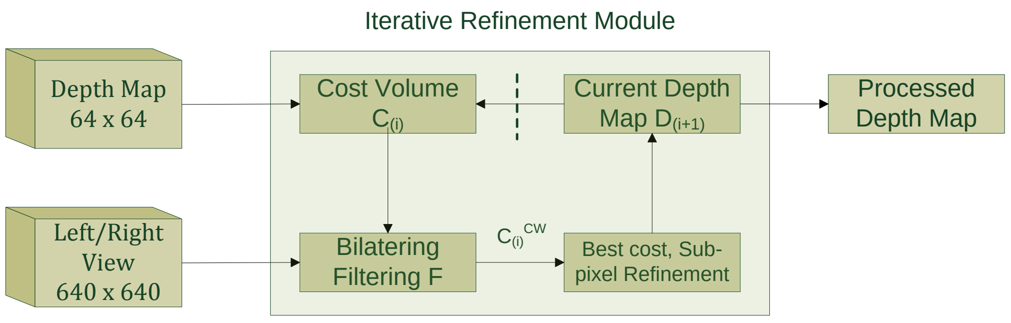


Figure 4.1: *Framework for spatial-depth super resolution for range images.*

The iterative refinement process shown in *Figure 4.1* has three major steps. First, a cost volume C_i is generated by up-scaling the current depth map $D_{(i)}$. Second, a bilateral Filtering is performed for each slice using the left/right view and the up-scaled depth image i.e. the initial cost volume, to generate a new cost volume $C_{(i)}^{CW}$. Finally the depth hypothesis with minimal cost is selected. A sup-pixel estimation is applied to obtain the final depth map $D_{(i+1)}$ which serves as the depth image for next iteration.

4.1.2 Bilateral Filter

Bilateral Filter forms one of the integral part in the iterative refinement process. It has been previously used with great success for stereo algorithms by Yang *et al.* in [25] and Yoon *et al.* in [26]. The next section will elaborately discuss on the mathematical representation of this filtering scheme.

The purpose of our filtering approach is to smooth the image while preserving the edges [1]. It utilizes both the intensity and range information from all color channels and also the spatial distance of the neighboring pixels to specify suitable weights for the filter. The filter favors pixels with similar intensities and also the ones with close proximity as explained by Smirnov in [13]. Thus pixels on the other side of an edge will always have lower weight and will have negligible contribution to the filtering process. *Figure 4.2* tries to illustrate the filtering approach and compares it to the Gaussian averaging.

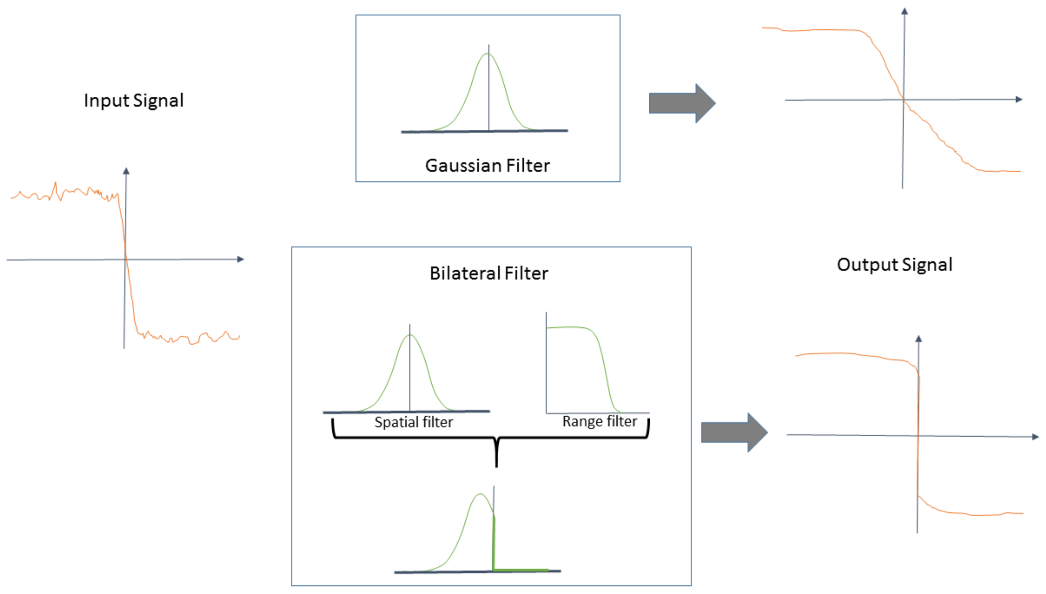


Figure 4.2: *Bilateral Filter representation.*

Mathematically the above idea can be formulated as:

$$\hat{Z}(k) = \frac{1}{F} \sum_{m \in \Gamma(k)} \{e^{-\frac{\|m-k\|^2}{2\sigma_s^2}}\}\{e^{-\frac{|I(m)-I(k)|}{2\sigma_i^2}} I(m)\} \tag{4.1}$$

$$F = \sum_{m \in \Gamma(k)} e^{-\frac{\|m-k\|^2}{2\sigma_s^2}} e^{-\frac{|I(m)-I(k)|}{2\sigma_i^2}} \tag{4.2}$$

where σ_s and σ_i controls the weights in spatial and intensity domains. The second term inside the summation resembles Gaussian averaging. Thus when the σ_i increases, the Gaussian curve widens or in other words becomes more flat. Thus the bilateral filter starts to approximate the Gaussian convolution more closely [15].

The normalization factor F ensures pixel weight sum to be unity and $\Gamma(k)$ represents a square window. For our implementation, I is the color image i.e. left or right view image in this case, $I(m)$ is the value of pixel within the window for the current iteration m and k indicates center of the window. If the difference in pixel intensity ($|I(m) - I(k)|$) of the color image turns out to be large, indicating a boundary pixel, the whole exponential term will be infinitely small. Thus, the pixels on the other side of the edge has no contribution to the filtered depth map. Similarly during the window operation, for pixels which are spatially further away from the center (determined by the tern $\|m - k\|$), the exponential term will again become small and will have lesser contribution to the overall filtering process.

4.2 Depth map filtering approach

As discussed earlier, this thesis aims to restore compressed and low resolution depth maps. The major issue here are compression artifacts and smoothened depth boundaries due to up-scaling. Bilateral filter described above, is a very good candidate for such problems. The current application estimates the filter weights using the color (left/right view) image and applies it to restore the compressed range image. This approach was described by Smirnov *et al.* in [14].

An important consideration for using the bilateral filter was that the depth of any surface is piece-wise smooth and it has direct correlation with scene color inside a region.

As described in *Figure 4.1*, first step is to calculate all the cost volumes from several depth hypothesizes. Subsequently, the one with the minimum cost is selected as shown in *Eq.* (4.3).

$$C_{(i)}(x,z) = \min(\delta * K, (z - \hat{Z}_{(i)}(x))^2) \tag{4.3}$$

where C is the cost volume at the *i-th* iteration, K is the search window size based on the constant δ, z is the potential depth candidate and $\hat{Z}$ is the current depth estimate at coordinate x.

Once the cost volumes are obtained, the bilateral filter represented by *Eq.* (4.1) was applied on individual cost volumes. This filtered or smoothened cost volume is used to find the potential depth candidate z in *Eq.* (4.4). The resulting filtered depth map can be seen in *Figure 4.3*.

$$\hat{Z}_{(i+1)}(x) = arg \min_{z}(\widehat{C_(i)}(x,z)) \tag{4.4}$$

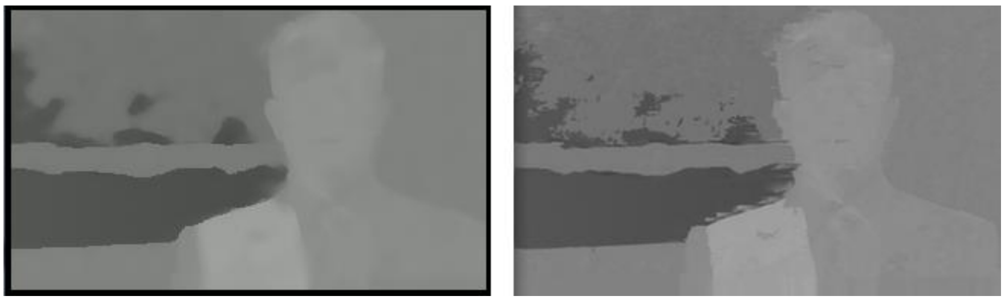


Figure 4.3: *Bullinger video sequence. On the left is the filtered depth map and the original one is on the right*

The MEX implementation of this algorithm showed satisfactory results for both quantitative and qualitative experiments. It was quite relevant for the mobile 3D TV kind of setup. Now, the aim was to make the filtering work in real-time. In order to do that the algorithm was ported on to a GPGPU (General Purpose Graphics Processing Unit).

4.3 Implementation in OpenCL

The bilateral filtering algorithm is specially suitable for parallel computation. The reason being, every pixel can be processed independently. The operations involved for a particular pixel do not depend on the results of the surrounding or neighboring pixels. Thus, once the data for the neighborhood is loaded into the local memory of the multiprocessor, simultaneous processing for several pixels can be carried out by the device. The only limiting factor is the number of threads that can be generated or the number of processing units available.

The algorithm first calculates the filter weights for each pixel using *Eq. 4.1*. The weights are derived from the difference in the intensity values and its spatial distance from the pixel under consideration. For each pixel only a window of certain radius is considered. If the radius is 'r', the size of the window is given as $(2r + 1)^2$.

For OpenCL implementation of this algorithm, every pixel is assigned a thread. This was done using available OpenCL APIs. Each of these threads was responsible for executing *Eq. 4.4*. However, the threads needed to access the data from the neighboring pixels within the window. The image data has to be arranged in such way that it can be accessed very fast. This is done by placing the data in the memory cache and not the global memory.

To address the above issue in OpenCL, the image buffers were used in place of normal buffers. This helps when the multiple threads need to access the same data simultaneously. Each thread loops over the entire window to calculate *Eq.*((4.1)) for single pixel. In a GPU, this windowing operation is performed for several pixels simultaneously. That is where most of the speed-ups come from.

In order to make the implementation more efficient, some modifications were proposed by Smirnov *et al.* in [14]. It was observed that there was no need to form cost volume in order to obtain the depth estimate for every pixel and all iterations. Instead, the cost function could be formed only for required neighborhood before filtering it. Furthermore, not all depth hypotheses were applicable for a given pixel. Thus, computation cost can be reduced by taking into consideration only depth within certain range. Now, in case of compressed depth maps with blocky artifacts the depth range appears to be sparse due to the quantization effect. Such depth maps were scaled to further reduce the number of hypothesizes [13].

The implementation was made in two parts. The host, i.e. the piece of code running on the CPU was written in C++, whereas the device code used to program the GPU device was written in OpenCL. The objective of the host code was to set up the device, allocate the memory buffers, transfer data to the device and put the kernel code on the queue for execution on the device. The device or kernel code actually performed all the operations related to the filtering of the depth map at the pixel level.

The OpenCL itself is capable of dividing the workload for its multiprocessors i.e. the programmer might not bother about how to arrange the data in the device memory. However, there might be some effect on the performance of the device if the data is not arranged properly in the local memory of the work group.

Additionally, it is also to be noted that the data transfer between the CPU and the GPU is considerably slower. So, the design has to be such that the transfers are kept as minimal as possible. The local memory of the Compute Units are considerably faster. All data is first arranged in those local cache before the processing begins; such that the processors does not need to wait for the data during execution.

Inside the GPU, the texture memory cache is faster than the constant cache. An useful technique might be to pack all the image data (RGB image and gray scale depth map) into one image structure and put it on to the texture cache using the OpenCL APIs. While reading from the cache, one may use the inbuilt API which reads single pixel data (R, G, B and depth) into a floating point vector in a single clock cycle. Most operations on the data were done on vectors taking the inline functions into use, making the implementation more efficient.

4.3.1 Installing OpenCL

A Xeon Dual core CPU with 3 GHz clock frequency and 3 MB RAM was used. As GPU, the NVIDIA GeForce 240 was used, which comes with 96 cores, graphics clock of 550 MHz, processor clock of 1340 MHz and 512 MB memory. For more details refer to Appendix B.1. There are few things that needs to be done before starting to develop application in OpenCL.

1. Install a development environment like Microsoft Visual Studio.
2. Install drivers for the GPU.
3. Download the OpenCL headers and libraries included in the OpenCL SDK from the web page [41].
4. Configure the compiler so that it can locate the OpenCL headers and libraries.
5. Edit the registry value for `GPU_MAX_HEAP_SIZE` to 512. Usually it is set to 256, which means the OpenCL could only use 256 Mb of it's 512 Mb memory (for this particular GPU).

4.3.2 Programming in OpenCL

An OpenCL program consists of two parts: Host code and Kernel code. The host code runs on the CPU where as the Kernel code runs on the GPU. An useful starting guide for programming in OpenCL can be found at website for ATI Stream [40].

Kernel Code

The Kernel code is in-fact an array of characters or a string. It can be in the same file as the host code which is written in C++; or it can be in a separate file, (e.g. *.cl), as was the case with this thesis work. As long as it starts with the keyword "_kernel" and the location of the file is known, the OpenCL API will able to locate it.

For this application, the OpenCL the API is defined as below. The complete OpenCL code is found at Appendix Listing A.1. All the arguments was defined in the host code which is explained in the next section. This section briefly describes how parallelization was achieved for the algorithm in OpenCL.

The API for the Kernel function

```
_kernel void hypothesis_filter_gpu (_global float* edge,
                                    _global float* depth,
                                    _global float* filtered,
                                    _global float* wt_table,
                                    _global float* dist_table,
                                    const _global int height,
                                    const _global int width,
                                    const _global int fltr_radius,
                                    const _global float search_limit)
```

The _global identifier informs the compiler that those buffers and variables are to be accessed from the device memory. They are put there by the host code using OpenCL APIs. The buffer "edge" contains the RGB image and "depth" contains the depth map. "wt_table" and "dist_table" are the look-up table for calculating the filter weights.

The OpenCL distributes the process in such a way that each worker thread gets one pixel to process. Since it was a 2-dimensional image, the indexing can be done as,

```
x = get_global_id(0);
y = get_global_id(1);

int index = ((y)+(x)*(h));
float Dnow = depth [ index ];
```

Similar indexing was done for the RGB image and a window around the pixel (x,y) was extracted. The `get_global_id(0)` and `get_global_id(1)` functions identifies the the horizontal "x" and vertical "y" indices for the current pixel in the image. They can be combined "((y)+(x)*(h))" to obtain unique index for the pixel. Then by just using "depth [index]" we can get the value at the "index" location of the depth map.

It is to be noted that the several indices are generated simultaneously by the OpenCL runtime. The "index" computed above helps the compute unit to identify with pixel it is supposed to process.

A little optimization was done such that for a block which is very smooth, which means the difference between the maximum and minimum depth value is very small, the depth value is left unchanged and no further processing was needed. This could be done in the same way as in a C program even though it is a part of the kernel code.

```
for(int i=0; i<window; i++)
{
    Dmax = (blkDepth[i] > Dmax) ? blkDepth[i] : Dmax;
    Dmin = (blkDepth[i] < Dmin) ? blkDepth[i] : Dmin;
}

if(abs(Dmax - Dmin) < 0.01)//search_step)
{
    Filtered[index] = Dnow;
    continue;
}
```

where, "blkDepth" is the window region from the depth image for a single pixel.

The next step was to calculate the weights for bilateral filter using the range information from all the color channels and the spatial distance of that neighboring pixel. For this the SAD (Sum of Absolute Difference) similarity measure was utilized between the current and the neighboring pixel. The SAD value was then multiplied to the distance measure between the same two pixels. This filter was then applied to the window extracted from the depth image.

Even though the steps involved in the filtering operation for CPU (MEX) explained in [14] and GPU (OpenCl) are identical, there is one major difference in the way the same instructions were executed. In CPU, the instructions were being executed sequentially; one pixel after another. Whereas in the OpenCL, the exact same instructions were processed simultaneously for several pixels over multiple cores. Using the "index" variable the OpenCL runtime kept track of pixels in the device memory space.

Host Code

A sequence of steps had to be performed to prepare the device to run the code. They are ordered as follows: initialize OpenCL, compile the kernel code, transfer all the necessary data to the device memory, execute the kernel and finally read the data back to the CPU side.

1. *Initializing OpenCL:* This will initialize the OpenCL platforms and devices. Then set up the environment for executing the OpenCL application. For this first identify the platforms using the API `clGetPlatformIDs` and then get the devices associated with each platform using the API `clGetDeviceIDs`.

2. *Creating context:* Now that the devices available are known, use the API `clCreateContext` to create context for each device. The context for OpenCL is created to keep track of all the OpenCL devices. This is then used by OpenCL runtime for managing command queue, memory, kernels and also for executing kernels on one or more OpenCL devices.

3. *Create command queues:* This is one of the most critical step for OpenCL programming. It defines which commands are to be executed by which device and in what order. That is performed using the API `clCreateCommandQueue` by passing the context and the device as parameters.

4. *Compile kernel code:* First a kernel has to be initialized and then compiled using an API. Interestingly, it can be divided into several strings. For this

purpose, all the characters of the kernel code are put into a character array and then passed to a function called `clCreateProgramWithSource` which creates an `cl_program` object. This object is then passed to the function `clBuildProgram` for "just-in-time" (JIT) compilation. Finally, create a handle to the compiled OpenCL function using the API `clCreateKernel`.

5. *Creating kernel arguments:* The arguments that have to be passed to the kernel have to be first initialized and allocated for both the CPU and the GPU memory. The implementation described here had three buffers for the color data, depth data and filtered depth output data; two smaller matrices for filtering; and constant scalar parameters like width, height, filter radius and search limit. Allocation on the CPU side was done with standard C library function `malloc` and on the GPU side with the OpenCL API `clCreateBuffer`. On the GPU side these buffers are then initialized using `clSetKernelArg`.

6. *Writing data to device memory:* Once the buffers are created on both sides, the API `clEnqueueWriteBuffer` copies each buffer to the device memory. This is the memory that is accessed by the GPU.

7. *Executing the kernel code:* The execution of the kernel code is distributed over several cores of the GPU. In order to do that the OpenCL runtime has to create worker threads which are executed by each Processing Unit. The number and dimension of worker threads can be specified by the programmer. This is done using the API `clEnqueueNDRangeKernel`. For this application, the work group was 2-dimensional i.e. the height and width of the color image.

8. *Retrieving data back to CPU:* Once all the computation is done the resulting depth map is copied back to the CPU buffer with the help of the function `clEnqueueReadBuffer`. This can now be saved to a desired file format or be used by any other algorithm running on the CPU.

4.3.3 OpenCL Tools

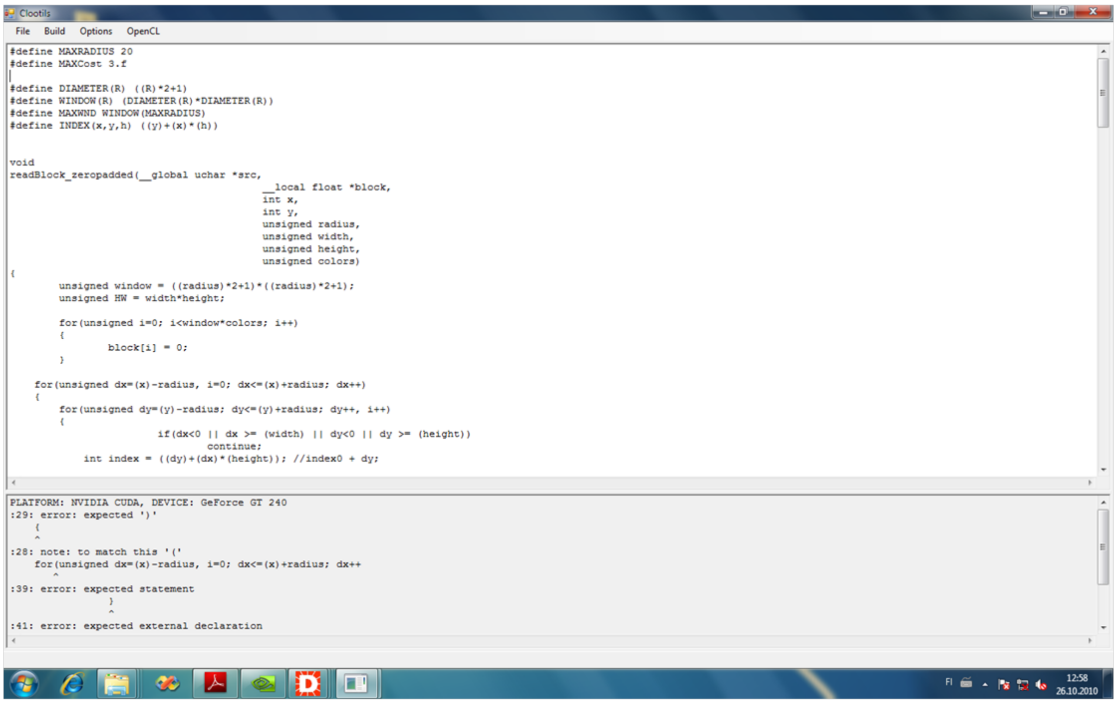
Since the OpenCL compilation is done using the JIT compilation, compilation time errors are difficult to detect. Hence a tool called Cloo was used to help with

OpenCL compile time errors. To analyze the efficiency of the program over the GPU, a separate tool called gDEBugger was used which is described below.

Editor and Compiler

Editors like Microsoft Visual Studio will be able to compile the OpenCL kernel code, although it cannot point out the compilation errors. It indicates failure as a return value to the OpenCL API but does not point out the exact error. That makes it hard to identify and fix the problem in the kernel code.

The Khonos group provided a simple and easy to use editor called Cloo. One can write the kernel code in the editor and then compile it. In case of compilation errors, Cloo precisely points out the error. One such example is shown in the screenshot in *Figure 4.4*.



```
#define MAXRADIUS 20
#define MAXCost 3.f

#define DIAMETER(R)  ((R)*2+1)
#define WINDOW(R)  (DIAMETER(R)*DIAMETER(R))
#define MAXWND WINDOW(MAXRADIUS)
#define INDEX(x,y,h)  ((y)+(x)*(h))


void
readBlock_zeropadded(__global uchar *src,
                                        __local float *block,
                                        int x,
                                        int y,
                                        unsigned radius,
                                        unsigned width,
                                        unsigned height,
                                        unsigned colors)
{
        unsigned window = ((radius)*2+1)*((radius)*2+1);
        unsigned HW = width*height;

        for(unsigned i=0; i<window*colors; i++)
        {
                block[i] = 0;
        }

    for(unsigned dx=(x)-radius, i=0; dx<=(x)+radius; dx++)
    {
        for(unsigned dy=(y)-radius; dy<=(y)+radius; dy++, i++)
        {
                        if(dx<0 || dx >= (width) || dy<0 || dy >= (height))
                                continue;
            int index = ((dy)+(dx)*(height)); //index0 + dy;
```

```
PLATFORM: NVIDIA CUDA, DEVICE: GeForce GT 240
:29: error: expected ')'
    {
    ^
:28: note: to match this '('
    for(unsigned dx=(x)-radius, i=0; dx<=(x)+radius; dx++
       ^
:39: error: expected statement
                }
                ^
:41: error: expected external declaration
```

Figure 4.4: *Cloo: The OpenCL compiler and editor*

Debugger Profiler and Memory Analyzer

Once the kernel code compiles, it becomes necessary to investigate how efficiently the available resources are being utilized. For this purpose gDEBugger was used, which served as the debugger, profiler and memory analyzer for OpenCL applications. It helps to sort out the OpenCL related bugs and also helps trace how the processing work load was distributed within the OpenCL platform. Detailed information regarding the tool can be found at the website [39]. The tool solves three basic requirements:

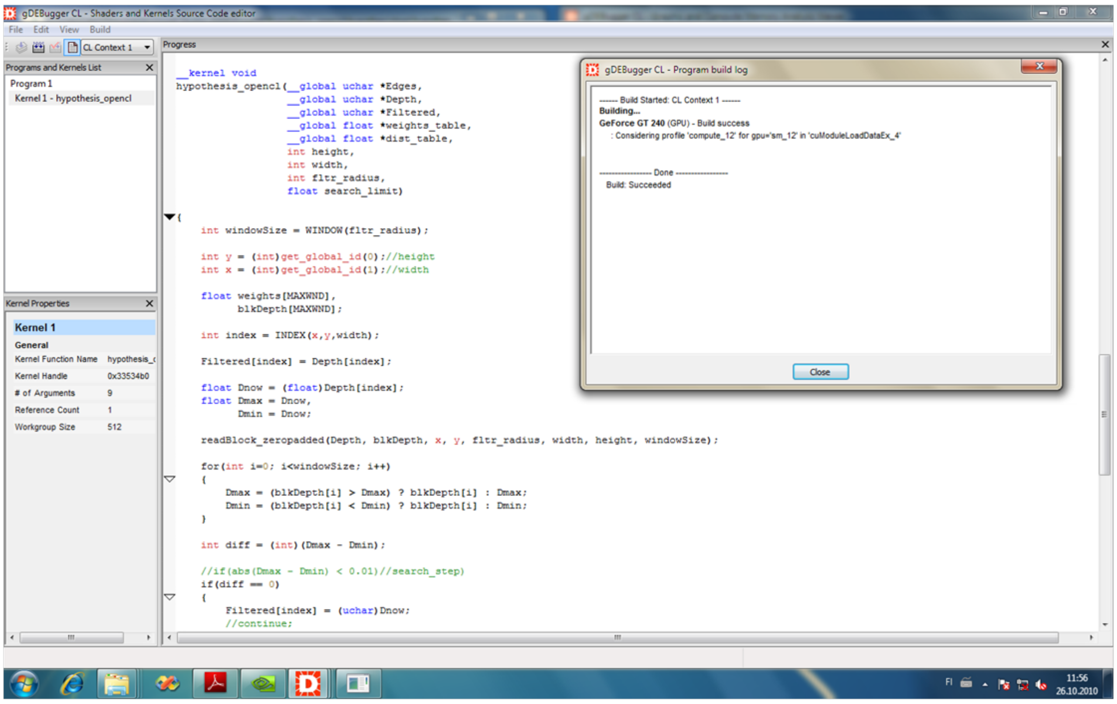
Debug It helps to locate OpenCL errors in the source code. The option can be set such that the execution breaks automatically when it encounters an OpenCL error. Appropriate lines of the source code which might have caused the error can be accessed via the Call Stack Viewer. Additionally there is the provision to check memory leaks and even break the process when leaks are detected.

The kernel code can be viewed from Shaders and Kernels Source code editor shown in the screenshot in *Figure 4.5*. Here the code can be edited and built. The build logs appears in a pop-up window.

One important feature of this debugger is the capability to set breakpoints to any OpenCL function. Finally, the Texture, Buffer and Image Viewer allow user to visualize the image and buffer information. For image data, one can zoom in or out and by just clicking on an individual pixel the data related to it can be seen.

Profile The results of the profiling for the Hypothesis filter can be seen in the bottom half of *Figure 4.6*. It shows the performance statistics for the distribution of workload between the CPU and GPU devices. On the top portion of figure, it can be seen that the profile information was taken during the execution of the function `clEnqueueNDRangeKernel`. The top right shows the list of functions, number of calls and percentage of calls for each function.

For this application, it was observed that the CPU resources were relatively free and the OpenCL kernel commands were utilizing 96 percent of GPU resources. The queue idle time was very small, only 3.6 percent. This shows that the code


gDEBugger CL - Shaders and Kernels Source Code editor
File Edit View Build
CL Context 1
Progress
Programs and Kernels List
Program 1
Kernel 1 - hypothesis_opencl
Kernel Properties
gDEBugger CL - Program build log
Close

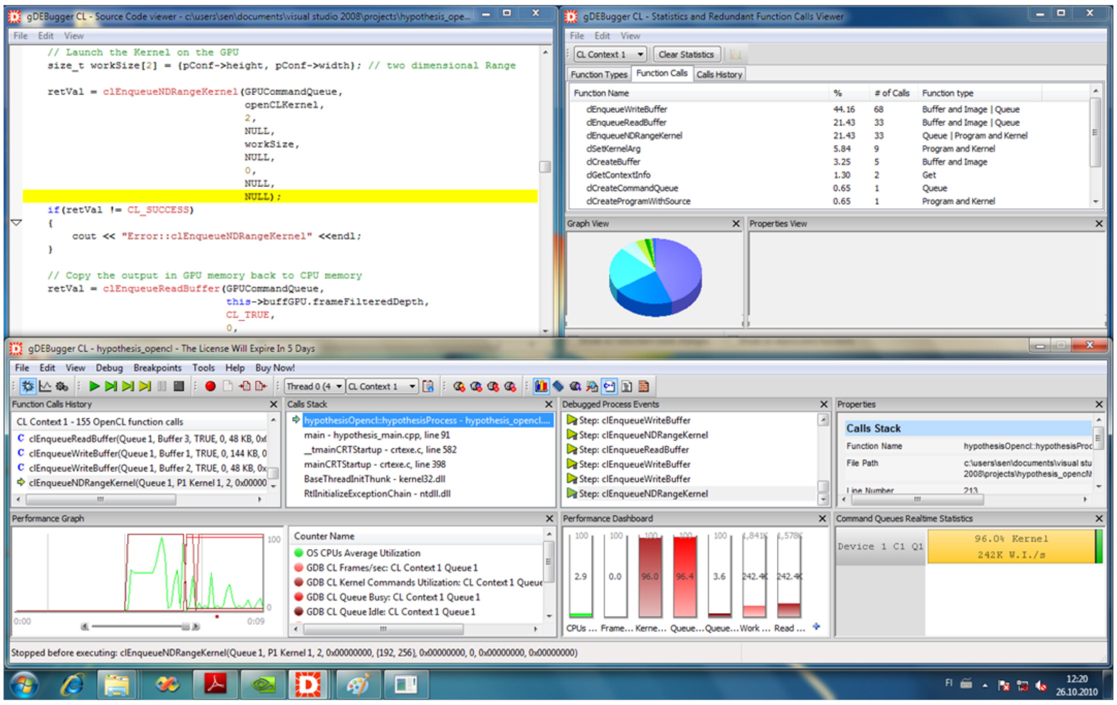
Kernel 1	
General	
Kernel Function Name	hypothesis_
Kernel Handle	0x33534b0
# of Arguments	9
Reference Count	1
Workgroup Size	512

```
__kernel void
hypothesis_opencl(__global uchar *Edges,
                  __global uchar *Depth,
                  __global uchar *Filtered,
                  __global float *weights_table,
                  __global float *dist_table,
                  int height,
                  int width,
                  int fltr_radius,
                  float search_limit)

{
    int windowSize = WINDOW(fltr_radius);

    int y = (int)get_global_id(0);//height
    int x = (int)get_global_id(1);//width

    float weights[MAXWND],
          blkDepth[MAXWND];

    int index = INDEX(x,y,width);

    Filtered[index] = Depth[index];

    float Dnow = (float)Depth[index];
    float Dmax = Dnow,
          Dmin = Dnow;

    readBlock_zeropadded(Depth, blkDepth, x, y, fltr_radius, width, height, windowSize);

    for(int i=0; i<windowSize; i++)
    {
        Dmax = (blkDepth[i] > Dmax) ? blkDepth[i] : Dmax;
        Dmin = (blkDepth[i] < Dmin) ? blkDepth[i] : Dmin;
    }

    int diff = (int)(Dmax - Dmin);

    //if(abs(Dmax - Dmin) < 0.01)//search_step)
    if(diff == 0)
    {
        Filtered[index] = (uchar)Dnow;
        //continue;
```

```
------- Build Started: CL Context 1 -------
Building...
GeForce GT 240 (GPU) - Build success
    : Considering profile 'compute_12' for gpu='sm_12' in 'cuModuleLoadDataEx_4'


------------------ Done ------------------
  Build: Succeeded
```

Figure 4.5: *gDEBugger: Shaders and Kernels Source code editor*

was well optimized for parallel processing. Since the analysis was done in the Debug mode the execution was very slow.



gDEBugger CL - Source Code viewer - c:\users\sen\documents\visual studio 2008\projects\hypothesis_ope...

```
    // Launch the Kernel on the GPU
    size_t workSize[2] = {pConf->height, pConf->width}; // two dimensional Range

    retVal = clEnqueueNDRangeKernel(GPUCommandQueue,
                                    openCLKernel,
                                    2,
                                    NULL,
                                    workSize,
                                    NULL,
                                    0,
                                    NULL,
                                    NULL);
    if(retVal != CL_SUCCESS)
    {
        cout << "Error::clEnqueueNDRangeKernel" <<endl;
    }

    // Copy the output in GPU memory back to CPU memory
    retVal = clEnqueueReadBuffer(GPUCommandQueue,
                                 this->buffGPU.frameFilteredDepth,
                                 CL_TRUE,
                                 0,
```

gDEBugger CL - Statistics and Redundant Function Calls Viewer

Function Name	%	# of Calls	Function type
clEnqueueWriteBuffer	44.16	68	Buffer and Image | Queue
clEnqueueReadBuffer	21.43	33	Buffer and Image | Queue
clEnqueueNDRangeKernel	21.43	33	Queue | Program and Kernel
clSetKernelArg	5.84	9	Program and Kernel
clCreateBuffer	3.25	5	Buffer and Image
clGetContextInfo	1.30	2	Get
clCreateCommandQueue	0.65	1	Queue
clCreateProgramWithSource	0.65	1	Program and Kernel

gDEBugger CL - hypothesis_opencl - The License Will Expire In 5 Days

Function Calls History

CL Context 1 - 155 OpenCL function calls

```
C clEnqueueReadBuffer(Queue 1, Buffer 3, TRUE, 0, 48 KB, 0x
C clEnqueueWriteBuffer(Queue 1, Buffer 1, TRUE, 0, 144 KB, 0
C clEnqueueWriteBuffer(Queue 1, Buffer 2, TRUE, 0, 48 KB, 0x
  clEnqueueNDRangeKernel(Queue 1, P1 Kernel 1, 2, 0x00000
```

Calls Stack

```
hypothesisOpencl::hypothesisProcess - hypothesis_opencl....
main - hypothesis_main.cpp, line 91
__tmainCRTStartup - crtexe.c, line 582
mainCRTStartup - crtexe.c, line 398
BaseThreadInitThunk - kernel32.dll
RtlInitializeExceptionChain - ntdll.dll
```

Debugged Process Events

```
Step: clEnqueueWriteBuffer
Step: clEnqueueNDRangeKernel
Step: clEnqueueReadBuffer
Step: clEnqueueWriteBuffer
Step: clEnqueueWriteBuffer
Step: clEnqueueNDRangeKernel
```

Properties

Calls Stack

Function Name	hypothesisOpencl::hypothesisProc
File Path	c:\users\sen\documents\visual stu 2008\projects\hypothesis_opencl\
Line Number	213

Performance Graph

Counter Name
- ● OS CPUs Average Utilization
- ● GDB CL Frames/sec: CL Context 1 Queue 1
- ● GDB CL Kernel Commands Utilization: CL Context 1 Queue
- ● GDB CL Queue Busy: CL Context 1 Queue 1
- ● GDB CL Queue Idle: CL Context 1 Queue 1

Performance Dashboard

Command Queues Realtime Statistics

```
Device 1 C1 Q1    96.0% Kernel
                  242K W.I./s
```

Stopped before executing: clEnqueueNDRangeKernel(Queue 1, P1 Kernel 1, 2, 0x00000000, {192, 256}, 0x00000000, 0, 0x00000000, 0x00000000)

Figure 4.6: *gDEBugger: Debugger, Profiler and Memory Analyzer*

5. RESULTS AND ANALYSIS

The objective of this thesis work was to confirm that the algorithms like bilateral filter can be effectively ported to a GPU, making it much more efficient for real-time use cases. For the Mobile 3DTV project, this filter was being used to restore depth maps which are then used for 3D view rendering on mobile devices. This chapter discusses the findings related to the filter implementation. The results points to the fact that bilateral filter is well suited for GPU devices. Furthermore, OpenCL as a platform makes it convenient for programmers to port algorithms written in C or MatLab to any such devices.

For the purpose of this thesis, all the 3D video sequences including depth maps used were taken from the Mobile 3DTV project database [42]. The specific sequences are Bullinger, Caterpillar, Car and Book Arrival.

A PC with Xeon Dual core CPU, running at 3 GHz clock frequency and 3 MB RAM was used. It also had a GPU (NVIDIA GeForce 240), which comes with 96 cores, graphics clock of 550 MHz, processor clock of 1340 MHz and 512 MB memory. For more details please refer to Appendix B.1.

There are multiple things that were tried in the scope of this project. First, to port the algorithm discussed by Smirnov et al. in [14] to a GPU device using OpenCL. Second, measure the gains in the processing time using GPU over the CPU implementation which was written in C and MatLab as MEX file. Thirdly, to verify if compressing the depth map with different quantization parameters (QP) have any effect on filtering speed.

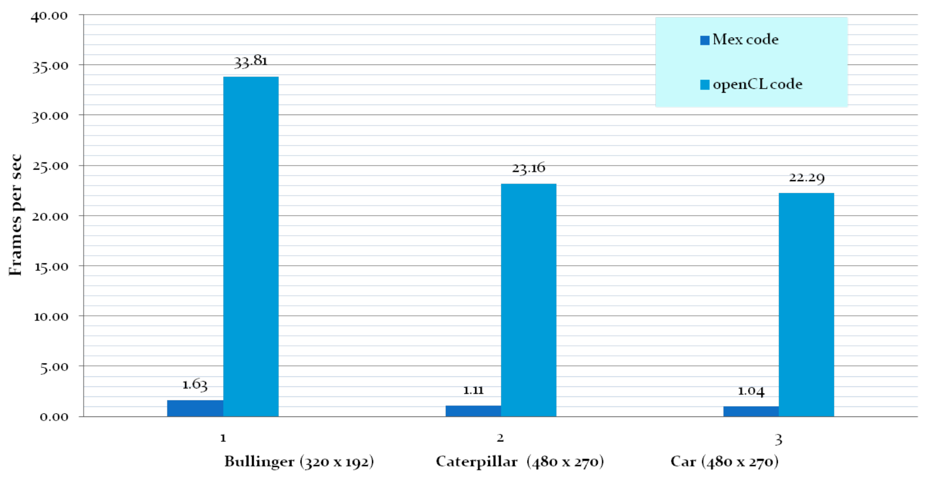


Figure 5.1: *Performance analysis between OpenCl and Mex code. Filter size used was 9.*

5.1 Improvements in Processing Time

The processing time received a significant boost from the GPU implementation as shown in *Figure 5.1*. It shows time measurements for three different video sequences: Bullinger, Caterpillar and Car. During the experiment, all the processes running on the PC were suspended, leaving only the operating system and few other essential ones. After this, first the MEX implementation was run for all the three sequences which generated the time log. Next, the OpenCL implementation was run which also generated a similar log file. All the data was put in an MS Excel worksheet to generate the graphs.

The timing was first noted for 100 frames and then the Frames per Second (*fps*) value was calculated. The idea behind choosing *fps* as the metric was that the ultimate intention was to find out if this filtering operation can be performed in real-time (around 25 *fps*).

From the results it was deduced that the operation was around 20.74x, 20.81x and 21.43x faster on GPU for Bullinger, Caterpillar and Car respectively. On average about 21x improvement in processing speed was achieved.

It is to be noted that the cache hit/miss rate is one of the key factor in determining how effectively the GPU is being utilized. It was observed that the hit/miss rate decreased with the increase in the window size. For out largest window size, only 0.2 percent of the data access happened from the device memory and rest of it was accessed from the fast cache.

Once it was confirmed that the OpenCL implementation was much more efficient in terms of processing time, the next step was to test processing times for videos with different resolutions. The filtering operation was run for three sequences: Bullinger (320 x 192), Caterpillar (480 x 270) and Car (480 x 270). The results are shown in *Figure 5.2*. From the graph, it can be inferred, that for this particular GPU, only data set having resolution of around (320 x 192) can achieve near real-time performance with our implementation having a filter size of 9.

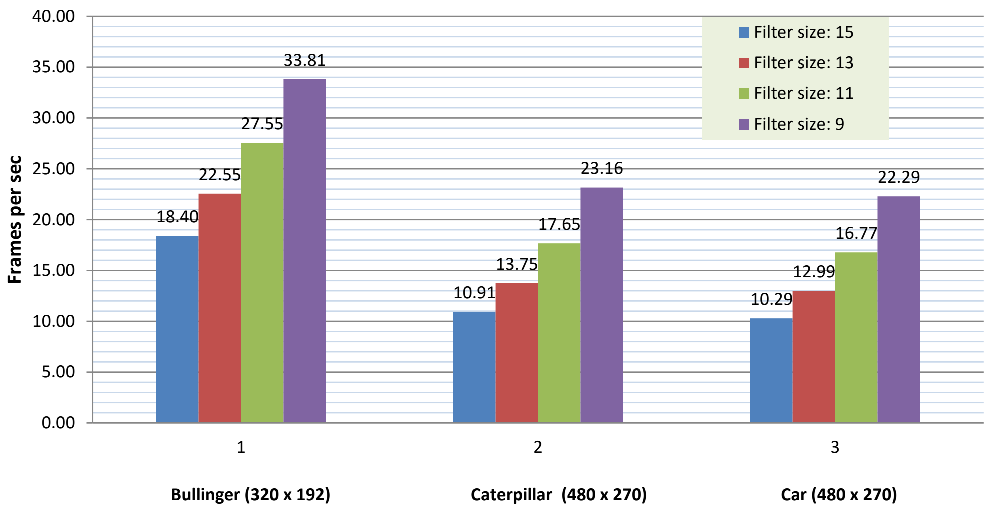


Figure 5.2: *Processing time over different video sequences: 1. Bullinger (320 x 192) 2. Caterpillar (480 x 270) 3. Car (480 x 270) for varying filter sizes (9, 11, 13 and 15)*

5.2 Effect of Compression Artifacts on Processing Time

Next study was made to understand if the amount of degradation in image quality has any effect on filtering time. Three sequences were chosen; Bullinger, Caterpillar and Car. Using H.264 encoder all depth maps were compressed with quantization factors of 25, 30, 35 and 40. It was observed in the results shown in *Figure 5.3* that the different levels of compression artifacts had no effect on the processing time.

H.264 is a block based video encoding scheme which is inherently lossy and hence degrades the image quality to certain extent. However, it tries to minimize the effect by removing only the truly redundant image data. The Quantization Parameter dictates how much of the spatial details can be compromised. When the QP is small, almost all pixel information is retained and vice-versa. The effect of this compression technique with QP = 35 is seen in *Figure 5.4 (b)*.

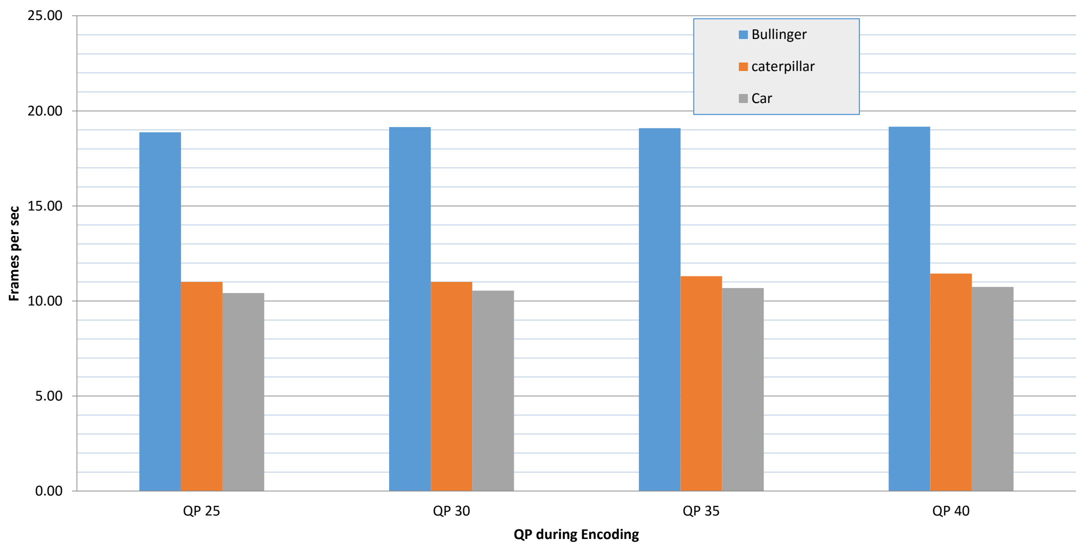


Figure 5.3: *Variation of execution speed for different data sets encoded using H264 encoder with different values of Quantization Parameters (25, 30, 35 and 40). Filter size used was 15.*

5.3 Bilateral Filtering Results

Figure 5.4 shows the filtering results from the Car sequence. The top row has the original frame 90 on the left. On the right there is the degraded frame after it was compressed with QP 35. The artifacts are seen quite prominently on the back of the car. The edges are rough and also broken at places.

Figure 5.4 (c), (d), (e), (f) shows the effect of bilateral filter on the compressed depth map *(Figure 5.4 (b))* with filter size of 9, 11, 13 and 15 respectively. It can be observed that the image becomes smoother with the increase in filter size. Furthermore, the boundaries, which are one of the most essential features of a depth map become distinct and sharp. Overall, the results from the OpenCL code looked very promising.

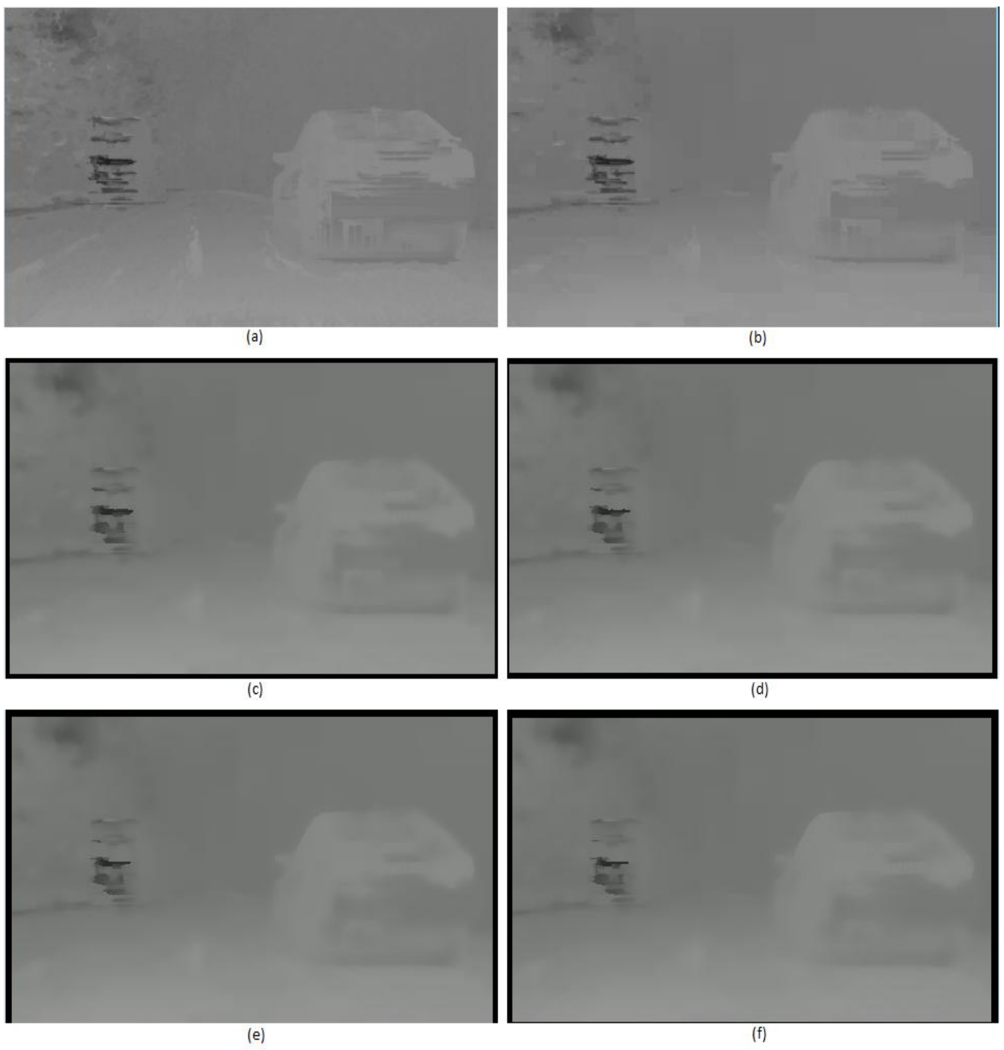


Figure 5.4: *Car, Frame no. 90, Depth encoded with QP = 35. (a) Original depth (b) Depth encoded with QP = 35 (c) Filtered with block size = 9 (d) Filtered with block size = 11 (e) Filtered with block size = 13 (f) Filtered with block size = 15*

6. CONCLUSION

Mobile3DTV project dealt with the capture, coding and transmission of 3D video to mobile devices. It also studied different 3D video representation formats for their applicability in such mobile delivery scenario. In the scope of this thesis, the DIBR model was chosen. Here, one view plus depth was to be transmitted to a mobile device and the second view was to be rendered on the device at real-time. This was a good technique since single channel depth data is relatively small and uses far less bandwidth compared to RGB intensity data.

Depth maps can be degraded with several different artifacts which can negatively affect the view synthesis. So, they need to undergo restoration before being used for further processing. This thesis work used a modified version of bilateral filter for the restoration process. The modification were made such that the weights for the filter was calculated using the RGB image (left/right view) and then applied to the depth image. The reason being, the textures and edges are better preserved in the intensity image.

Since the filtering process had to run in real-time, it was decided to take GPU into use. At the end of our study, it can be safely said that for these kind of algorithms GPU is the way forward. Since the computations for one pixel is independent of the results from it's neighbors, each pixel can be assigned to a GPU core, which means computation on several pixels can be done simultaneously. Thus, the number of pixels that can be processed independently is directly proportional to the number of GPU cores.

The study shows that under certain constrains, real-time performance is achievable for our restoration algorithm. Experiments show Bullinger video sequence with size (320 x 192) attained near real-time performance whereas the other two sequences Car and Caterpillar with size (480 x 270) fell slightly short. There is also another very important factor to be considered here. Since the GPU is handling most of the computation, CPU is meanwhile free to take up other tasks.

Another conclusion from this study was that OpenCL is quite an useful platform for implementing and porting algorithms to GPU. The language in which the actual kernel code is written is very similar to C. That makes it easy for most programmers who are already familiar with C. Moreover, in the field of Computer Vision most core algorithms are written in C, thus it is convenient to port those algorithms to OpenCL.

The improvement in processing efficiency of the OpenCL version compared to the implementation on CPU was optimistic. The OpenCL code was not the most optimized and there are rooms for further improvements. For this implementation most of the memory utilization was left to OpenCL; which might not be the best coding practice. Thus, the implementation can be made more efficient if one can develop better understanding of the OpenCL architecture.

With the sharp rise in the GPU enabled devices and parallel computing platforms, it is a good idea to investigate if the particular task can be parallelized. For algorithms similar to the one that is described in this thesis, GPU implementations should be encouraged. Moreover, the introduction of free and device independent standards like OpenCL will support the wide adoption of GPU in the coming years.

REFERENCES

[1] C. Tomasi, R. Manduchi, Bilateral Filtering for Gray and Color Images, 6th International Conference on Computer Vision, 1998, ICCV, pp.839

[2] M. Elad, On the origin of the bilateral filter and ways to improve it, Image Processing, IEEE Transactions on 11.10 (2002), pp. 1141-1151

[3] R. Lange, P. Seitz, Solid-state time-of-flight range camera, IEEE Journal of Quantum Electronics, Vol 37 no. 3, 2001, pp 390-397.

[4] M. Subbarao, G. Surya, Depth from defocus: a spatial domain approach, International Journal of Computer Vision Vol 13 no.3, 1994, pp 271-294.

[5] A. Levin, R. Fergus, F. Durand, W. T. Freeman, Image and depth from a conventional camera with a coded aperture, ACM Transactions on Graphics (TOG), Vol. 26, No. 3, 2007, p 70.

[6] C. Zhou, S. Lin, S. K. Nayar. Coded aperture pairs for depth from defocus and defocus deblurring, International journal of computer vision Vol. 93 no. 1, 2011, pp 53-72.

[7] M. Solh, G. AlRegib. A no-reference quality measure for DIBR-based 3D videos, IEEE International Conference on Multimedia and Expo (ICME), 2011, pp. 1-6

[8] C. Fehn, A 3D-TV approach using depth-image-based rendering (DIBR), Processing of Visualization Imaging and Image Processing, Vol. 3 No. 3, 2003

[9] L. Zhang, W J Tam, Stereoscopic image generation based on depth images for 3D TV, IEEE Transactions on Broadcasting, Vol. 51 no. 2, 2005, pp. 191-199

[10] LM. Po, S. Zhang, X. XU, Y. Zhu, A new multidirectional extrapolation hole-filling method for depth-image-based rendering, IEEE International Conference on Image Processing, 2011, ICIP, pp. 2589-2592

[11] M. Solh, G. AlRegib, Hierarchical hole-filling for depth-based view synthesis in FTV and 3D video, IEEE Journal of Selected Topics in Signal Processing, Vol 6 no. 5, 2012, pp. 495-504

[12] S. Smirnov, A. Gotchev, K. Egiazarian, Methods for restoration of compressed depth maps: a comparative study, Fourth International Workshop on Video Processing and Quality Metrics for Consumer Electronics, VPQM, 2009, p. 6

[13] S. Smirnov, A. Gotchev, S. Sen, G. Tech, H. Brust, 3D Video Processing Algorithms Part I, MOBILE3DTV, Project No. 216503, p. 22

[14] S. Smirnov, A. Gotchev, K. Egiazarian, A Memory-Efficient and Time-Consistent Filtering of Depth Map Sequences, Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, FEB 2010

[15] S. Paris, P. Kornprobst, J. Tumblin, F. Durand, Bilateral filtering: Theory and applications, Now Publishers Inc, 2009, p. 7

[16] Q. Yang, R. Yang, J. Davis, D. NistÂ´er, Spatial-Depth Super Resolution for Range Images, IEEE Conference on Computer Vision and Pattern Recognition, 2007

[17] J. Park, H. Kim, Y.W. Tai, Y.S. Brown, High quality depth map upsampling for 3D-TOF cameras, IEEE International Conference on Computer Vision, ICCV, 2011

[18] S. Ryu, D.H. Kim, K. Sohn, Stereoscopic image quality metric based on binocular perception model, 19th IEEE International Conference on Image Processing, 2012, pp. 609-612

[19] M. Matthieu, M. N. Do, Joint encoding of the depth image based representation using shape-adaptive wavelets, 15th IEEE International Conference on Image Processing, 2008, pp. 1768-1771

[20] R. Krishnamurthy, C Bing-Bing, H. Tao, S Sethuraman, Compression and transmission of depth maps for image-based rendering, Proceedings of International Conference on Image Processing, 2001, vol. 3, pp. 828-831.

[21] W. S. Kim, A. Ortega, P. Lai, D. Tian, C. Gomila, Depth map coding with distortion estimation of rendered view, SPIE Electronic Imaging International Society for Optics and Photonics, pp. 75430B-75430B

[22] Q. Huynh-Thu, P. Le Callet, M. Barkowsky, Video quality assessment: From 2D to 3D Challenges and future trends, 17th IEEE International Conference on Image Processing, 2010, pp. 4025-4028

[23] A. Benoit, P. Le Callet, P. Campisi, Quality assessment of stereoscopic images, EURASIP Journal on Image and Video Processing, Springer, 2008, Article-ID 659024

[24] C.T.E.R. Hewage, T.E.R. Chaminda, M.G. Martini, Reduced-reference quality metric for 3D depth map transmission, 3DTV-Conference: The True Vision-Capture, Transmission and Display of 3D Video, IEEE, 2010, pp. 1-4

[25] Q. Yang, L. Wang, R. Yang, H. StewÂ´enius, D. NistÂ´er, Stereo Matching with Color-Weighted Correlation, Hierarchical Belief Propagation and Occlusion Handling, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 31 no. 3, 2009, pp. 492-504

[26] K.J. Yoon, S. Kweon, Adaptive Support-Weight Approach for Correspondence Search, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 28, no. 4, 2006

[27] K. Dabov, A. Foi, V. Katkovnik, K. Egiazarian, Image Denoising by Sparse 3-D Transform-Domain Collaborative Filtering, IEEE Transactions on Image Processing, Vol 16, 2007

[28] F. Zilly, J. Kluger, P. Kauff, Production Rules for Stereo Acquisition, Proceedings of the IEEE, Vol. 99, No. 4, 2011

[29] A. Smolic, P. Kauff, S. Knorr, A. Hornung, M. Kunter, M. Muller, M. Lang, Three-Dimensional Video Postproduction and Processing, Proceedings of the IEEE, Vol. 99, No. 4, 2011

[30] M. Karsten, P. Merkle, T. Wiegand, 3-D video representation using depth maps, Proceedings of the IEEE, Vol 99, No. 4, 2011

[31] J.M. Rolfe, K.J. Staples, Flight Simulation, Cambridge University Press, 1986, p. 134

[32] D. Scharstein, R. Szeliski, A taxonomy and evaluation of dense two-frame stereo correspondence algorithms, International Journal of Computer Vision, vol. 47 no. 1, 2002, pp. 7-42

[33] A. Bhatti, Current Advancements in Stereo Vision, Chapter 1, InTech, 2012

[34] G. Vogiatzis, P.H.S Torr, R Cipolla, Multi-view stereo via volumetric graph-cuts, IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Vol. 2, 2005, pp 391-398

[35] M.F. Tappen, W.T. Freeman, Comparison of graph cuts with belief propagation for stereo, using identical MRF parameters, IEEE International Conference on Computer Vision, vol. 2, 2003, pp. 900-906

[36] D. Gallup, J.M. Frahm, P. Mordohai, Q. Yang, M. Pollefeys, Real-time plane-sweeping stereo with multiple sweeping directions, IEEE Conference on Computer Vision and Pattern Recognition, 2007, pp 1-8

[37] A. Munshi "The OpenCL Specification", Khronos OpenCL Working Group, Version 1.2

[38] D. Scharstein, A. Vandenberg-Rodes,R. Szeliski, 2003 Stereo datasets with ground truth, Available (accessed on 10.02.2016) http://vision.middlebury.edu/stereo/data/scenes2003/

[39] Introduction to gDEBugger, Graphic Remedy, Available(accessed on 17.01.2016): http://www.gremedy.com/tutorial/

[40] ATI Stream OpenCL Technical Overview Video Series, AMD Developer Central, Available(accessed on 17.01.2016): http://developer.amd.com/resources/documentation-articles/videos/ati-stream-opencl-technical-overview-video-series/

[41] CUDA toolkit, NVIDIA Corporation, Available(accessed on 17.01.2016): https://developer.nvidia.com/cuda-toolkit

[42] 3D Video Database, Mobile 3DTV, Available(accessed on 17.01.2016): http://sp.cs.tut.fi/mobile3dtv/video-plus-depth/

A. CODE LISTING

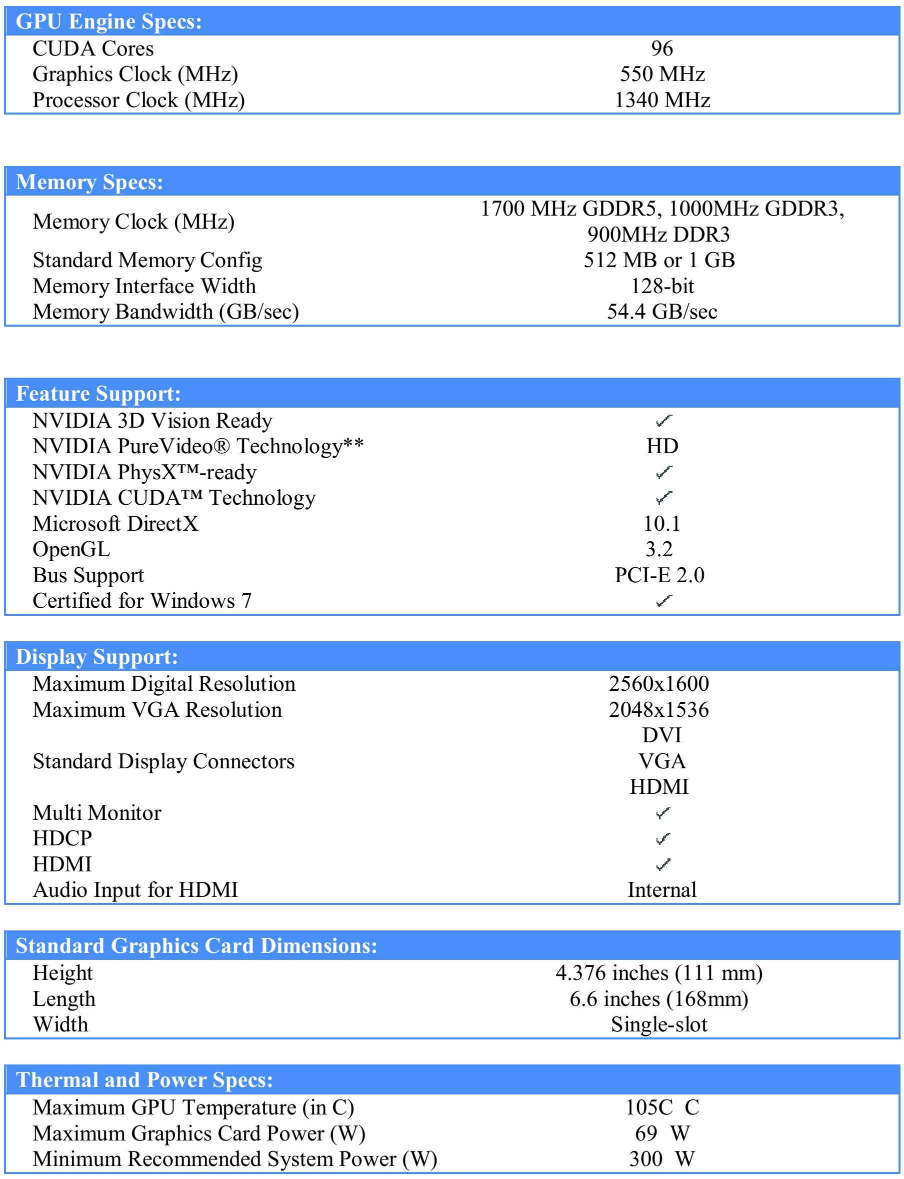
Listing A.1: OpenCL Kernel code for Hypothesis filter

```
__kernel void hypothesis_filter_gpu
                    (__global float* edge,
                     __global float* depth,
                     __global float* filtered,
                     __global float* wt_table,
                     __global float* dist_table,
                     const __global int height,
                     const __global int width,
                     const __global int fltr_radius,
                     const __global float search_limit)

{
    x = get_global_id(0);
    y = get_global_id(1);

    float weights[MAXWND], blkDepth[MAXWND];

    int index = INDEX(x,y,height);

    float Dnow = Depth[index];
    float Dmax = Dnow, Dmin = Dnow;

    readBlockFloat_zeropadded((float *)Depth, blkDepth, x, y,
     radius, width, height);

    for(int i=0; i<window; i++)
```

```
{
    Dmax = (blkDepth[i] > Dmax) ? blkDepth[i] : Dmax;
    Dmin = (blkDepth[i] < Dmin) ? blkDepth[i] : Dmin;
}
if(abs(Dmax - Dmin) < 0.01)) /*search_step*/
{
    Filtered[index] = Dnow;
    continue;
}
Dmin = floor(Dmin-1);
Dmax = ceil(Dmax+1);

calculateWeights_zeropadded((UINT8*)Edges, weights,
wt_table, radius, x, y, width, height);

for(int i=0; i<window; i++)
{
    weights[i] *= dist_table[i];
}
normalizeWeights(weights, radius);

float Cost[255];
int hypoMax = (int)ceil((Dmax-Dmin));
float Cbest = MAXCost*5;
float Dbest = Dnow;
int Hbest = -1;

for(int hypo = 0; hypo <= hypoMax; hypo++)
{
    float d = (float)Dmin + hypo;
    float value = 0;
    for(int i=0; i<window; i++)
    {
        float diff = d-blkDepth[i];
```

```
            diff *= diff;
            diff = diff>search_limit ? search_limit : diff;
            value += diff*weights[i];
        }
        Cost[hypo] = value;

        if(value <= Cbest)
        {
            Cbest = value;
            Dbest = d;
            Hbest = hypo;
        }
    }

    float Cdown = (Hbest > 0) ? Cost[Hbest-1] : MAXCost;
    float Cup = (Hbest < hypoMax) ? Cost[Hbest+1] : MAXCost;

    Filtered[index] =
        Dbest - (Cup-Cdown)/(2*(Cup+Cdown-2*Cbest));

        return 0;
}
```

B. GPU SPECIFICATIONS



GPU Engine Specs:	
CUDA Cores	96
Graphics Clock (MHz)	550 MHz
Processor Clock (MHz)	1340 MHz

Memory Specs:	
Memory Clock (MHz)	1700 MHz GDDR5, 1000MHz GDDR3, 900MHz DDR3
Standard Memory Config	512 MB or 1 GB
Memory Interface Width	128-bit
Memory Bandwidth (GB/sec)	54.4 GB/sec

Feature Support:	
NVIDIA 3D Vision Ready	✓
NVIDIA PureVideo® Technology**	HD
NVIDIA PhysX™-ready	✓
NVIDIA CUDA™ Technology	✓
Microsoft DirectX	10.1
OpenGL	3.2
Bus Support	PCI-E 2.0
Certified for Windows 7	✓

Display Support:	
Maximum Digital Resolution	2560x1600
Maximum VGA Resolution	2048x1536
Standard Display Connectors	DVI VGA HDMI
Multi Monitor	✓
HDCP	✓
HDMI	✓
Audio Input for HDMI	Internal

Standard Graphics Card Dimensions:	
Height	4.376 inches (111 mm)
Length	6.6 inches (168mm)
Width	Single-slot

Thermal and Power Specs:	
Maximum GPU Temperature (in C)	105C C
Maximum Graphics Card Power (W)	69 W
Minimum Recommended System Power (W)	300 W

Figure B.1: NVIDIA GeForce 240 Engine Specification