Using High Performance Computing to Improve Image Guided Cancer Treatment

Dante Gama Dessavre

August 24, 2012

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2012
Abstract

Radiotherapy is one of the main cancer treatments used today. It is a complex process that relies on finding the cancer in the images of the patients with the most accuracy possible in order to minimize the radiation that the surrounding organs receive. Given that a typical radiotherapy treatment process lasts for 6 weeks, ideally, a system that performs this analysis in real time reliably would enable a better treatment process.

The Western General Hospital and the Edinburgh Cancer Research Center have a prototype system based on texture features analysis of small lung and rectum cancer patients’ images that is currently being developed and used for their research. It was implemented in MATLAB and we used and compared four different technologies to optimize the software: the MATLAB Parallel Toolbox using a small computer cluster, the GPU computing support given by that toolbox, the Jacket add on for GPU computations for MATLAB and finally C/CUDA native function calls from MATLAB, all these GPU technologies were carried on using a NVIDIA Tesla C2050 card.
# Contents

1 Introduction .................................................. 1

2 Background theory ............................................ 3
   2.1 Cancer Treatment ........................................ 3
   2.2 Image Segmentation ..................................... 6
   2.3 Statistical Texture Analysis .............................. 7
      2.3.1 First Order Statistics ............................ 8
      2.3.2 Second Order Statistics ......................... 10
      2.3.3 Higher Order Statistics ......................... 12

3 Design .......................................................... 15
   3.1 Analysis of the Original Software ....................... 15
   3.2 Complexity Analysis .................................... 18
      3.2.1 First Order Statistics ............................ 18
      3.2.2 Second Order Statistics ......................... 20
      3.2.3 Higher Order Statistics ......................... 21
      3.2.4 Theoretical Run Time Comparison ............... 21
   3.3 Parallelization Strategies ............................. 22
      3.3.1 Distributed Memory Cluster ..................... 22
      3.3.2 GPGPU ............................................... 23

4 Implementation ................................................. 25
   4.1 MATLAB Parallel Toolbox ................................ 26
   4.2 MPT/GPU .................................................. 28
   4.3 Jacket ..................................................... 30
   4.4 C/CUDA .................................................... 32
   4.5 Effort Required and Comparison ....................... 35

5 Results ........................................................ 37
   5.1 Micro Benchmarking ....................................... 38
      5.1.1 Individual Function Results ...................... 38
      5.1.2 Execution Time Distribution ..................... 39
   5.2 Scalability Results .................................... 44
      5.2.1 Matrix Size Effects on Data Parallelism ........ 44
      5.2.2 Matrix Size Effects on Task Parallelism ........ 45
5.3 Complete System Results ................................................. 46
  5.3.1 GPGPU Results .................................................. 46
  5.3.2 Cluster Results .................................................. 48
  5.4 Additional results .................................................. 48

6 Conclusions .................................................................. 51

A Statistical Features Formulas ...................................... 54
  A.1 First Order Statistics ................................................. 54
  A.2 Second Order Statistics ............................................ 55
  A.3 Higher Order Statistics .............................................. 57

B Supporting Code ......................................................... 59
List of Tables

3.1 Complexity estimation of the code processing an image of size 512 × 512, . . . 22
3.2 Complexity estimation of the code processing an image of size 20 × 20, . . . 22
List of Figures

2.1 CT Image Example ......................................................... 4
2.2 Typical treatment process .............................................. 4
2.3 Graphical example of volume delineations generated in the planning process. 5
2.4 The checker board and warped checker board textures .......................... 10

3.1 Components of the WGH hospital provided software ............................... 16
3.2 Profiling of the original software with the image in figure 3.4 .................. 17
3.3 Profiling of the original software with the image in figure 3.5 .................. 18
3.4 Example CT scan .......................................................... 19
3.5 Example of typical sub images used of a given scan ............................. 19
3.6 MATLAB Parallel Toolbox Cluster Setup ........................................... 23

5.1 Speed up comparison between strategies and calculations for the FOS Mean, FOS Variance and GLRLM .................................................. 39
5.2 FOS execution time distribution using a big matrix .................................. 40
5.3 FOS execution time distribution using a small matrix ............................ 40
5.4 SOS execution time distribution using a big matrix ................................ 41
5.5 FOS execution time distribution using a small matrix ............................ 41
5.6 HOS execution time distribution using a big matrix ................................ 42
5.7 HOS execution time distribution using a big matrix ................................ 42
5.8 Statistical features groups execution time distribution using a big matrix .... 43
5.9 Statistical features groups execution time distribution using a small matrix ... 43
5.10 Matrix Size Effects on Data Parallelism ............................................. 44
5.11 Matrix Size Effects on Task Parallelism ............................................. 45
5.12 Speed up achieved using big box configuration ..................................... 47
5.13 Speed up achieved using many small boxes configuration ........................ 47
Acknowledgements

I would like to offer my deepest gratitude to Dr. Mario Antonioletti for his guidance and (almost infinite) patience.

I would also like to offer my sincerest gratitude Dr. William Nailon and Dr. George Becket for their support and knowledge during the project.

I would like to thank also my parents and my sister who have always supported me and are the most important persons in my life.
Chapter 1

Introduction

Radiotherapy is one of the main cancer treatments used today, given to one third of the cancer patients [2]. A linear particle accelerator is used to fire ionizing radiation beams at the cancer cells, which causes DNA damage leading to cellular death, in order to kill or control the spread of cancer tumours. The radiation beams are aimed from several angles in order to intersect at the tumour, maximizing the radiation dose there while limiting the radiation in the surrounding (healthy) tissue.

In order to perform the treatment, a radiation treatment planning process is done by a team consisting of oncologists, radiation therapists, medical physicists and dosimetrists in order to have an appropriate plan to perform the best radiotherapy possible. A typical planning process consists of using medical imaging to simulate the actual treatment to plan the geometric and radiological aspects of it, which include beam type, intensity and arrangement. In order to perform the simulation a key part resides in identifying reliably the limits of the cancer tumour.

The process of identifying the different components in an image is called segmentation. This is a very complex process, and there is no single segmentation algorithm suited to every kind of applications, in fact there are many kinds of algorithms suited for different imaging applications (computer vision, facial recognition among others). Many segmentation algorithms have been explored in medical image applications, in particular the Western General Hospital (WGH) and the Edinburgh Cancer Research Center (ECC) have a prototype image analysis system based on two- and three-dimensional texture analysis. The code to this system was provided by Dr. William Nailon, who has is currently performing and overseeing texture analysis research [3].

In the context of this project, a texture is, basically, the visual pattern that covers an image or a segment of it. The most clear example is the black and white squares that cover a checker board, that non surprisingly is known as a checker board texture. The objective of the WGH software is to analyze the textures present on images pertaining to cancer patients (in particular small lung tumour and rectum cancer patients), based mainly on seeing statistical features derived from the image. The simplest example perhaps being the average value of the colors present in the image (or the average of the gray levels in a grayscale image). The purpose of this analysis is to build a knowledge base necessary to classify different regions of interest (related to cancer
tumours and nearby organs) based on their (statistical) texture features. This is done in order to enable a computer based automatic classification system that can have the accuracy required for radiotherapy applications to be constructed.

The aim of this project was to study the use of various parallelization technologies to reduce the execution times of the texture analysis algorithms. As the WGH software was implemented in MATLAB, and as the main focus of the researchers is to explore the algorithms and their results and not spend their time in difficult parallelization or optimization strategies that required too much effort in things like using low level message passing libraries (like MPI), 4 technologies that are either part of MATLAB or can interact with MATLAB were explored:

1. **MATLAB Parallel Toolbox (MPT)** - A MATLAB library that supports shared and distributed memory parallelization utilizing traditional CPUs [4].

2. **MATLAB Parallel Toolbox GPU Capabilities (MPT/GPU)** - The MPT also includes some support for using the GPUs for general computations (GPGPU).

3. **AccelerEyes Jacket** - A GPU computing add-on that predates the MPT/GPU and provides support for execution of more functions than the current native MATLAB offerings [5].

4. **Compute Unified Device Architecture (CUDA)** - MATLAB has the capability to call CUDA codes written in C (CUDA/C). CUDA is a parallel computing platform and programming model to use a Graphics Processing Unit (GPU) for general computations. CUDA/C refers to the use of the language C with CUDA extensions [6].

The process of utilizing the technologies, the speedup results and the difficulty of implementation of all these technologies was studied in order to provide the researchers at the WGH the information they need in order to utilize the technologies that better adapt to their needs.

The theoretical background and context that motivates this work is presented in Chapter 2. Chapter 3 includes the analysis of the software provided and the parallelization technologies that were used in the project. Chapter 4 describes the implementation process and compares the difficulty of using the different approaches. Chapter 5 presents the results of the optimizations performed to the system and their discussion, and finally we present our conclusions in Chapter 6.
Chapter 2

Background theory

This chapter presents the theoretical background and context that motivates this work.

2.1 Cancer Treatment

Radiotherapy is a treatment for cancer that consists in the use of radiation beams to try to kill cancer cells. It can be performed on its own or in combination with other treatments. Its aim is to lethally irradiate the diseased region whilst sparing the surrounding (healthy) tissue. Overexposure of healthy tissues has very serious side effects, like fibrosis\(^1\). Therefore the main objective of radiotherapy can be seen as maximizing the irradiation to the cancer region while minimizing the damage to the surrounding tissue.

In order to perform radiotherapy, localization of the cancer tumours is necessary, therein lies the importance of acquisition of large sets of images using modern medical imaging technologies, such as Computed Tomography (CT) and Magnetic Resonance Imaging (MRI). An example of a typical CT image can be seen in figure 2.1. Treatment planning and review increasingly rely on analysis and manipulation of larger and larger sets of images in order to reveal the precise location, size and shape of cancerous tumours.

A typical treatment process has four main stages, that can be schematically seen in figure 2.2 and lasts for 6 weeks \(^7\). The planning stage consists of the following phases \(^9\):

- The imaging step consists of the images acquisition by the technologies previously mentioned. After this step the images are made available in DICOM format\(^2\).

- A radiologist then creates outlines of both the visible cancerous tumour, called the Gross

\(^1\)Formation of excess fibrous connective tissue in an organ or tissue in a reparative or reactive process which can be detrimental to the organ’s functions.

\(^2\)Digital Imaging and Communications in Medicine (DICOM) is a standard for handling, storing, printing, and transmitting information in medical imaging \(^8\).
Tumour Volume (GTV), which is the target of the treatment, and the nearby organs, which are the ones that could potentially receive collateral damage during the treatment. These are called Organs at Risk (OAR).

- The Clinical Target Volume (CTV) is then identified. It consists of the GTV plus the tissue that contains microscopic malignant disease, and the surrounding tissue considered to be at risk (and therefore requiring treatment). This extension of the GTV is mainly done by a radiation oncologist through clinical examination, the pathology of the tumour and their experience.

- The Planning Target Volume (PTV) is then identified. It consists of the CTV plus an additional margin to minimize the effect of the potential uncertainty when delivering a dose. It can be defined as the CTV plus an additional margin (e.g. PTV + x mm). This margin is based on the specific characteristics of the machine to be utilized, systematic and random uncertainties in the patient positioning system, the immobilization device, internal organ characteristics and motion, and the respiratory motion effects.

One of the focus of modern research is the reduction of the margin in the PTV by using image guided radiation therapy in order to spare the surrounding (healthy) tissue in the most effective way possible. It is noticeable that most OARs usually are very sensitive organs with low tolerance to radiation dose. Besides the delineation of all this volumes, oncologists also specify the prescription dose to the target and maximum dose limitation to the OARs. A diagram exemplifying this volumes can be seen in figure 2.3.
Before the actual radiation treatment, a plan is generated at a treatment planning system\(^3\). This plan specifies the patient’s setup, the radiation beam angle and shape, the modifiers to these beams, treatment time needed to satisfy the dose prescription. This generates the three dimensional dose profile, and a summary of this profile in the form of a dose-volume histogram for plan evaluation.

As mentioned previously, one of the main purposes of computer assisted treatment planning is minimizing the radiation dose that bodies adjacent to cancer tumours receive. However, implementing this technologies represent a time consuming part of the process that most centers follow. Typically the identification of the different volumes in the sets of images is done in workstations by drawing contours around the anatomical sections as perceived by the operators. Software tools supporting this procedure are provided in most commercial treatment planning systems, and use state of the art display and interaction technologies. But even then, this process still is subjective and time consuming, so therefore there is a natural interest in automating this processes in order to reduce the time consumption and improve the accuracy.

Given that the object of work are sets of (medical) images, clearly the field of computer vision has the most relevant subjects to this process. In particular, computer algorithms for the delineation of anatomical structures and other regions of interest are a key component in assisting and automating specific radiological tasks. These algorithms are called image segmentation algorithms. A wide variety of segmentation techniques and methods have been developed, intended for many different imaging applications.

\(^3\)A treatment planning system refers to the whole set of hardware and software involved in all stages of radiotherapy, including the image scanner, the controller of the linear accelerator, among other components.
2.2 Image Segmentation

Not only can computers potentially reduce the time required to identify the different volumes of interest and improve the accuracy, but also with the increasing size (due to resolution improvements) and number of medical images, the use of automated computer systems to facilitate this processing is becoming a necessity.

Formally, image segmentation can be defined as the process of partitioning an image $I$ into $|S|$ distinct subregions or subsets $S_i$, which are homogeneous with respect to a set of defining characteristics. These subsets satisfy the relation

$$S_i \subset I, \quad I = \bigcup_{i=1}^{|S|} S_i, \quad S_i \cap S_j = \emptyset$$  \hspace{1cm} (2.1)

where every $S_i$ is a connected subset. If this constraint is relaxed, then they are called classes. Additionally the constraint of non-intersection can be dropped, depending on what the interests to be accomplished by the image segmentation are.

Image segmentation is one of the most challenging problems in image processing today. For medical purposes it is particularly important for quantifying the progression of diseases and identifying anatomical variations. In general, in medical applications a high degree of accuracy is sought, given the importance of having a correct segmentation (as was motivated in the previous section), and as the anatomical variations can be very small, this can lead them to be confused with the segmentation errors. Thus medical image segmentation is a particularly daunting task. Cancer tumours pose a challenge for producing accurate segmentations and treatment plans, in part because of tumour motion due to natural body movements like respiration. With conventional treatment planning (described in the previous section) significant artifacts can occur related to tumour volume, shape and position changes that can occur during the typical 6 week treatment period.

Investigators have carried numerous intensive studies of image segmentation techniques, developing a wide variety of methods intended for many different applications. John Canny \[10\] introduced the first edge detection method, based on local gradients of the image. Another important type of algorithms are region growing algorithms, these start with an initial region that, through a problem of minimization (or maximization), is grown until it reaches the desire subset. A good review on these algorithms can be found in \[11\].

The characteristics of the existing (and emerging) segmentation methods vary significantly according to the specific application and imaging modality. No single scheme yields acceptable results over the entire spectrum of medical image types that may need to be considered.

A category of algorithms that is the central focus for this work is the field of texture analysis. Humans can easily recognize textures by visual inspection, but they are not a trivial thing to define formally. This difficulty is clear given the number of different definitions given by vision researchers. These definitions depend on the particular application and there is no general consensus or agreed upon definition. Some definitions are perceptually motivate, meanwhile others are driven by the goals of the application area and the algorithms used. A useful defi-
nition of texture for this work is as a function of the spatial variation in pixel intensities (the grey values in grey-scale images). This definition has already been subject to intense study by many researchers.

An immediate application of texture analysis is the recognition of image regions using texture properties. Texture classification is when the texture of an image is used as the most important visual cue for its identification. The goal of texture classification is producing a classification map of the input images where the textured regions can be identified (and therefore segmented) by the texture class that it belongs to.

The ECC and the WGH have an image analysis system based on using statistical texture analysis for identification of different regions on CT images. This has the longer term potential to be used as the basis for the segmentation of these images, or in other words, identifying the shapes of the different volumes of interest.

2.3 Statistical Texture Analysis

As stated previously, texture analysis is an important and useful area of research in machine vision. Texture analysis methods have been utilized in a variety of application domains, which include remote sensing and surface inspection.

The visual process that allows humans to separate objects in figures using texture cues is the basic motivation among psychologists for studying texture perception. This makes the human visual system the base of comparison against which texture analysis algorithms are evaluated.

Julesz has made extensive perception studies in the context of texture discrimination [12]. His aim is to see when two textures in an image can be discriminated from each other given that they have the same brightness, contrast and color. His research focused on the spatial statistics of the grey level in images while keeping the other illumination related properties the same. His work has two fundamental measures:

1. **First Order Statistics (FOS)** - These characteristics are computed from the histogram of pixel intensities in an image. They are a measure of the likelihood of observing a grey value at a randomly chosen location in the image. They depend only on individual pixel values, and not on the interaction or co-occurrence of neighboring pixel values. The average pixel value in an image is the basic example of a FOS.

2. **Second Order Statistics (SOS)** - These are defined as the likelihood of observing two grey values occurring at the endpoints of a needle of random length placed in the image at a random location and orientation. In other words, these are properties of pairs of pixel values.

Two textures may coincide in their FOS values and yet be clearly different. Julesz conjectured
that two textures are not unconsciously or pre-attentively\(^4\) discriminable if their SOS are identical, so to a human, if the SOS values match the textures would appear to be identical at first sight.

The textural properties of an image to be computed and analyzed depend on the application domain in which they are being used. In particular there are many cases of their use in medical image analysis.

For example, Sutton and Hall [13] discuss the classification of pulmonary disease using texture features. Some diseases, like interstitial fibrosis, affect the lungs in a manner that the changes in the images can be seen by the change of texture instead of any geometry changes, which makes texture analysis a clear choice in this area. They used three types of texture features to distinguish normal lungs from diseased ones\(^5\).

Harms et al. [14] used image texture and color features to diagnose leukemic malignancy in samples of blood cells. They extracted the edges of textures and something they called textons between this edges. These textons are regions with almost uniform color. From these they extracted a number of texture features, including total number of pixels that had a specific color, the mean texton radius and texton size for each color, amongst other features. The texture features significantly improved the correct classification rate of blood cell types when compared to pure color analysis.

Other examples include Ladeweerd and Gelsema [15] extracted various FOS as well as SOS to differentiate various types of white blood cells. Insana et al. [16] used textural features to estimate tissue scattering parameters. They used the physical characteristic of the ultrasound imaging process and the tissue characteristics to design the texture model.

In summary, texture features analysis has been used to good success in the field of medical imaging. The software provided by the WGH uses three groups of features to study the images, using both FOS and SOS features, and adding the use of some higher order statistics. The following sections present some of the features it uses and explains them a little. A complete list of the features it uses can be found in Appendix A.

### 2.3.1 First Order Statistics

As was stated in the previous section, FOS measures are statistics calculated from the original image values, and do not consider pixel neighborhood relationships. In particular, a histogram approach is based on the intensity value concentrations on an image or parts of it.

The histogram of intensity levels is a concise and simple summary of the information contained in the image. Its calculation is only based on the values of single pixels, and therefore contains

\(^4\)This means that it would take a human being a closer conscious look to discern between the two textures with identical SOS.

\(^5\)These features are computed based on an isotropic contrast measure, a directional contrast measure, and a Fourier domain energy sampling, they obtained best results using the directional contrast measure.
the first order statistical information about it. Formally, if \( M(i) \) is the number of pixels with intensity \( i \) and \( N \) is the total number of pixels in an image, it follows that the histogram, or pixel occurrence probability, is given by

\[
p(i) = \frac{M(i)}{N}
\]  

(2.2)

, alternatively we denote \( p(i) \) as \( p_i \). The shape of the histogram also provides a way to visually see characteristics of the image. For example, a narrow histogram indicates a low contrast image. A bimodal\(^6\) histogram suggests that the image contains an object with a narrow intensity range against a clearly distinct background.

In the literature a large number of features have been proposed, although they are not independent \([8]\). The WGH software uses 7 FOS. Three of these features are:

1. Mean

\[
\mu = \frac{\sum p_i}{G}
\]  

(2.3)

This measure represents the average value in a texture, for example, in the checkerboard pattern of figure 2.4, the average value would correspond to a 50% grey color.

2. Variance

\[
\sigma^2 = \sum p_i(x_i - \mu)^2
\]  

(2.4)

Indicates how much color variation exists in a texture. A single colored image would have zero variance, and depending how many colors are present in the image the variance increments.

The standard deviation is the root square of the variance and is represented by \( \sigma \).

3. Coarseness

\[
C = 1 - \frac{1}{1 + \sigma^2}
\]  

(2.5)

Represents how fine or coarse an image or texture is. In general a texture composed of large groups of colors will have a bigger coarseness measure than one composed of smaller groups.

The remaining features are listed in Appendix A.1.

\(^6\)Bimodal refers to distribution with two modes.
FOS based texture analysis’ main limitation is the lack of information about the relative position and values of pixels with respect to each other. A clear example of this disadvantage can be seen in figure 2.4. Both images have 50% black and 50% white values, even though they are clearly different images they produce the same level histogram, therefore they also possess the same FOS characteristics and they cannot be distinguished at all using FOS analysis.

2.3.2 Second Order Statistics

Haralick [9] suggested the use of Gray Level Co-occurrence Matrices (GLCM) to study the SOS of an image. These matrices estimate image properties related the SOS by extracting statistical information from the image regarding the distribution of values of pairs of pixels.

A GLCM of a given direction and distance is computed by the distribution of co-occurring values at this offset. In other words, by seeing the values of pairs of pixels separated by the distance defined by the direction and angle, and counting the number of pairs of pixels that possess a given pair of values. An important difference against the FOS is that instead of having an array of size  with the frequency values of this distribution, for each direction and distance we have a matrix of size  with entries that correspond to the frequencies of co-occurrence of pairs of values.

We use the following notation:

-  is the  entry in the GCLM
- The projection onto the x axis: .
• The projection onto the y axis \( p_y(j) = \sum_i p(i, j) \).

• Projections onto the diagonals:

\[
p_{x+y} = \sum_i \sum_j p(i, j)
\]

\[
p_{x-y} = \sum_i \sum_j p(i, j) \quad |i-j| = k
\]

The measures proposed by Haralick, and implemented in the Hospital’s software:

1. Angular Second Moment

\[
f_1 = \sum_i \sum_j p(i, j)^2
\]

(2.6)

This is a measure of the homogeneity of a texture. In a very homogeneous texture there are few grey tone transitions, which is what \( f_1 \) is measuring, unlike a non homogeneous image which has a greater amount of transitions.

2. Contrast

\[
f_2 = \sum_{n=0}^{N} n^2 \{ \sum_i \sum_j p(i, j), |i - j| = n \}
\]

(2.7)

This is a measure of the local variations between pixel values, in other words the checker board pattern in figure 2.4 has a bigger contrast value than an identical checker pattern with the black squares substituted by grey squares, given that the change from white to black are greater than white to grey.

3. Correlation

\[
f_3 = \frac{\sum_i \sum_j (i \cdot j) p(i, j) - \mu_x \mu_y}{\sigma_x \sigma_y}
\]

(2.8)

where \( \mu_x, \mu_y, \sigma_x \) and \( \sigma_y \) are the means and standard deviations of \( p_x \) and \( p_y \).

This correlation measures the the colour value dependencies in the direction given by the offset of the co-occurrence matrix, and provides a more detailed measure to distinguish between images. In particular the checkerboard and warped checkerboard patterns of image 2.4 have different correlations in spite of having the same colour values, given that the correlation is highly sensitive to the exact location of the pixels and not only their value.
4. Variance

\[ f_4 = \sum_i \sum_j (i - \mu)^2 p(i, j) \]  

(2.9)

Unlike the variance in the FOS, which measures the change of pixel values, this variance measures the change of the pixel values in the direction of the offset of the co-occurrence matrix. In other words, it measures the same thing as the FOS variance, but specific to the offset, therefore a group of variances in a number of offsets gives a better characterization of the texture, unlike the FOS variance that cannot distinguish among clearly different textures (like the checkerboard and warped checker board textures).

2.3.3 Higher Order Statistics

In order to produce a more comprehensive texture classification, Higher Order Statistics (HOS) can be used. These refer to measures that contain information of the relationship between three or more pixels.

One way to approach these is seeing how grey level runs behave in an image. One distinction that is not necessarily captures with FOS and SOS features is a definitive distinction between coarse and fine textures. In a coarse texture long grey level runs occur more often, meanwhile fine textures contain predominantly short runs. Given this, Galloway [10] proposed the use of a Run-Length Matrix (RLM or GLRLM for grey scale images).

In this work we focus on grey scan images, for a given image, the \((i,j)\) entry of the GLRLM is defined as the number of runs with pixels of grey level \(i\) and run length \(j\). Therefore, given a direction, the RLM measures how many times there are runs of \(j\) consecutive pixels with the same value, with \(j\) going from 2 to the length of the longest. Even though there can be defined many RLM for a given image, similarly to SOS normally 4 matrices are computed, for the horizontal, vertical and diagonal directions.

Some of the parameters that can be computed from the RLM include the short and long run emphasis (which indicate the coarseness of the textures), which measure the proportion of runs occurring in the image that have short/long length. The fraction of the image in runs is the percentage of image pixels that are part of any of the runs for the matrix computing (i.e. what part of the image contains runs opposed to isolated pixel values).

Tang[11] provides a good summary of some of the features obtained from the GRLM that have been studied.

In the following we use a notation analogue to the one used in the SOS section, replacing \(p\) with \(q\) to denote that we are using the GLRLM instead of the GLCM. Also, \(n_r\) denotes the total number of runs.

---

7A run is a consecutive sequence of pixels in any given direction with the same value/grey level intensity.
Galloway proposed 5 numerical measures to study the RLM:

1. Short Run Emphasis (SRE):

\[ g_1 = \frac{1}{n_r} \sum_i \sum_j q(i,j) \frac{j^2}{f^2} \]  

(2.10)

The value of this measure indicates how much is a texture composed of runs of short length in a given direction.

2. Long Run Emphasis (LRE):

\[ g_2 = \frac{1}{n_r} \sum_i \sum_j q(i,j)j^2 \]  

(2.11)

Similar to the SRE, this measure is an indicative of how much a texture is composed by longer runs. These two measures give a more in depth information of the coarseness of an image.

3. Grey-Level Non-uniformity (GLN):

\[ g_3 = \frac{1}{n_r} \sum_i \left( \sum_j q(i,j) \right)^2 \]  

(2.12)

This is a measure of the uniformity of the (gray) pixel values in a given direction. It is somewhat similar to the SOS Second Angular Moment, but gives a finer level of detail when describing a texture.

4. Run Length Non-uniformity (RLN):

\[ g_4 = \frac{1}{n_r} \sum_j \left( \sum_i q(i,j) \right)^2 \]  

(2.13)

This measure indicates the uniformity of the run lengths, i.e. how much variation there is among the values of run lengths present in an image. A checker board like texture has a low RLN value in the horizontal direction given that all the runs have the same length. This measure is related to how random a texture is.

5. Run Percentage (RP):

\[ g_5 = \frac{n_r}{N} \]  

(2.14)
Even though these measure is not formally a percentage in spite of its name, is a measure of how much do the pixels of a texture are organized in runs and how many pixels are isolated.

We have described what are the theoretical aspects behind the software provided by the WGH. It is based on the analysis of three groups of statistical features (FOS, SOS and HOS) of medical images to characterize medical images of cancer patients. The next chapter focuses on describing how the software works and what was the optimization strategies utilized, taking into consideration both the theoretical aspects and the way the software is implemented.
Chapter 3

Design

3.1 Analysis of the Original Software

The software provided by the Western General Hospital is called DicomGUI. It has additional functionality other than the ones that are the focus of this work. To perform the texture analysis the software follows the following process:

1. The DICOM images to be processed are opened.
2. Boxes that define the areas for which the features are to be calculated are generated. In some of the image sets provided typical box configurations were already provided.
3. The features of the sub images defined by boxes are calculated.

The software consists of two main group of components, one is the GUI code and the other is the files containing the code for the actual texture analysis calculations. These files containing the analysis code are divided in two main groups, first is a file calle DGTextures.m that controls the calls to the files performing the individual feature calculations, these last contained in the files fos.m, haralick.m and glrlm.m corresponding to the calculation of the FOS, SOS and HOS features respectively. These can be seen in image 3.1. Even though there is a clear idea to the organization of the code, it is important to note that the different layers are not completely abstracted from each other, with the GUI code executing functions that would in theory be done i the DGTextures.m file, like dividing the images using the boxes defined by the user.
The code was profiled, using the MATLAB profiler, in order to identify the regions that could benefit the most from parallelization. In order to have a better understanding, given the difference in size and number of images derived from different box configurations, two sets of image were used to profile the code:

1. A set of images in which there was only one box per image covering the entire region, see figure 3.4. In other words, the calculations were done over entire images. The profiling can be seen in figure 3.2.

2. The same set of images but the features are calculated in many small sub images derived by an a priori made box configuration by researches in the WGH, seen in figure 3.5. The profiling can be seen in figure 3.3.

In the profiles, disregarding the execution time corresponding to the GUI features (i.e., observing the functions corresponding to the statistical features calculations), the higher order statistics dominated the execution time by far, with upwards of 70% of the time of execution. Given that the higher order statistical features are not inherently more complex to calculate than the first or second order ones (analyzed in the next section), this clearly suggests that the matrix calculation is a particularly expensive step of the process, much more so than the co-occurrence matrix calculation for the SOS, therefore a deeper analysis of the process and potential parallelization was required, which is presented in the next section.

---

1 The profile images are screenshots of the MATLAB profiler.
### Profile Summary

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Calls</th>
<th>Total Time</th>
<th>Self Time*</th>
</tr>
</thead>
<tbody>
<tr>
<td>glslm</td>
<td>88</td>
<td>768.328 s</td>
<td>768.328 s</td>
</tr>
<tr>
<td>boxcount_bill</td>
<td>88</td>
<td>74.160 s</td>
<td>73.913 s</td>
</tr>
<tr>
<td>fos</td>
<td>88</td>
<td>49.565 s</td>
<td>49.565 s</td>
</tr>
<tr>
<td>graycomatrix&gt;computeGLCM</td>
<td>176</td>
<td>4.271 s</td>
<td>4.107 s</td>
</tr>
<tr>
<td>DicomGui&gt;pushbutton1_Callback</td>
<td>1</td>
<td>10.784 s</td>
<td>4.095 s</td>
</tr>
<tr>
<td>inline</td>
<td>72510</td>
<td>5.363 s</td>
<td>3.990 s</td>
</tr>
<tr>
<td>impixelinfo</td>
<td>10</td>
<td>3.708 s</td>
<td>3.344 s</td>
</tr>
<tr>
<td>images/private/poly2edgelist</td>
<td>263</td>
<td>8.695 s</td>
<td>3.332 s</td>
</tr>
<tr>
<td>regionprops</td>
<td>789</td>
<td>8.355 s</td>
<td>2.804 s</td>
</tr>
<tr>
<td>ind2sub</td>
<td>526</td>
<td>2.499 s</td>
<td>2.499 s</td>
</tr>
<tr>
<td>DicomGui&gt;CentroidInformation_Callback</td>
<td>1</td>
<td>20.007 s</td>
<td>1.901 s</td>
</tr>
<tr>
<td>dialog</td>
<td>6</td>
<td>2.370 s</td>
<td>1.889 s</td>
</tr>
<tr>
<td>DicomGui&gt;menu_tools_compute_Callback</td>
<td>1</td>
<td>904.245 s</td>
<td>1.666 s</td>
</tr>
<tr>
<td>flipud</td>
<td>68578</td>
<td>1.373 s</td>
<td>1.373 s</td>
</tr>
<tr>
<td>interp2</td>
<td>88</td>
<td>1.525 s</td>
<td>1.138 s</td>
</tr>
<tr>
<td>diicominfo&gt;convertRawAttr</td>
<td>10278</td>
<td>2.241 s</td>
<td>1.058 s</td>
</tr>
<tr>
<td>..._gmodel&gt;imagemodel.getNumberFormatFcn</td>
<td>318</td>
<td>1.068 s</td>
<td>0.986 s</td>
</tr>
<tr>
<td>iptformats/private/dicomparse (MEX-file)</td>
<td>147</td>
<td>0.950 s</td>
<td>0.950 s</td>
</tr>
<tr>
<td>Haralick_features</td>
<td>176</td>
<td>0.739 s</td>
<td>0.739 s</td>
</tr>
</tbody>
</table>

Figure 3.2: Profiling of the original software with the image in figure 3.4.
### Profile Summary

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Calls</th>
<th>Total Time</th>
<th>Self Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>glrim</td>
<td>3093</td>
<td>187.345 s</td>
<td>187.345 s</td>
</tr>
<tr>
<td>inpolygon &gt; vec_inpolygon</td>
<td>1098</td>
<td>37.156 s</td>
<td>37.166 s</td>
</tr>
<tr>
<td>Haralick features</td>
<td>6185</td>
<td>24.434 s</td>
<td>24.074 s</td>
</tr>
<tr>
<td>boxcount_bill</td>
<td>3093</td>
<td>14.792 s</td>
<td>8.623 s</td>
</tr>
<tr>
<td>fos</td>
<td>3093</td>
<td>6.130 s</td>
<td>6.130 s</td>
</tr>
<tr>
<td>intline</td>
<td>72510</td>
<td>5.393 s</td>
<td>4.157 s</td>
</tr>
<tr>
<td>DicomGui&gt;pushbutton1_Callback</td>
<td>1</td>
<td>10.337 s</td>
<td>3.822 s</td>
</tr>
<tr>
<td>images/private/poly2edgelist</td>
<td>263</td>
<td>9.145 s</td>
<td>3.748 s</td>
</tr>
<tr>
<td>ind2sub</td>
<td>526</td>
<td>3.301 s</td>
<td>3.301 s</td>
</tr>
<tr>
<td>regionprops</td>
<td>789</td>
<td>9.630 s</td>
<td>3.015 s</td>
</tr>
<tr>
<td>DicomGui&gt;CentroidInformation_Callback</td>
<td>1</td>
<td>23.007 s</td>
<td>2.904 s</td>
</tr>
<tr>
<td>interp2</td>
<td>3093</td>
<td>6.651 s</td>
<td>2.668 s</td>
</tr>
<tr>
<td>DQfeatures</td>
<td>3093</td>
<td>246.637 s</td>
<td>2.643 s</td>
</tr>
</tbody>
</table>

Figure 3.3: Profiling of the original software with the image in figure 3.5

### 3.2 Complexity Analysis

In order to understand the difference between both runs, a deeper analysis of the calculations was required. The software scales the images for each group of features, therefore we denote this number of scaled gray values as $G_{\text{FOS}}$, $G_{\text{SOS}}$ and $G_{\text{HOS}}$ for each respective group.

#### 3.2.1 First Order Statistics

Calculation of the FOS is fairly straightforward, which consists on two steps:

- The histogram is an array of size $G_{\text{FOS}}$ that contains the frequencies of each level of gray. In order to get the histogram each of the image values needs to be visited once, therefore the number of operations is $N \times M$.
- All of the FOS features are then calculated doing single sums over the values of the
Figure 3.4: Example CT scan.

Figure 3.5: Example of typical sub images used of a given scan.
histogram (the formulas can be seen in Appendix A.1), therefore the complexity is $O(G_{FOS})$ for each one.

Overall the complexity of the FOS calculation, given $n_{fos}$ number of FOS features, is:

$$O(N \times M + n_{FOS} \times G_{FOS})$$

(3.1)

The software is configured to have the image scaled to $2^6$ grey values. In particular, it reduces the images we worked with from $2^{16}$ grey values initially to $2^6$. An alternative way to state this reduction is that the resolution from the 16 bit image was dropped to 6 bits. Therefore the software uses a $2^6$ sized array to store the histogram, instead of one with size $2^{16}$.

### 3.2.2 Second Order Statistics

The SOS go through an analogue, if somewhat more complex, process:

- The co-occurrence matrices are calculated, which is a similar algorithm to the histogram in the FOS. Given an image with $G_{SOS}$ levels of gray, the dimension of a co-occurrence matrix corresponding to a given offset is of dimension $G_{SOS} \times G_{SOS}$. In order to calculate the entries, each value of the image is visited and then the value of the image corresponding to the offset is checked, and the value corresponding to that pair of gray scale values is incremented. Therefore the calculation of the matrix also consists of $N \times M$ operations.

- Since most of the SOS features correspond to sums over the values of the matrix (seen as double sums in their formulas in Appendix A.2) they possess a complexity of $O(G_{SOS}^2)$ with two notable exceptions:
  - Contrast: Corresponds to a triple sum through the number of gray scale values (see Appendix A.2), therefore it takes $O(G_{SOS}^3)$ operations.
  - Maximal Correlation Coefficient: Consists in getting the second largest eigenvalue of a matrix that costs $O(G_{SOS}^3)$ operations to build (see Appendix A.2), therefore it is also $O(G_{SOS}^3)$.

The software calculates two matrices (corresponding to two offsets) by default. Changing what offsets are used would be undistinguishable performance wise, since the same procedure of having to calculate the histogram of the pairs of values is done in the same way for any offset. Unlike what happened in the FOS features, the image is scaled to 3 bits (i.e. to $2^3$ gray values). Thus, the co-occurrence matrix corresponding to a given offset has dimension $8 \times 8$. The software calculates matrices for two offsets.

Overall the complexity of the FOS calculation, given $n_{fos}$ number of FOS features and $n_{glcm}$ the number of co-occurrence matrices built, is:
\begin{equation}
O(n_{glcm} \times N \times M + (n_{SOS} - 2) \times G_{SOS}^2 + 2 \times G_{SOS}^3) \tag{3.2}
\end{equation}

3.2.3 Higher Order Statistics

The higher order statistics, or run length features calculations process is very similar to the previous ones, consisting of two main steps:

- Initially, the horizontal run length matrix is defined having dimensions of \( M \times G_{HOS} \). It is calculated by iterating through each row, counting the number of run lengths by visiting each pixel, and if there is a change from one pixel to the next incrementing the corresponding entry to the gray value and the run length. Therefore it also is a \( O(N \times M) \) calculation, although it is potentially significantly slower than the previous structure calculations (the histogram in FOS and the co-occurrence matrix in SOS), since it contains several conditional statements to see the value change to determine the run lengths (overall the original code uses approximately 14 instructions per loop iteration), therefore it is modeled as \( O(14 \times N \times M) \).

- The calculation of the run length matrices is an analogue process to the FOS features, being sums over the entries of the run length matrices, also taking \( O(N \times M) \) operations each. The feature calculations all consist of summations on the elements of the GLRLM, therefore they have complexity \( O(M \times G_{HOS}) \) per run length matrix.

The software scales the images to 6 bit values for the higher order calculations, the same number as for the FOS calculations.

Overall the complexity of the FOS calculation, given \( n_{hos} \) number of FOS features and \( n_{rlm} \) the number of run length matrices built, is:

\begin{equation}
O(n_{rlm} \times 14 \times 4 \times N \times M + n_{rlm} \times n_{hos} \times M \times G_{sos}) \tag{3.3}
\end{equation}

3.2.4 Theoretical Run Time Comparison

Using the actual data of the images used for the profiling of section 3.1, with dimensions \( M = N = 512 \), we reach the approximation in table 3.1.

Clearly the higher order are considerably more expensive, which was what was observed in both profiles.

To see the difference between the FOS and SOS calculations in both profiles there are a few important factors. First the co-occurrence matrix calculation is a built in MATLAB function, that means that it is very likely to be heavily optimized in comparison. Additionally in table 3.2 it can be observed what happens with the number of operations with the small matrices.
Complexity formula: $O(N \times M + n_{FOSS} \times G_{FOSS})$

Operations estimation: $O(7 \times 32 + 512^2) \approx O(262,000)$

Table 3.1: Complexity estimation of the code processing an image of size $512 \times 512$.

Even supposing that the cost of calculating the co-occurrence matrix was negligible, the SOS calculations would still take $O(2600)$ operation, which means that the optimized GLCM calculation does not make up for the feature calculation difference.

After this analysis it two main things were learned: First, the calculation of the GLRLM is a very important and time consuming step of the process, the calculation of the GLCM done by MATLAB is very efficient, and the percentage of time that the different feature groups take is different depending upon the input images.

### 3.3 Parallelization Strategies

This section describes the parallelization strategies and technologies used. The software was ran on two systems. The main system had the following specifications: 4 six core Xeon X5650 at 2.67 Ghz, 48 GB total RAM and 4 Tesla C2050 GPUs with 3GB of VRAM each. The second system for testing purposes was the hospital cluster consisted of a number of (different) workstations connected through ethernet.

#### 3.3.1 Distributed Memory Cluster

A distributed memory system refers to a multi processor system in which each processor has its own private memory and can only operate on local data. To work with remote data on the

2This computer is the node *fermi0* in the hydra machine at EPCC.
other processors communication between the processes is necessary.

In order to run MATLAB in parallel in a cluster two additional components are required: MATLAB Parallel Toolbox and the MATLAB Distributer Computing Server software. They are additional components that can be purchased as add ons to a standard MATLAB system. They provide explicit parallel constructs such as parallel for loops and code blocks, distributed arrays, some parallel numerical algorithms and message passing functions.

In MATLAB terminology, the section of the code to be parallelized is called a job, with is broken into segments called tasks. The MATLAB session that invokes the program is called the client session. It uses the MPT to define the jobs and tasks in order to run them either on a cluster or on the local machine. The Distributed Computing Server, as it name implies, is the responsible for the execution of the job in a cluster. This can be graphically seen in figure 3.6.

The first parallelization strategy was using the MPT in a distributed memory environment. In principle this seemed like it was going to be the easiest optimization strategy, and the WGH already had a cluster, whose main purpose was running MPT codes, even though each node is also used by the researchers and personal as their personal workstations. It is a small sized cluster consisting of a number of workstations linked by ethernet. It does not contain any other distributed memory parallel library (like MPI), and the workstations are actually regular computers.

3.3.2 GPGPU

The other strategies were focuses in using the general computation capabilities of modern GPUs (i.e. GPGPU strategies).

The MPT provides some support for using GPUs, once again this provided a path with the least complications while providing a reasonable acceleration.

---

3The cluster was initially built to have 32 cores. At the moment of this work 14 cores were connected and operational.
The next strategy was to use the Jacket add-on for MATLAB. Jacket is a GPU add-on for MATLAB code produced by a company called AccelerEyes. According to them, an a number of third party reports and benchmarks [17] it promised to be faster than the MPT/GPU facilities, while supporting a much broader set of functionality. It is developed by a company called AccelerEyes, and it includes a comprehensive set of GPU based operations including: matrix manipulation, linear algebra functions and interfacing with programs written in other languages. Even though we did not used this last capability it is important to know it is a possibility that could be used. Jacket promised a much better acceleration than the MPT/GPU capabilities, while also supporting a (much) broader range of built in functions that could potentially improve the ease of implementation.

The last strategy utilized was to use CUDA/C calls from within the MATLAB code. Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by NVIDIA for general processing in graphics processors. C/CUDA refers to C with NVIDIA extensions and certain restrictions to code algorithms for execution on the GPU. MATLAB supports making calls to external C/CUDA files in order to maximize the acceleration for the most important segments of code. The main characteristic of coding for GPUs is that they have a parallel throughput architecture that emphasizes executing many concurrent threads (even though each one can be executed potentially slower than a CPU thread), rather than executing a single thread quickly (like traditional CPUs). Clearly this strategy had the potential to be the fastest, while being the hardest to implement, probably in a very significant way in both.
Chapter 4

Implementation

This chapter describes in detail the implementation process and details of the parallelization strategies described in the previous chapter. This is of particular importance for the researchers, since the algorithms are part of active research and therefore can (and potentially) change in order to accommodate their needs. So, the description of how to implement the parallelizations and optimizations could prove to be very valuable for them.

The most important decision about the calculations is whether to use task parallelism or data parallelism. This is due to the difference of the execution timings of the different sections, seen in the previous chapter, and potential acceleration when calculating the matrices and features of a whole image (or big boxes) when compared to calculating them when dealing with small boxes (see figures 3.4 and 3.5).

Overall the software originally included the calculation of 26 features, of which 25 were optimized in this work. The calculation of the Maximal Correlation Coefficient, seen in Appendix A.2, that consists of calculating the eigenvalues of a particular matrix wasn’t done mainly because the calculation of eigenvalues is a very studied problem, and for every technology pre-made code would’ve been used (all MATLAB libraries have the capability built in and for CUDA many options already made exist). This was to focus the development time on the other 25 features that required manual implementation of the calculations.

In order to understand the implementation of the different kinds of functions and compare them, three feature calculations were used in this chapter to illustrate the process that each strategy requires. The functions were:

• FOS: Mean

The calculation of the mean, as was done in the software, was an example of the simplest kind of operations present in the project. The main importance of it is that it consists of using built-in MATLAB functions (sum) and .\*\[1\]. The original code was:

\[1\] .\* denotes a per entry multiplication in MATLAB, unlike * that denotes a complete matrix multiplication.
Listing 4.1: Original FOS Mean calculation code.

```matlab
1  % P_fo is the histogram of the image
2  mea = sum(P_fo.*(0:(grey_levels-1)));
```

- **FOS**: Variance

  The variance is a function that already includes looping, and therefore represents a very significant portion of the functions. The original code was:

Listing 4.2: Original FOS Variance calculation code.

```matlab
1  varianc = 0.0;
2  for b = 0:(grey_levels - 1),
3      varianc = varianc + (((b - mea) * (b - mea)) * P_fo(b+1));
4  end
```

It is noticeable that the SOS are analogue to the calculation of this variance, except that instead of working with the histogram they work in the co-occurrence matrix, but their implementation is similar. Additionally, the fact that the calculations are sums make them reduction operations.

- **GLRLM**: Run Length Matrix construction

  The construction of the GLRLM consists of double looping and many conditionals, with significantly more instructions than most of the calculations. Also an important difference is that it produces another matrix instead of a scalar value. The code can be seen in Appendix B.

### 4.1 MATLAB Parallel Toolbox

The MATLAB Parallel Toolbox has a rather simplistic approach to both CPU and GPU parallelization. CPU wise it supports both data and task parallelism via its parfor and spmd\(^2\) constructs.

The parfor is a construct that comes with the MPT that enables splitting computations into different workers. This can be accomplished on either a single multicore CPU or via a cluster of connected CPUs. The spmd is an MPT construct that enables bigger control over what each of the workers executes. Unlike the parfor loop, the workers used for an spmd statement each have a unique value identifying them, which allows programmers to customize the execution of each worker. In our case it allowed us to execute the task parallelization. One key difference is that the parfor loop distributes automatically the loop iterations among the workers, in-

---

\(^2\)spmd stands for *single program multiple data*, and also supports data parallelism with a greater degree of control than parfor, but we only used it for task parallelism.
cluding the data distribution necessary, meanwhile for the \texttt{spmd} the data is distributed by the programmer.

But prior to running a parallel task, the \texttt{matlabpool} must be configured. The \texttt{matlabpool} command enables the parallel language features mentioned above by starting a parallel job that connects this MATLAB client to a number of workers.

For data parallelism a sub image is divided among different elements in a cluster, and given that the features are reductions, this implies a significant amount of communication required for their calculations (since the data of all the elements of the cluster is needed to performed said reduction). Given that the GWH's cluster is connected by ethernet and that the machines are not even in the same room, the communication cost present a very significant challenge for data parallelism. Therefore for the small box size simulations, a task parallelism is the clear choice. Nonetheless data parallelism was also explored for the bigger box sizes.

The use of \texttt{parfor} is pretty much straightforward. For example, to split the calculation of the variance in the FOS features, the for loop is simply substituted by the corresponding \texttt{parfor}:

```
Listing 4.3: Data parallel MPT FOS Variance calculation code.
1 % The \texttt{for} is replaced \texttt{for} \texttt{parfor}, which allows \texttt{using} reduction
2 % variables, in \texttt{this case} \texttt{varianc}, which enables the code to
3 % stay the same
4 varianc = 0.0;
5 parfor b = 0:(grey_levels - 1),
6     varianc = varianc + (((b - mea) * (b - mea)) * P_fo(b+1));
7 end
```

The \texttt{spmd} code was:

```
Listing 4.4: Task parallel MPT FOS Variance calculation code.
1 % Now the histogram is distributed to the workers
2
3 % Each worker calculates the variance of its corresponding image
4 spmd
5 % The code inside the spmd is the same as the serial version
6 varianc = 0.0;
7 for b = 0:(grey_levels - 1),
8     varianc = varianc + (((b - mea) * (b - mea)) * P_fo(b+1));
9 end
```

The process of porting regular MATLAB code into MPT code is a very straightforward process, therefore there is no need to put more examples in this section.

Even though we explored the usage for both task and data parallelization, given the WGH
cluster configuration and typical smaller box sizes that the researchers tend to use, it was deemed that the benefit of decomposing the matrix calculation using data parallelism was very limited, and therefore we implemented only a task parallel version.

4.2 MPT/GPU

The MPT also has support for using the GPU for accelerating MATLAB code.

In order to tell MATLAB to execute a piece of code in the GPU some things need to happen: the data transfer from CPU to GPU memory is done using the function GPUArray, and, either the code consists of only the MPT/GPU supported functions or the code needs to be isolated in a function and it must be called by using arrayFun: \[ \text{arrayfun}(\text{func}, \text{A1}, \ldots, \text{An}) \] calls the function \( \text{func} \), giving it the parameters \( \text{A1}, \ldots, \text{An} \) and receiving the results in \( \text{B1}, \ldots, \text{Bn} \).

The FOS mean calculation needs an additional function (to be called using arrayFun) even though \text{sum} is one of the functions supported to be executed on the GPU. The reason for this is the matrix operation done before the sum, which exemplifies the added complexity of needing additional functions. So the external function would be:

\[ \text{arrayfun}(\text{func}, \text{A1}, \ldots, \text{An}) \]

The pre configured data set used overlapping sub images of dimensions 20 × 20.

The list of supported functions can be seen at http://www.mathworks.co.uk/help/toolbox/distcomp/bsic4n4-1.html, when this functions are called using GPUArray parameters instead of regular ones, MATLAB automatically executes them in the GPU.
P_fo_gpu = arrayfun(@gpu_mean, P_fo_gpu, gr_lv);

% The sum of the result of the product is then done in the GPU, since sum is a GPU supported function.
mea = sum(P_fo_gpu);

There are some tradeoffs to this approach, starting with the need for the two additional arrays that occupy significant portions of the video memory, or only one additional array and modification of P_fo in the GPU which means that we would need to copy it again from the CPU to the GPU to use it (which is the case since it is reused in other statistics).

Even though this function is a good candidate for the MPT, the difficulties in using it can be seen from the fact of requiring an additional external function, along the memory management and the different way of calling the function.

Notably this statistic can also be calculated as follows:

Listing 4.7: Alternative MPT/GPU FOS Mean calculation

% The mean function of a matrix returns a vector
% containing the average of each column, therefore
% mean(mean(matrix)) returns the average of all the values
mea = mean(mean(image_in))

It is not used in the original version since it is less efficient (according to the original software documentation), but it is much easier to port to a MPT/GPU code, as can be seen:

Listing 4.8: Alternative MPT/GPU FOS Mean calculation

% The only additional step needed is transferring the data to the GPU since mean of a GPUArray returns a GPUArray
image_in_gpu = gpu_array(image_in);
mea = mean(mean(image_in_gpu));

Since the mean function is another function that automatically executes in the GPU if it is called with a gpuArray, the porting was very easy since only the data transfer was necessary.

It is very important to note that the MPT does not support a distributed for loop execution entirely on the GPU, so when encountered with these kind of codes there are three options, either use the parfor construct (which is intended for clusters with multiple GPUs) or convert the for loop into matrix/vector operations which can be quite challenging or impossible in the case when iterations are not independent. Therefore the operation inside the for loop is divided in two, the matrix multiplication and the sum. The matrix multiplication is put into an external function as follows:
% In the original variance calculation each loop calculates
% the variance by doing
% \( \text{variance} = \text{variance} + ((b - \text{mea}) \times (b - \text{mea})) \times P_fo(b+1); \)
% This function corresponds to \( (b - \text{mea}) \times (b - \text{mea}) \times P_fo(b+1) \)

function [o1, o2] = gpu_variance(P_fo, gr_lv)

Then, since \text{sum} execution is done in the \text{GPU} if it is called with a \text{gpuArray} as argument, in the FOS function the code would be:

Apart from the added effort of changing the code to adapt it to the MPT/GPU execution, the acceleration also suffers slightly.

But there is one more significant limitation to the \text{arrayfun} construct of \text{MATLAB}: The call \( A = \text{arrayfun}(\text{FUN}, B) \) applies the function specified by \text{FUN} to each element of the \text{GPUArray} \( B \), and returns the results in \text{GPUArray} \( A \). \( A \) is the same size as \( B \), and \( A(i,j,...) \) is equal to \( \text{FUN}(B(i,j,...)) \) element wise. \text{FUN} is a function handle to a function that takes one input argument and returns a scalar value, i.e. the function is applied to an array element. This makes the possibilities of what can be calculated with the MPT/GPU rather limited, in particular matrix constructions, like the run length matrix, cannot be done.

### 4.3 Jacket

While the MPT/GPU supports 172 native functions for GPU execution\(^5\) Jacket supports more than 500. The latest version is 2.2, although the version installed in \textit{fermi0} is 1.8.2. Also, in addition to supporting NVIDIA CUDA devices, it adds supports to Intel and AMD OpenCL ones.

In order to start using Jacket with \text{MATLAB}, the path to the installation of Jacket is given to \text{MATLAB}:

\(^5\)As of release R2012a.
addpath <jacket_root>/engine

Where <jacket_root> is the folder that contains the Jacket installation, in particular in fermi0 the command is:

addpath /usr/local/jacket/engine

Finally, a very important advantage is that it has a parallel for loop, called gfor, lack of which is one of the main drawbacks of the MPT/GPU as was seen in the previous section. Additionally it is not necessary to put the element wise operations (like .*) in external functions, they can be used as they would normally in MATLAB, and as long as they are applied to data on the GPU, the execution is done there. This means that external functions are not necessary, and therefore the porting process is significantly simpler than the equivalent MPT/GPU one:

Listing 4.11: Jacket FOS Variance function

```matlab
1 % The transfer from CPU to GPU memory is done using gdouble()
2 X_gpu = gdouble(image_in);
3 P_fo_gpu = gdouble(P_fo);
4 grey_level_vector = gdouble(0:(grey_levels-1))
5
6 % The mean calculation remains the same as the original one
7 mea = sum(P_fo.*grey_level_vector);
```

Even though the added capability is clear in comparison to the MPT/GPU there is one significant drawback: the gfor construct does not support simple reduction operations (unlike parfor in the MPT). Therefore we needed to use temporary arrays/matrices in these kind of functions, which not only is suboptimal from the programmer’s point of view (requiring extra work to have temporary vectors before doing reduction operations), it also affects the potential acceleration obtainable.

Listing 4.12: Jacket FOS Variance function

```matlab
1 % A temporary vector to store the values of the loop before the sum
2 varianc_vector = gdouble(zeros(grey_levels));
3 gfor b = 0:(grey_levels),
4     variance_vector(b) = (((b - mea) * (b - mea)) * P_fo(b));
5 gend
6 varianc = sum(variance_vector);
```

On the other hand the existence of the gfor construct allows the parallelization of the matrix calculations. Even though the restriction of no nested gfor loops meant that it is not possible to simply port it as is, we found an elegant way of using the gfor for the task parallelism. Given a set of sub images (potentially overlapping) with common dimensions among them, we constructed a 3d array in which each slice of the array corresponds to each of the sub images. Then we used the gfor loop along this dimension in order to make the GPU process the sub images in parallel. This naturally arises the problem that most of the features possess
reductions, which limit the potential acceleration that the GPU can provide. Yet, as we will see in the next chapter, depending on the size of the sub images, if we process enough images in parallel the total time of execution significantly decreases. Remembering from last chapter’s profiling that the run length matrix calculation is a particularly expensive part of the system, this proved to be a key optimization as we show in the next chapter.

The disadvantage of this task parallel scheme is that it would require significant changes to the original DicomGUI software in order to integrate it (the changes are even to the GUI code since it is not completely separate to the other components). Given that the version that we posses of DicomGUI is already outdated, and significant portions of the research is done through scripting, it was not very useful to integrate the accelerations into the GUI.

4.4 C/CUDA

This parallelization path is the one that promised the best performance, given that it deals directly with the GPU hardware, and also avoids the limitations of MPT/GPU and Jacket, since it makes available all the capabilities of the GPU to the programmers.

NVIDIA GPUs have a two level hierarchy. Each GPU is made of multiple Stream Multi-processors (SMs), each featuring multiple simple cores (usually called CUDA cores). This architecture is then abstracted as a grid of thread blocks. These blocks map onto the SMs, and the threads map onto the cores. The programmers are encouraged to oversubscribe threads and blocks, and let the device perform the schedule automatically. This is done in order to hide the memory latency, and also code programmed like that will be portable (and potentially efficient) across different GPU versions and architectures. The code executed by the threads is called a kernel, and threads on the same block execute the same kernel.

CUDA extends C, defining a new syntax for kernel implementation and launching. The __global__ specifier defines a kernel, that when called runs in parallel in CUDA threads on the GPU. New delimiter «<...>» syntax is used to specify the grid and block dimension when making the kernel call.

Since we are dealing with C, proper GPU memory management must be done by the programmer (i.e. allocating and freeing things), although the same principle of having to transmit data to the GPU and then collect it as was the case with both MPT/GPU and Jacket still holds.

An important feature that needs to be implemented for almost all the features is a sum reduction operation. We used an open source code found in Google Code as a basis for this reduction operation

66This code corresponds to a Stanford programming course, and is distributed under the Apache License. It can be found in http://code.google.com/p/stanford-cs193g-sp2010/source/browse/trunk/tutorials/sum_reduction.cu.
Listing 4.13: C/CUDA reduction function

```c
// Algorithm input: *input - the data to be summed
//                  * n - the size of the data (block wise)
// Algorithm output: *per_block_results = the sums of the block

__global__ void block_sum(const float *input,
                          const size_t n,
                          float *per_block_results) {

  // Create a shared array to hold the input data
  extern __shared__ float sdata[];
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

  // load input into shared memory
  float x = 0;
  if (i < n)
    x = input[i];
  sdata[threadIdx.x] = x;

  // Wait until all threads load their data to the shared memory
  __syncthreads();

  // Each thread sums elements that are contiguous
  for(int offset = blockDim.x / 2;
      offset > 0;
      offset >>= 1){

    // add a partial sum upstream to our own
    if(threadIdx.x < offset){
      sdata[threadIdx.x] += sdata[threadIdx.x + offset];
    }

    // wait until all threads in the block have
    // updated their partial sums
    __syncthreads();
  }

  // thread 0 writes the final result
  if(threadIdx.x == 0)
    per_block_results[blockIdx.x] = sdata[0];
}
```

There are libraries like Thrust and CUDPP\footnote{CUDPP is the CUDA Data Parallel Primitives Library.} that contain auto tuned, very well optimized of commonly used functions like the reduction, probably a lot better optimized than what we
achieved here, that could be used. We didn’t opt to use it only to avoid adding complexity to the already complex process of using CUDA with MATLAB in the timeframe for this project.

In order to perform the calculation of the FOS mean we need to do a process analogue to what we did with the MPT/GPU, which is having a kernel for multiplying and then performing the reduction. The code for the kernel doing the matrix multiplication is:

Listing 4.14: MPT/GPU variance function

```c
__global__ void mean(double *mean_vector,
                       const double *P_fo,
                       const double *grey_levels,
                       const int n)
{
    int idx = threadIdx.x;

    // Each thread calculates one entry of the multiplication
    if (idx<n)
        mean_vector[idx] = P_fo[idx] * grey_levels[idx];
}
```

The code for each kernel is contained in a .cu file, reduce.cu and mean.cu respectively. This files are then compiled into .ptx files using nvcc, the NVIDIA CUDA compiler. Afterwards, the code to call the kernels from MATLAB is:

Listing 4.15: MPT/GPU variance function

```matlab
k1 = parallel.gpu.CUDAKernel('reduce.ptx', 'reduce.cu');
k1.threadBlockSize = N;
k2 = parallel.gpu.CUDAKernel('fos_mean.ptx', 'fos_mean.cu');
k2.threadBlockSize = N;
mean_vector = feval(k1, P_fo, [0:(grey_levels-1)]);
mean = feval(k2, image, image_size, 32);
```

The code for calculating the variance is be very similar, since in this case the fact that the original mean code calculated via vector operations did not matter, since when translated to CUDA we used the equivalent loop version, which is analogue to the variance code:

Listing 4.16: MPT/GPU variance function

```c
// The original code was
// varianc = varianc + (((b - mea) * (b - mea)) * P_fo(b+1))
// This kernel calculates the vector operation
```
The calculation of the GLRLM was a bigger challenge since it is considerably more complex. The way it was achieved was by making each thread process a row/column/diagonal line of the image (depending upon the particular GLRLM being constructed), given the existent dependencies upon previous values in said row/column/diagonal. Even though this process made the construction of a single GLRLM in the GPU a slower process, it allowed the construction of many GLRLMs in parallel to overcome this deficit (assigning one block). The CUDA code and additional details for the calculation of the GLRLM can be found in appendix B.

In order to achieve the task parallelism distribution in CUDA, the natural approach is to assign each sub image to a block of threads. In order to assign the data there are two possible approaches:

- Constructing a 3d array from the sub images in a similar fashion to the Jacket parallelization or, in other words, repackaging the 2d data into a 3d array.

- Using the indexes in the kernels to make each block access the corresponding sub image.

We decided to use the first approach (the same as Jacket), given that the index access meant a potential introduction of bugs given the care that needs to be have with indexes in arrays in the GPU and the fact that the repackaging data routines are not particularly expensive in MATLAB. If the sub images don’t overlap in fact it is a O(1) operation (the data is not moved, the only description of it changes), otherwise it takes linear time over the added size of the sub images (copy the subimages to the 3d array).

### 4.5 Effort Required and Comparison

After describing each of the parallelization strategy characteristics, it was seen that there are significant differences regarding the details of implementation for each one.

The MPT is the technology that requires the least effort to implement, which is a consequence

---

8In particular, errors that would cause segmentation faults in CPU code and only cause the program to crash, in CUDA and in Jacket code an out of bounds array access often causes the graphics card status to get corrupted, making it necessary to relaunch MATLAB in order to re establish the initial state of the graphics card.
of using additional CPUs (either in a shared or distributed memory way) to provide its acceleration. Most of the effort required resides in the data distribution in order to minimize the potential communication penalties that a parallelization can incur in. In particular, for the typical matrix sizes that we worked with, with sub $512 \times 512$ size, the potential costs of communication to divide a matrix among computers in the cluster can be a very high.

The MPT/GPU, given that it uses the GPU, naturally requires bigger changes to the code than just using the plain MPT. But even when compared to alternatives like Jacket the changes can be significantly higher. If the code being parallelized is not composed of functions contained in the rather limited set of functions supported by MPT/GPU, it requires the creation of additional functions in order to be able to run custom code in the data, and even then what it allows to execute is limited to calculations performed to each entry of the matrix being used\(^9\). This meant that we couldn’t find a way of calculating additional matrices, like the run length matrices. This immediately limited the usefulness of the MPT/GPU given how time consuming in the overall profile of the software these matrix calculations are, which was seen in Chapter 3.

The Jacket add on provides a significant larger of supported functions for GPU execution, by a factor of 5 to 1. Additionally it provides the possibility of parallelizing for loops in the GPU, which allows far greater flexibility than the MPT/GPU, and enabled us to calculate the additional matrices without much additional effort, and overall the changes to the code required to take advantage of Jacket were fairly moderate when compared to the other GPU alternatives.

Using C/CUDA to parallelize the software was by far the most challenging strategy. Not only does it require using another language (C), it requires that the programmers deal with almost every detail of how to use the GPUs computational abilities, like the number of blocks and number of threads and how they access the data. This provides an extremely flexible and powerful tool in comparison to the other GPU alternatives, but also makes it the hardest technology to implement by a wide margin. It even adds an additional complexity layer by requiring a compilation cycle before being usable by MATLAB. Some of the difficulties in implementing some functions, like reductions, can be alleviated by using existing libraries like Thrust, but even then their integration with MATLAB adds complexity due to requiring the use of the external language interfaces of MATLAB to use these libraries.

In the next chapter the acceleration results provided by each of the technologies is presented, which gives the remaining considerations to analyze the benefits of each parallelization strategy.

---

\(^{9}\)Even using indexing to access other entries is not supported, see [http://www.mathworks.co.uk/help/toolbox/distcomp/bsic4n4-1.html#bsnx7h8-1](http://www.mathworks.co.uk/help/toolbox/distcomp/bsic4n4-1.html#bsnx7h8-1).
Chapter 5

Results

This sections presents the results of the parallelisation performed on the original MATLAB code and their analysis.

In order to have some baseline scenarios to provide as a basis for comparison, the original software was ran (without the GUI) first by limiting MATLAB to one computing thread, and then running it without this restriction, which made it run on 12 threads on 12 of the 24 CPU As of its releas 2012a, cores of fermi0 in hydra, the machine described in chapter 3, which are presented as serial and multithreaded in the result graphs.

All the calculations were performed using double precision floating point numbers. Given that the matrices used are square, in the graphs the size of the matrix refers to the number of columns (rows) of it.

The first section of the chapter provides the result and analysis of the performance of each of the feature calculations. After that a scalability analysis is performed with the speedup results obtained in the system where we used different matrix sizes to see the behaviour of the different parallelization strategies. Following that the result for the complete system are shown and analyzed. Finally some additional results are presented, that include comparing different matrix structures and report of the error observed in the calculations.

\footnote{MATLAB only allows either single threaded or general multithreaded execution with no control over the exact number of threads/cores it uses, during our usage of hydra it used 12 threads.}
5.1 Micro Benchmarking

5.1.1 Individual Function Results

In this section the results of the optimizations made to the functions used as examples in the previous chapter are given. This was done to get an idea of the kind of results that can be expected in an ideal scenario (minimizing the data transfers, no GUI, etc.).

The results of the GPU parallelization micro benchmarks can be seen in figure 5.1 including the execution times of the FOS mean, FOS variance and the GLRLM, alongside the serial version results and the results provided by MATLAB when ran in its multithreaded mode.

The multithreaded acceleration was very consistent regarding the calculation of both the FOS mean and the FOS variance, and also the run length matrix calculation, which is not surprising given that it is the same processor as the serial run, only 12 cores of it. It shows that MATLAB has a good support of shared memory multithreading, although it was limited to using 12 cores instead of the available 24 cores, which seems to be a MATLAB imposed limitation (either technical or license wise).

The GPU results show a more diverse panorama of speed up results. Starting with the MPT/GPU, the first notable thing, as stated in the previous chapter, was the inability to get it to compute the GLRLM given its limited support for executing custom code, which made the acceleration of calculation of the GLRLM features suffer very drastically, barely being 30% faster than the serial version. This is a clear consequence of the importance that the calculation of the GLRLM has, which was presented in chapter 2. On the other hand the feature calculations showed an acceleration comparable to what was obtained by the multithreaded version, a very respectable result given that the graphics card loses half its possible computational performance when using double precision numbers.

The results for Jacket were very different and immediately show the limitations in MPT/GPU. The feature calculations had a speed up more than twice the MPT/GPU one (and the multithreaded CPU results also). But more importantly, given the calculation of the GLRLM using the gfor loop, the GLRLM features also showed a significant speedup when compared to the serial version. Nonetheless it is important to note that the potential difficulties identified in the previous chapter when describing the GLRLM implementation in Jacket showed clearly in the fact that the speed up obtained was only half of what the 12 core CPU did. Not necessarily a bad result overall, but very telling that the results from the optimization do not show one parallelization strategy as being categorically superior.

The C/CUDA results were in fact very similar to the Jacket ones in their behaviour, except for being faster across the board. The fact that the GLRLM was still slower than the 12 core CPU seem to indicate that the GLRLM calculation is limited by the architecture of the GPU and it is not a fault of either the Jacket or CUDA software stack (or our implementation could be improved).
Figure 5.1: Speed up comparison between strategies and calculations for the FOS Mean, FOS Variance and GLRLM.

5.1.2 Execution Time Distribution

This section provides the results of how the execution time was distributed in the different feature groups. The results presented are for two cases, a big matrix of size $512 \times 512$, and a small matrix of size $16 \times 16$. They were processed using the plain MATLAB code (with multithreading turned on), and the 3 GPU technologies, and the results were averaged to get an idea of the overall importance of each of the calculations in the overall execution time of the software. Even though the code was only ran once
The FOS execution time distribution for the big box configuration can be seen in image 5.2. The most important result is that the time is divided pretty much uniformly among all the feature calculations, even though they have differences in the calculations performed in the loops. The results for the many boxes configuration show a similar result, indicating that there is no dominating feature and all the FOS features contribute almost equally to the overall execution time, which is consistent with the original profiling of the code. In figure 5.3, the many box results can be seen, with a very similar result of almost uniformity in their time.

![Figure 5.2: FOS execution time distribution using a big matrix.](image)

![Figure 5.3: FOS execution time distribution using a small matrix.](image)
The SOS execution time distribution for the big box configuration can be seen in image 5.4 and the results for the many box configuration are in figure 5.5. The first notable thing is that the feature calculation, except for $f_2$, had a very uniform time distribution similar to the SOS case. The exception, $f_2$, corresponds to the calculation of the contrast, which, as was noted in chapter 3, has cubic complexity unlike the rest of the features with quadratic complexities. The behaviour of the feature calculations was very similar among both the big and small matrix execution times.

There was a significant difference in the calculation of the GLCM, with the big matrix GLCM calculation being significantly more expensive in the relative amount of processing time used compared to the feature calculations. This was consistent with both the profiling results and complexity analysis of chapter 3, and shows a significantly different behaviour of the execution time distribution in the small matrix case, in which the GLCM calculation is significantly less dominant.

Figure 5.4: SOS execution time distribution using a big matrix.

Figure 5.5: FOS execution time distribution using a small matrix.
The HOS execution time distribution for the big box configuration can be seen in image 5.4 and the results for the many box configuration are in figure 5.5. Unlike the SOS case, the time execution distribution was very similar between the big and small matrix cases. The GLRLM calculation clearly took the biggest share of execution time, while the feature calculation times were fairly uniformly distributed.

Perhaps the most important results of the this section are the execution time distribution among the FOS, SOS and HOS overall calculation times, which can be seen in figure 5.8 for the big matrix and figure 5.9 for the small matrix.

Figure 5.6: HOS execution time distribution using a big matrix.

Figure 5.7: HOS execution time distribution using a big matrix.
The results observed in these two sections provide a good summary of the behaviour of each of the feature calculations, both individually and in how do their execution times are relative to each other. All of these influence the results of the complete system runs that are presented later in this chapter.
5.2 Scalability Results

In this section we study the effects that changing the matrix size had in the results, using random matrices in order to see the relationship between the CPU (multithreaded) and the GPU performance.

5.2.1 Matrix Size Effects on Data Parallelism

For analyzing the data parallelism performance we used matrix sizes from $16 \times 16$ up to $16384 \times 16384$.

When using data parallelism the scalability results can be seen in Figure 5.10. The CPU was faster when seeing the performance on matrices up to size $512 \times 512$ when compared to CUDA, and up to size $4096 \times 4096$ This supports the hypothesis that the fact that the CPU had better performance that the GPU was not due to lack of capabilities of this last one, but it is due to the relatively small image size. An alternative way of stating this result is that the GPU seems to be very good at weak scaling\(^2\) and therefore when doing data parallelism of calculations like the ones we are dealing with, the GPUs seem good suited for resolution or data size improvements.

![Figure 5.10: Matrix Size Effects on Data Parallelism.](image)

---

\(^2\)Weak scaling refers to how the execution time varies with the number of processors for a fixed problem size per processor.
5.2.2 Matrix Size Effects on Task Parallelism

The effects of changing the data size in the task parallelism approach prove to be considerably more complex to analyze, but a particularly useful graph to see what is happening is presented in figure 5.11. We focus on comparing CUDA with the multithreaded behaviour, given that the Jacket implementation has a similar behaviour to the former, only being slower. We used data sizes from $16 \times 16$ up to $1024 \times 1024$, using 150 images in each case in order to isolate the effects of the different matrix size. In the actual software, typically, a bigger matrix size (derived from bigger boxes) would mean less matrices to process, although this is not always the case since the matrices can overlap.

There is a clear advantage of the GPU for matrix workloads with size up to $128 \times 128$, where the massive parallelism overcomes the inherent disadvantage that the CUDA cores have in comparison to the CPU cores in the calculation of the GLRLM. Even though up to dimension of $1024$ the matrices fit in the GPUs memory, clearly the massive parallelism advantage gets negated by the the limited computing capability of the CUDA cores. Going to sizes of $2048$ and above would not be practical, since the fact that when even one of the sub images does not fit in the GPU memory (along the associated matrices like GLRLM), then the execution time doubles since this image needs to be processed after the first batch of images finishes. At $1024 \times 1024$ the 150 matrices already use 1.25 GB of the video memory.

![Figure 5.11: Matrix Size Effects on Task Parallelism.](image)

45
5.3 Complete System Results

Once seen the individual feature and the scalability results, we present two main results: Figure 5.12 shows the result of the task parallelization made in a big box configuration and Figure 5.13 shows the result for the task parallelism implementation using a many small boxes configuration. Both were run 10 times and their results averaged.

5.3.1 GPGPU Results

The big box configuration meant using a set of 88 images, each having a $512 \times 512$ size, performing the calculations on the whole images. The measurements are done using the built in MATLAB timing functions. The base running time for the serial version was 2879.98 seconds. The CPU multithreaded code achieved the best speed up by a significant margin, which indicates a few things. The most important one is that the size of the images was not big enough to show the potential advantage that the GPU can have when dividing a matrix. The $512 \times 512$ double precision images only used approximately 2 MB of memory in the GPU of the available 3 GB. Doing quick simulations with random matrices indicated that for CUDA the GPU started to surpass the CPU performance at a size of $2000 \times 2000$, and have a significant lead when dealing with a dimension of $10000 \times 10000$, that corresponds to using 800 MB, a much bigger percentage of the available memory. This indicates that for this approach the GPU needs to deal with better resolution, therefore changing the scaling that the software uses would have been an interesting path to explore. Although even then it is not clear whether the GPUs would have made a significant leap given that the added resolution not only means dealing with bigger matrices (good for the GPUs), it also means a considerably more expensive process of building them (potentially not so good for the GPUs), so only the empirical results would be able to show conclusive evidence of whether bigger resolutions would be good or bad for the GPUs.

The many small box configuration refers to the case where the features on many small sub images are calculated, using a task parallelism approach instead of a data parallelism approach. In particular, a set of 22 images was used, calculating the features on sub images of dimension $16 \times 16$ (not uniformly distributed on the images). In total there were 3093 sub images. The serial running time was of 10,049.98 seconds. The results of this approach give a completely different scenario than the previous result. Given the ample pool of memory available in the GPU, all the sub images for a given image fit in its memory at the same time ($150 \ 16 \times 16$ double precision images take less than 310 KB). Additionally given the capability of the GPU of having a large number of blocks and large numbers of threads per block meant that it was capable of processing all 150 images at the same time, which clearly shows up in the massive speed up achieved compared to the serial version in both the CUDA and Jacket implementations, overcoming the fact that each CUDA core is slower than the GPU cores in calculating the GLRLM. In fact, given this massive parallelism showed, the Jacket version was very close to the CUDA version in spite of the native CUDA speed advantage.
Figure 5.12: Speed up achieved using big box configuration.

Figure 5.13: Speed up achieved using many small boxes configuration.
5.3.2 Cluster Results

The cluster that we used was composed of regular pc’s connected via ethernet. The node we used to access MATLAB remotely and administer the jobs submitted to the cluster had a dual core Intel Xeon processor at 3.2 Ghz with 6 GB of RAM, being a typical representative of the nodes that conform the cluster.

The cluster consists of 32 nodes, although during the development of our project less cores were operational\(^3\) and at the moment of running the tests only 14 cores were operational. Given this instability in the clusters configuration, we didn’t rely on it for development, using a different machine (the hydra machine that was used for the GPGPU tests) to write the code and test that it worked, and only used the cluster to run the final version to get a picture of the potential acceleration it can provide.

Using data parallelism for processing the $512 \times 512$ big matrix test, the cluster actually caused a slow down of 300%, i.e. the single node was 3 times faster processing the big matrix. This was a clear indicator of the high communication costs that the cluster had. This made the approach of task parallelization a potential better approach to use the capabilities of the cluster.

Given that we were using CPUs for the task parallelization, unlike the GPU case where each processing core is slower, we chose to use 150 matrices of size $512 \times 512$ in order to assess the potential performance of the cluster. Distributing the matrices previous to their processing, in this case the results were the opposite than the data parallelism one, with the cluster providing a speedup of 7.2 when compared to the single node result. This clearly supports our hypothesis that the task parallelization approach was the correct one for exploiting the cluster’s capabilities. More interestingly, unlike the GPUs result that suggest they are most adequate for the many small boxes configuration, the cluster is adequate for task parallelization with boxes of any size, since it can even accelerate the computation of calculations on whole images if desired.

5.4 Additional results

In this section we study the performance effects of different image (or matrix) structures, in order to see the potential use of the optimizations beyond the image sets provided.

The error in the calculation of the histogram in FOS and the GLRLM was non existent. The error in the calculations of the various features for most features was in the order of $10^{-16}$, with a couple ones in the order of $10^{-15}$ due to some reordering of operations, but overall the errors were not significant.

Additionally tests were conducted to see if there were performance differences depending on the structure of the matrix being processed. Four sets of data were used: lung cancer CT

\(^3\)Operational only refers to whether the core was available to the cluster and not to the status of the node itself.
scans, benchmark ct scans, random matrices and sparse random matrices. After running them, no significant differences were found, although it is possible that some optimizations could be made for sparse matrices, overall it seems that the performance is unrelated to the matrix structure.
Chapter 6

Conclusions

This dissertation investigated the use of various parallelization technologies in the optimization of a texture analysis system in the context of cancer research provided by the Edinburgh Cancer Research Center. This software’s purpose was to analyze the textures in medical images in order to enable their classification, and in order to accomplish data large numbers of data sets, corresponding to cancer patients, must be processed. Therefore accelerating this process could have great benefits. In fact, the holy grail of the field would be a real time system that enables the identification and classification of images at any point during the cancer treatment process.

Hardware wise we used two technologies, a cluster in the WGH and GPGPU technologies. Even though the cluster was a fairly modest one, with high communication costs, and we didn’t use it until the final stages of the project, it was shown to be a very good option if the algorithms are parallelized with minimizing communication in mind. The biggest advantage to using a cluster like this, with MPT, is in the relative easiness of parallelization when compared to any GPGPU technology, in fact there are no fundamental changes needed to the code of the algorithms thanks to the built in capabilities of the MPT, and there is no need to deal with the communication explicitly. Its main disadvantage is actually not due to the technology itself, it is due to the need of an additional piece of software (the Distributed Computing Server) to have access to more flexibility for job scheduling and data distribution.

On the other side of the spectrum the GPGPU technologies showed very different results amongst them. Even though the theoretical speedups that a GPU can provide are huge, its underlying architecture makes it better suited to some kind of computations. Even though entry wise matrix operations usually are a good fit, the fact of the matter is that the reductions required in almost all of the calculations of the code already decreased the potential speedups that the GPGUS could achieve. Additionally, the size of the data is very important in order to get closer to the theoretical performance benefits of GPUs, meaning that they are very adept at weak scaling. For our project this meant either the need for bigger sized images (derived from higher resolutions) in order to make data decomposition a feasible approach to use the GPUs. This lead us to explore another approach, exploiting the fact that one of the typical ways of processing the images in the software was to decompose it in smaller sub images, which we used
to create a task parallelization approach making the gpu process many of those small matrices at the same time, acting in a similar way of how we used the cluster.

When talking about the particular GPGPU technologies used in this project (MPT/GPU, Jacket and C/CUDA), there are many important things to remark, important from both the computational aspects of the system and the medical research ones. Currently, the MPT/GPU offers a rather limited feature set to exploit the capabilities of GPUs. If a code is limited to using the MATLAB functions that the MPT/GPU supports natively, and doesn’t need much additional functionality beyond that, then it could prove to be a good solution. Additionally if a system is already using the general MPT, then using MPT/GPU to add some GPU accelerations could be a good idea, given that the MPT is already there and includes the MPT/GPU. But, for the system that was our object of study, and given the current state of MPT/GPU, to recommend its use beyond those scenarios, particularly given the availability of the Jacket add on alternative.

Jacket is developed by a company called AccelerEyes, which is dedicated exclusively to developing programming tools for parallel programming and visual computing on GPUs. This means they are a more focused company than Mathworks (the MATLAB developers), and that is clearly noticeable when comparing it with the MPT/GPU. Not only was it faster across the board and supported more built in functions, it was also the easiest GPGPU technology to implement. Added to that, but it supports accelerators beyond those offered by NVIDIA (like the AMD offerings), unlike the MPT/GPU that is currently limited to CUDA devices (which still mean NVIDIA GPUs). This evidently means greater flexibility and possible hardware choices. This is particularly interesting at this point in time given the strong performance that the latest GCN-based AMD GPUs.

Using C/CUDA was, arguably, an order of magnitude harder than any other parallelization technology we used. Not only do one needs to deal with all the details of the implementation, like the arrays index (which often can get tricky in CUDA codes), but also the programmer needs to deal with manually tuning the settings of the number and organization of threads and blocks. This adds even more complexity to achieving the best levels of optimization.

These lower level technologies, like CUDA and OpenCL, could probably be very useful in the final implementation of a real time or near real time system that can analyze images at any point during the treatment process, but at this point when the features studied are evolving and the exact requirements are not a fixed target (given that the features are still being studied and developed), technologies that provide better flexibility while still providing good results like Jacket and the MPT in clusters are, probably, more adequate strategies that can give the best acceleration/effort benefit.

Even though this project achieved its aims quite satisfactorily with many interesting and noteworthy results, there are many possibilities to extend this work in order to both have a better

---

1 GCN stands for Graphics Core Next, and represented a major shift in the architecture of AMD GPUs, particularly strengthening their computing capabilities very significantly. At the moment of this writing, AMD’s GCN GPUs have better computing performance than the first models of NVIDIA Kepler GPUs, which potentially could change later this year with the release of their Big Kepler based GPUs, but still means that the GPGPU market is growing in alternatives.
understanding of the algorithms, and achieving the best possible optimization result. The software scales the images to fewer bits, 6 or 3 bits, and we did not have the time to explore the consequences of altering this values, that according to the complexity analysis we performed should have significant effects in both the number of operations that the features take to calculate and the memory requirements of the different matrices constructed. Exploring this, and possibly using external assistance from the medical researchers to relate the execution time and the quality of the results of the analysis, could be an extremely interesting and useful path to explore.

An important focus of our project was to use all the technologies to understand their behaviour in order to assess their potential contributions, but within each technology there are many paths that can be explored in order to see what are the best approaches for each one. For example, often there are more than one way to calculate the statistical features (like the FOS mean that had the vector multiplication form or using two times the mean function), seeing the performance of the technologies using different methods of calculation would give a better understanding of each technology that could enable a better use of them. Other factors to explore include thread and block factors, different decompositions and using matrices that don’t have sizes that are power of 2 (this could affect the performance in the GPUs significantly).

Additionally we restricted our efforts to the use of a Tesla C2050 card, exploring other options, not only limited to the specialized products from the manufacturers (the Tesla and Firststream lines), but also including more common consumer GPUs could prove to be extremely useful. Given the typically huge price difference that can exist when getting a professional workstation with Tesla/Firestream cards and regular PCs with consumer level cards (GeForce and Radeon), it would not be far fetched to imagine the benefits of the consumer grade hardware being attractive, to say the least. But usually this consumer level cards are restricted in some ways, for example the have less double precision performance, so only by experimenting could a quality judgment be made.
Appendix A

Statistical Features Formulas

This section contains the formulas for the features not explicitly stated in Chapter 2.

A.1 First Order Statistics

1. Mean

\[ \mu = \frac{\sum p_i G}{G} \]  \hspace{1cm} (A.1)

2. Variance

\[ \sigma^2 = \sum p_i (x_i - \mu)^2 \]  \hspace{1cm} (A.2)

The standard deviation is the root square of the variance and is represented by \( \sigma \).

3. Coarseness

\[ C = 1 - \frac{1}{1 + \sigma^2} \]  \hspace{1cm} (A.3)

4. Skewness

\[ Sk = \frac{\frac{1}{n} \sum (p_i - \mu)^3}{\left(\frac{1}{n} \sum (p_i - \mu)^2\right)^{3/2}} \] \hspace{1cm} (A.4)

5. Kurtosis
\[ K = \frac{1}{n} \sum (p_i - \mu)^4 \left( \frac{1}{n} \sum (p_i - \mu)^2 \right)^2 - 3 \] \tag{A.5}

6. Energy
\[ e = \sum p_i^2 \] \tag{A.6}

7. Entropy
\[ H = - \sum p_i \log(p_i) \] \tag{A.7}

### A.2 Second Order Statistics

1. Angular Second Moment
\[ f_1 = \sum_i \sum_j p(i, j)^2 \] \tag{A.8}

2. Contrast
\[ f_2 = \sum_{n=0}^{N} n^2 \left\{ \sum_i \sum_j p(i, j), |i - j| = n \right\} \] \tag{A.9}

3. Correlation
\[ f_3 = \sum_i \sum_j p(i, j) \frac{(i - \mu_x)(j - \mu_y)}{\sigma_x \sigma_y} \] \tag{A.10}

where \( \mu_x, \mu_y, \sigma_x \) and \( \sigma_y \) are the means and standard deviations of \( p_x \) and \( p_y \).

4. Variance
\[ f_4 = \sum_i \sum_j (i - \mu)^2 p(i, j) \] \tag{A.11}

5. Inverse Difference Moment
\[ f_5 = \sum_i \sum_j \frac{1}{1 + (i - j)} p(i, j) \] \tag{A.12}
6. Sum Average

\[ f_6 = \sum_{i=2}^{2G} ip_{x+y} \] \hspace{1cm} (A.13)

7. Sum entropy

\[ f_7 = -\sum p_{x+y}(i)\log(p_{x+y}) \] \hspace{1cm} (A.14)

8. Sum variance

\[ f_8 = \sum i = 2(i - f_7)^2 p_{x+y}(i) \] \hspace{1cm} (A.15)

9. Entropy

\[ f_9 = -\sum \sum p(i, j)\log(p(i, j)) \] \hspace{1cm} (A.16)

10. Difference Entropy

\[ f_{10} = -\sum p_{x-y}(i)\log(p_{x-y}) \] \hspace{1cm} (A.17)

11. Difference Variance

\[ f_{11} = \sum i = 2(i - f_{10})^2 p_{x-y}(i) \] \hspace{1cm} (A.18)

12. Information Measures of Correlation:

\[ f_{12} = \frac{f_9 - H_{xy1}}{\max\{H_x, H_y\}} \] \hspace{1cm} (A.19)

\[ f_{13} = 1 - e^{(-2(H_{xy2} - f_9))^{1/2}} \] \hspace{1cm} (A.20)

where \( H_x \) and \( H_y \) are the entropies of \( p_x \) and \( p_y \) and:

\[ H_{xy1} = -\sum \sum p(i, j)\log(p_x(i)p_y(j)) \]
\[
H_{xy2} = - \sum_{i} \sum_{j} p(i, j) \log(p_x(i)p_y(j))
\]

13. Maximal Correlation Coefficient

\[f_{14} = \lambda_2^{(1/2)}\]  \hfill (A.21)

where \( \lambda_2 \) is the second largest eigenvalue of the matrix defined by:

\[
Q(i, j) = \sum_k \frac{p(i, k)p(j, k)}{p_x(i)p_y(k)}
\]

### A.3 Higher Order Statistics

The Higher Order Statistics implemented in the version of the code provided were:

1. Short Run Emphasis (SRE):

\[g_1 = \frac{1}{n_r} \sum_i \sum_j \frac{q(i, j)}{j^2}\]  \hfill (A.22)

2. Long Run Emphasis (LRE):

\[g_2 = \frac{1}{n_r} \sum_i \sum_j q(i, j)j^2\]  \hfill (A.23)

3. Grey-Level Non-uniformity (GLN):

\[g_3 = \frac{1}{n_r} \sum_i \left( \sum_j q(i, j) \right)^2\]  \hfill (A.24)

4. Run Length Non-uniformity (RLN):

\[g_4 = \frac{1}{n_r} \sum_j \left( \sum_i q(i, j) \right)^2\]  \hfill (A.25)

5. Run Percentage (RP):
Based on the fact that the aforementioned described features are only functions of the frequency of the run length and don’t consider the gray level information contained in the GRLM, Chu et al.[12] proposed two new features, which have been implemented in subsequent versions of the WGH software, and we mention for completeness:

1. Low Gray Level Run Emphasis (LGRE):

\[ g_6 = \frac{1}{n_r} \sum_i \sum_j p(i, j) \frac{i^2}{i^2} \]  \hspace{1cm} (A.27)

2. High Gray Level Run Emphasis (HGRE):

\[ g_7 = \frac{1}{n_r} \sum_i \sum_j p(i, j) i^2 \]  \hspace{1cm} (A.28)

Dasarathy and Holder[13] described another 4 new features that consist of joint statistical measures that combine information of gray levels and run lengths:

1. Short Run Low Gray-Level Emphasis (SRLGE):

\[ g_8 = \frac{1}{n_r} \sum_i \sum_j p(i, j) i^2 j^2 \]  \hspace{1cm} (A.29)

2. Short Run High Gray-Level Emphasis (SRHGE):

\[ g_9 = \frac{1}{n_r} \sum_i \sum_j p(i, j) \frac{i^2}{j^2} \]  \hspace{1cm} (A.30)

3. Long Run Low Gray-Level Emphasis (LRLGE):

\[ g_{10} = \frac{1}{n_r} \sum_i \sum_j p(i, j) i j^2 \]  \hspace{1cm} (A.31)

4. Long Run High Gray-Level Emphasis (LRHGE):

\[ g_{11} = \frac{1}{n_r} \sum_i \sum_j p(i, j) i^2 j^2 \]  \hspace{1cm} (A.32)
Appendix B

Supporting Code

The original code for the GLRLM calculation was:

Listing B.1: MPT/GPU variance function

```matlab
run_length = 1;
col_num = 0;
row_num = 0;

for row_num = 0:row_max,
    run_detect = 0; % Variable to detect the end of a run.
    col_num = 0;
    run_length = 1;
    grey_val = GLRLM_data_in(col_num+1,row_num+1);
    while (col_num<col_max),
        grey_val_next = GLRLM_data_in(col_num + 2,row_num+1);
        if (grey_val == grey_val_next),
            run_detect = 1; % detected a run of length > 1
            run_length = run_length + 1;
        else
            grey_val = grey_val_next;

        end
    end
    % Now check for runs that finish at the end of a row. Have to set run_detect.
    if (col_num == (col_max - 1)),
        run_detect = 0;
    end
else
    run_detect = 0; %indicated that we are at the end of a run.
    if (col_num == (col_max - 1)),
        GLRLM_horiz(2,grey_val_next+1) =GLRLM_horiz(2,←
                                    grey_val_next+1) + 1;
    end
end
```
An optimized CUDA version, explained with comments, was:

Listing B.2: MPT/GPU variance function

```c
__global__ void glrlm( double *glrlm,
const int rows,
const int cols,
const int rows_sub,
const int cols_sub,
const double *image,
const int n) {

  // Get the thread index values to know what parts of the image
  // and glrlm arrays each thread has to access
  int bx = blockIdx.x + blockIdx.y*blockDim.x;
  int tx = threadIdx.x;

  int j, run_length, run_detect;

  run_detect = 0;
  run_length = 1;

  // This is the same loop as the original
  for(j = 0; j < cols_sub-1; j++){
    // Instead of using conditional statements, assigning the logical
    // values to a variable enable the calculations while avoiding
    // thread divergence
    run_detect = ((image[bx*rows_sub*cols_sub + j*rows_sub + tx]
      == image[bx*rows_sub*cols_sub + j*rows_sub + tx])
      || (j == (cols-1)));

    // This is the code that gets executed in the first conditional in
    // the original code, if run_detect is zero the run_length stays the
```
run_length = run_length + run_detect;

// We store the value of the pixel in an int in order to use is
// as index to the array

temp = (int)image[bx*rows_sub*cols_sub + n*rows_sub + tx];

// This is the content of the second conditionals, the ones with
// the condition (run_detect == 0), we reproduce the behaviour
// by substracting run_detect from 1.

    glrlm[bx + rows*temp] = glrlm[bx + rows*temp] +
                        (1-run_detect)*run_length;
    run_length = (1-run_detect)*run_length;

}
}

It is important to note that this version of the algorithm was developed almost at the end of
the project and was not the version used and tested, therefore it could contain mistakes. The
version used was very similar to the MATLAB code using conditionals, but we deemed adequate
to present this more sophisticated version since it better illustrates and supports our claim that
C/CUDA codes were an order of magnitude more difficult to implement when compared to any
other strategy.
Bibliography


