GPU-Based Low-Level image processing for object recognition using HDR images

Master Thesis of

Jonatan Dominguez Tejera

At the Institute for Anthropomatics Humanoids and Intelligence Systems Laboratories

Advisor: Prof. Dr. Dillmann
Second advisor: Prof. Dr. T. Asfour
Reviewer: D. Gonzalez-Aguirre

Contents

1. Introduction 1
   1.1. Motivation .............................................. 1
   1.2. Approach .................................................. 2
   1.3. Overview .................................................. 5

2. State of the Art 7
   2.1. FPGA ..................................................... 7
   2.2. GPU in Real Time Applications............................. 8
   2.3. Human Detection ........................................... 8
   2.4. Point Detectors ............................................ 9
   2.5. Object Detection .......................................... 9

3. Fundamentals 11
   3.1. Density Estimation ....................................... 11
   3.1.1. Probability Density Function ......................... 11
   3.1.2. Weighting Function .................................... 12
   3.2. Synthesis from Low Dynamic Range Images .................. 14
   3.3. Convolution ................................................ 15
   3.3.1. Discrete Signals and Linear Systems Invariant ........... 15
   3.3.2. Frequency Domain Representation ....................... 17
   3.3.3. Gabor Filter ........................................... 18

4. Parallel Programming 19
   4.1. CPUs and GPU’s Evolution ................................ 19
   4.2. GPU Architecture ......................................... 20
   4.3. CUDA, Compute Unified Device Architecture ............... 21
   4.4. OpenCL as CUDA Alternative ............................... 22
   4.5. CUDA Model Programming ................................ 22
   4.6. Memory Structure .......................................... 28
   4.7. Programming Interface .................................... 30
   4.7.1. Functions Type Qualifiers ................................ 30
   4.7.2. Variable Type Qualifiers ................................ 31
   4.7.3. Built-in Vector Types ................................... 31
   4.7.4. Built-in Variable ....................................... 31

5. Implementation 33
   5.1. Noise Filtering ............................................ 33
   5.2. HDR Synthesize ............................................ 36
   5.3. Gabor Edge Extraction ..................................... 39
6. Evaluation

6.1. Hardware and Software ................................................. 45
6.2. Optimal Execution Configuration ...................................... 45
   6.2.1. Grids as Vector of Blocks ...................................... 46
   6.2.2. Grids as Matrix of Blocks ...................................... 47
6.3. Modules Evaluation .................................................... 51
   6.3.1. Fusion Module .......................................................... 51
   6.3.2. HDR Synthesize Module ........................................... 56
   6.3.3. RMS-Convergence by HDR Synthesis ............................ 57

7. Conclusion ......................................................................... 59
   7.1. Summary and Results ................................................... 59
   7.2. Future Work ............................................................... 60

Bibliography .......................................................................... 63

Appendix .................................................................................. 67
   A. Evaluation of Kernel Execution Configuration .................... 67
      A.1. Evaluation of Execution Time 1-Dimension Grid Configuration 67
      A.2. Evaluation of Execution Time 2-Dimension Grid Configuration 71
   B. Implementation Structure .................................................. 72
      B.1. Index Map on Parallel Programming Model ..................... 72
      B.2. Texture Memory of GPU .............................................. 74
      B.3. Structure of Implemented Code .................................... 77
1. Introduction

1.1. Motivation

The improvement of the life quality and the aim of getting benefits that humans are not able to achieve -high strength, for example in construction, or precision in surgery- are two of the most important reasons why science developed humanoid robots capable of interacting with people. The application of a robot capable of moving, interacting with humans and helping them is undoubtedly one of the greatest technological feats of the century. Following this purpose, research focuses on developing humanoids which are able to work in a “made-for-humans” environment\textsuperscript{1}.

The physical structure of humanoids mimics the human body. This fact allows their easy adaptation to these environments. It also enables an accessible transfer of skills from the humans by means or imitation learning of programming by demonstration [BCDS08]. Consequently, their design imposes constraints on the components and the sensors used in its development.

The robot interacts with the environment. Therefore, the robot should be able to extract the relevant information from the scene to give an adequate answer to orders or stimuli. In this context, several kinds of sensors, cameras and microphones allow the robot to sense, see and hear. Nevertheless, these devices are not ideal. Instability in sensors is caused by both external and internal noise sources, such as variations on scene illumination or other actuators conditions.

Focusing on image acquisition and trying to reduce the errors caused by these instabilities, the algorithms must be designed in order to solve these problems and allow the humanoid robot to work with the clearest and best information obtained from the environment.

This work, using low-level image processing, is focused on obtain an edge-image from the scene.\textsuperscript{1,3} In a real “made-for-humans” scenario, for example a kitchen, the robot is desired to open the dishwasher, the fridge or a cupboard, and move objects from

\textsuperscript{1}Made-for-humans environment is the environment created by man for the purpose of fulfilling his needs in order to make his life more convenient and easy
one place to another. Therefore, the humanoid has to be able to recognize the edges of the objects in order to identify them.

1.2. Approach

In [AWD08] and in [WAD09] both composition and applications of the ARMAR-III are respectively described. In these cases, the desired human appearance of the robot involves restrictions on its design. Due to space limitations, several devices must be placed in the head of a humanoid robot. Their simultaneous operations produce electric, magnetic and thermal perturbations which deteriorate the quality of the sensor signals, as explained in [DD11]. Space limitations and reduced power supply also restrict the size and weight of the cameras.

The noise added to the captured images is caused by external and internal sources. It is filtered using methods that provide robust and general solutions. By using these methods, the time required for the humanoid to respond to a given order is of the order of teens of seconds. The robustness of these methods makes them suitable for laboratory setups but because of their processing time, they are not suitable to be implemented in real-time applications.

The features of the current methods are analyzed in order to find faster alternatives. This analysis shows that the instructions implemented are highly parallelizable at pixel level. In other words, the process consists on the execution of several instructions on a large amount of input data. Concerning the mentioned feature the parallel programing is seen as a suitable alternative in order to reduce the time required in these methods. This programming model is focused on algorithms based on Simple Instruction Multiple Data (SIMD). It helps to reduce the time required through the execution of the same instruction over each input data simultaneously.

A new programming model implies a new programming paradigm. The C language allows writing structured programing and C++ allows writing object oriented programs. To achieve it, the programming paradigm behind each language has to be previously understood. Parallel programming has its own paradigm with specific guidelines, such as modular structure, strong cohesion, high profitability, etc. It would be an error to think of developing a parallel programing-based program as a reusing straightforward from any C or C++ program. The architecture of the Graphic Processing Units (GPUs) makes them more effective than Central Processing Unit (CPU) to process large blocks of data in parallel, see section 4.2. The instructions are executed using parallel threads run on GPU and input data must be previously allocated in device memory. Besides, specific characteristics such as organization of data, memory accesses, both CPU and GPU, programing flow, etc. must also be taken into account.

This thesis is focused on the analysis and application of the parallel programming model at early steps of the images acquisition process and edge extraction. It can be divided into three modules, each one solves a different problem with the aim of obtaining “quasi-noiseless” and high quality images in order to extract the relevant

---

2The instructions on the image pixels are not executed exactly at the same time. Threads within a warp are executed physically in parallel. However, warps and blocks are executed logically in parallel, see section 4.5.
information from the scene. Later, this information will be used in the following steps for object recognition.

The first module filters the noise from the captured images. The straightforward filters estimate the "noiseless" image from a single noise-contaminated image. However, these filters present severe artifacts when the noise affects an area which comprises two or more pixels. This problem is avoided using image fusion methods based on exploiting the available information from two or more images. The noise filter implemented in this work is based on image fusion methods. The content of the scene have to remain static during the sampling period. Firstly, several images are captured from the scene. Finally, the intensity value for each pixel is based on statistic estimators from the information of images. This method is suitable for humanoids as they are able to capture a large amount of images per second.

Humanoid robots should also be able to visually explore and recognize objects in their environment under poor lighting conditions. These conditions can be caused by high-contrast and dynamically varying illumination. For example, the interior of an oven or cupboard or on a reflective surface. Lightin becomes a limitation for extracting the relevant information from the environment. Both the flexibility and applicability of humanoids are strongly reduced due to these limitations. Therefore, images must be analyzed and processed in order to reduce lighting issues.

Traditional imaging methods are constrained by limitations in technology and do not use physical values from the scene. These methods obtain Low Dynamic Range (LDR) images. These images present very dark and bright areas of the scene named under-exposed and over-exposed areas, see fig. 1.1.

Figure 1.1.: Example of three LDR captured with different exposure time. It can be appreciated how the regions a), b) and c) are under or over-exposed according to this exposure time. High radiance areas, such as a), have to be sampled by the short-exposed images, unlike low radiance areas, such as c), which must be sampled by long-exposed images.

In the last decades the High Dynamic Range (HDR) images were introduced. Based on these images, the physically correct light values from the scene can be captured. Consequently, under-exposed and over-exposed area can be avoided. However, HDR cameras are not widespread especially in applications with several tight constraints, such as high frame rate, light weight, reduced space or low power consumption. Because of these restrictions, an appropriate mechanism for capturing HDR images by means of LDR cameras has been introduced in [DM97]. An overview of this mechanism can be seen in fig. 1.2-HDR Synthesis. The mechanism of the HDR Synthesize explained in [GAAD10] consists of:
Firstly, a radiometric calibration which consist on a nonlinear transformation from the scene radiance to the discrete values of the images is computed based on [DM97]. Secondly, a collection of images, defined as Wycoff set in [MP94], are captured with different exposure time determined by the shutter speed. Finally, the HDR image is obtained by fusing this set of images according to the short-exposed images sample the high radiance regions of the scene, and the long-exposed images sample the low radiance regions.

Figure 1.2.: The modular pipeline of low level image processing [GAAD10]

As illustrated in fig. 1.2-Gabor Edge Detection, a set of $n$ Gabor kernels $\Psi$, each one oriented on $\alpha$ direction, computes the receptive saliency extraction convoluting this set of kernels with the HDR image. The different visual features which contributes to attentive selection are combined into one saliency map, as proposed in [KU85]. This map integrates the normalized information from the features of images into one global measure of visibility. Later, considering the saliency direction coherency $\mu$ and the saliency norm, the dual non-maximal suppression $\gamma$ selects the pixels located close to the edge in order to extract the edge-graphs.

Figure 1.3.: Example of low-level image processing. In this thesis the methods required to extract edge-image (right image) from a “made-for-human” scenario (left image) is implemented.
1.3. Overview

In this part of the introduction chapter, an overview of the following chapters is given. An outline of state-of-the-art is given in the second chapter. Chapter 2 introduces the parallel programming applied on robotic presenting several processes based on this concept. First, Field Programmable Gate Array (FPGA) is described due to its applications on digital signal processing. Later, a review of Graphics Processing Unit (GPU) and diverse applications on human detection and object recognition are introduced. Chapter 3 introduces the reader into the fundamentals on which this work is based on. The concepts presented are kernel density estimation (KDE), which is a statistic estimator of the probability density function of a random variable; an algorithm implemented for recovering the irradiance value of images captured by cameras, and a short description of convolution process on two-dimensional signals. In chapter 4, parallel programming concepts as the architecture of the GPUs, memory structure, paradigm and model programming of the Single Instruction Multiple Thread (SIMT), are explained in detail. The image fusion, HDR image synthesize and edge extraction modules implemented in order to extract edge-image are introduced in chapter 5. In chapter 6, experimental results are shown after a short overview of the applied hardware. The final chapter presents what has been achieved in a summarizing and evaluation the thesis and it finishes with future work.
2. State of the Art

Image processing applications are very suitable to be implemented by single instructions multiple data (SIMD) methods because the instructions implemented have to be executed over a huge number of input data. Besides, the execution time of applications implemented on human robots must be short in order to use them for real-time applications. For these reasons, the parallel programming model is used to develop applications in robot vision.

The target of these applications can be very diverse, from simple image filtering to object or human detection. Operations such as convolution or image filtering are commonly used to achieve these targets. In this chapter several algorithms, based on images features extraction using parallel programming methods, are introduced to provide an overview of the implementation of low-level image processing operations on GPU.

In several algorithms, almost all functions implemented are processed by the CPU and the parallel implementation on GPU is reserved to filter image by a simple mask. However, other algorithms are more complex, such as key point matching or object detection, and they are implemented completely on GPU, avoiding CPU data transfers or calculations.

2.1. FPGA

Before developing robot vision applications, the adequate architecture has to be chosen. This architecture must be suitable to implement operations on a huge number of input data simultaneously, commonly used for image processing applications. The Field-Programmable Gate Array (FPGA) is used to implement these operations. Recently, the Graphic Processing Unit (GPU) has become an alternative. Both architectures are difficult to compare due to the constant changes in technology. The characteristics of them are improved for each new version. However, operations such as the memory access and the data computation can be analyzed to obtain a general overview of their performance.

From the evaluation in [PTD+11], it is concluded that GPU is an adequate architecture to develop the edge extraction process of this work. The required time to develop
2. State of the Art

Low-level image processing operations such as convolutions is larger for FPGA than GPU, because of GPU uses an interface similar to C language that simplifies the implementation of many operations. However, FPGA is an embedded platform that allows better performance for bitwise operations. Moreover, the extended libraries developed for GPU based on C language allow implementing easily instructions such as data transfers between CPU and GPU memories. The multi-frame applications achieve better performance implemented on GPU than FPGA because of the texture memory unit of the GPU which improve the irregular memory access. However, the bandwidth of this memory causes that the operations which require short latency response are suitable to be implemented on FPGA.

2.2. GPU in Real Time Applications

In robotics, processing sensory data in real-time is one of the most important features in order to react to the changes in the environment. Many image processing algorithms require considerable execution time implemented on CPU. However, they can be considered suitable for real-time applications implemented on GPU. Filtering images by neighborhood filters, such as Sobel [ZCW10], morphologic operators or processing images with Look-Up-Tables belong to this group of algorithms. In [KP10] the execution times are measured on both CPU and GPU. The results of these evaluations show that, depending on the mask size, the processing time on GPU is around 5 times smaller than the corresponding processing on CPU.

2.3. Human Detection

Robot vision systems are usually implemented for object or human detection and recognition. However, most of the low-level algorithms are similar for these applications. In this section, human detection on robot vision is presented to analyze the algorithms and to compare them with the algorithms implemented for object recognition applications, both implemented on GPU.

Human detection by shape-based and motion-based [BSR11] implement several algorithms using CUDA. In shape-based detection, the parallelized processes are based on filtering and computing integrals on the images. Also, updating the pixels value to model the background image and computing the foreground image are the algorithms implied on motion detection.

The human detector method HOG-LBP is based on the combination of the Histogram of Oriented Gradients (HOG) and Local Binary Pattern (LBP). HOG is presented as a human detection algorithm with excellent results. LBP is a simple method included in applications of texture classification and segmentation such as face recognition. The HOG method convolutes two kernel masks with the input image in order to obtain the magnitude and orientation of gradient. For color images, the gradient is calculate separately for each RGB channel. The LBP texture operator computes the value for every pixel of input image from the neighboring pixels and binarizes the result. Because of implementing this human detector method on GPU, [FMS11], 10x speed up is obtained compared to CPU implementation.

Despite both presented applications are developed for human detection, many low-level algorithms presented are common to those of this work, for example filtering,
2.4. Point Detectors

Algorithms implemented to detect key points respect variations in the size, rotation and translation of the image are also suitable to be implemented using parallel programming model.

In [KPC+09], a process for fast feature extraction is implemented using OpenMP and SSE for CPU parallel implementation, and CUDA model for GPU. The low-level algorithms implemented are the multi-scaled convolution by Gaussian kernel mask and the second-order derivative of the input image. The computational time is compared to the proposed implementations and different existing methods. As a conclusion, the GPU-parallel method is shown faster than other implementations including CPU-parallel methods.

Another method implemented on GPU which takes advantage of the parallel programming for feature extraction is the scale invariant point detector Harris-Hessian (H-H) method presented in [XGZ+10]. In edge extraction process from input images, false local maxima are detected in the neighborhood of edges for Harris detector then, the determinant of Hessian matrix is used to detect them. The GPU implementation of the convolutions and differential operations achieves a 10-20x speedup compared to CPU.

A framework for robot vision to find coincidences between to images implemented completely in GPU is presented in [SW09]. It is divided into feature extraction, comparing of feature with feature descriptions and feature matching. All of them implemented on GPU without CPU calculations or memory transfers. The features extraction is based on corner-based detector because of corners are much more stable against scale change. In order to recognize keypoints, descriptors are used to distinguish detected keypoints and determine correspondences. Finally, the matching is done inspired by SURF prematching which implies no scale search or rotational invariance. The results show that whole computation time for GPU implementation below 3 ms for image size of 752x480 pixels.

Despite of the huge different execution time required between CPU and parallel models on GPU, the configuration parameters for parallel execution are not optimize for these methods. Unlike these methods, in this work these configuration parameters are analyzed to optimize the execution time of the parallel executions. Besides, in this work the feature extraction is also computed using either one image or a set of images in order to obtain robust features extraction algorithms against the noise, as opposed to the presented methods.

2.5. Object Detection

Object detection is another important task of robot vision. These detectors are characterized by extracting features from objects using large number of images, and searching these features in input given image. In [CBLN09] this process is implemented by four different objects: coffee mugs, paper coffee cups, office staplers and
bananas. First, a dictionary with hundreds of patches for object is constructed from small image fragments from the objects, these patches are obtained from 150 input images which contain between one and four instance of an object. These patches are defined by three variables, patch identification number, a rectangle specifying its approximate location relative to the object center, and the image channel from which it was extracted. The patches from the dictionary are cross-correlated with input image given for the corresponding image channel and taking the maximum response over the patch rectangle.

Shape-Based matching (SBM) detection method for partial occluded objects is presented in [Pet07]. These objects can be translated, rotated and scaled in order to be recognized. The edges of the object to be recognized are extracted from an image in a clear environment, controlled illumination and without occlusion or clutter. This method also uses the cross-correlation to locate the object in the input image. This process is executed through pyramidal resolution. The searching is done on the coarsest level of the pyramid and the candidates extracted are tracked down the levels of the pyramid until it is found in the original search image.

These two methods exploit at the maximum the parallel programming implementation on GPU due to the huge amount of data, such as dictionary of patches or amount of multi-resolution images, which have to be cross-correlated with the features extracted previously. Moreover, this correlation must be calculated also for several numbers of scales and rotations. These processes are complex and a huge number of instructions are executed on a large amount of data. Because of that, they are even more parallelizable and allow to obtain approximately 100 times faster running on GPU than executed on CPU.
3. Fundamentals

The fundamentals of the algorithms developed in this work are covered in this chapter. First, an overview of the density estimation is presented. This estimator is implemented in order to obtain the most probable pixel values in the image by the fusion method. Second, the process to synthesize HDR image from LDR images implemented in [DM97] is shown. In section 5.2 an improvement extension of this process is developed. Finally, the fundamentals of the convolution and the Fourier Transform (FT) are described in order to introduce the Gabor filter. This filter is suitable to be implemented for edge extraction because of its multiple properties.

3.1. Density Estimation

3.1.1. Probability Density Function

The probability density function (pdf) $f(x)$ defines the probability that a random variable $X$ takes a value in a given interval. This probability can be defined as:

$$P(a \leq X \leq b) = \int_a^b f(x) dx$$  \hspace{1cm} (3.1)

The range of the random variable $X$ is the set of values that $X$ can assume. In case of all the values of this range are not known, the pdf of the random variable can be estimated from the known values of its range.

The pdf can be estimated using a parametric approach in order to obtain a stable estimations. However, this approach assumes that the data come from a type of probability distribution and restrictions on these data are imposed. If the assumption is incorrect the estimation can be inaccurate. As an example, the normal distribution assumes that the data are symmetric, see fig. 3.1 Therefore, skewed pdfs cannot be represented using this parametric distribution.
Normal Distributions

\[ \mu = 0.0, \sigma^2 = 0.4 \]
\[ \mu = 0.0, \sigma^2 = 1.0 \]
\[ \mu = 0.0, \sigma^2 = 4.0 \]
\[ \mu = -3.0, \sigma^2 = 0.6 \]

Figure 3.1.: Normal distribution is a parametric estimator. This estimator imposes restrictions such as symmetrical distribution.

The non-parametrical estimator of pdf without assuming anything about the data. The histogram is a straightforward estimator within this group. The probability distribution is represented directly from the data using rectangles which height is proportional to the frequency of each range value, see fig. 3.2. The range of the random variable is divided in sub-intervals, bins. Consequently, a histogram is a non-continuous representation of the pdf and depends on the width of bins.

Figure 3.2.: Example of a histogram from a gray image. The range of intensity values is divided in bins.

3.1.2. Weighting Function

The weighting functions allow estimating a continuous representation of pdf, \( \hat{f}(x) \), from a histogram removing the dependence on the bins. From equation 3.1 and defining a window which width is equal to \( 2h \),

\[
P(x - h < X < x + h) = \int_{x-h}^{x+h} f(t)dt \approx 2hf(x)
\] (3.2)
the pdf can be approximated by
\[ f(x) \approx \frac{1}{2h} P(x - h < X < x + h) \] (3.3)

Therefore, the pdf can be estimated by
\[ \hat{f}(x) = \frac{1}{2h} \frac{\text{number of samples} \in (x - h, x + h)}{\text{Total number of samples}} \] (3.4)

This estimation can be mathematically written as
\[ \hat{f}(x) = \frac{1}{n} \sum_{t=1}^{n} W(x - x_t, h) \] (3.5)

where \( n \) is the total number of samples, \( x_1, x_2, ..., x_n \) are the samples of the histogram and \( W(t, h) \) is the weighting function known defined as
\[ W(t, h) = K \left( \frac{t}{h} \right) \] (3.6)

where \( K(u) \) is a function named kernel defined as
\[ K \left( \frac{t}{h} \right) = \begin{cases} \frac{1}{2h} & \text{if } t < |h| \\ 0 & \text{if } t > |h| \end{cases} \] (3.7)

The parameter \( h \) defines the smoothing factor of \( \hat{f}(x) \). This parameter is also named bandwidth or smoothing constant. The kernel determines the appearance of the weighting function. The functions that satisfy the conditions of equation 3.8 can be defined as kernels. The tab. 3.1 shows smoothing kernels commonly used to kernel density estimation.

\[ \int K(u)du = 1, \int u \cdot K(u)du = 0, \int u^2 \cdot K(u)du < \infty \] (3.8)
### 3. Fundamentals

<table>
<thead>
<tr>
<th>Kernel Type</th>
<th>Kernel K(u)</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>$K(u) = \frac{1}{2} 1_{{</td>
<td>u</td>
</tr>
<tr>
<td>Triangular</td>
<td>$K(u) = (1 -</td>
<td>u</td>
</tr>
<tr>
<td>Epanechnikov</td>
<td>$K(u) = \frac{3}{4} (1 -</td>
<td>u</td>
</tr>
<tr>
<td>Gaussian</td>
<td>$K(u) = \frac{1}{\sqrt{2\pi}} \exp -\frac{1}{2} u^2$</td>
<td></td>
</tr>
<tr>
<td>Biweight</td>
<td>$K(u) = \frac{15}{16} (1 - u^2)^2 1_{{</td>
<td>u</td>
</tr>
</tbody>
</table>

Table 3.1.: Example of smoothing kernels used for kernel density estimation

### 3.2. Synthesis from Low Dynamic Range Images

From a real scenario, the difference between the brightest and darkest points is defined as dynamic range of the scene. Usually, this range is larger than the dynamic range that can be represented in a LDR images. Therefore, according to the exposure time of the camera, these images presents under- or over-exposed areas, see fig. 1.1.

The HDR synthesis presented in [DM97] consist on an algorithm that allows recovering the irradiance values from the scene in order to assign the pixel values in the image.

The exposure $X$ is defined as the product of the irradiance $E$ and the exposure time $\Delta t$. During the process of capturing image, the pixel value $Z$ in the image is obtained from a nonlinear mapping, $f$, of the exposure $X$. The pixel value can be expressed as,

$$Z_{ij} = f(E_i \Delta t_j)$$  \hspace{1cm} (3.9)

where $i$ is a spatial index over pixel and $j$ indexes over exposure times $\Delta t_j$. Because of the mapping $f$ and irradiance $E$ are unknown, the pixel value cannot be defined directly for a given exposure time. The function $f$ is assumed monotonic.
3.3. Convolution

Consequently, it is invertible:

$$f^{-1}(Z_{ij}) = E_i \Delta t_j$$  \hspace{1cm} (3.10)

Taking the logarithm of both sides:

$$\ln f^{-1}(Z_{ij}) = \ln E_i + \ln \Delta t_j$$  \hspace{1cm} (3.11)

Defining function $g = \ln f^{-1}$ the previous equation can be written as:

$$g(Z_{ij}) = \ln E_i + \ln \Delta t_j$$  \hspace{1cm} (3.12)

The algorithm required to recover the function $g$ is explained in [DM97]. By using this function, the irradiance $E$ can be obtained as:

$$\ln E_i = g(Z_{ij}) - \ln \Delta t_j.$$  \hspace{1cm} (3.13)

From the equation 3.13 the dynamic range of the scene can be completely defined from a set of images captured by different exposure times. In short, it can be observed that the function $g$ allows the synthesize of HDR images from the irradiance values of the scene, which avoids the over- and under-exposition areas in the image. However, the function $g$ presents several problems for large and low values of $\Delta t$. In section 5.2 of this work, an improvement extension of this function $g$ is presented.

### 3.3. Convolution

#### 3.3.1. Discrete Signals and Linear Systems Invariant

A discrete system is defined as operator $T\{ \}$ that modify the input discrete sequence $(x[n])$ into an output sequence $(y[n])$. Given the unit sample sequence

$$\delta[n] = \begin{cases} 0, & n \neq 0, \\ 1, & n = 0 \end{cases}$$  \hspace{1cm} (3.14)

any input sequence $x[n]$ can be expressed as a lineal combination of displaced $\delta[n],$

$$x[n] = \sum_{k=-\infty}^{\infty} x[k] \delta[n - k]$$  \hspace{1cm} (3.15)

Therefore, the discrete system can be expressed as

$$y[n] = T\{x[n]\} = T\left\{ \sum_{k=-\infty}^{\infty} x[k] \delta[n - k] \right\}.$$  \hspace{1cm} (3.16)

A system which is both linear and time invariant is defined as LTI. Using the properties of this kind of systems, it is possible to know the response of the system for every input signal. Firstly, the linear properties allow writing it as
where \( h_k[n] \) is defined as response to the displaced impulse \( \delta[n - k] \). Secondly, using the time invariant property, the last equation can be expressed as

\[
y[n] = \sum_{k=-\infty}^{\infty} x[k] T\{\delta[n - k]\} = \sum_{k=-\infty}^{\infty} x[k] h_k[n] = x[n] \ast h[n]
\] (3.18)

where \( h[n] \) corresponds to the output system when \( \delta[n] \) is the input sequence. LTI system is completely characterized by the impulse response \( h[n] \). Therefore, for any sequence \( x[n] \) and \( h[n] \) it is possible to compute each sample of the output sequence \( y[n] \) by using the operator \( \ast \) known as convolution. A graphic illustration of convolution is shown below.

The two-dimensional LTI systems can be analyzed from the one-dimensional case. The two-dimensional discrete convolution is a common operation in the image processing, for example to filter images. In the convolution, the value of one output pixel is calculated using the weighted sum of neighboring pixels, see fig. 3.4. The convolution is computed between the image and a mask. This mask is defined by a matrix (coefficients of the filter) to filter the image. The convolution between \( x[m, n] \) and \( h[m, n] \), where \( x[m, n] \) represents an image and \( h[m, n] \) is a convolution mask of MxM coefficients, is represented as

\[
y[m, n] = x[m, n] \ast h[m, n] = \sum_{j=-M_0}^{M_0} \sum_{k=-M_0}^{M_0} x[m + j, n + k] h[j + M_0, k + M_0]
\] (3.19)

where \( M_0 \) is defined to center the index of the mask as

\[
M_0 = \frac{M - 1}{2}
\] (3.20)

The filtered effect in the image is according to the coefficients of the convolution mask and its size, see tab. 3.2.
3.3. Convolution

Figure 3.4.: The value of the output pixel is computed using neighboring pixels according to the size of the convolution mask.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Mask</th>
</tr>
</thead>
</table>
| Average         | \[
\begin{pmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{pmatrix}
\] \[\frac{1}{9}\] |
| Weighted average| \[
\begin{pmatrix}
1 & 1 & 1 \\
1 & 2 & 1 \\
1 & 1 & 1 \\
\end{pmatrix}
\] \[\frac{1}{10}\] |
| Laplacian       | \[
\begin{pmatrix}
0 & 1 & 0 \\
1 & -4 & 1 \\
0 & 1 & 0 \\
\end{pmatrix}
\] |
| Horizontal Sobel| \[
\begin{pmatrix}
1 & 2 & 1 \\
0 & 0 & 0 \\
-1 & -2 & -1 \\
\end{pmatrix}
\] |
| Vertical Sobel  | \[
\begin{pmatrix}
-1 & 0 & 1 \\
-2 & 0 & 2 \\
-1 & 0 & 1 \\
\end{pmatrix}
\] |

Table 3.2.: Example of different kind of kernels and its coefficients in a 3x3 mask

3.3.2. Frequency Domain Representation

For image processing applications, it is interesting to represent a time signal in the frequency domain. In this case, the signal is represented as sum of periodic signals with different frequency. An important representation in the frequency domain is the Fourier Transform (FT). This transform represents an image as a sum of complex exponentials with different magnitude, phase and frequency. The FT is used for several applications, such as convolutions, enhancing image, features extraction or compression.

For one-dimensional case, the Fourier transform is defined as

\[
F[x(n)] = X(k) = \sum_{k=0}^{N-1} x(n) \exp\frac{j2\pi kn}{N} ; \quad k = 0, 1, \ldots, N - 1
\] (3.21)

The two-dimensional case is defined as
\[ \begin{align*}
F[x(k, l)] &= X(k, l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x(m, n) \exp \left[-\frac{2\pi}{M} km \right] \exp \left[-\frac{2\pi}{N} ln \right] \\
&= \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x(m, n) \exp \left[-\frac{2\pi}{M} km \right] \exp \left[-\frac{2\pi}{N} ln \right] \\
k &= 0, 1, \ldots, M - 1; \\
l &= 0, 1, \ldots, N - 1
\end{align*} \]

The values of \( X(k, l) \) are the coefficients of the FT from the image \( x(m, n) \). From the FT, fast convolution can be defined. This process is based on that convoluting two signals in the space domain is equal to the multiplication of the FT of these signals.

### 3.3.3. Gabor Filter

The Gabor filter is especially important for image processing applications because of its characteristics approximate certain cells in the visual cortex of the mammals, [Dau85]. This filter is used in many applications, such as texture segmentation, target detection or edge detection. It can be represented as a sinusoidal signal modulated by a Gaussian envelope. It can be written as

\[ h(x, y) = s(x, y)g(x, y) \]  
\[ s(x, y) = \exp \left[-j2\pi(u_0x + v_0y) \right] \]  
\[ g(x, y) = \frac{1}{\sqrt{2\pi\sigma}} \exp \left[-\frac{1}{2} \frac{x^2}{\sigma_x^2} + \frac{y^2}{\sigma_y^2} \right] \]

the Fourier Transform FT of this filter is obtained as,

\[ H(u, v) = G(u - u_0, v - v_0) = 2\pi\sigma_x\sigma_y \exp \left[-2\pi^2((u-u_0)^2\sigma_x^2 + (v-v_0)^2\sigma_y^2) \right] \]

where,

\[ \sigma_u = \frac{1}{2\pi\sigma_x}; \quad \sigma_v = \frac{1}{2\pi\sigma_y} \]

The equation 3.26 shows that these filters have optimal localization properties in both spatial and frequency domain. The frequency response of the filter is equal to translating the Gaussian function in the frequency domain.
4. Parallel Programming

4.1. CPU’s and GPU’s Evolution

The Central Processing Units (CPU) improved their performance in the last decades as mentioned in Moore’s Law [Moo65]. These improvements affected processors which execute a single thread simultaneously. The applications implemented on these processors exploited the computational power of the CPUs. Therefore, the improvement in the applications was achieved when the new version of CPU was developed.

However, problems to dissipate the high temperature generated by integrated circuits working at very high clock frequencies, the gap between the computational speed of a CPU and memory access time, and the difficulty to improve the number of instructions executed per cycle affected the evolution of the CPUs. For these reasons it was necessary to introduce the Chip-Multi-Processors (CMPs), commonly known as multicore processors, in order to improve the applications implemented on CPUs.

Otherwise, the Graphics Processing Units (GPUs) were focused in the world of graphics, its architecture consisted mainly of two types of functional units replicated: the pixel shaders and the vertex shaders. Initially these units were designed for specific function; therefore, they are not suitable to be used for other applications. In recent years, a change in the design of these devices has been introduced in order to use the GPUs for any application. Unlike with CPUs, advances in transistor technology allowed improving the features of the GPUs, see fig. 4.1 because these advances were used to increase the number of functional units replicated on each device.

The GPUs are specialized in parallel computation. In their design, more transistors are involved in data processing rather than the storage control and data flow. An illustration of the difference in the number of transistors dedicated to provide computer units (green) in both technologies is shown in fig. 4.2. Despite this, these

---

1In the field of computer graphics, a shader is a programmable function in display adapters that offers a graphics application programmer flexibility in rendering an image
devices have to be considered as accelerator complements of CPUs, because the cores of GPUs are more simple and limited than the CPUs ones. This new kind of architecture leads to develop applications based on parallel programing model that offer an alternative to the limitations encountered on the CPUs.

4.2. GPU Architecture

A GPU could be typecast in the group of vector processors. These processors are able to execute the same operation on multiple data (vector data) simultaneously. This process allows reducing the required time to execute these operations.

A general diagram of the architecture of a GPU is shown in fig. 4.3. GPU contains processing units, called Streaming Multiprocessors or SMs, the amount of SMs depends on the type of graphic card. Each SM is formed by eight simpler cores or processors, SP, which are capable of executing instructions on different data simultaneously. Also, each SM contains a per-block shared memory (PBSM). In addition,
several SMs are grouped in Thread Processing Clusters or TCPs, where the first level texture cache (L1 Cache) is located in order to share memory by different multiprocessors. A second level of this type of cache memory, L2 Cache, shares memory by all the TCPs.

![Diagram of the GPU Architecture](image)

The SMs have two instruction units (IU) in order to perform operations on data of integer or floating point single precision. In case of floating point operations in double precision, only a single instruction unit is shared. Unlike CPU, GPUs have no branch prediction mechanism, the execution takes place in order.

4.3. CUDA, Compute Unified Device Architecture

Initially, the GPUs had been developed for 3D visualization process. The programmable elements and tools designed to use them were strongly oriented for graphics applications and video games. The shaders had restrictions such as fixed amount of input and output data with a limited in length, the number of variables was limited, access to textures have to follow certain patterns, etc. They were specializing on multiply vectors by matrices of floating point numbers. However, these operations are also used on scientific fields.

The GPGPU (General-Purpose Computing on Graphics Processing Units) appeared for the motivation to use the computing power of graphics cards for any area. Because of the increasing popularity of GPGPU and the use of shaders for other purposes, CUDA (Compute Unified Device Architecture) was developed by NVIDIA. CUDA is a parallel computing architecture that provides an application programming interface, API, to use the parallel processors of the GPU.

The hardware of the GPUs became generalized. This design was called Unified shader model and was not only adopted by NVIDIA, but it was the first to have a

---

2Without branch prediction mechanism the thread execution is serialized for conditions such as if-else, see fig. 4.10
compatible product on the market, other companies have adopted this architecture in several products such as AMD / ATI, Intel or S3.

NVIDIA developed an extension to the C language that allows the execution of programs written in this language on its hardware. Also, this hardware has less restrictions than the shaders, this means for example, that the GPU memory can be accessed easier and the input and output data are not fixed.

### 4.4. OpenCL as CUDA Alternative

OpenCL (Open Computing Language) was initially an Apple product. After June 2008, when Khronos Group was created, it became a standard cross-platform for general-purpose computing on graphics processing units (GPGPU). Companies such as AMD/ATI, NVIDIA and Intel cooperate in order to develop this standard.

OpenCL is a low-level API similar to CUDA. This language can be used to develop applications which run on CUDA architecture. However, the low- and high-level API from CUDA are optimized for NVIDIA graphic card, which is used to develop this thesis. In addition, CUDA is widespread language. It has extensive documentation, analysis and debug tools on both Windows and Linux, active forums and huge amount of application are developed by the universities based on its architecture.

Although the possible benefits and advantages of the OpenCL, it has not yet the level of maturity and popularity of CUDA. OpenCL has a very promising future, being a completely open standard and being supported for many processors.

### 4.5. CUDA Model Programming

An application based on CUDA language can be divided into two contexts host and device, see fig. 4.4. The first one includes the code executed on CPU using the serial host thread. The second one includes the code executed on GPU using multiple parallel threads, this code is also named kernel and is executed by each thread of the device.

The threads are grouped into blocks, see fig. 4.5. a block represents the smallest unit of allocation between these threads and multiprocessor device. There is a trade-off between the number of threads and blocks. Consequently, if the number of threads per block is increased, the number of blocks per multiprocessor running in parallel is decreased; and if the number of threads per block decrease, just the opposite.

In the same way, blocks are grouped into a structure called grid. This structure represents the execution configuration for a kernel. A kernel can be called through different configurations of threads and blocks without requiring any modification of the code.
Figure 4.4.: Structure of CUDA program execution. Serial code is executed on the CPU, while parallel code is executed on the GPU. Extracted from [CUDA]
CUDA is also a scalable programming model. The execution configuration must be defined previously at the kernel execution. Then, the blocks are put together into a grid. However, the device is the one that schedules the execution of each block. Therefore, thread blocks can run both concurrently or sequentially. As illustrated in fig. 4.6, the blocks of threads are executed independently from each other. This independence allows scalability; this means that, a kernel execution is scaled for any number of cores.
As illustrated in fig. 4.7, the host determines when the device starts to run each of their kernels. Otherwise, the GPU memory is independent of CPU memory. Therefore, the GPU needs explicit data transfers between CPU and GPU memory spaces. Also, the host is involved in allocating and freeing memory space on the device and performing data transfers between the two areas.

Figure 4.7.: Steps at CUDA’s execution model

From the illustration 4.7, it can be observed that the threads and blocks can be disposed into multiple dimensions. The built-in variables from the API of CUDA allow mapping the threads in a multidimensional grid.

Threads grouped into a block can be organized in three dimensions, which can be indexed locally for each block using threadIdx.\([x, y, z]\) variables. In addition, blocks can be organized in two dimensions into a grid and indexed using the variables blockIdx.\([x, y]\). Furthermore, the variables gridDim.\([x, y]\) and blockDim.\([x, y, z]\) describe the size of a grid and a block according to the number of blocks and threads respectively. All these variables are predetermined and accessible from any kernel. These dimensions allow mapping the multidimensional structures threads more easily. Figure 4.8 shows a declaration of a kernel. This example code adds two vectors and writes the result value in a third.
__global__ void my_kernel (int* a, int* b, int* c) {
    int LocalId = threadIdx.x;
    int GlobalId = blockIdx.x*blockDim.x + LocalId;
    c[GlobalId] = a[GlobalId] + b[GlobalId];
}

Figure 4.8.: Example of thread index in a kernel

Each thread is defined by the global identifier GlobalId, which is used to access the position of the vectors that must be computed. This identifier is calculated as the global displacement of the thread with respect to other blocks, plus the local identifier of the thread within the block which is LocalId or threadIdx.x

Figure 4.9 describes the instructions required to execute a kernel. In this example, the size of vectors is equal to 1024, and the execution configuration defines 64 threads per block and a grid of 16 blocks. A, B and C represent vectors, where the suffixes _h and _d indicate whether the pointer refers to the structure found in host memory or in device memory, respectively. Although the instructions of this example can be deleted or be in a different order, depending of the program.

In order to setup a kernel, CUDA incorporates execution configuration syntax. The grid and block dimensions are set between the <<<...>>> operators before the arguments of kernel, which are declared as in a C function.

It is important to know that the invocation of a kernel is asynchronous; this means that the execution control is returned to the host before the kernel code has finished. To make sure that the host waits until the execution of the kernel is finish, cudaThreadSynchronize() function is recommended to use. This function acts as barrier until all threads have finished their kernel work. The use of this function can be omitted if a transfer of memory is make, because the cudaMemcpy() is a blocking function, waiting until the kernel has finished.

At low level, the threads are grouped into warps. A warp is nothing but a group of 32 threads, which running in parallel. All threads of a warp run at the same time and following the same instructions of code, but they can take different branch. Actually, it runs first a half-warp and then the other half, this is an important factor to achieve the efficient access to memory or to reduce the divergent threads (running different code than the rest of half-warp). The divergence of threads must to be avoided because of serialization, see fig. 4.10. When a group of threads take a different path from the rest of the half-warp, all threads that do not follow this path are blocked and they wait until the first group of threads finishes the execution.
// Variables Declaration
int *A_h, *B_h, *C_h // Host pointers
int *A_d, *B_d, *C_d // Device pointers

// Variables Initialization
...

// Device Memory Allocation
cudaMalloc((void**)&A_d, N*sizeof(int));
cudaMalloc((void**)&B_d, N*sizeof(int));
cudaMalloc((void**)&C_d, N*sizeof(int));

// Host to Device Transferences
cudaMemcpy(A_d, A_h, N*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, N*sizeof(int), cudaMemcpyHostToDevice);

// Kernel Configuration
dim3 block(64,1,1);
dim3 grid(1024/64,1,1);

// Kernel Launching (GPU Execution)
my_kernel<<<grid, block>>>(A_d, B_d, C_d);

// Device To Host Transferences
cudaMemcpy(C_d, C_h, N*sizeof(int), cudaMemcpyDeviceToHost);

// Host Memory Deallocation
free(A_h);
free(B_h);
free(C_h);

// Device Memory Deallocation
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);

Figure 4.9.: Example of kernel invocation
Figure 4.10.: Example of serial execution of the jumps for a condition if-else. Few threads of the warp execute the instructions of if condition meanwhile, the rest of the threads must wait until they finish. After that, the instructions of else condition are executed for the rest of threads. Finally, all threads of warp converge, at this point the code can be executed for all the threads simultaneously again. extracted from [Rei10].

4.6. Memory Structure

Figure 4.11 shows the six kinds of memories available on GPU and their scope of visibility. A description of these memories is presented below:

Figure 4.11.: In GPU there are six kinds of memories divided according to its location and scope of visibility. Extracted from [Zel08]
• **Global memory** is the memory used to transfer data between host and device. It is the largest memory and it is outside the chip. The lifetime of data is equal to the life of the program, and these are visible to any thread of any block that is running on the device. The threads access to it is in groups of half-warps. For best performance, access must be coalesced. A coalesced access is defined for a half-warf if:
  
  – Each thread accesses to a memory element which size is 4, 8 or 16 bytes.
  
  – The address of the first element to be accessed is aligned to 16 times the size of the memory element.
  
  – Consecutive threads access to consecutive memory elements.

• **Local memory** is contained into global memory. It can be considered as a global memory. However, the visibility and lifetime of the data is limited at thread-level. This memory area is used by the compiler and the runtime when the register memory is full or the structure data is not suitable to be stored as registers.

• **Registers** are the fastest storage resource of GPUs. It is located in the processing chip. The visibility and life-time is limited at thread-level.

• **Shared memory** is located in chip and, unlike the local registers and memory, visibility is at the block level and the lifetime of the data is at the kernel level. It is one of the most important memories of the device because of all threads contained in the same block can share information with a latency equal to the registers. It is a very limited memory, and the transfers between global and shared memory must be managed manually. The memory is accessed in groups of half-warps.

• **Constant memory** provides a cache level above the global memory. This is a read only memory at GPU level. Hence, data must be transferred from the host before proceeding with the execution of a kernel to access these items. The visibility and lifetime are similar to those of the global memory. A reading over the constant memory is equal to global memory access device in case of cache miss, and faster as access to a register in case of success. This optimal access is achieved when all the threads of the same half-warp read the same memory location. If threads access to different addresses, the memory accesses are serialized, and access cost is increased linearly as a function of number of different memory locations.

• **Texture memory** was used initially to read image textures in graphics applications. Like constant memory, it provides a cache level above global memory, but with different access restrictions. In addition, this cache is optimized for data access in two dimensions. As a constant memory, if data are not cached, the access is to global memory.

The Table 4.1 shows a summary of the device memories described, where X indicates memory cached only on devices of compute capability:\[3\]2.x:

---

3 The compute Capability describes the features supported by a CUDA hardware.
### 4. Parallel Programming

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location on/off chip</th>
<th>Cached</th>
<th>GPU Access</th>
<th>Scope</th>
<th>Life-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>On</td>
<td>n/a</td>
<td>R/W</td>
<td>1 thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off</td>
<td>X</td>
<td>R/W</td>
<td>1 thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On</td>
<td>n/a</td>
<td>R/W</td>
<td>All threads in block</td>
<td>Block</td>
</tr>
<tr>
<td>Global</td>
<td>Off</td>
<td>X</td>
<td>R/W</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
<tr>
<td>Constant</td>
<td>Off</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
<tr>
<td>Texture</td>
<td>Off</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
</tbody>
</table>

*Table 4.1.: Summary of the GPU Memories*

### 4.7. Programming Interface

The firmware of the computer architecture of NVIDIA GPUs is different from previous GPGPU models. It allows creating GPU programs using the standard syntax of C language with and minimal extensions from the CUDA runtime library. The runtime provides C functions executed on host in order to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple device, etc.

Kernels must be compiled into binary code by `nvcc` compiler. Full C++ is supported for the host code. However, only a subset of C++ is fully supported for the device code. In the fig. 4.12 is shown an example of a CUDA compiling program using either C++ for host code, and C with CUDA extensions for the device code.

```bash
GpuDensityFusion.o: GpuDensityFusion.cu
    nvcc -g -O0 -c GpuDensityFusion.cu -o GpuDensityFusion.o
DensityFusion.o: DensityFusion.cpp
    g++ -g -O0 -c DensityFusion.cpp -o DensityFusion.o
```

*Figure 4.12.: Example of makefile. CUDA nvcc compiler is used to compile *.cu files, which contain GPU kernels code, g++ compiler is used for *.cpp files usually for the C++ code.*

Below are a short description of several runtime C language extensions, these are mostly prefixes used to define specific features of functions or variables and variable types derived from the basics.

#### 4.7.1. Functions Type Qualifiers

These qualifiers defines both where the function is executed on host or device, and from where this function can be called

- `__host__` qualifier is defined for functions called from host and executed on host. These kinds of functions are equal to the normal functions of C language.
4.7. Programming Interface

- __global__ qualifier is defined for functions called from the host and executed on device. These kinds of functions are also known as kernels.

- __device__ qualifier is defined for functions called from device and executed on device.

4.7.2. Variable Type Qualifiers

These qualifiers define the memory location of the declared variables. The variables declared without qualifiers are considered registers:

- __device__: a variable declare using this qualifier is located in global memory of the GPU. It has the same lifetime as the application and is accessible by all threads and also the CPU using functions from the CUDA library.

- __constant__: a variable declare using this qualifier is located on the GPU constant memory, has the same lifetime as the application and is accessible by all threads and also the CPU using functions from the CUDA library.

- __shared__: a variable declare using this qualifier is located in shared memory, has the same lifetime of the block and is accessible by all threads in the block.

4.7.3. Built-in Vector Types

As explained in section B3 of [CUD11], these are vector types derived from the basic integer and floating-point types. They could be structured on four components accessible though the fields $x,y,z$ and $w$. There is a large list of this kind of vector but, basically the most used are:

- uint3: this type is an unsigned integer vector type based on a vector of type int with value $(x,y,z)$.
- dim3: this type is an integer vector type based on uint3 that is used to specify dimensions.

When a dim3 variable is defined, any unspecified component is initialized to 1. However, when the unspecified component of uint3 left, is initialized to 0.

4.7.4. Built-in Variable

Built-in variables define parameters of the execution configuration of the kernels. These variables are only declared for functions executed on device.

- grimDim is of type dim3 and contains the dimensions of the grid.
- blockIdx is of type uint3 and contains the block index within the grid.
- blockDim is of type uint3 and contains the dimensions of the block.
- threadIdx is of type uint3 and contains the thread index within the block.
5. Implementation

5.1. Noise Filtering

In this module, the quality of the captured images is improved based on multiple-image fusion methods. These methods enhance images by extracting the relevant information from two or more other images. The algorithms are developed according to the nature of the input images and the synthesizing type. These methods can be divided into: Super resolution methods, High-dynamic-range, Multiple-focus imaging, Image based rendering or simply fusion image captured in a different spectral range. This thesis is focused on obtaining quasi-noiseless images regarding the following conditions, see [DD11]:

- Non-prior knowledge of the content of the images,
- Assumption that the images to fuse are captured in a semi-static environment,
- The parameters of the camera are fixed.

The Figure 5.1 shows the schema of the variables location in both host and device memory and the required kernels in order to implement fusion method. The images captured by the cameras are allocated on host. These images can be parameterized by the image width $w$ and height $h$ and area $\Omega$. Every pixel $x$ contained in $\Omega$ has associated an independent random variable $I: \mathbb{N}^2 \rightarrow \mathbb{N}$. The discrete domain of the intensity $I$ is defined by $\Theta$.

\[
x \in \Omega := \{ x \mid (1,1) \leq x \leq (w,h) \} \subset \mathbb{N}^2, \quad (5.1)
\]

\[
I_t(x) \in \Theta := \{ i \mid 0 \leq i \leq (2^m - 1) \} \subset \mathbb{N} \quad (5.2)
\]

where the temporal subindex $t$ index over time and $m$ implies the (usually 8) bit per pixel.
The input images are transferred to GPU memory. The histogram of the intensity $I(x)$ and statistical parameters corresponding to each pixel are updated for each thread by the kernel AddSample:

- **Bin Maximum** indicates the maximum value of the intensity range.
- **Bin Minimum** indicates the minimum value of the intensity range.
- **Bin Maximal Frequency** defines the maximal density value.

Data type Bin can be defined as **char** or **unsigned short** according to the amount of images to be fused, the range value of the statistical parameters is contained in $[0, 255]$ or $[0, 65.535]$, respectively. In order to optimize the use of the GPU memory, if the number of image to be fused is lower than 255 Bin is defined as char. Moreover, to optimize the standard deviation computation of the equation 5.3, two accumulators are also precomputed in this kernel:

$$I(x) = \sum_{i=1}^{n} I_i(x)^2 , \quad J(x) = \left( \sum_{i=1}^{n} I_i(x) \right)^2$$  \hspace{1cm} (5.3)

where $n$ is the total amount of added images for the fusion.

When the required images are captured and added, the representative value from each pixel is extracted by the Fusion kernel by a two-stage interval analysis. Firstly, the global maximum of histogram narrows, by minimum $L(x)$ and maximum $U(x)$, locally the area where the maximal probability density is allocated. The
KDE is computed for the evenly spaced subsamples to obtain an accurate value of the maximal density fusion, which is expressed as

\[ \hat{I}_n(x) := \arg\max_{j \in \mathbb{R}} \left[ \sum_{t=1}^{k} K_h(i - I_t(x)) \right] U(x)_{j = \alpha + L(x), s \in \mathbb{N}_0} \]  

(5.4)

Where the \( K_h \) denotes the smoothing kernel with bandwidth \( h \) and \( \hat{I}_n(x) \) is the first approximation to the maximal density intensity \( \hat{I}(x) \). This expression is graphically represented in fig. 5.2.

![Figure 5.2](image)

Figure 5.2.: Example of probability density function estimated by kernel density estimation (KDE) from a histogram. Extracted from [BPT11]

This process is executed on every pixel independently. The value of bandwidth factor \( h \) is estimated according to the smoothing kernel. In this work, the KDE can be computed by Epanechnikov and Gaussian smoothing kernels. The bandwidth factor for Epanechnikov kernel must be a known parameter defined previously. However, for Gaussian kernel the value of \( h : \mathbb{N}^2 \mapsto \mathbb{R}^+ \) is calculated from the standard deviation \( \sigma : \mathbb{N}^2 \mapsto \mathbb{R}^+ \) for every pixel, using the equations of 5.3, as

\[ \sigma(x) = \sqrt{\frac{1}{n} \left( \bar{I}(x) - \frac{1}{n} \bar{J}(x) \right)} \]  

(5.5)

As explained in [Sil88], the bandwidth factor \( h(x) \) for one-dimensional discrete samples is obtained from \( \sigma(x) \) as

\[ h(x) = \left( \frac{4\sigma(x)^5}{3n} \right)^{\frac{1}{5}} \]  

(5.6)

where \( n \) is the number of samples until this point in time. To estimate the pdf, the smoothing kernel \( K \) is defined to Gaussian case as:

\[ K_{h(x)}(i - I(x)) = \exp \left( -\frac{1}{2} \frac{(i - I(x))^2}{h(x)^2} \right) \]  

(5.7)
or Epanechnikov as:

\[
K_h(i - \mathbf{I}(\mathbf{x})) = 1 - \frac{(i - \mathbf{I}(\mathbf{x}))^2}{h^2}
\]  

(5.8)

Therefore, the KDE from the equation 5.4 can be expressed as

\[
\hat{\mathbf{I}}_\alpha(\mathbf{x}) := \arg\max_{j \in \mathbb{R}} \left[ \sum_{t=-h}^{h} K_h(\mathbf{x}) (i - \mathbf{I}_{i-t}(\mathbf{x})) \cdot H(\mathbf{I}_{i-t}(\mathbf{x})) \right]_{j=\alpha s + \mathbf{L}(\mathbf{x}), s \in \mathbb{N}_0}^U(\mathbf{x})
\]  

(5.9)

Where \( \mathbf{I}_{i-t}(\mathbf{x}) \) is the discrete intensity value of the histogram in \( i - t \), \( H(\mathbf{I}_{i-t}(\mathbf{x})) \) indicates the amount of samples on it.

![Figure 5.3.](image)

Figure 5.3.: The straightforward fusion methods such as mean or median obtain lower density values than the fusion method developed for this work. Extracted from [DD11]

Secondly, by using an iterative method, higher resolution of the representative intensity value is obtained by using the KDE close to the maximum density value \( \hat{\mathbf{I}}_\alpha \).

By defining \( \beta < \alpha \), the value is obtained as

\[
\hat{\mathbf{I}}(\mathbf{x}) := \arg\max_{j \in \mathbb{R}} \left[ \sum_{t=-h}^{h} K_h(\mathbf{x}) (i - \mathbf{I}_{i-t}(\mathbf{x})) \cdot H(\mathbf{I}_{i-t}(\mathbf{x})) \right]_{j=\beta s + \hat{\mathbf{I}}_\alpha(\mathbf{x}) - \alpha, s \in \mathbb{N}_0}^{\hat{\mathbf{I}}_\alpha(\mathbf{x}) + \alpha - \beta}
\]  

(5.10)

5.2. HDR Synthesize

Figure 5.4 shows the schema of the HDR synthesize method. The intensity values of the LDR input images may come from the CPU memory space, which information must be transferred to GPU before the synthesis is executed. However, due to the duality memory space of both CPU and GPU, the output buffer of the fusion method, which is allocated on GPU, can be associated directly as input of HDR synthesis avoiding any data transference.

Previously to the HDR image synthesize, a radiometric calibration has to be done in order to obtain a transformation from the scene radiance to the discrete values of the images. Due to the image-content independence, the radiometric calibration \( g_\lambda : \mathbb{N} \rightarrow \mathbb{R} \) is based on computation described in section 3.2. However, the implementation of this calibration has two problems:
5.2. HDR Synthesize

- There is no explicit criterion for the selection of the $\lambda$-smoothing regression factor.
- The transformation function between exposure and intensity values presents irregularities on its extremes, see fig. 5.5-a.

In order to avoid the inconsistency on the extremes (low and high intensity) that can be presented in HDR images caused by noisy calibration, an improvement extension for calibration presented in [GAAD10] is developed:

By computing exponential function of the discrete calibration curve

$$g^*_\lambda(I_t(x)) := \exp(g_\lambda(I_t(x))) \quad (5.11)$$

a continuous model can be obtained, see fig. 5.5-b.

$$g^L_\lambda : \mathbb{R} \rightarrow \mathbb{R}; g^L_\lambda(I_t(x)) \approx g^*_\lambda(I_t(x)) \quad (5.12)$$

This calibration function avoid the issues on the extremes of the intensity values, see fig. 5.5-c.

$$g^M_\lambda (\bar{I}_n^j(x)) = \log \left( g^L_\lambda (\bar{I}_n^j(x)) \right) \quad (5.13)$$

where $\bar{I}_n^j(x)$ is the image obtained by fusing $n$ images for the exposure time $j$ and $g^M_\lambda$ is the standard domain of the radiometric calibration model. The parameters

Figure 5.4.: Schema of the HDR synthesize method algorithm
Figure 5.5.: a) The Debevec $\lambda$-optimal radiometric camera calibration by experimental evaluation of the $\lambda$-smoothing factor. b) The model calibration $g^L_\lambda$ is accomplished by a weighed linear regression in the exponential domain. The magenta dashed weighting Gaussian kernel weights the Debevec $\lambda$-optimal $g^L_\lambda$ Debevec calibration in order to estimate the linear model calibration $g^L_\lambda$. c) The $\lambda$-factor selection is based on the minimal deviation of the regression model, the graph shows the Debevec $\lambda$-optimal, the continuous model $g^M_\lambda$ and their mutual deviations. Extracted from [GAAD10]

of the radiometric calibration model $g^M_\lambda$ are different for every color channel. In kernel AddLDRImage, the calibration is computed according to the color channel associated to each pixel. Because of the cameras capture the images using the Bayer filter pattern. A channel map is implemented in this kernel, based on Bayer pattern, to obtain the channel associated to every pixel.

By fusing a set of LDR images captured by different exposure time, the radiance from the scene is represented in HDR image. The short-expose images sample the high radiance regions of the scene. Inversely, the long-expose images sample the low radiance regions. The synthesis algorithm is executed in Synthesize kernel. However, the summations required to synthesize the HDR image are previously calculated in AddLDRImage kernel

$$R(x) = \sum_{j=1}^{m} N(\mathcal{T}_j(x))(g^M_\lambda(\mathcal{T}_n(x)) - \log(\Delta_{ij}))) \quad (5.14)$$

$$K(x) = \sum_{j=1}^{m} N(\mathcal{T}_j(x)) \quad (5.15)$$

where $N$ is the Gaussian kernel is used as weighting function for LDR images, see fig. 5.5b. From the equations 5.15 and 5.14 the HDR image synthesis is defined as

$$\Phi(x) = \exp \left[ \frac{R(x)}{K(x)} \right]. \quad (5.16)$$

The synthesized image can be also represented on homomorphic space. In this case,
the image is represented in a logarithmic scale,

$$\log(\Phi(x)) = \frac{\mathcal{R}(x)}{K(x)}$$  \hspace{1cm} (5.17)

Figure 5.6.: Example of the HDR synthesis. Both images are obtained by fusion of sixty-five LDR image. Left image shows the result of linear scale representation and the right is a homomorphic representation.

### 5.3. Gabor Edge Extraction

The schema of the last stage to edge extraction is illustrated in fig. 5.7. Like HDR synthesize method, input images may come from CPU or GPU memory space.

The receptive radiance saliency

$$\Gamma: \mathbb{N}^2 \mapsto \mathbb{R}^2$$

is computed by Gabor kernel. This filter is commonly used for image analysis due to their biological relevance and computational properties, see section 3.3.3. The shapes of the Gabor kernels are located by a spatial sinusoidal Gaussian window, see fig. 5.8. The filter is defined as a family of filter bank where each one is rotated with respect to another, see fig. 5.9 and it can be defined as

$$\Psi(x, \alpha, \gamma_1, \gamma_2) = \exp \left(-\frac{1}{2} x^T \Sigma^{-1} x\right) \sin \left(\pi \frac{U_{\alpha}\cdot x}{\lambda}\right),$$  \hspace{1cm} (5.18)

where the covariance matrix $\Sigma$ adjusts the smoothing extraction trade-off and it is defined as

$$\Sigma = \text{diag} [\gamma_1 \gamma_2] [U_{\alpha} V_{\alpha}]$$  \hspace{1cm} (5.19)

and the vectors $U_{\alpha}$ and $V_{\alpha}$ allow the saliency extraction in a $\alpha$-target direction

$$U_{\alpha} = [\cos \alpha \sin \alpha]^T \text{ and } V_{\alpha} = [-\sin \alpha \cos \alpha]^T$$  \hspace{1cm} (5.20)

The computation of the receptive radiance saliency $\Gamma$ is executed by the convolution kernel and it requires $k \geq 1$ band-pairs $(\alpha, q)$ of orientation-complementary kernels,

---

1Radiance saliency is the receptive radiance energy for edge extraction, see [GAAD10].
\[
\Gamma(x) = \sum_{p=1}^{k} \sum_{q=0}^{l} U_{<p\pi/2k>} \left\{ \Phi(x) * \Psi(x, p\pi/2k + \frac{q\pi}{2}, \gamma_1, \gamma_2, \lambda) \right\}
\] (5.21)

where \( * \) denotes the convolution operator. From the receptive radiance saliency the magnitude and phase images can be obtained, see fig. 5.10.

Figure 5.10.: Example of obtained images from the receptive radiance saliency \( \Gamma \) computed by a set of forty pair of kernels, each one has radius \( r = 2 \). The magnitude (left) and phase (right) of \( \Gamma \) are illustrated.
From the normalized saliency direction ,

\[
\hat{\Gamma}(x) = \Gamma(x) \frac{1}{\|\Gamma(x)\|} = \begin{bmatrix} \hat{\Gamma}_x(x) & \hat{\Gamma}_y(x) \end{bmatrix}^T \tag{5.22}
\]

the neighbors pixels \( \mathbf{x}_\pm \) are defined as

\[
\mathbf{x}_\pm = x \pm \eta\left(\hat{\Gamma}(x)\right) \tag{5.23}
\]

where the function \( \eta : \mathbb{R}^2 \mapsto \{-1, 0, 1\} \) defines the location \( \mathbf{x}_\pm \in \mathbb{N}^2 \) in the image relative to \( x \),

\[
\eta\left(\hat{\Gamma}(x)\right) = [\eta_x(x) \eta_y(x)]^T \tag{5.24}
\]

according to magnitude of \( \hat{\Gamma}_x(x) \) and \( \hat{\Gamma}_y(x) \) the neighborhood condition is defined for each component as
\[
\eta_k(x) = \begin{cases} 
1, & \hat{\Gamma}_k(x) > L \\
0, & -L < \hat{\Gamma}_k(x) < L, \\
-1, & \hat{\Gamma}_k(x) < -L,
\end{cases}
\] (5.25)

where \( L \) is a parameter defined as orientation limit.

The dual non-maximal suppression \( \gamma : \mathbb{N}^2 \mapsto \{0, 1\} \) selects pixels close to the edge by considering the saliency norm \( \|\Gamma(x)\| \) and the saliency direction coherency \( \mu \),

\[
\gamma(x) = \begin{cases} 
1, & \langle \|\Gamma(x)\| > \|\Gamma(\tilde{x}_+)\| \rangle \wedge \langle |\hat{\Gamma}(x)\hat{\Gamma}(\tilde{x}_+)\rangle > \mu \rangle \wedge \\
0, & \langle \|\Gamma(x)\| > \|\Gamma(\tilde{x}_-)\| \rangle \wedge \langle |\hat{\Gamma}(x)\hat{\Gamma}(\tilde{x}_-)\rangle > \mu \rangle \wedge \\
\end{cases}
\] (5.26)

As illustrated in fig. 5.11, isolated peaks are obtained from \( \gamma \) as edge pixels. At last stage of the edge extraction process, these peaks are removed from image, see fig. 5.12.

Figure 5.11.: Example of image obtained by Non-maximal suppression algorithm.
Figure 5.12.: Example of edge-image where the isolated pixels have been removed.
6. Evaluation

In this chapter, the analysis of methods, experiments and results are described. Before that, a short description of the hardware and software where the algorithms were implemented is given. The following results are divided into three steps. Firstly, the results that allow defining the optimal execution configuration for each kernel. Secondly, the execution time of the kernels using the optimal configurations is evaluated. Finally, the quality of the images obtained from the density fusion process and the HDR synthesis is analyzed.

6.1. Hardware and Software

The evaluation presented in this chapter has been performed on a workstation equipped with an Intel Core i7-870 running at 2.93 GHz with 8 MB Cache. The GPU GeForce GTX 460 is equipped with 336 cores, a memory of 2048 MB, memory clock at 1800 MHz and processor clock at 810 MHz.

The Eclipse IDE for C/C++ Developers Indigo Service Release 1 has been used as development environment and compiler. Also, the applied key components of CUDA used in this work are:

- NVIDIA Accelerated Linux Driver Set.
- CUDA Toolkit version 3.2 Release.
- CUDA SDK 3.2 for Linux

6.2. Optimal Execution Configuration

The execution configuration defines the dimension of the grid and blocks that will be used to execute the kernels. A kernel can be executed using different execution configurations and the execution time of the kernel will be different for different configurations. As explained in [Gui12], the most important performance consideration in programming for CUDA architecture is the coalescing of global memory access.

\[\text{The coalesced memory access is defined in section 4.6}\]
Because of the global device memory loads and stores by threads are the bottleneck of any process implemented on GPU. Therefore, certain technical specifications have to be considered to define the parameters of execution configuration:

- The maximum number of threads per block is 1024.
- The maximum number of active blocks per multiprocessor is 8.
- The maximum number of active warps per multiprocessor is 48.

The number of active block is conditioned by the amount registers and shared memory in use. This trade-off between occupancy and efficiency leads to think about try out different scenarios, because performance it cannot be derived from a generic function of the parameters. Therefore, the optimal distribution of the threads and blocks per grid has to be independently evaluated for each kernel implemented.

According to these conditions the grids are analyzed taken into account that the amount of threads per block must be multiple of warp. Kernels can be splitted into two groups according the dimensionality of the grid, these grids can be defined as vector or matrix blocks according to the instructions to be executed within the kernel.

### 6.2.1. Grids as Vector of Blocks

The kernels within this group are characterized by an execution configuration based on vector of blocks:

```c
dim3 BlockDim(BlockDimX, 1, 1);
dim3 Grid(Area / BlockDimX, 1, 1);
Kernelxxx<<<Grid, BlockDim>>>()
```

This configuration is defined for kernels which arguments are pointers to device memory allocated as 1-dimension buffers. The variable BlockDim is a matrix of threads with one row of BlockDimX number of threads. The amount of blocks per grid is defined in Grid, as a one row with Area/BlockDimX number of blocks, where Area is the area of the input image.

In order to obtain the optimal execution configuration for each kernel, the statistics parameters are obtained after hundred trials are executed for each execution configuration. As an illustration, the results obtained for one of these kernels are shown in fig. 6.2.1.

Amount of threads per block represented on axis X, corresponds to variable BlockDimX. That means, knowing the number of threads per block, the BlockDim and Grid variables are implicitly defined. The execution time of axis Y is expressed in milliseconds. The represented values are the maximum, minimum and mean. The optimal execution configuration corresponds to the configuration with the minimal of minimums execution time. The table shows the number of threads per block corresponding to the optimal configuration for each kernel implemented in this work.
6.2. Optimal Execution Configuration

Figure 6.1.: For every execution configuration is represented the maximum, minimum and mean times. Appendix A.1 contains the illustrations for each of these kernels

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Optimal Number of Threads per Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>KernelAddSample</td>
<td>32</td>
</tr>
<tr>
<td>KernelSetImageMask</td>
<td>64</td>
</tr>
<tr>
<td>KernelFusionByGaussianKDE</td>
<td>64</td>
</tr>
<tr>
<td>KernelFusionByEpanechnikovKDE</td>
<td>160</td>
</tr>
<tr>
<td>KernelAddLDRSample</td>
<td>64</td>
</tr>
<tr>
<td>KernelResetSamples</td>
<td>64</td>
</tr>
<tr>
<td>KernelSythesisHDRLineal</td>
<td>480</td>
</tr>
<tr>
<td>KernelSythesisHDRHomomorphic</td>
<td>736</td>
</tr>
</tbody>
</table>

Table 6.1.: Optimal number of threads per block for every execution configuration.

6.2.2. Grids as Matrix of Blocks

The kernels within this group are characterized by an execution configuration based on grids defined as a matrix of blocks:

\[
\text{dim3 BlockDim}(\text{BlockDimX}, \text{BlockDimY}, 1);
\]

\[
\text{dim3 Grid}(\text{Width}/\text{BlockDimX}, \text{Height}/\text{BlockDimY}, 1);
\]

\[
\text{Kernelxxx} \lll<<\lll \text{Grid}, \text{BlockDim}>>();
\]

This configuration is defined for kernels which arguments are pointers to device memory allocated as 2-dimensions buffers, as well as, the kernels which use the *texture references*. The dimension of the *thread* matrix (BlockDim) is defined by BlockDimX and BlockDimY values. Also, the number of blocks for each dimension of the Grid matrix is defined by Width/BlockDimX and Height/BlockDimY respectively. Where the width and height are parameters of the input image. It must be taken into account that, certain execution configurations are not suitable, for instance,

\[
\frac{\text{Width}}{\text{BlockDimX}} \leq 1 \text{ or } \frac{\text{Height}}{\text{BlockDimY}} \leq 1
\]  

(6.1)
this two cases defines the Grid matrix as a \((0, \text{Height} / \text{BlockDimY}, 1)\) or \((\text{Width} / \text{BlockDimX}, 0, 1)\). As an illustration, the results of execution time for one kernel within this group are shown below:

The X axis represents the enumeration of execution configurations and in axis Y, the number of pair of Gabor kernels defined for the convolution. This illustration shows the execution time in milliseconds for each case. For this evaluation, the radius of the kernels is constant \(r = 2\). Due to the 4-dimensionality of the function to be represented, an execution configuration identification number (Execution Configuration Id), is defined by each \((\text{BlockDimX}, \text{BlockDimY}, 1)\) configuration. In order to represent graphically the results, these configuration ids are sorted based on the minimum execution times.
These illustrations show the minimum execution times for 1 and 63 pairs of complementary kernels. It can be observed that the minimum execution times correspond to configurations where values of BlockDimX are high and values of BlockDimY are low, and the maximum corresponds to configurations where values of BlockDimX are low and BlockDimY value are high. However, it has not a linear behavior. This means that, increasing linearly BlockDimX and decreasing BlockDimY do not imply a linear increasing of the execution time.

Otherwise, the execution configurations can be also evaluated varying the radius length for a fixed number of Gabor kernels as illustrated below.
In this evaluation, the configurations with large number of threads in BlockDimX and low of them in BlockDimY obtain faster executions that the rest. The results of these evaluations are grouped in the following table.

<table>
<thead>
<tr>
<th>Radius Length</th>
<th>Number of Kernels</th>
<th>BlockDimX</th>
<th>BlockDimY</th>
<th>Execution Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>352</td>
<td>1</td>
<td>0.1724</td>
<td>14.26x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>32</td>
<td>2.458</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>336</td>
<td>2</td>
<td>3.655</td>
<td>8.84x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>480</td>
<td>32.3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>336</td>
<td>2</td>
<td>0.2971</td>
<td>10.63x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>32</td>
<td>3.159</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>63</td>
<td>336</td>
<td>2</td>
<td>13.09</td>
<td>10.5x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>480</td>
<td>137.4</td>
<td></td>
</tr>
</tbody>
</table>

In this evaluation, the configurations with large number of threads in BlockDimX and low of them in BlockDimY obtain faster executions that the rest. The results of these evaluations are grouped in the following table.

<table>
<thead>
<tr>
<th>Radius Length</th>
<th>Number of Kernels</th>
<th>BlockDimX</th>
<th>BlockDimY</th>
<th>Execution Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>352</td>
<td>1</td>
<td>0.1724</td>
<td>14.26x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>32</td>
<td>2.458</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>336</td>
<td>2</td>
<td>3.655</td>
<td>8.84x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>480</td>
<td>32.3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>336</td>
<td>2</td>
<td>0.2971</td>
<td>10.63x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>32</td>
<td>3.159</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>63</td>
<td>336</td>
<td>2</td>
<td>13.09</td>
<td>10.5x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>480</td>
<td>137.4</td>
<td></td>
</tr>
</tbody>
</table>
From these results, it can be appreciated that for an implemented kernel the execution times are highly dependent on the execution configuration. The ratio between the minimum times for the faster and slower configurations is approximately 10x speedup. The tab. 6.2 contains the optimal execution configuration for the kernels within this group.

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Number of Threads per Block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BlockDimX</td>
</tr>
<tr>
<td>KernelConvolutionMagnitudePhase</td>
<td>336</td>
</tr>
<tr>
<td>KernelSelectPeakMultipleLevelCheck</td>
<td>352</td>
</tr>
<tr>
<td>KernelClearIsolatedPeak</td>
<td>352</td>
</tr>
</tbody>
</table>

Table 6.2.: Optimal execution configuration for kernel based on the minimal execution time. Appendix A.2 contains the illustrations for each of these kernels.

6.3. Modules Evaluation

After the optimal execution configuration for each kernel has been evaluated, in this section the implemented modules are analyzed in order to obtain relevant information. Firstly, the execution times of the last steps of both fusion method and HDR synthesis are evaluated. Secondly, the quality of the images obtained by these methods is analyzed by root mean square (rms).

6.3.1. Fusion Module

The intensity value for every pixel is defined from kernel density estimation (KDE), as explained in Section 3.1. In this work, these values are estimated using Gaussian or Epanechnikov kernels.

![Execution Time of Fusion by Gaussian KDE for Different Number Added Images](image)

In this illustration, the number of images used to fuse is represented in axis X. In axis Y the execution time in milliseconds is represented. The maximum time by the fusion is required for four images added. The table 6.3 shows the execution time depending on the number of images used by the fusion method. For every
amount of image added, the % indicates the ratio of the time needed in relation to the maximum execution time:

<table>
<thead>
<tr>
<th>Execution Time for scenario 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Time</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>94.37 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution Time for scenario 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Time</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>90.75 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution Time for scenario 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Time</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>123.63 ms</td>
</tr>
</tbody>
</table>

Table 6.3.: The fusion by Gaussian KDE is analyzed using three scenarios showed in fig. 6.2.

The result of these tables indicates that, when the number of images is near than 100, fusing time is reduced until around third part. After this number of images, execution time slows down into a linear behavior, the difference of ratio time between fusing 100 or 500 images is only around 9%. The reason of the time difference between fusing small or large amount of images can be explained by the Central Limit Theorem (CLT). The CLT explain that, if $S_n$ is the sum of $n$ independent random variables, the distribution function of $S_n$ can be approximated by a normal distribution.

The intensity values of the pixels can be considered as a random variable $I$, which values are obtained from the sum of several independent random variables such as illumination variations, electric, electromagnetic and thermal perturbations, etc. Therefore, for a large number of images added, the distribution function of the intensity values, $S_I$, can be considered as normal distribution. Consequently, the function $S_I$ will contain one local maximum. However, for small amount of images, $S_I$ cannot be considered normal distributed; hence, $S_I$ can include several local maximums. In conclusion, the required time to obtain the global maximum of a distribution function by KDE is faster for normal distribution, as a result, it is faster for large amount of image added.
Upper illustration shows the times obtained when fusion method is executed using Epanechnikov KDE. Unlike Gaussian KDE, these times have not monotonic decreasing behavior. In this case, it is able to find an specific number of images where the minimum execution time for this algorithm is obtained.

<table>
<thead>
<tr>
<th>Execution time for scenario 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Time</td>
</tr>
<tr>
<td>17.78 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution time for scenario 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Time</td>
</tr>
<tr>
<td>17.58 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution time for scenario 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Time</td>
</tr>
<tr>
<td>18.88 ms</td>
</tr>
</tbody>
</table>

Table 6.4.: The images of scenarios used are showed in fig. 6.2. Like in Gaussian KDE, the maximum time is obtained for four images added.

Figure 6.2.: These three scenarios are used to analyze the execution time of the KDE. The references in this section of Scenario 1, Scenario 2 and Scenario 3 correspond to the left, center and right images respectively.
The optimal number of images is approximately the same number of images for Gaussian KDE fusion. Fusing a different number of images increase the needed time by the algorithm.

However, the execution time is not the only parameter to be analyzed. The number of frames per second that the cameras are able to capture is commonly the bottleneck. For example, for fusion by Gaussian KDE, even when the execution time is lower for 500 images than 100 images, it is important to think about the time required for the cameras to capture this number of images. Moreover, the quality of the images also depends on the number of images used by the fusion. The convergence of the density fusion process is analyzed below.

![Graph showing rms-convergence of density fusion process](image)

Figure 6.3.: Illustration of the rms-convergence of density fusion process for the scenarios of fig. 6.4.

In order to analyze the image quality improvement relative to the amount of fused images, horizon fusion image is created by fusing 512 images for each of 32 scenarios analyzed, see fig. 6.3. Then, the deviation showed in these illustrations indicates the root mean square (rms) for a partial k number of fused images. The right zoom image from the rms-convergence shows that after 4 samples the maximal deviation is reached. Near of 150 samples the convergence slows down into a lineal behavior until around 450 samples, these values depend on the scene.
Figure 6.4.: These scenarios are used to compute the rms-convergence of the density fusion
At this point, the different parameters to taking into account in order to fuse the optimal number of images are explained. However, this optimal value is obtained having the previous knowledge of the information from the scene but, in real-time applications the robot has not previous information about the scene. Therefore, the trade-off between the execution time and the quality of the image has to be taking into account when defining the number of images needed to fuse.

### 6.3.2. HDR Synthesize Module

HDR synthesis allows obtaining two kinds of image depending on the used kernel. One of these output images is represented in lineal scale, the other is scaled as logarithmic. The following illustration shows the execution times for both of these kernels.

Where the execution time in milliseconds is represented in axis Y, and in axis X are represented the number of LDR images used for the synthesis. The execution times represented in this illustration are from the kernels which execute the last step of the HDR synthesis. These values do not consider the stage where the samples are added. Without consider the execution times locally, the results have a constant behavior along the number of images synthesized. In conclusion, the execution time of synthesis can be considered independent of the number of synthesized images, because it is a division and an exponential per pixel, independently of the amount of images.

As seen in equations \[5.16\] and \[5.17\] the difference between both methods is to compute an exponential function. For a \(k\) number of images, the difference between execution time of both kernels is equal to compute exponential function. This function is implemented by CUDA libraries. Functions from these libraries are faster versions of the same ones from the mathematical standard libraries. From the table below is deduced that compute an exponential function implemented by CUDA libraries for all pixel from a 640x480 image cost 0.6 microseconds. This fact, once again place on record, the high capability to parallelize process of CUDA architecture.

<table>
<thead>
<tr>
<th>Mean Execution Time (microseconds) of HDR synthesis methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lineal Synthesis</td>
</tr>
<tr>
<td>56.3</td>
</tr>
</tbody>
</table>
6.3.3. RMS-Convergence by HDR Synthesis

The exposure times of the camera must be set manually in order to capture every LDR image. These times must be set using the exposure value parameter. However, as it is shown in fig. 6.5, the relationship between exposure value and exposure time is not linear.

![Exposure Time of the Camera according to Exposure Value](image)

Figure 6.5.: The exposure time and exposure values have not a direct variation (red line). From exposure value = 73 the slope of the function is increased. The range of exposure values must be subsampled (green), in order to exposure time becomes directly proportional to exposure value (blue).

Subsampling the range of exposure values allows to defining the amount of LDR images manually. From this, the fusion method can be analyzed for several exposure times. An HDR image is obtained from the scenario of the fig. 6.6 by a set of 180 LDR images. For every of these LDR images, the rms-convergence is analyzed in fig. 6.7. The deviation values indicate that the variations in the intensity of the image are higher for short exposure times. The analysis of section 6.3.1 showed that for a few number of images fused, the deviation of the rms was also higher. Therefore, the images that satisfy both conditions are strongly affected by the noise.

![Figure 6.6: This scenario is used to analyze the convergence for several LDR images.](image)
Figure 6.7.: The rms-convergence is evaluated for a set of 180 LDR images. The X axis represents the number of fused images at HDR synthesis. The exposure times of the Y axis are represented by 180 subsamples which are evenly-spaced as explained in fig. 6.5.

Consequently, in order to obtain a set of LDR images which accomplish to have a certain error value, the LDR image for each exposure time must be obtained by fusing different amount of images. The figure 6.8 shows the level-curves of the rms-convergence. The general behavior of these curves indicates that the amount of image to be fused must be larger by short exposure time. The deviation for $k$ fused images in short exposures is higher than in long exposure times. Consequently, to obtain a certain error value for all the LDR images, the amount of images that must to be fused is larger for short values of exposure time.

Figure 6.8.: The rms-convergence is represented by the level-curves. The deviation values represented are $\{0.2, 0.4, 0.6, 0.8, 1.0, 1.2\}$, from blue until brown respectively.
7. Conclusion

7.1. Summary and Results

The aim of this work was firstly, studying the paradigm of the parallel programming model; secondly, reformulating the algorithms of low-level image processing to adapt them for GPU and finally, analyzing the execution configurations to optimize the execution of the kernels.

As the parallel programming model was introduced recently, in ch. 4 a detailed description of GPU architecture, CUDA model programming and memory structure is shown. In chapter 2 the state of the art shows a compact view of few robot vision application which are implemented, partially or completely in GPU, in order to reduce drastically the execution time. The targets of these are also distinct showing that the parallel programming on GPU is a good model for whether features extraction or object and human recognition.

In this work, the implemented algorithms were divided into three modules: fusion, HDR synthesize and edge extraction. In the main part, the reformulation of these modules is described based on the data structure of each module, the memory transactions between CPU and GPU and the programming interface of CUDA.

The execution configurations defined by the kernels are exhaustively analyzed in order to optimize the execution times. This analysis concludes that for a given kernel the configuration of the number of threads per block and blocks per grid can grow the speed up of the kernels execution in 10 times. Using optimal execution configurations, an example of timing process is shown in tab. 7.1.

The implementation of these modules allows executing the whole process of edge extraction sequentially on GPU, without data transfers between CPU and GPU memories. In other words, the input data are allocated in device from the image acquisition and the CPU is set free until edge extraction is concluded. This allows executing in CPU the image acquisition and the data transmission to device memory, while the GPU computes the required kernels.

Consequently from the table, it can be observed that the cameras are the application bottleneck. The maximal rate acquisition of the cameras is 52.5fps. Therefore, the
robot captures one image each 19 milliseconds approximately using the maximal rate acquisition. Usually the frame rate is less than the maximum rate. For example, at stage of LDR images acquisition this rate is decreased to obtain images from large exposure time. Even considering the fusion, synthesis and edge extraction methods together, without the stage of acquisition images, the execution time is negligible and hidden by the capture images from the cameras.

### 7.2. Future Work

The analysis of the rms-convergence indicates that the quality of the images obtained from the density fusion process depends on the scene and the exposition time. As shown in this work, this analysis allows defining, approximately, a minimum number of images required to be fused according to the exposure time. Moreover, the images obtained by shorter exposure times are captured faster but large amount of them are required, and the opposite happens for large values of exposure time. Because of the imaging capture is the bottleneck, it would be interesting to obtain more accurate number of images required by the fusion. An exhaustive analysis of images with several dynamic range would allow obtaining it. In addition, fusion execution time is highly variable according to the weighted function used for KDE. Therefore, the fusion process could be evaluated by several weighted functions. Both analysis would allow defining a minimum number of input images to obtain images with the required quality in a faster way.

The HDR images contain completely the range of the intensity values from the scene by synthesis of LDR images. However, the minimum number of LDR images required to capture the intensity range of the scene is an unknown parameter. It would be interesting to know the minimum number of images needed. The evaluation shows that the time to synthesize HDR images is constant and independent of the number of input images. Therefore, the execution time of the synthesis is not an interesting parameter to define this number. It can be defined only in order to include completely the intensity value range of the scene in the output HDR image.

Otherwise, there is the edge extraction. The optimal execution configurations of its kernels are based on the mean of the minimum execution times. However, the
optimal number of threads per block is different according to the parameters of the kernel mask such as size or amount of complementary kernels. Therefore, these configurations should be defined according to these parameters. One idea to increase the efficiency is to develop a system which sets the number of threads per block before the edge extraction process is executed. This system could define this amount for each execution configuration from the parameters of the kernel mask automatically.

Despite this, the bottleneck will still be the camera’s acquisition. The fusion process for noise filtering requires 30 frames and the HDR synthesis needs 16 exposures. Hence, for single HDR noiseless image are required 480 images. Because of it, the 52.5fps from the cameras of ARMAR-III becomes so slow compared with the edge extraction process based on parallel programming model, as seen in tab. [7.1]. Actually, there are cameras that allow capturing over 150fps. Including these cameras to the humanoid, the required time to execute this process can be reduced drastically. Moreover, several of these cameras are able to capture the images for the scene directly as a HDR images, which would be an interesting complement or even, certain methods implemented in this work could be partially reduced.
Bibliography


Appendix

A. Evaluation of Kernel Execution Configuration

The illustrations of this appendix show the execution time for each of the kernels implemented in this work. This time is evaluated for the different number of threads per block of the execution configuration. The obtained values are the maximum, minimum and mean of thousand trials. For each kernel, the optimal amount of threads per block and the minimum execution time for it is showed.

A.1. Evaluation of Execution Time 1-Dimension Grid Configuration

![Graph of Kernel Add Sample Execution Time]

<table>
<thead>
<tr>
<th>Amount Threads per Block</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1.5</td>
</tr>
<tr>
<td>128</td>
<td>1.7</td>
</tr>
<tr>
<td>192</td>
<td>1.9</td>
</tr>
<tr>
<td>256</td>
<td>2.1</td>
</tr>
<tr>
<td>320</td>
<td>2.3</td>
</tr>
<tr>
<td>384</td>
<td>2.5</td>
</tr>
<tr>
<td>448</td>
<td>2.7</td>
</tr>
<tr>
<td>512</td>
<td>2.9</td>
</tr>
<tr>
<td>576</td>
<td>3.1</td>
</tr>
<tr>
<td>640</td>
<td>3.3</td>
</tr>
<tr>
<td>704</td>
<td>3.5</td>
</tr>
<tr>
<td>768</td>
<td>3.7</td>
</tr>
</tbody>
</table>

Max
Mean
Min
b) Kernel Set Image Mask Execution Time

Max
Mean
Min
Amount of Threads = 64
Execution Time = 0.2316 ms

0   64  128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
0.2
0.25
0.3
0.35
0.4
0.45
0.5
0.55
Amount Threads per Block

c) Kernel Fusion by Gaussian KDE Execution Time

Max
Mean
Min
Amount of Threads = 64
Execution Time = 28.42 ms

0   64  128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
28
29
30
31
32
33
34
35
36
Amount Threads per Block

d) Kernel Fusion by Epanechnikov KDE Execution Time

Max
Mean
Min
Amount of Threads = 128
Minimal Execution Time = 14.81 ms
Maximal Execution Time = 15.28 ms
Amount of Threads = 160
Minimal Execution Time = 14.81 ms
Maximal Execution Time = 15.26 ms

0   64  128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
14
14.5
15
15.5
16
16.5
17
17.5
18
18.5
19
Execution Time (ms)

Appendix
A. Evaluation of Kernel Execution Configuration

---

e) Kernel Add LDR Sample Execution Time

- Amount of Threads = 64
- Execution Time = 0.1571 ms

---

f) Kernel Reset Samples Execution Time

- Amount of Threads = 64
- Execution Time = 0.05392 ms

---

g) Kernel Synthesis HDR Lineal Execution Time

- Amount of Threads = 480
- Execution Time = 0.05507 ms
h) Kernel Synthesis HDR Homomorphic Execution Time

- Max
- Mean
- Min

Amount of Threads = 736
Execution Time = 0.05683 ms

Minimal Execution Time of Fusion Method by Gaussian and Epanechnikov KDE

- Gaussian
- Epanechnikov
A.2. Evaluation of Execution Time 2-Dimension Grid Configuration

The illustrations of this section correspond with the edge extraction kernels. The execution times of these illustrations are evaluated by \textit{MinimalMagnitude} = 0 and \textit{Coherence} = 60°. Like the illustrations of the previous section, these values are obtained of thousand trials and they represent the minimum execution time for every execution configuration.
B. Implementation Structure

This section helps to understand how the implementation of the thesis is done, by definition of common patterns, structures and CUDA instructions. Firstly, the common pattern to identifier threads from any execution configuration is adapted to be suitable for this work. Secondly, the texture memory of GPU optimizes accesses for two-dimension address, which is commonly used for algorithms such as convolutions. Finally, an overview of the code structure is presented. Host code is implemented by C++ language and kernels by C language and CUDA extensions. Both codes must to be organized in order to optimize the functionality of algorithms.

B.1. Index Map on Parallel Programming Model

CUDA language allows to identifier the threads using identification numbers based on the structure of blocks and threads declared in order to execute kernels. The architecture of the NVIDIA graphic cards executes instructions on multiple data simultaneously using parallel threads. This capability allows to think about use each of this threads to compute the information about each pixel from images. Also, this architecture allows to use a common pattern to map these indexes in one, two or three dimensions, see an example of one dimension map in Fig B.1.

Assuming: Buffer size = 16 and blockDim=4 => 4 blocks
int idx = blockDim.x * blockIdx.x + threadIdx.x

Figure B.1.: Common pattern map from local index to global index

The images used on this thesis can be allocated on one or two dimensions buffers. Therefore, in order to define a global index for images based on this CUDA pattern, the one dimension buffers are defined as,

\[
\text{const uint CurrentPixel = blockIdx.x * blockDim.x + threadIdx.x;}
\]

and on two dimensional structure the following:
The kernels are the CUDA functions which execute the parallel instructions. In the functions implemented in this thesis, the “if-else” conditions are used into the kernels in order to be sure that the threads compute on the pixels of the images. The following example, where Area, Width and Height are parameters of the images, shows that:

```c
if (CurrentPixel < Area) {
    // Kernel instructions
}
if (X < Width && Y < Height) {
    // Kernel Instructions
}
```

Left condition is invoked when the argument of the kernel is a one dimension buffer, unlike right condition is used for two dimension buffer arguments.

The arguments of the kernels must be pointers which point to data allocated in device memory or registers. The nomenclature used on these pointers is defined as pDeviceXXXBuffer. Where the prefix pDevice indicates that it is a pointer to device memory, suffix Buffer shows that this pointer contains the information of the host buffer encapsulated inXXX object class. The common pattern to transfer data between host and device is showed below,

```
// Data transfer from host to device memory
pDeviceSourceImageBuffer = GPU_CREATE(float, Area, "ErrorMessage");
GPU_COPYHOSTTODEVICE(pDeviceSourceImageBuffer, SourceImage.m_pBuffer, float, Area, "ErrorMessage");

// Data transfer from device to host memory
GPU_COPYDEVICETOHOST(SourceImage.m_pBuffer, pDeviceSourceImageBuffer, float, Area, "ErrorMessage");
```

However, in the kernels it is necessary to point each data of these buffers separately in order to execute the instructions over these simultaneously. For this purpose, a new pointer is defined as:

```
<datatype>* pCurrentXXXPixel = pDeviceXXXBuffer + CurrentPixel;
```

or in two dimensional case:

```
<datatype>* pCurrentXXXPixel = pDeviceXXXBuffer + (Y * Width + X);
```

Where datatype is the kind of data pointed by pDeviceXXXBuffer. Using this nomenclature the instructions executed over the pointer pCurrentXXXPixel run over the entire buffer simultaneously. This pattern is adequate to be use in case of threads access sequentially to device memory.

```c
const uint X = blockIdx.x * blockDim.x + threadIdx.x;
const uint Y = blockIdx.y * blockDim.y + threadIdx.y;
```
B.2. Texture Memory of GPU

The instructions that define not sequential accesses to device memory (usually 2-Dimension memory access), such as convolution or checking the neighbor of a pixel, cause that threads read to two or more not sequential memory address, see section 4.6. In these cases, the memory access is not coalesced and the reading from the memory is more expensive. Therefore, it must be done using an optimized two dimensional memory, named texture memory. The texture memory is allocated off chip, however, it is cached and optimized for two dimensional reading. This memory is read from kernels using device functions. The texture memory must be accessed using a texture reference which attributes are explained in [CUD11]. In this appendix, a short description of these attributes and examples of the application of them in this work are showed.

A texture reference defines which area of memory or CUDA array is fetched, called a texture memory. It must be declared as a static global variable and it is not possible to be passed as an argument to a function. It can be addressed as a one, two or three-dimensional array using one, two or three texture coordinates respectively. Elements of the array are called texels, short for “texture elements”. A texture reference is declared at file scope as a variable of type texture:

```
texture<DataType, Type, ReadMode> texRef;
```

where:

- **DataType**: specifies the type of data that is returned when fetching texture. It is restricted to integer, floating-point types and any of the 1-, 2-, and 4-component CUDA Built-in Vector Types.

- **Type**: specifies the type of the texture reference. It can be cudaTextureType1D, cudaTextureType2D, or cudaTextureType3D, for a one-dimensional, two-dimensional, or three-dimensional texture, respectively.

- **ReadMode**: maps the returned value. If it is cudaReadModeElementType there is no conversion on it. If it is cudaReadModeNormalizedFloat and DataType is a 16-bit or 8-bit integer type, the value is mapped to [0.0, 1.0] for unsigned integer type and [-1.0, 1.0] for signed integer type.

The texture type is defined in the high-level API as a structure publicly derived from the textureReference type defined in the low-level API:

```
struct textureReference {
    int normalized;
    enum cudaTextureFilterMode filterMode;
    enum cudaTextureAddressMode addressMode[3];
    struct cudaChannelFormatDesc channelDesc;
}
```

- **normalized** specifies whether texture coordinates are normalized or not. If its value is zero, all elements in the texture are addressed in the range [0, width-1], [0, height-1], or [0, depth-1] where width, height, and depth are the texture sizes. In other case, texture coordinates are addressed in the range [0,1].
• **filterMode** specifies the filtering mode, it can be **cudaFilterModePoint** or **cudaFilterModeLinear**. It defines the value returned. If it is **cudaFilterModePoint**, the returned value is the texel which is close to the input texture coordinates. If it is **cudaFilterModeLinear**, the returned value is obtained as the linear interpolation of the two (for a one-dimensional texture), four (for a two-dimensional texture), or eight (for a three-dimensional texture) texels which are close to the input texture coordinates.

• **addressMode** specifies the addressing mode. It it is equal to **cudaAddressModeClamp**, any out of range texture coordinates are wrapped to valid range. If it is **cudaAddressModeWrap** any case out-of-range texture coordinates are clamped to the valid range.

• **channelDesc** describes the format of the value that is returned when fetching the texture; **channelDesc** is of the following type:

```c
struct cudaChannelFormatDesc {
    int x, y, z, w;
    enum cudaChannelFormatKind f;
}
```

where x, y, z and w are equal to the number of bits of each component of the returned value and f is:

- **cudaChannelFormatKindSigned** if these components are of signed integer type,
- **cudaChannelFormatKindUnSigned** if they are of unsigned integer type,
- **cudaChannelFormatKindFloat** if they are of floating point type.

**normalized**, **addressMode**, and **filterMode** may be directly modified in host code.

The following code sample applies some simple lecture from texture memory,

```c
// 2D float texture
texture<float, cudaTextureType2D, cudaReadModeElementType>
    g_TextureSource;

// Simple reading kernel
__global__ void KernelReadTexture2D(float* pDeviceOutputBuffer, int Width)
{
    // Calculate normalized texture coordinates
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;

    // Read from texture and write to global memory
    pDeviceOutputBuffer[Y * Width + X] = tex2D(g_TextureSource, X, Y);
}

// Host code
...
// Allocate GPU memory,
float* pDeviceInputBuffer;
cudaMalloc((void**) &pDeviceInputBuffer, size);

// Copy to device memory some data located at address h_data
// in host memory
cudaMemcpy(pDeviceInputBuffer, pHostBuffer, size,
cudaMemcpyHostToDevice);

// Set texture parameters
g_TextureSource.addressMode[0] = cudaAddressModeClamp;
g_TextureSource.addressMode[1] = cudaAddressModeClamp;
g_TextureSource.filterMode = cudaFilterModePoint;
g_TextureSource.normalized = false;

// Bind the array to the texture reference:
// If the device memory pointer was returned from cudaMalloc(),
// the offset is guaranteed to be 0 and NULL may be passed
// as the offset parameter in cudaBindTexture2D()
const uint Pitch = Width * sizeof(float);
cudaChannelFormatDesc channelDesc =
    cudaCreateChannelDesc(sizeof(float) * 8,
        0, 0, 0, cudaChannelFormatKindFloat);
cudaBindTexture2D(NULL, g_TextureSource, pDeviceInputBuffer,
    channelDesc, Width, Height, Pitch);

// Allocate result of transformation in device memory
float* pDeviceOutputBuffer
cudaMalloc((void**) &pDeviceOutputBuffer, size);

// Invoke kernel
dim3 threads(_TILE_DIM_, _TILE_DIM_, 1);
dim3 grid(Width / _TILE_DIM_, Height / _TILE_DIM_, 1);
KernelReadTexture2D<<<grid, threads>>>(pDeviceOutputBuffer, Width)

// Free device memory
cudaFree(pDeviceInputBuffer);
cudaFree(pDeviceOutputBuffer);
...
B. Implementation Structure

B.3. Structure of Implemented Code

The implementation of the code on this thesis is restricted by CUDA compiler. It only supports a subset of C++ for the device code. However, a full C++ language is supported for the host code. This fact causes that the host code is implemented using C++ language and the device one is implemented on C. The file structure is organized based on this implementation. Each module is programmed in both files, *.cpp and *.cu. Firsts one are compiled by gcc and is used for C++, and seconds by nvcc, it is the NVIDIA compiler and allows to use de C code, see section 4.7.

In order to use together C and C++ language and paradigms, some different kind of functions are defined based on its functionality and they are characterized by the use of prefixes:

- **Kernelxxx()**: these functions are the CUDA kernels and they are the basic units of the parallel programming. They are executed on device, use the __global__ type qualifier and must have void return type. If a pointer must be passed to these as an argument, it must be a pointer to a device memory. The pointers to host memory are not allowed. An example of a kernel definition is showed following:

  ```
  __global__ void KernelAddSample(const byte* pDeviceSourceImageBuffer, const uint Area, DensityFusionPixel* pDeviceDensityFusionPixelBuffer);
  ```

- **Devicexxx()**: the main functionality of these is define the adequate size of the grid and the blocks of threads in order to execute a kernel. They are executed on host but they use CUDA instructions therefore, are implemented in the *.cu files on C language. How it is showed on the following example:

  ```
  void DeviceAddSample(const byte* pDeviceSourceBuffer, const uint Area, DensityFusionPixel* pDeviceDensityFusionBuffer)
  {
     dim3 threads(_BLOCK_DIM_, 1, 1);
     dim3 grid(Area / _BLOCK_DIM_, 1, 1);
     KernelAddSample<<<grid,threads>>>(pDeviceSourceBuffer, Area, pDeviceDensityFusionBuffer);
  }
  ```

- **GPU_xxx()**: are protected members of C++ class. They are the first level between the C++ public functions members of the class and functions implemented on C language. In these ones the Devicexxx() functions are called using as arguments the required variable members of the C++ class. Also, at the end of the C functions execution, .

- **No prefix functions**: These kinds of functions are common members of any C++ class. However, these functions must be able to allocate memory in device, copy data from host to device before the GPU_xxx() are called. After that, the returned values must be passed from device to the variable members again. Finally, this functions have to free memory allocated by both, host and device variables.