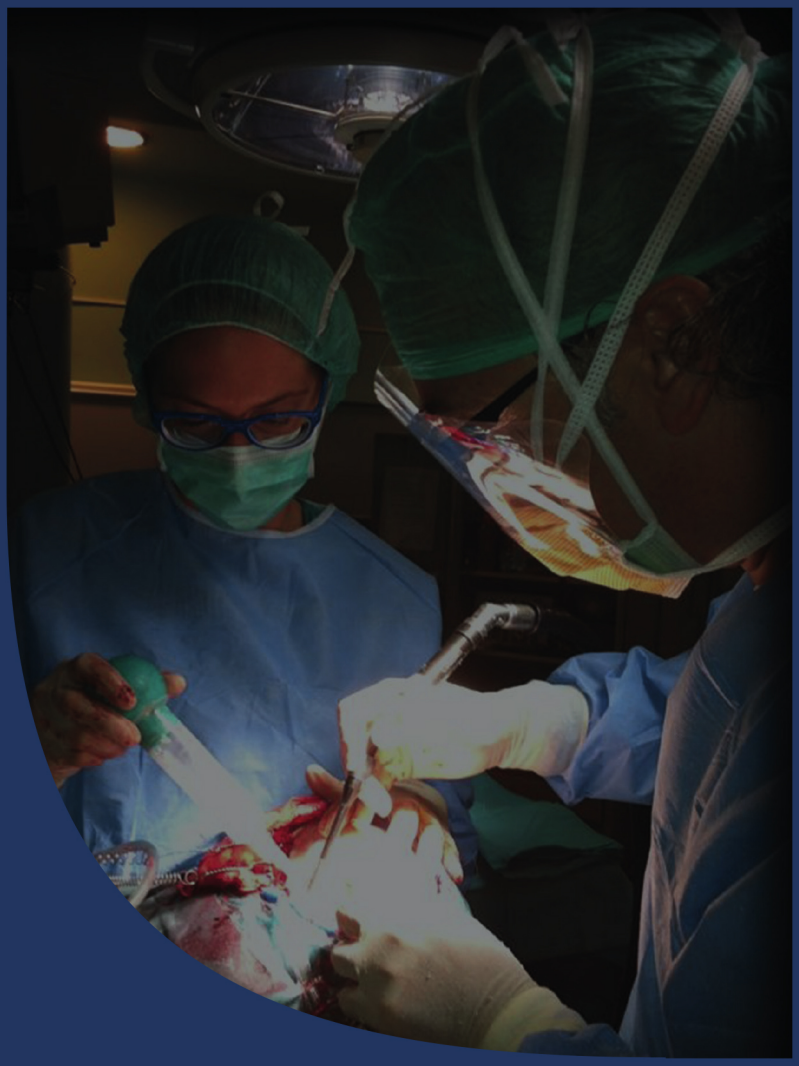
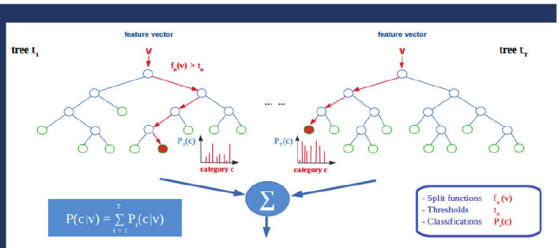
UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

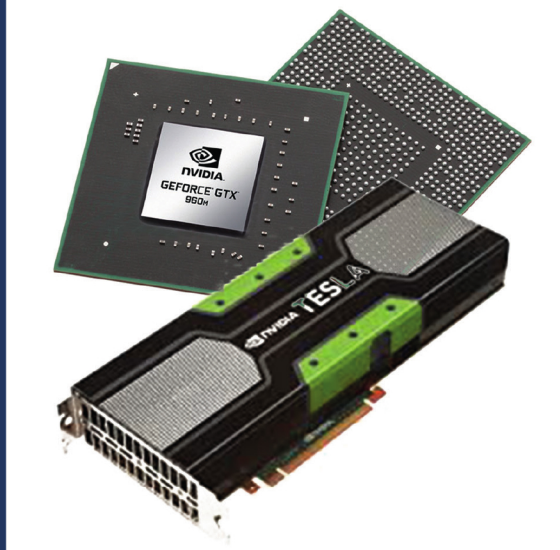
IUMA

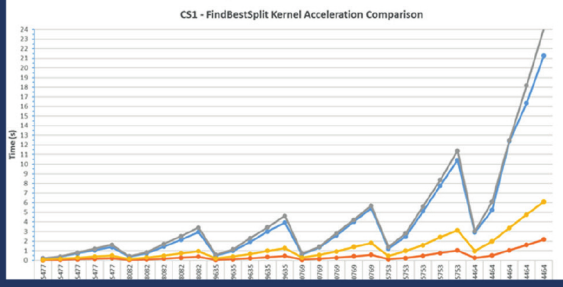
IMPLEMENTATION OF HYPERSPECTRAL IMAGE CLASSIFICATION ALGORITHMS FOR BRAIN TUMOUR DETECTION USING GRAPHICAL PROCESSING UNITS (GPUS)

IMPLEMENTACIÓN DE ALGORITMOS DE CLASIFICACIÓN DE IMÁGENES HIPERESPECTRALES PARA LA DETECCIÓN DE TUMORES CEREBRALES SOBRE TARJETAS GRÁFICAS PROGRAMABLES (GPUS)









Titulación: Máster en Tecnologías de Telecomunicación

Autor: D. Abián Hernández Guedes
Tutor 1: Dr. Gustavo Marrero Callicó
Tutor 2: D. Himar Fabelo Gómez

Fecha: Diciembre de 2016

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Implementación de Algoritmos de Clasificación de Imágenes Hiperespectrales Para la Detección De Tumores Cerebrales sobre Tarjetas Gráficas Programables (GPUs)

Autor: **Abián Hernández Guedes**

Tutor(es): **Dr. Gustavo Iván Marrero Callicó**

D.Himar Fabelo Gómez

Fecha: **diciembre - 2016**

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es
f +34 928 451 083 | www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Implementación de Algoritmos de Clasificación de Imágenes Hiperespectrales Para la Detección De Tumores Cerebrales sobre Tarjetas Gráficas Programables (GPUs)

HOJA DE FIRMAS

Alumno/a: Abián Hernández Guedes Fdo.:

Tutor/a: Dr. Gustavo Iván Marrero Callicó Fdo.:

Tutor/a: D.Himar Fabelo Gómez Fdo.:

Fecha: diciembre - 2016

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira
f +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Implementación de Algoritmos de Clasificación de Imágenes Hiperespectrales Para la Detección De Tumores Cerebrales sobre Tarjetas Gráficas Programables (GPUs)

HOJA DE EVALUACIÓN

Calificación: .....

Presidente: Fdo.:

Secretario: Fdo.:

Vocal: Fdo.:

Fecha: **diciembre - 2016**

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira
f +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

TABLE OF CONTENT

FIGURES

TABLES

RESUMEN

Las unidades de procesamiento gráfico (GPUs) se han vuelto enormemente populares en el área de computación de alto rendimiento debido a su arquitectura de hardware masivamente paralela. Su arquitectura permite explotar abundantemente el paralelismo a nivel de datos mientras reduce el consumo de energía en la búsqueda, decodificación y emisión de instrucciones. Por esta razón, las GPUs son unas plataformas perfectas para acelerar las tareas de clasificación de imágenes hiperespectrales, las cuales son una tecnología emergente para el diagnóstico médico. Los sensores de imágenes hiperespectrales miden el brillo de los materiales dentro de cada área de píxeles usando un número muy grande de bandas formadas por longitudes de onda espectrales contiguas y explotando el hecho de que todos los materiales reflejan, absorben o emiten energía electromagnética, a longitudes de onda específicas, en patrones distintivos relacionados con su composición molecular.

Los datos hiperespectrales se pueden procesar utilizando múltiples algoritmos de aprendizaje supervisados para detectar el tejido tumoral en cerebros humano. Random Forest, un método de aprendizaje automático que se ha popularizado en tareas de detección de objetos en la comunidad de la visión por computador, ha demostrado ser un buen candidato para clasificar las imágenes hiperespectrales. En general, la formación de un modelo de Random Forest con grandes conjuntos de datos supone una elevada carga computacional y dificulta la investigación científica, ya que el proceso requiere mucho tiempo de cómputo si no existe la disponibilidad de una plataforma de computación de alto rendimiento.

El objetivo de este Trabajo de Fin de Máster es acelerar la fase de entrenamiento de Random Forest utilizando GPUs, partiendo de una implementación eficiente secuencial de este algoritmo. A lo largo del documento, se presentan múltiples cuellos de botella identificados en la fase de entrenamiento y la solución a estos cuellos de botella para acelerar los algoritmos. Las diferentes soluciones de este estudio han demostrado que la aceleración obtenida por las GPUs es prometedora para generar modelos en un tiempo más reducido, permitiendo la posibilidad de realizar este proceso en tiempo real en un futuro no muy lejano.

ABSTRACT

Graphics Processing Units (GPUs) have become extremely popular in the high-performance computing area due to its massively parallel hardware architecture. This architecture allows to exploit abundant data level parallelism while reducing power consumption in the instruction fetching, decoding, and issuing. For this reason, GPUs are suitable platforms to accelerate the classification of hyperspectral images which are an emerging technology for medical diagnosis. Hyperspectral imaging sensors measure the radiance of the materials within each pixel area at a very large number of contiguous spectral wavelengths, exploiting the fact that all materials reflect, absorb or emit electromagnetic energy, at specific wavelengths, in distinctive patterns related to their molecular composition.

Hyperspectral data can be processed using multiples different supervised learning algorithms to detect human brain tumour tissue. Random Forest, a machine learning method that has become popular in object detection tasks in the computer vision community, has proved to be a good candidate in order to classify hyperspectral images. Generally, training a Random Forest model on large datasets is computationally demanding and makes scientific research difficult since the process requires too much computational time if there is not available a high performance computing platform.

The goal of this Master's Thesis is focused in the Random Forest training phase acceleration using GPUs, starting from an efficiently sequential implementation of this algorithm. We present multiple bottlenecks identified in the training phase and a solution for these bottlenecks in order to accelerate the algorithms. The different bottleneck solutions achieved in this research study have demonstrated that GPU acceleration is promising in order to generate models in a shorter time, giving the possibility to perform this process in real-time in a close future.

CHAPTER 1: INTRODUCTION

1.1 INTRODUCTION

Hyperspectral images are an extension of the concept of digital image. This kind of images progressively increases the number of spectral bands collected by the imaging instrument and the spectral resolution too (modern instruments not only have more bands, but the bands are also narrower or closer to each other). Particularly, hyperspectral images comprise hundreds of narrow spectral bands. Due to this sampling strategy, hyperspectral images provide much more information about the captured scene than traditional solutions based on panchromatic or multispectral approaches.

One of the major benefits from such technology is likely to be in the removal of brain tumours. There are several reasons for this. Brain tumours, more than any other cancers, can resemble the normal surrounding brain making them difficult to differentiate. Unlike many tumours, they infiltrate the surrounding tissue and thus their borders are indistinct and difficult to identify. The surrounding brain is also very eloquent and there is no redundancy as is seen in many other organs where it is normal to remove the tumour with a surrounding rim of healthy tissue. This is not possible in the brain where it is essential to identify accurately the borders between normal and disease. Although malignant primary brain tumours in adults occupy the 13th place in frequency of all cancers, due to their particularly poor prognosis they are the fifth most common cause of cancer death in the under 65-year-old population. Moreover, they are the second most common cancer in children and the most common cause of cancer death in children.

Currently, the main tool for differentiating normal from malignant tissue remains the human naked eye. Other techniques have been developed but none has succeeded in reliable tissue differentiation. Neuronavigation is plagued by brain shift, ultrasound is highly operator dependent and intraoperative MRI (Magnetic Resonance Imaging) fails to provide real time images obtaining just an occasional snapshot during surgery. Under these circumstances, HSI (Hyperspectral Imaging) arises as a potential solution that allows a precise detection of the edges of the malignant tissues in real time, while assisting guidance for diagnosis during surgical interventions and treatment. Moreover, the cost associated with hyperspectral imaging instrumentation is significantly lower than the aforementioned techniques as it is based on conventional optical imaging technology. HSI supposes a non-contact, non-ionizing and minimal-invasive sensing technique based on registering

extremely small wavelengths (normally in the nanometre range) of the tissues in order to determine their histological characteristics.

Since HSI collects high amount of data, it is needed the utilization of high-performance computer platforms where the processing algorithms are implemented. One of the possibilities to address this issue is using a high-performance computer together with a Graphical Processing Unit (GPU) where the algorithms can be highly parallelized in order to obtain the results of the processing in real time.

1.2 OBJECTIVES

The main goal of this project is to implement the Random Forest (RF) algorithm in a GPU so as to accelerate the training and the classification process of hyperspectral images. The experiments performed in this project are framed in the context of a bioengineering research project where hyperspectral images are used to distinguish between tumour and normal tissue during neurosurgical operations. In this application, real-time is highly necessary since the neurosurgeons need to have the results of the processing during the time framework of the surgery.

Although this project is focused in this concrete application, the work performed can be extrapolated to other fields where the acceleration of the RF algorithm will be necessary.

1.3 CONTEXT AND MOTIVATIONS

The main motivation to carry out this project is the European Project HELICoiD "HypErspectral Imaging Cancer Detection" (FP7-618080) from the Institute for Applied Microelectronics (IUMA) at the University of Las Palmas de Gran Canaria. This project is coordinated by Dr. Gustavo Marrero Callicó as the principal investigator (PI), has the main goal of applying hyperspectral imaging techniques in order to accurately identify the margins of malignant tumours during surgical procedures in real-time. The HELICoiD project develops an experimental intraoperative setup based on non-invasive hyperspectral cameras connected to a platform running a set of algorithms capable of discriminating between healthy or tumour tissues.

On the other hand, to understand the context of this project it is necessary to know that the Integrated System Design Division, DSI, from IUMA is specialized in the treatment of hyperspectral images. DSI team has already undergoing projects such as:

CCSDS Lossless Compression IP-core Space Applications (ITT-No. AO/1-8032/14/NL/AK)

The main objective of this ITT (invitation to Tender) is to implement two separate IP-cores corresponding to the CCSDS 123 and CCSDS 121 standards respectively. The former corresponds to a Lossless Multispectral and Hyperspectral Image Compression architecture, while latter is a Lossless Data Compressor. Both IP-cores will be mapped for space qualified FPGAs (from Microsemi and Xilinx) and also for radiation hardened standard cells (180 nm ATMEL ATC18RHA).

REBECCA: Resilient Embedded Electronic Systems for Controlling Cities under Atypical Situations (TEC2014-58036-C4-4-R)

REBECCA is oriented to the Smart City paradigm. This topic brings up important challenges in different areas related with the sustainable development of the city and the provision of services to citizens. Among these areas, REBECCA focuses on urban security for large public spaces and/or

celebration of major events. In this context, REBECCA works on the design of a platform for sensing and distributed computing of visible and multi-hyper-spectral image processing.

ENABLE-S3: European initiative to Enable Validation for Highly Automated Safe and Secure Systems

ENABLE-S3 is a strongly industry-driven project. It will pave the way for accelerated application of highly automated and autonomous systems in the mobility domains automotive, aerospace, rail, maritime and health, through provision of highly effective test and validation methodology and platforms. ENBALE-S· will help the European industry to gain leadership in the strategic field of autonomous systems due to faster development and test of new products.

HYLOC: Multispectral and Hyperspectral Image Compression System

The objective of this industrial project is the implementation of a prototype suitable for its implementation on a space-qualified FPGA for the compression of multispectral and hyperspectral images based on the standard CCSDS-123. The effect of the several configuration parameters on the compression efficiency and hardware complexity is taken into consideration to provide flexibility in such a way that the implementation can be adapted to different applications scenarios.

1.4 MEMORY ORGANIZATION

This memory is structured as follows:

Chapter 1: The first chapter of this document consists on a brief introduction to the research work that will be described in this document.

Chapter 2: In this chapter, it is presented the material employed in this work. First, it will introduce the hyperspectral image and analysis methods where we will focus in medical applications with special mention to the HELICoiD project. Later, this chapter introduces the supervised pixel-wise algorithms which will be used in order to classify hyperspectral images, detailing the Random Forest (RF) algorithm. Finally, we will introduce the General-Purpose computing on Graphics Process Units (GLGPU) and the NVIDIA's CUDA Architecture.

Chapter 3: This chapter is dedicated to explain how the dataset used in this work is extracted . Here we explain the HELICoiD demonstrator that it is used to extract a hyperspectral image database so as well neurosurgeons and neuropathologists work to generate the ground truth for the training of the supervised algorithm.

Chapter 4: In this part of the document, we explain the different parallel implementations of RF, both the CPU and the GPU implementations.

Chapter 5: This chapter presents the results obtained in the acceleration performed in the training part of the Random Forest algorithm.

Chapter 6: The last chapter sums up the conclusions achieved during the growth of this research work. Also, the on-going work is described.

CHAPTER 2: RELATED WORK

2.1 INTRODUCTION

This chapter presents a brief description of the concepts of hyperspectral imaging and its use in the medical field. Furthermore, the classification algorithm selected for the analysis of this kind of images is detailed as well as the platform where this algorithm has been implemented.

About the selected classification algorithm, its operation is detailed and special attention will be given to the training phase of the algorithm that is where this project will focus.

Finally, the platform selected is NVIDIA K40, a CUDA GPU. We will explain how the CUDA platform (architecture and memory hierarchy) works and we will focus on the K40's architecture.

2.2 HYPERSPECTRAL IMAGING

Spectral imaging, also known as imaging spectroscopy, refers to the technology that integrates conventional imaging and spectroscopy methods to obtain both spatial and spectral information of an object [1]. Hyperspectral imaging sensors measure the radiance of the materials within each pixel area at a very large number of contiguous spectral wavelength bands [3] exploiting the fact that all materials reflect, absorb, or emit electromagnetic energy, at specific wavelengths, in distinctive patterns related to their molecular composition [2]. The basic task underlying many HSI applications is to identify different materials based on their reflectance spectrum. In this respect, the concept of a spectral signature, which uniquely characterizes any given material, is highly attractive and widely used [2].

As a result of spatial and spectral sampling, hyperspectral imaging (HSI) sensors produce a three-dimensional (3D) data structure (with both spatial-spectral components), referred to as a hypercube. Figure 2-1 shows an example of such a data cube [3]. If we extract all pixels in the same spatial location and plot their spectral values as a function of wavelength, the result is the average spectrum of all the materials in the corresponding ground resolution cell. In contrast, the values of all pixels in the same spectral band, plotted in spatial coordinates, result in a grayscale image depicting the spatial distribution of the reflectance of the scene in the corresponding spectral wavelength [3].

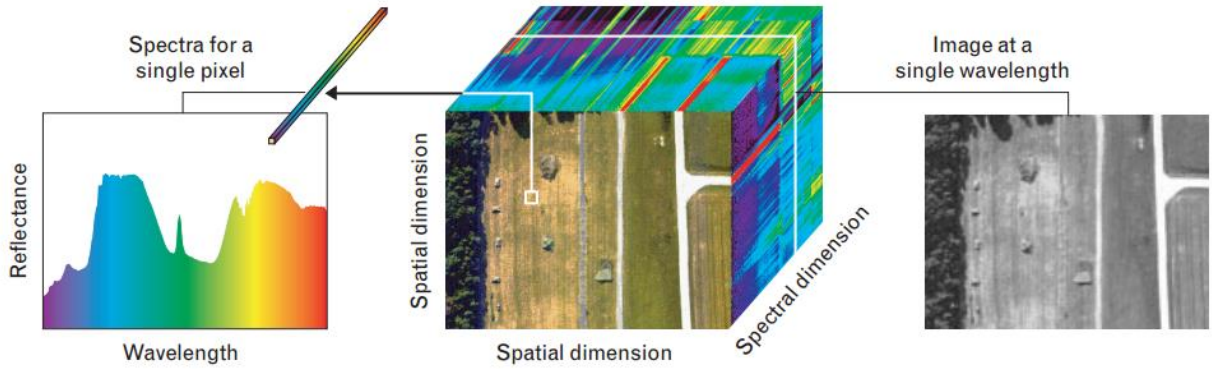


Figure 2-1: Basic data-cube structure (centre) in hyperspectral imaging, illustrating the simultaneous spatial and spectral character of the data. The data cube can be visualized as a set of spectra (left), each for a single pixel, or as a stack of images (right), each for a single spectral channel.

To acquire a 3-D data cube, spatial (2-D) and wavelength as a third dimension, a spectral imaging system needs to be employed. The typical hardware framework of a typical spectral imaging system generally consists of four parts: collection optics or instruments, spectral dispersion element, detector, and system control and data collection module [1]. The spectral dispersion element is the heart of the system that enables the separation of the light into different wavelengths.

Although hyperspectral image analysis methods have been intensively investigated in the remote sensing area, their development and application has been extended to other fields. For instance, in recent years, hyperspectral imaging has gained a wide recognition as a non-destructive and fast quality and safety analysis and assessment method for a wide range of food products [4]. Another important application of hyperspectral imaging technologies is that of agriculture, and in particular, precision agriculture. Precision agriculture can be broadly defined as the use of observations to optimize the use of resources and management of farming practices [5]. Hyperspectral Imaging is an emerging technology for clinical diagnosis. Some studies have proven that interaction between electromagnetic radiation and tissue carries useful information for diagnosis proposals [6].

2.2.1 Hyperspectral Imaging for Cancer Detection

Hyperspectral imaging is an emerging imaging modality for medical applications, especially in disease diagnosis and image-guided surgery [6]. This technology shows some advantages compared to the currently techniques employed for cancer detection, such as Magnetic Resonance (MR), Computed Tomography (CT), Ultrasound (US) and Positron Emission Tomography (PET). MR exposes patients to potentially harmful radiation, requires a trained operator and is expensive. US suffers from the disadvantage of having low image contrast. PET and CT are also expensive and uses high doses of radiation. Since most of these methods are costly and require trained operators, patient access to these important life-saving measures is limited. The long-term goal of hyperspectral imaging in cancer detection is to develop a simple-to-use, non-invasive, and risk-free tool that will provide early and affordable detection of potentially life threatening malignant tumours. This technology can be use both for screening enhancement and for quantitative analysis of tissue [7].

Some research studies that employ hyperspectral imaging as diagnosis tool are show on [6]. The differences between the different research works are the specimens used for each study, the kind of the analysed disease and the acquisition technique employed.

The study of in-vivo tissue allows providing a non-invasive disease diagnosis and an automatic guidance tool for surgery. After a cancer resection surgery, a pathology report notes that the surgical

margin is one of the following three types: clear, positive, or close. In case of clear margin, normal tissue surrounds cancer cells. In case of positive margin, cancer cells appear on the margins, which would lead to an additional surgery to remove the remaining cancer tissue. It is reported that 20-50% of breast cancer surgeries performed require a further surgery to remove remaining cancer tissues. In the research work proposed on [8] a method for improving the detection of breast cancer positive margins during surgery has been developed in order to prevent the need for additional surgeries by using hyperspectral imaging and image classification techniques. The results of this research study have found that examining ex-vivo breast cancer hyperspectral images tagged by a pathologist, the developed classification approach is shown to achieve a sensitivity of about 98% and a specificity of about 99%.

Although few studies explain the basic principles and instrumental systems for in-vivo HSI system in the biomedical field [9], only limited work deals with in-vivo detection of gastrointestinal (GI) cancer. In one of them [10], by employing rigid endoscopy, the authors propose a novel pre-processing stage to detect cancer cells in the larynx, overcoming several problems related with image interference(s): i) mis-registration of single images in a HS cube due to patient heartbeat; ii) image noise; and iii) specular reflections. Another interesting study in the field of upper GI endoscopy can be found in [11]. In this work, the authors present the calibration and test results obtained by mean of a HS reflectance and flexible video endoscope setup for the in-vivo GI cancer detection.

On the histological field, some studies have employed this technology for quantitative analysis of different diseases from in-vitro samples. On [8] microscopic hyperspectral imaging technology instead of traditional method has been employed to evaluate the therapeutic effects of Erythropoietin (EPO) on diabetic rats, founding that spectral imaging can contribute to significant quantitative analysis of such kind of diseases. HSI can also improve the detection of malignancy from histopathological GI samples, by removing subjectivity and intra/inter-observer variations [12]. In a recent work, microscopic HSI can distinguish between normal and cancerous gastric cells with 95% accuracy [13]. Maggioni et al [14] presented a classification technique for discriminating normal, precancerous, and cancerous colonic lesions. HS data are collected in the range of 440-700 nm, while setting the microscopic magnification factor to be 400x. A 97.1% classification accuracy has been reported when nuclei were extracted from all the samples. Chadded et al [15] performed classification of multispectral colon biopsy images, achieving reasonable classification accuracy in discriminating different types of colon tissues such as carcinoma, Intraepithelial Neoplasia, and Benign Hyperplasia. Additionally, Akbari et al [16] proposed a method of colonic cancer detection by using a broad band light source to illuminate the tissue slide and a HS camera to capture wavelength bands from 450-950 nm. Using Support Vector Machine (SVM) to classify the given tissues, 98.3% specificity and 96.2% sensitivity was observed for colon cancer data set.

The use of HSI can be mainly divided in the detection of cancer and other pathologies. Regarding with cancer detection, this technique has been mainly used in the detection of cervical cancer (both in-vivo [17][18] and in-vitro [19], for the detection of breast cancer (both in-vivo [20] and in-vitro [21], for the detection of skin cancer (both in-vivo [22] and in-vitro [23], and for the detection of head and neck cancer (both in-vivo [24][25] and in-vitro [16]. Regarding with other pathologies, HSI has been also used for the study of heart and circulatory pathology (both in-vivo [26] and in-vitro [27], and for the study of retinal diseases [28]. Recently, this technique has been proven to be of special relevance to guide the surgeons in different operations as could be the mastectomy [29], the gall bladder surgery [30], the renal surgery [31] and the abdominal surgery [32].

2.2.2 HELICoiD Project

In addition to radiotherapy and chemotherapy, surgery is one of the major treatment options for brain tumours [33]. However, because brain tumours infiltrate and diffuse into the surrounding normal brain, the surgeon's naked eye is often unable to accurately distinguish between the tumour and normal brain tissue. Inevitably, tumour tissue is either unintentionally left behind during surgery or too much normal brain tissue is taken out. Studies have shown that tumour tissue left behind during surgery is the most common cause of tumour recurrence and is a major cause of morbidity and mortality [34][35][36]. On the other hand, over-resection of brain tumour tissues has also been shown to cause permanent neurological deficits that affect patients' quality of life [37].

Intra-operative neuro-navigation, intra-operative Magnetic Resonance Imaging (iMRI) and fluorescent tumour markers such as 5-aminolevulinic acid (5-ALA) have been developed as adjuncts to surgery to help with brain tumour delineation. Although these adjuncts have improved the accuracy of brain tumour resections, they have a number of limitations. Neuro-navigation (Figure 2-2) is rendered inaccurate at locating tumour margins due to brain shift and changes in tumour volume that occurs during resections [38][39].

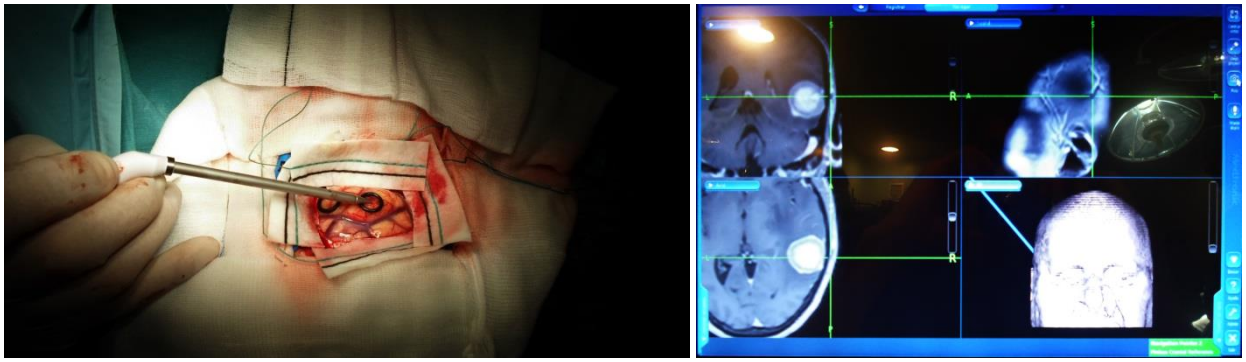


Figure 2-2: Image-guided stereotaxis system and screen capture used to locate the position of a tumour marker in a MRI

Intra-operative Magnetic Resonance Imaging was developed as a solution to intra-operative brain shift as it's capable of providing near real time tumour margin mapping. However, this has been found to have poor spatial resolution, to largely extend the surgery time and it is very expensive [40].

Fluorescent tumour markers such as 5-aminolevulinic acid (5-ALA) are excellent at identifying tumours but can only be used for high grade tumours, produce important knock-on effects and are poor at defining tumour margins mainly due to the diffuse nature of brain tumours [41][42].

Despite the improvement in the accuracy of brain tumour resections, neurosurgery is still unable to accurately define brain tumour margins. Under these circumstances, hyperspectral imaging arises as a potential solution that allows a precise detection of the edges of the malignant tissues in real time, while assisting guidance for diagnosis during surgical interventions and treatment. Moreover, the cost associated with hyperspectral imaging instrumentation is significantly lower than the aforementioned techniques as it is based on conventional optical imaging technology. Hyperspectral imaging supposes a non-contact, non-ionizing and minimal-invasive sensing technique based on registering extremely small wavelengths (normally in the nanometre range) of the tissues in order to determine their histological characteristics.

Figure 2-3: Intra-operative Magnetic Resonance Imaging system

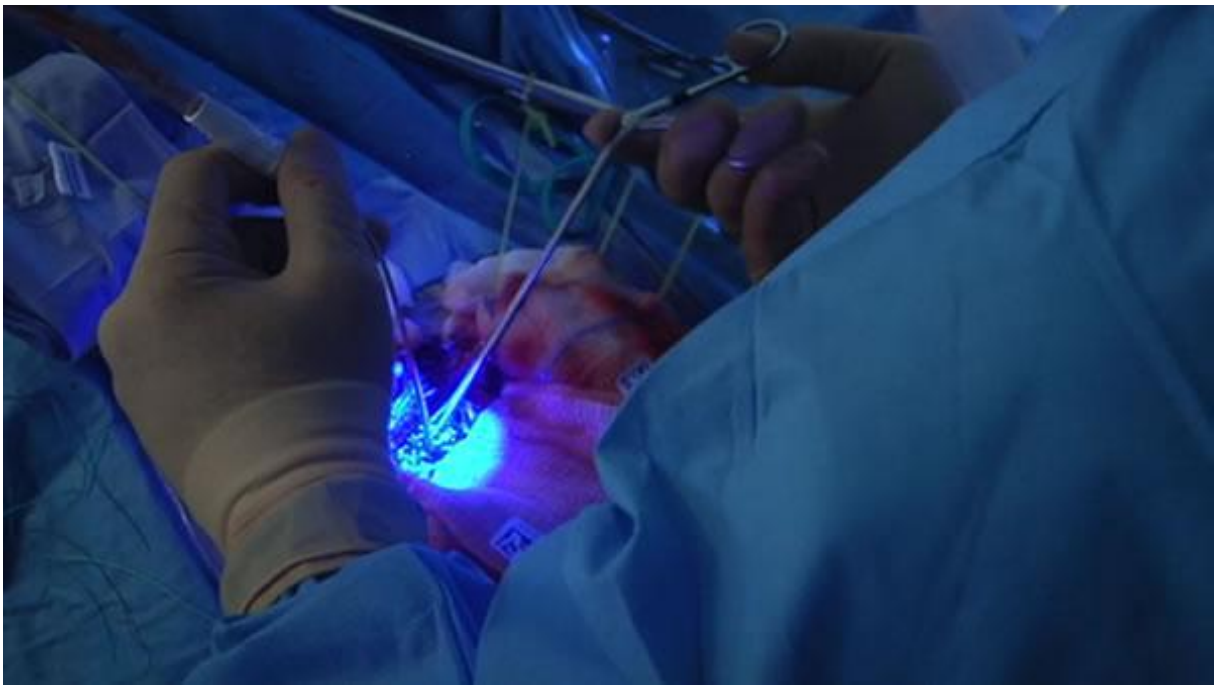


Figure 2-4: Fluorescence-guided surgery using 5-Ala system

HELICoiD (HyperEspectraL Imaging Cancer Detection) is a European collaborative project funded by the Research Executive Agency (www.helicoid.eu), through the Future and Emerging Technologies (FET-Open) programme, under the 7th Framework Programme of the European Union. It is formed by four universities, two university hospitals and three leading industry partners. The main goal of this project is to efficiently differentiate between healthy and diseased tissues and so lead to better surgical removal using the aforementioned hyperspectral images.

Starting with some specific types of cancers, this project tries to generalize the methodology to discriminate between healthy and malignant tissues in real-time during surgical procedures. Using the hyperspectral signatures of the healthy tissues and the same tissues affected by cancer, a model

of how cancer affects to the hyperspectral signature is derived. The research has started with the challenging task of brain cancer detection. A precise resection of the gliomas will minimize the negative effect of removing brain cells while assuring an effective tumour resection. As cancer supposes a change in the cellular physiology, it should be detected as a change in the hyperspectral signature. This project tries to determine if there is a certain pattern that could be identified as a cancer hyperspectral signature. This information is provided, through different display devices to the surgeon, overlapping normal viewing images with simulated colours that indicates the cancer density in a certain tissue area exposed during every instant of the surgical procedure.

A high-efficiency hardware/software prototype has been developed with the aim of recognizing cancer tissues on real time. The processing of hyperspectral images requires a huge amount of computation due to its natural high dimensionality. Instead of the typical three bands found in normal images (RGB), hyperspectral images can have more than one thousand bands.

This master thesis has the mail goal to accelerate part of the algorithms used in the HELICoiD project to classify the hyperspectral images using NVIDIA GPUs for the computation of the brain cancer presence/absence in real-time.

2.3 CLASSIFICATION ALGORITHM FOR HYPERSPECTRAL IMAGE ANALYSIS

Nowadays, machine learning is used in many research fields because it offers automated procedures that it allows to predict a behaviour based on multiples past observations. The purpose of this work is to use the machine learning for classification of hyperspectral images using supervised algorithms.

2.3.1 Supervised pixel-wise algorithms

Data analysis and machine learning have become an integrative part of the modern scientific methodology, offering automated procedures for the prediction of a phenomenon based on past observations, unravelling underlying patterns in data and providing insights about the problem. However, it must to be considered as a methodology and not as a black-box tool because it is necessary a rational thought process that is dependent on the problem under study. In particular, the use of algorithms should ideally require a reasonable understanding of their mechanisms, properties and limitations, in order to better understand and interpret their results **¡Error! No se encuentra el origen de la referencia.**.

Supervised pixel-wise algorithms are a specific branch from machine learning whose purpose is to use a set of observations, the training set, to find a decision function. The main feature is that it is necessary to have a labelled training data, therefore, the training data consists of an input object (usually a feature vector) and a desired output value.

There are two kinds of supervised learning: regression and classification. The main difference between both is that regression works with continues values, for example predicting fuel consumption of a car, while classification works with discrete values.

2.3.1.1 Random Forest (RF)

Random Forest (RF) is a supervised learning method that can be applied to solve classification or regression problem. It is constituted by a combination of tree predictors such that each tree depends on the values of a random vector **¡Error! No se encuentra el origen de la referencia.**. This method allows quick prediction but requires a long training phase when the dataset is large. Random Forest became recently popular in the computer vision community thanks to the great result that this method generates. One of the most famous applications of Random Forest is the recognition of human poses in real time to the Microsoft Kinect system [47].

Historically, the earliest mention of ensemble of decision trees is due to Kwok and Carter in 1990 but L. Breimann was one of the earliest to show, theoretically and empirically, that aggregating multiple versions of an estimator into an ensemble can give substantial gains in accuracy.

The main idea behind Random Forests is to generate many decision trees from the same dataset thus it is significantly improved in classification result accuracy by the vote for the most popular class realized by the trees [43]. The rationale behind this method is that the combination of learning models increases the classification accuracy.

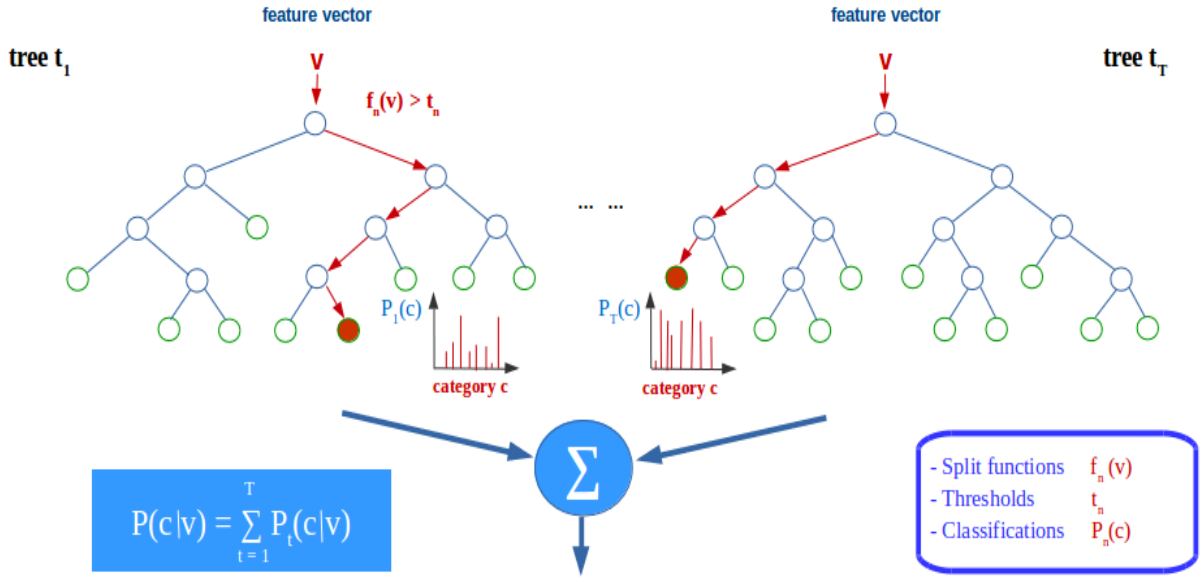


Figure 2-5: Random Forests representation

A decision tree consists of a hierarchy of questions that are used to map a multi-dimensional input value to a scalar output. The scalar output can be a real value (regression) or a class label (classification). In this paper, we focus on decision trees and forests for classification.

When the output space is a finite set of values, like in classification, where $y = \{c_1, c_2, \dots, c_j\}$, another way of looking at a supervised learning problem is to notice that Y defines a partition over the universe Ω, that is

$$\Omega = \Omega_{c_1} \cup \Omega_{c_2} \cup \dots \cup \Omega_{c_j} \quad (2\text{-}1)$$

where Ω_{ck} is the set of objects for which Y has value c_k. Thus, classifier φ can also be regarded as a partition of the universe Ω and it defines an approximation $\bar{Y}$ of Y. This partition however is defined on the input space χ rather than directly on Ω, that is

$$\chi = \chi_{c_1}^{\varphi} \cup \chi_{c_2}^{\varphi} \cup \ldots \cup \chi_{c_j}^{\varphi} \tag{2-2}$$

where $\chi_{c_k}^{\varphi}$ is the set of description vectors $x \in \chi$ such that $\varphi(x) = c_k$.

In this terms, a decision tree can be defined as a model $\varphi: \chi \rightarrow y$ represented by a rooted tree, where any node t represents a subspace $\chi_t \subseteq \chi$ of the input space (see Figure 2-6) **¡Error! No se encuentra el origen de la referencia.**.

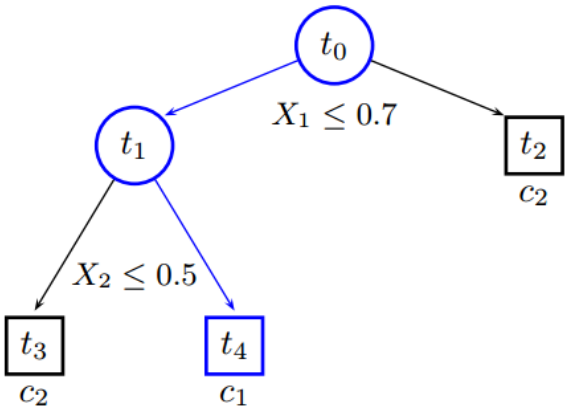


Figure 2-6: Decision tree example

As we could see in Figure 2-5 and it was explained above, Random Forest is a collection of classifiers that are structured as decision trees. The K decision trees in a random forest are trained independently and the output values for a set of descriptions vectors $x \in \chi$ are collected from all reached leaves in the decision trees and combined to generate a single classification. Common methods for generate a single classification are the majority defined by

$$P(c|x) = \sum_{k=1}^{K} P_k(c|x) \tag{2-3}$$

and the average of all probability distributions defined by

$$P(c|x) = \frac{1}{K} \sum_{k=1}^{K} P\left(c \middle| l_k(x)\right) \tag{2-4}$$

where $l_k(x)$ is the leaf node of the k-tree that has been reached when the classification is finished.

2.3.1.1.1 Random Forest Training

A characteristic of RF is that it works with random decision trees and the key differences between a decision tree and a random decision tree is the training phase. The idea of RF is to training multiple trees on a random subset of the dataset and a random subset of features.

A learning set of L consists of data $\{(y_n, x_n), n = 1, .., N\}$ where the $y's$ are class labels in a classification problem, albeit it could be a numerical response in a regression problem. In RF, we are given a sequence of learning sets $\{L_K\}$ each one consisting of N independent observations from the same underlying distribution as L. The main idea is to use the $\{L_K\}$ to get a better predictor than the single learning set predictor $\varphi(x, L)$, working with a sequence of predictors $\{\varphi(x, L_k)\}$ **¡Error! No se encuentra el origen de la referencia.**.

When we are training the RF, usually, we have a single learning set L and for this reason we use bootstrapping (random sampling with replacement) and take repeated bootstrap samples $\{L^{(B)}\}$ from L, and form a sequence of decision trees $\{\varphi(x, L^{(B)})\}$. This procedure is called bagging.

With the bootstrap sample, the next step in each node of the tree is the random selection of features. It is necessary to remember that x is a feature vector with M input variables or features, i.e $x_n = \{f_1, f_2, \dots, f_M\}$. For each node of the tree, $F \ll M$ features are randomly selected and thus each node tree trains with a random subset of features x_n^t .

Those previous steps for the training of each tree is necessary because the classification in RF is the result by the votes of the trees and, for this reason, it is necessary that the decision trees have the lowest possible correlation, remembering that the procedure of random selection of feature is realized in each node of each tree.

A common process to train a decision tree is Top-Down Induction of Decision Trees (TDIDT) which consists of two training phases **¡Error! No se encuentra el origen de la referencia.**:

1. Iterative growing the decision tree until a stopping criterion is reached
    a) Selecting a leaf node that is not yet pure.
    b) Selecting the best test that minimizes the impurity score.
    c) Splitting the leaf node into left and right according to the selected test.
    d) Continuing with a).
2. Prune the decision tree using a validation set.

In contrast to normal decision trees, in RF the random decision trees are developed to its maximum expansion, not pruned after training as they are less likely to overfit [43]. Breiman's random forests use Classification And Regression Tree (CART) [51] as tree growing algorithm [43] and are restricted to binary trees for reasons of simplicity. Whether the decision tree is balanced depends on the dataset and the impurity score function used for training.

2.3.1.1.2 Error estimation

Such as we could see above, RF is a bagged model, the trees are repeatedly fit to bootstrapped subsets of the observations, and in this kind of models there is a very simple way to estimate the test error without the need to perform cross-validations or the validation set approach. Thereby, the classification or regression error in RF is defined by the OOB (out-of-bag) concept. In RF, on average, each tree bagging uses two-thirds of the observations, the remaining third will not be used in the comments off-exchange (OOB). We can predict the response for the i^{th} observation using each of the trees in which that observation was OOB. In order to obtain a single prediction for the i^{th} observation, we can take a majority vote in classification or we can average these predicted responses for regression. An OOB prediction can be obtained in this way for each of the n observations, from which the overall OOB MSE (for a regression problem) or classification error (for a classification problem) can be computed.

The resulting OOB error is a valid estimate of the test error for the bagged model, since the response for each observation is predicted using only the trees that were not fit using that observation.**¡Error! No se encuentra el origen de la referencia.¡Error! No se encuentra el origen de la referencia.**.

2.3.1.1.3 Split selection

When we develop a classification decision tree, it is necessary to find the best split value in non-terminal nodes, the root and other internal nodes. The splitting is performed based on measures of the degree of impurity of the child nodes.

The Gini index has the purpose to evaluate the measure of impurity degree and the attribute that provides the largest reduction in impurity is chosen to split the node [47]. In the Gini index the impurity of the child nodes is being calculated by:

$$GINI(t) = \sum_{c \in y} P(t|c)(1 - P(t|c)) = \sum_{c \in y} P(t|c) - P(t|c)^2 = 1 - \sum_{c \in y} P(t|c)^2 \qquad (2\text{-}5)$$

where c belongs to set of classes $y = \{c_1, c_2, \dots, c_j\}$ and $P(t|c)$ is the relative frequency of class c in the node t **¡Error! No se encuentra el origen de la referencia.**.

2.4 GPUs for General-Purpose Computing

For 30 years, the most important method for improving the performance of computing devices has been to increase the speed at which the processor's clock operated. However, in recent years, manufacturers have been forced to look for alternatives because of heat and power restrictions as well as a rapidly approaching limit to transistor size **¡Error! No se encuentra el origen de la referencia.**. For this reason, the two main PC processor manufacturers, Intel and AMD, have had to a different approach, adding more cores to processors and trying to increase CPU clock rates and extract more instructions per clock through instruction-level parallelism.

Due to the limitations of the single-core processors, the use of last generation massively parallel hardware architectures such as Graphics Processing Units (GPUs) has become extremely popular in the high-performance computing area **¡Error! No se encuentra el origen de la referencia.**.

The GPU design philosophy (many-cores architecture) allows to perform better than a CPU (multi-core architecture). In a GPU, the idea is to optimize the throughput of many threads running in parallel, so that if any of them is waiting for the completion of an operation, it is assigned new work and it does not remain idle. However, the CPU provides better performance for sequential solutions.

As a result of the explained above, in a CPU the optimization is based in a complex logic control for the parallel execution of sequential code and the uses of caches to reduce latencies while a GPU uses a simpler control and it uses smaller caches to help to maintain the bandwidth defined by all the parallel threads [54].

Since 2001, GPUs use shaders programmable which are the data processing units in GPUs. Generally, each GPU has many shaders and with them the GPU can be programmed by aiming to increase the GPU works to process the data graphics in parallel. On a modern GPU, shader number or often called the Stream Processor (for stream input and output), has reached the hundreds or even thousands. GPU calculation abilities can reach Teraflops. The comparison of the GPU and the CPU is show in Table 2-1 and a schematic comparison in Figure 2-7.

GPUs are based in the single instruction, multiple data (SIMD) architecture. This architecture allows exploiting abundant data level parallelism while reducing power consumption in instruction fetch, decode, and issue. Basically, it means that this architecture can process elements that perform the same operation on multiple data points simultaneously, there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment. SIMD capability is widely adopted in general-purpose architectures and we can see in Cell Broadband Engine and Many Integrated Cores [58]. For instance, most modern CPU designs include SIMD instructions in order to improve the performance of multimedia use.

Nowadays, GPUs are devices presents in any PC. They perform basic task for the CPU such as rendering an image in memory and then displaying that image onto the screen **¡Error! No se encuentra el origen de la referencia.**. For this reason, GPUs are one of the most popular devices for general-purpose computing and the evolution of them has experienced a great growth thanks to the great demand of tasks in which they can be applied.

CPU	GPU
Parallelism through time multiplexing	Parallelism through space multiplexing
Emphasis on low memory latency	Emphasis on high memory throughput
Allow wide range of control flows + control flow optimization	Optimized for data parallel, throughput computation
Very high clock speed	Mid-tempo clock speed
Peak computation capability low	Higher peak computation capability
Off-chip bandwidth lower	Higher off-chip bandwidth
Handle sequential code well	Requires massively parallel computing
CPU are great for task parallelism	GPU are great for data parallelism

Table 2-1: Comparison between CPU and GPU

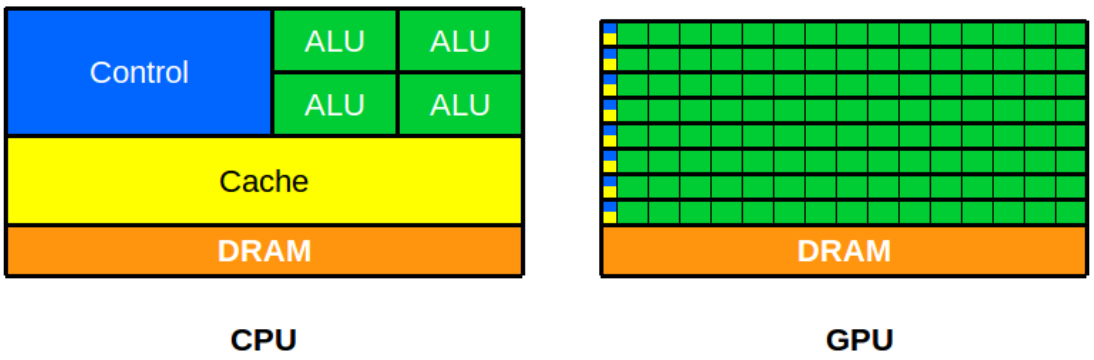


Figure 2-7: CPU vs GPU schematic comparison

2.5 CUDA OVERVIEW

NVIDIA unveiled the first GPU with CUDA architecture in November 2006, the Geforce 8800 GTX. With this GPU, NVIDIA intended this new family of graphics processors to be used for general-purpose computing **¡Error! No se encuentra el origen de la referencia.**.

Driven by the insatiable market demand for real-time, high-definition 3D graphics, the GPU has evolved into a highly parallel, multi-thread, many core processors with tremendous computational power and very high memory bandwidth **¡Error! No se encuentra el origen de la referencia.¡Error! No se encuentra el origen de la referencia.**[59] as illustrated on Figure 2-8.

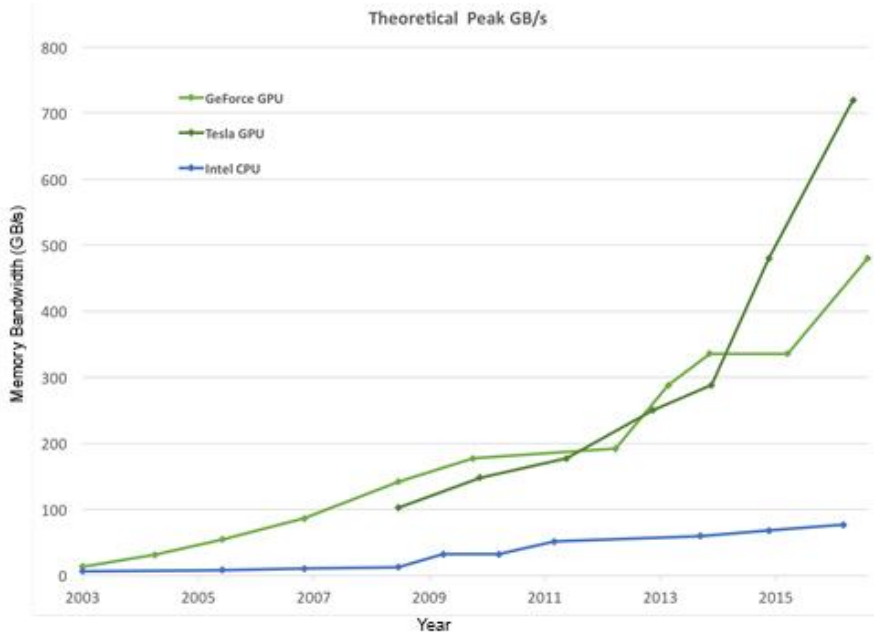


Figure 2-8 Memory Bandwidth for the CPU and GPU

2.5.1 Architecture

The CUDA performance and scalability lies in the simple partitioning of a computation into fixed size blocks of threads in the execution configuration. CUDA allows to efficiently solve many problems, parallelizing the problem to a scale that CPUs cannot do.

The CUDA architecture is built around an array of multithreaded streaming multiprocessors, known as SMs, in a NVIDIA GPU. The SM is a common architectural building block that a GPU replicate repeatedly. The number of SMs of CUDA GPUs can vary from a generation to another. The data-parallel computing kernels in an application are off-loaded for concurrent execution on the GPU device, while the remainder of the application is executed on the CPU host [55][59]. An example of what has just been explained, can be observed in Figure 2-9, where the CUDA programming model assumes that the CUDA threads are executed on a physically separate *device* that operates as a coprocessor to the *host* running the C program. For example, this is the case when the kernels execute in a GPU and the rest of the code is executed on a CPU.

As we saw above, each graphics card consists of several SMs and in each SM is attached many Stream Processors (SPs) which share instructions cache and control logic. The original 9800 GTX had 8 SMs, with 16, SPs, giving a total of 128 SPs **¡Error! No se encuentra el origen de la referencia.**. In the CUDA architecture this SPs are known as CUDA Cores. These multiprocessors are designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called SIMT (Single-Instruction, Multiple-Thread).

The SIMT architecture is akin to SIMD vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organization exposes the SIMD width to the software, whereas SIMT instructions specify the execution and branching behaviour of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads [55].

In the SIMT architecture, the multiprocessor creates, manages, schedules and executes treads in groups of 32 parallel threads called warps. Every thread of a warp starts together at the same program address, but each thread has his own instruction address counter and register state and are therefore free to branch and execute independently. When a multiprocessor must use one or more blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution [55].

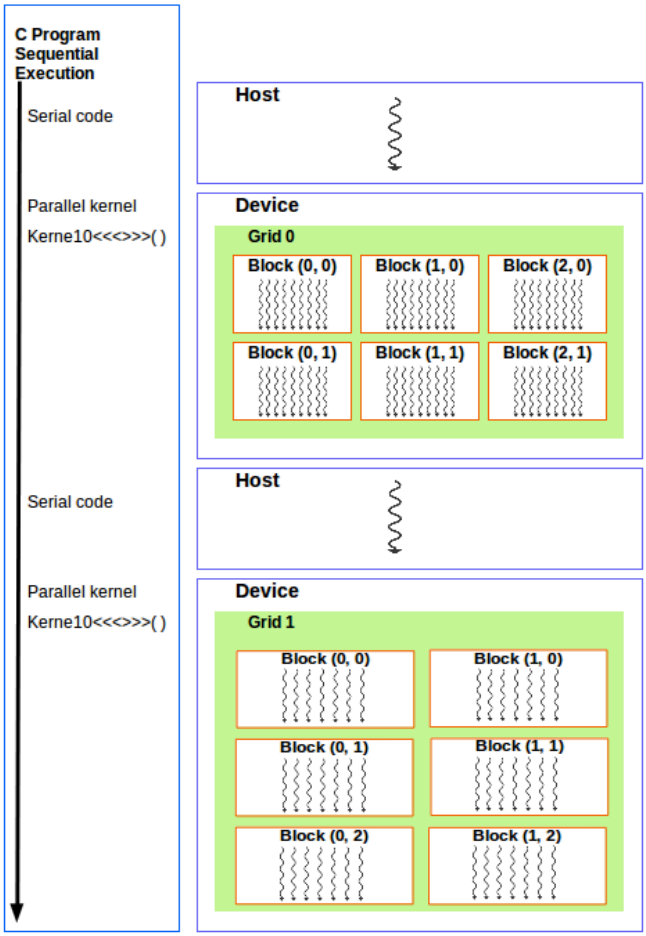


Figure 2-9 Execution of a CUDA Program (Heterogeneous Programming)

A warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. When we are working using CUDA, we have to understand that the index of a thread and its thread ID relate to each other in a direct way. The index of a thread is representing by a *threadIdx*, which is a 3-component vector and, for this reason, the thread can be identified using one-dimensional two dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

2.5.2 Memory Hierarchy

The CUDA threads dispose multiple memory which can be accessed during their execution process as illustrated in Figure 2-10. Each thread has private local memory, only used by this thread. Each block has shared memory visible to all threads of the block and with the same lifetime as the block. The global memory can be accessed by all threads, and it is the biggest memory that the GPU

device has. There are another two additional read-only memory spaces that can be accessed by each of the threads: the texture memory and the constant memory spaces. Every memory space include the global, constant and texture memory spaces and are optimized for non-similar memory functions [59][55].

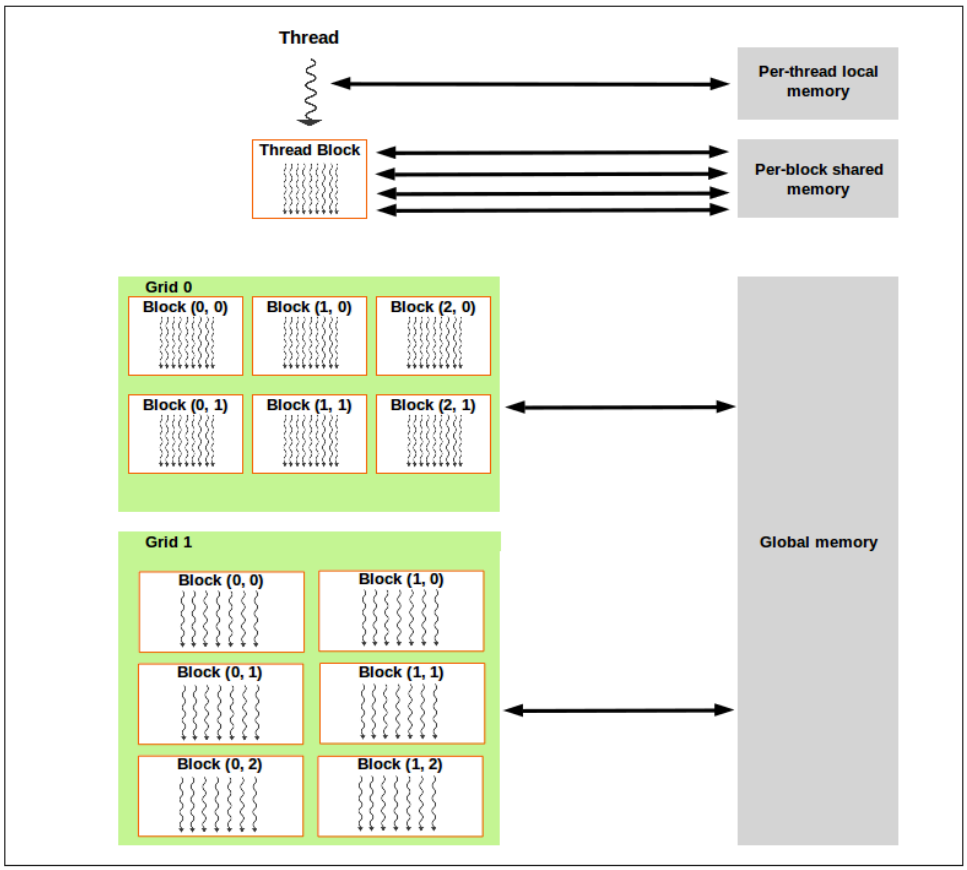


Figure 2-10 Memory Hierarchy

2.5.2.1 Global Memory

Global memory is the biggest off-chip memory, and basically all data resides in global memory during a kernel execution. This memory is very interesting because it is writable from both the GPU and the CPU. It can be accessed from any device on the PCI-E bus **¡Error! No se encuentra el origen de la referencia.**.

As discussed above, it can be accessed by all threads but the latency of accessing is hundreds of cycles greater than the rest of memories. All the same, this latency can be hidden by the large number of threads execution. Even though the bus between global memory and the SMs is quite wide, the massively parallel execution can easily be saturated by the limited bandwidth, and this often becomes the performance bottleneck in CUDA programs.

The development of CUDA programs must be careful in the use of the global memory, for this reason, understanding how to efficiently use global memory is essential for CUDA programmer.

Due to host memory copy, directly in GPU's global memory, all data are first presented in global memory, so data must be pre-fetched from global memory to lower latency memory, in general, the best option is the shared memory. However, CUDA Toolkit Documentation [55] has a specific

article dedicated to the optimization, and some chapters are devoted to optimization of global memory using, for instance, access pattern.

2.5.2.2 Shared Memory

Shared memory is the name given to the memory that threads from a same block can access (see Figure 2-10). As a programmer, you can use the CUDA Keyword *__shared__* to make a variable resident in the shared memory.

The CUDA Compiler creates a copy of the variable in the shared memory for each block that the programmer launch on the GPU. Every thread in that block shares the memory but, threads cannot see or modify the copy of this variable that is seen within other blocks. Unlike global memory, shared memory is an on-chip memory, it means that this memory resides physically on the GPU. Because of this, the latency to access shared memory tends to be far lower than typical buffers.

CUDA developers can efficiently communicate every thread within a block with the use of the shared memory, but if we expect to communicate between them, it is necessary a mechanism for synchronizing between threads. In CUDA C, *__syncthreads()* is used for synchronize threads in the same block **¡Error! No se encuentra el origen de la referencia.**. This call guarantees that every thread in the block has completed their instructions prior to the *__syncthreads()* before the hardware will execute the next one.

2.5.2.3 Constant Memory

The constant memory is an off-chip memory, albeit supports low latency and a high bandwidth because it has an associated cache on-chip. The host is responsible for writing on it **¡Error! No se encuentra el origen de la referencia.**[54]. There are two reasons for reading from a constant memory (64KB) that can save bandwidth over standards reads of global memory:

- A single read from constant memory can be broadcast to other nearby threads.
- Constant memory is cached, so consecutive reads of the same address will not incur any additional memory traffic.

2.5.2.4 Texture Memory

Like constant memory, texture memory is another variety of read-only memory that can improve performance and reduce memory traffic when reads have certain access patterns. Although texture memory was originally designed for traditional graphics applications, it can also be used in some GPU computing applications.

The texture units were designed for OpenGL and DirectX rendering pipeline but the texture memory may also be used for general-purpose computing. Like constant memory, the texture memory has an associated cache on-chip and this memory allow to reduce memory requests to off-chip DRAM.

This memory was designed for graphical applications where memory access patterns has a strong spatial locality.

Thereby, to use the texture memory is necessary to declare the inputs such as texture references. For instance: *texture<float> constSrc*. Nevertheless, we need to bind the references to the memory buffer using *cudaBindTexture()*. This basically tells the CUDA runtime two things:

- We intend to use the specified buffer as a texture.
- We intend to use the specified texture reference as the texture's "name".

When we are using Texture memory we cannot use square brackets to read from buffers; we need to use *tex1Dfetch()* when reading from memory. *text1Dfetch()* is a compiler intrinsic, for this reason we can no longer pass the input and/or output buffers like parameters because the compiler needs to know at compile time which textures *tex1Dfetch()* should be sampling.

Some problems can arise having two dimensional domains and therefore it can be convenient to use two-dimensional texture memory: e.g. *texture<float, 2> constSrc*. When we are working with two-dimensional texture memory, we access to the memory using *tex2D()* instead of using *tex1Dfetch()* and the bind reference using *cudaBindTexture2D()* **¡Error! No se encuentra el origen de la referencia.**.

2.6 NVIDIA TESLA K40

In order to accelerate most demanding High-Performance Computing (HPC) challenges, NVIDIA releases the Tesla family of GPUs. NVIDIA Tesla is the world's leading platform for accelerated datacentre, deployed by some of the world's largest supercomputers and enterprises.

The Tesla K40 appear in 2013 and it is based in NVIDIA Kepler compute architecture which was a more efficient architecture than the previous generation Fermi architecture.

Tesla K40 is a PCI-E, dual-slot computing module in the Tesla (267 nm length) form factor comprised of a single GK110B GPU. This GPU is designed for servers and it has 12 Gb of GDDR5 on-board memory and his main demand is for the use of High-Performance Computing (HPC). The Table 2-2 contains K40 key features [61][62].

Kepler architecture introduces some technologies improvements to obtain a better perfomance, compared with the previuos generation. Kepler architecture is the first one that enables GPU threads to automatically spawn new threads, this technology is known as Dynamic Parallelism. Hyper-Q technology allows multiple CPU cores to simultaneously use the CUDA cores on a single Kepler GPU, increasing GPU utilization and slashes CPU idle times.

Tesla K40	
Cuda capability	3.5
Peak double-precision floating point performance (board)	1.43 Teraflops
Peak singlle-precision floating point performance (board)	4.29 Teraflops
Number of processor cores	2880
Base core clocks	745 MHz
Boost core clocks	810 MHz and 875 MHz
Memory clock	3.0 GHz
Memory bandwidth	288 GB/sec
Total board memory (GDDR5)	12 GB

Table 2-2: Tesla K40 features

2.6.1 Kepler GK110 GPU Architecture

NVIDIA aims to cover high performance parallel computing demand that have increase across many areas of science, medicine, engineering, etc. NVIDIA's Kepler GK110 GPUs are designed to help solving the world's most difficult computing problems. One of the main premises of Kepler GK110 was the simplification in the creation of parallel programs and further revolutionize the high performance computing issues.

Kepler GK110 was built first and foremost for Tesla. The GK110 GPU exceeds the raw computational power delivered by Fermi architecture and it does it by efficiently reducing the consumption and generating less heat output.

A full Kepler GK110 include 15 SMX and six 64-bit memory controllers [56]. In the Figure 2-11 we can see the full chip diagram of a GK110.

A principal design goal for the Kepler architecture was improving power efficiency. The manufacturing in 28 nm was an importance role in lowering power consumption but NVIDIA engineers needed realize many modifications in the GPU architecture to further reduce power consumption while maintaining great performance.

Kepler GK110 supports CUDA capability 3.5, we can see a brief overview in the Table 2-3.

2.6.1.1 Streaming Multiprocessor (SMX) Architecture

In the GK110, each SMX units feature 192 single-precision CUDA cores, and each core has fully floating-point and integer arithmetic logic units.

NVIDIA engineers aimed to increase the GPU's delivered double precision performance because double precision arithmetic is at the heart of many HPC applications. Kepler GK110's SMX also retains the special function units (SFUs) for fast approximate transcendental operations as in previous- generation GPUs, providing 8x the number of SFUs of the Fermi GF110 SM [56].

For the first time, the cores within the GK110 SMX unit used the primary GPU clock. In previous architecture, the cores inside SMX used the 2x shader clock.

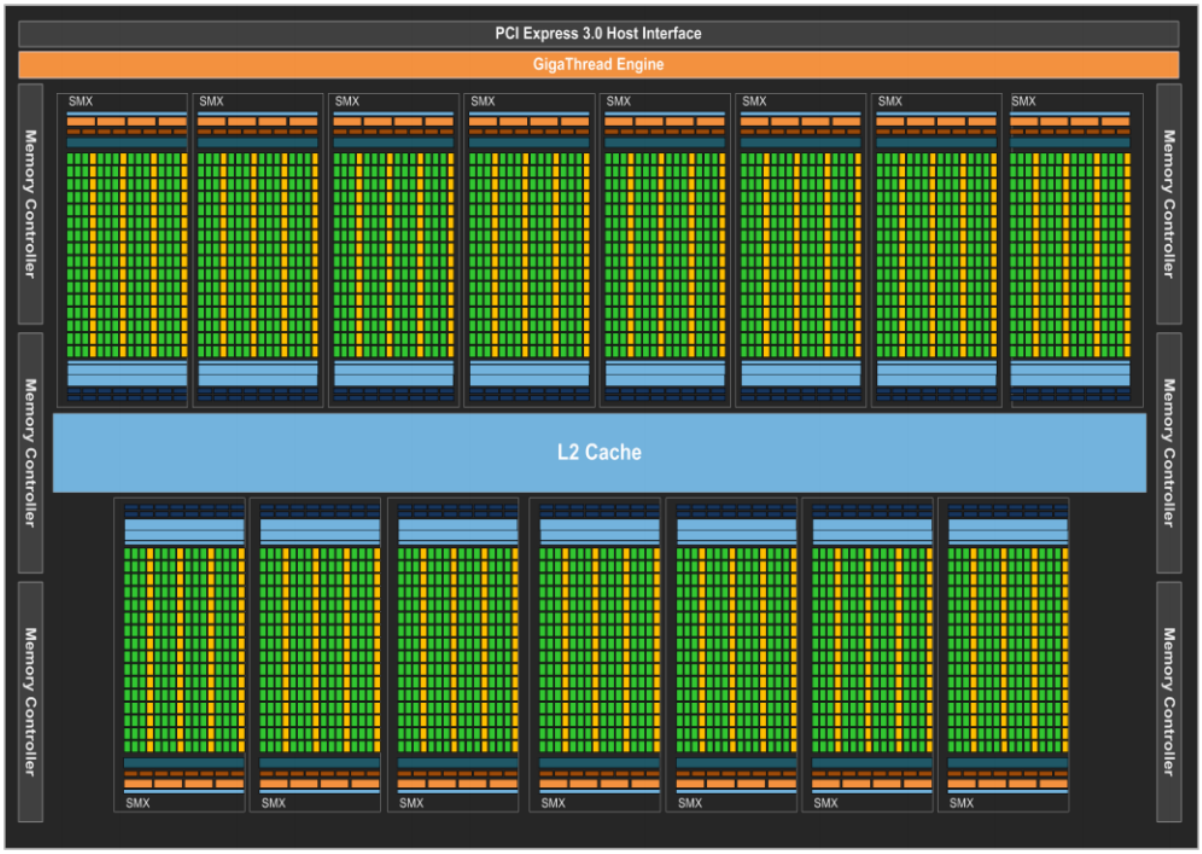


Figure 2-11: Kepler architecture.

Compute Capability	3.5
Threads / Warp	32
Max Threads / Thread Block	1024
Max Warps / Multiprocessor	64
Max Threads / Multiprocessor	2048
Max Thread Blocks / Multiprocessor	16
32-bit Registers / Multiprocessor	65536
Max Registers / Thread Block	65536
Max Registers / Thread	255
Max Shared Memory / Multiprocessor	48K
Max Shared Memory / Thread Block	48K
Max X Grid Dimension	2^{32}-1
Hyper-Q	Yes
Dynamic Parallelism	Yes

Table 2-3: Compute capability of Kepler GPUs

About the warp scheduler, recall the SMX schedules threads in groups of 32 parallel threads called warps. Each SMX features four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. Kepler has a quad warp scheduler and it can select four warps, and two independent instructions per warp can be dispatched each cycle. Kepler GK110 allows double precision instructions to be paired with other instructions.

2.6.1.2 Kepler Memory Subsystem – L1, L2, ECC

The Kepler architecture supports a unified memory request path for loads and stores, with an L1 cache per SMX multiprocessor. Kepler GK110 also enables compiler-directed use of an additional new cache for read-only data.

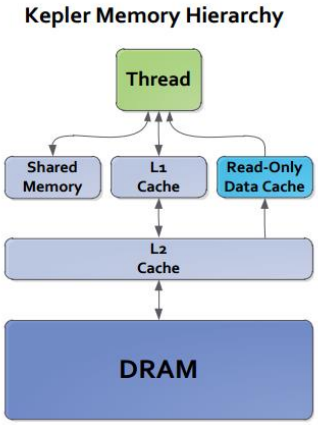


Figure 2-12: Kepler Memory Hierarchy

Kepler SMX has 64 Kb of on-chip memory that can be configured as 48 Kb of shared memory with 16 Kb of L1 cache, 16 Kb of shared memory and 48 Kb of L1 cache or 32 Kb/32 Kb split between both memoirs. The shared memory bandwidth is 256 B per core clock.

Kepler introduced a 48 Kb cache for data that is known to be read-only for the duration of the function. About the L2 cache memory, it has double the amount of L2 available in the Fermi architecture getting 1536 Kb.

2.7 SUMMARY

Hyperspectral image analysis methods have been intensively investigated in the remote sensing area and currently their development and application have been extended to other fields. It is an emerging imaging modality for medical applications, especially in disease diagnosis and image-guided surgery, this technology shows some advantages compared to the current techniques employed for cancer detection.

The study of in-vivo tissue allows providing a non-invasive disease diagnosis and an automatic guidance tool for surgery. After a cancer resection surgery, a pathology report notes about the surgical margins, that can be one of the following three types: clear, positive, or close. In case of clear margin, normal tissue surrounds cancer cells. In case of positive margin, cancer cells appear on the margins, which would lead to an additional surgery to remove the remaining cancer tissue. In case of close margin, cancer cells are close to the edge of the tissue, but not right at the edge. More surgery may be needed.

Surgery is one of the major treatment options for brain tumours, despite the improvements in the accuracy of brain tumour resections, neurosurgery is still unable to accurately define brain tumour margins in real-time. Under these circumstances, hyperspectral imaging arises as a potential solution that allows a precise detection of the edges of the malignant tissues in real time.

HELICoiD is a European collaborative project and its main goal is to efficiently differentiate between healthy and diseased tissues and so lead to better surgical removal using the

aforementioned hyperspectral images. This Master Thesis has the mail goal of accelerating part of the algorithms used in the HELICoiD project to classify the hyperspectral images using NVIDIA GPUs for the computation of the brain cancer presence/absence in real-time.

The classification is made by supervised pixel-wise algorithms, concretely, in this project we use the Random Forest algorithm. Supervised pixel-wise algorithms are a specific branch from machine learning whose purpose is to use a set of observations, the training set, to find a decision function. The main feature is that it is necessary to have a labelled training data, therefore, the training data consists of an input object (usually a feature vector) and a desired output value.

Random Forests is a supervised learning method that can be applied to solve classification or regression problems. It is constituted by a combination of tree predictors such that each tree depends on the values of a random vector. This method allows a quick prediction but requires a long training phase with a huge dataset.

The main idea behind Random Forests is to generate several decision trees from the same dataset. It will significantly improve the classification result by means of the vote for the most popular class spanned by the trees. In the training phase, the idea is to training multiple trees on a random subset of the dataset and a random subset of features. The error estimation is defined by the Out-Of-Bag concept and the split selection can be evaluated with different methods, for instance the Gini index.

As explained above, the main goal of this project is to accelerate hyperspectral images classification by using GPUs, specifically CUDA GPUs. For this reason, we will use a NVIDIA Tesla K40, a GPU develop by NVIDIA with the purpose of covering high performance parallel computing applications that have increased their computational requirements across many areas of science, medicine, engineering, etc. This GPU is based in the Kepler architecture with capability 3.5.

CHAPTER 3: HYPERSPECTRAL IN-VIVO BRAIN TISSUE DATABASE

3.1 INTRODUCTION

This section will describe the hyperspectral in-vivo brain tissue database employed to perform the experiments to evaluate the improvements performed in the hyperspectral classification algorithm. Moreover, the intra-operative hyperspectral acquisition system and the process to obtain and label the images during a neurosurgical operation will be briefly described.

3.2 INTRA-OPERATIVE HYPERSPECTRAL ACQUISITION SYSTEM

In order to obtain the hyperspectral images of the in-vivo human brain surface during the neurosurgical operations, the HELICoiD project has built a demonstrator capable of simultaneously obtaining two hyperspectral cubes [57]. The two hyperspectral cameras selected are the Hyperspec® VNIR A-Series and the Hyperspec® NIR X-Series, manufactured by HeadWall Photonics, Massachusetts, USA. The VNIR (visible and near infrared) camera ranges between 400 nm to 1000 nm. The NIR (near infrared) camera ranges between 900 nm to 1700 nm.

Figure 3-1 shows the main parts of the demonstrator. The most important elements of the system are located in the acquisition scanning platform. Table 3-1 presents the specifications of the two push-broom hyperspectral cameras. These cameras are fixed in a scanning unit composed by a stepper motor and a screw with a maximum path of 230 mm and a step resolution of 6.17 μm. Furthermore, a cold light emitter is located together with the cameras. The cold light emitter is connected to a 150 W Quartz Tungsten-Halogen system (QTH) (Figure 3-2.c), which offers broadband emission in the VIS (visible) and NIR spectral ranges (400 nm to 2200 nm), through an optical fibre. This system isolates the high temperatures produced by the halogen lamp, avoiding a direct emission to the brain surface.

Data pre-processing system is composed by a high-performance computer which manages the entire system, especially the acquisition scanning platform and the interaction with the user through

the graphical user interface (GUI). Finally, the processing sub-system platform has the goal of performing the hyperspectral classification in order to achieve the results in real-time.

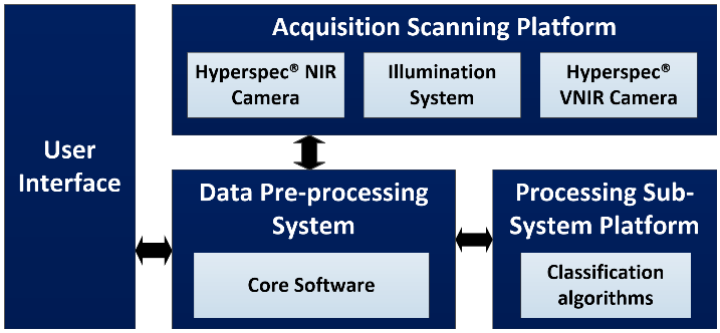
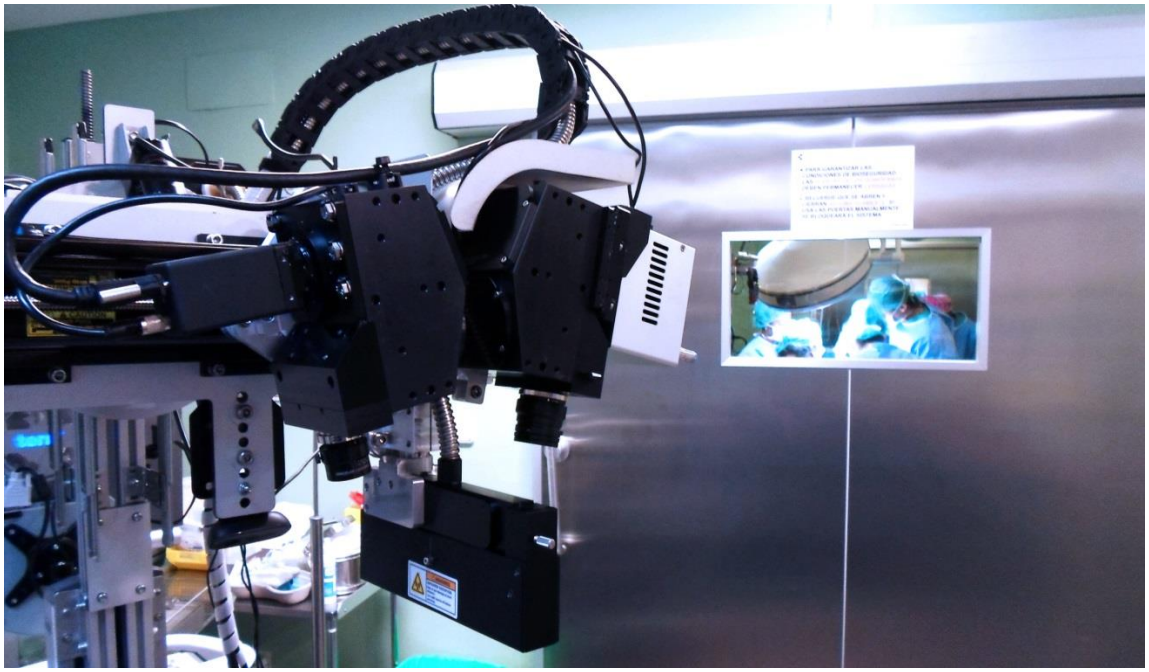


Figure 3-1: HELICoiD demonstrator main parts.

Figure 3-2.b shows the HELICoiD demonstrator inside the pre-operative area at the University Hospital Doctor Negrín in Las Palmas de Gran Canaria, Spain. Figure 3-2.a presents the acquisition platform where the cameras and the cold light element are located. On the left side of the platform, the VNIR camera is located, and on the right side, the NIR camera is placed. In the middle of the two cameras, the cold light emitter is located. These three elements are correctly aligned in order to obtain the images properly illuminated. Figure 3-2.d displays the stepper motor controller, which is in charge of managing the scanning platform shift.

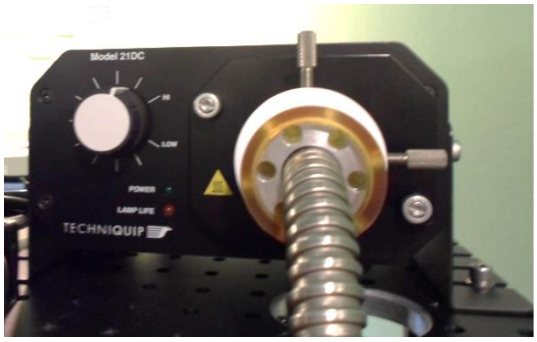
	Hyperspec® VNIR	Hyperspec® NIR
Spectral range (nm)	400 – 1000	900 – 1700
Spectral resolution (nm)	2 – 3	5
Slit (μm)	25	25
Spatial bands	1004	320
Spectral bands	826	172
Frame Height (FOV) (mm)	129.21	153.6
Pixel Dimensions (IFOV) (mm)	0.1287	0.4800
Max Pixels per Frame	1004	320
Max Frames per Capture	1825	489
Dispersion per pixel (nm)	0.74	4.8
Detector array	Silicon CCD	InGaAs
Frame rate (fps)	90	100

Table 3-1: Camera Specifications

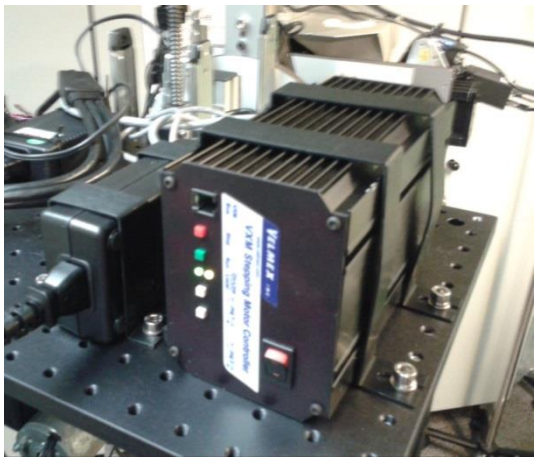


(a)

(b)



(c)



(d)

Figure 3-2: (a) Acquisition scanning platform, (b) complete HELICoiD demonstrator located in the pre-operative room of the neurosurgical operating theatre at the University Hospital Doctor Negrín, (c) Quartz Tungsten-Halogen system and (d) stepper motor controller.

3.3 HYPERSPECTRAL BRAIN IMAGE DATABASE

This section provides an overview of the procedure carried out to obtain the hyperspectral images of naked brain surface that have been generated in the HELICoiD database. Furthermore, the process to label the samples for the supervised algorithm development is described. Finally, the dataset of the images selected for the hyperspectral algorithm evaluation is presented.

3.3.1 Capturing Hyperspectral Images During Surgery

Before the operation, the patient has an Image Guide Stereotactic (IGS) compatible CT (Computed Tomography) and MRI (Magnetic Resonance Imaging) head which are up loaded on to the IGS system. Patient undergoes general anaesthesia and placed in a supine position and registered on to the IGS system. A scalp incision is made and the skull exposed before a burr hole/s is drilled using a high-speed drill. A craniotome is then inserted into the burr hole/s and a bone flap is cut out (craniotomy) is created using a craniotome. The dura is then cut with a knife (durotomy) to expose the brain surface.

Using the HELICoiD demonstrator, an in-vivo human brain hyperspectral image database has been created. The hyperspectral cubes have been obtained from 22 different patients at the University Hospital Doctor Negrín. The type of tumours captured in this study involves both primary secondary brain tumours. In order to obtain the samples correctly labelled, the four steps flowchart presented in Figure 3-3 has been followed. In the following sub-sections, each one of these steps will be described.

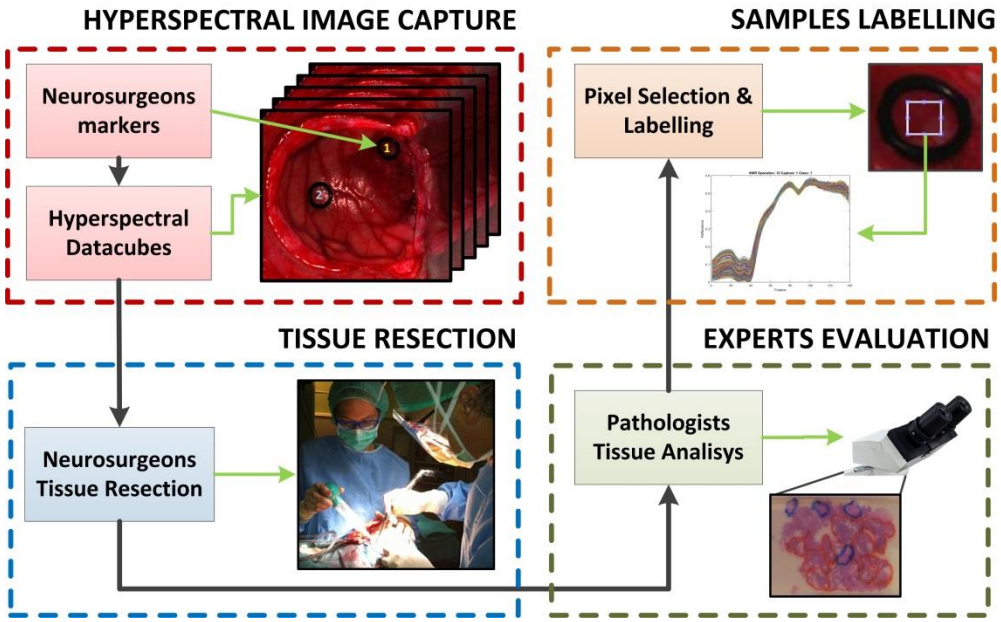


Figure 3-3: Data capture and labelling process.

3.3.1.1 Hyperspectral image capture

Initially, images are captured after durotomy before the arachnoid and pia have been breached if the tumour comes to the surface on imaging. If the tumour can be seen on the surface, two sterilised rubber ring markers are placed to identify the position of the tumour and that of normal brain as

judged by the operating surgeon where he/she can be quite confident that the brain tissue is normal. This judging is done based on visual appearance, anatomical relationship to sulci and gyri an image guidance feedback. If possible, the exact location of the markers is noted using the IGS system pointer so as to identify the location of the markers over the brain. This pointer allows knowing the position of the rubber ring markers with respect to a MRI or CT performed previously to the patient for the surgical procedure. Figure 3-4 and Figure 3-5 illustrates the use of the IGS system pointer to identify the position of the markers in a MRI.

After that, the operator of the HELICoiD demonstrator captures the hyperspectral image of the exposed brain surface with and without markers. The markers offer an area of the image where the pixels can be labelled with the surgeon prior evaluation and then, this prior evaluation is contrasted with the pathology results, as the inside of the rubber ring markers will be sent to pathology.

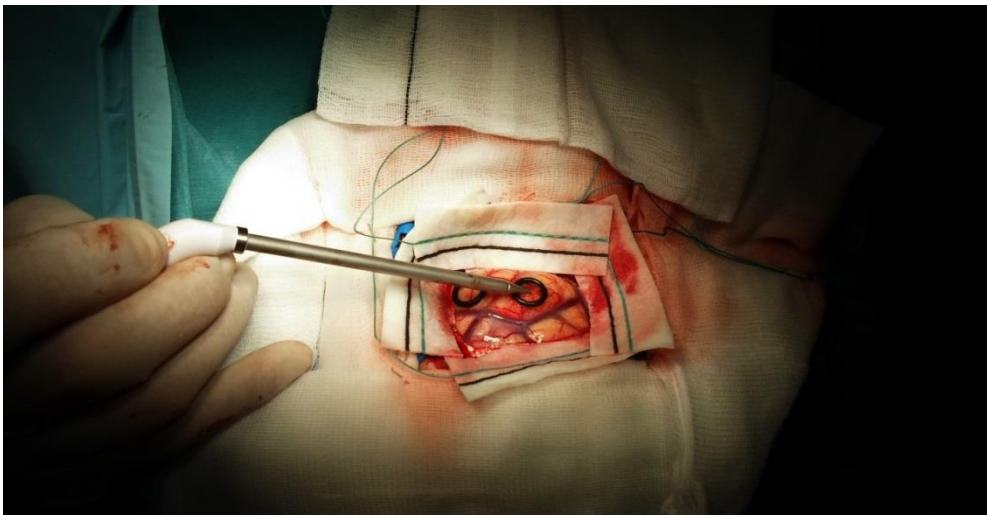


Figure 3-4: IGS system pointer over the HELICoiD tumour marker located on the exposed brain surface.

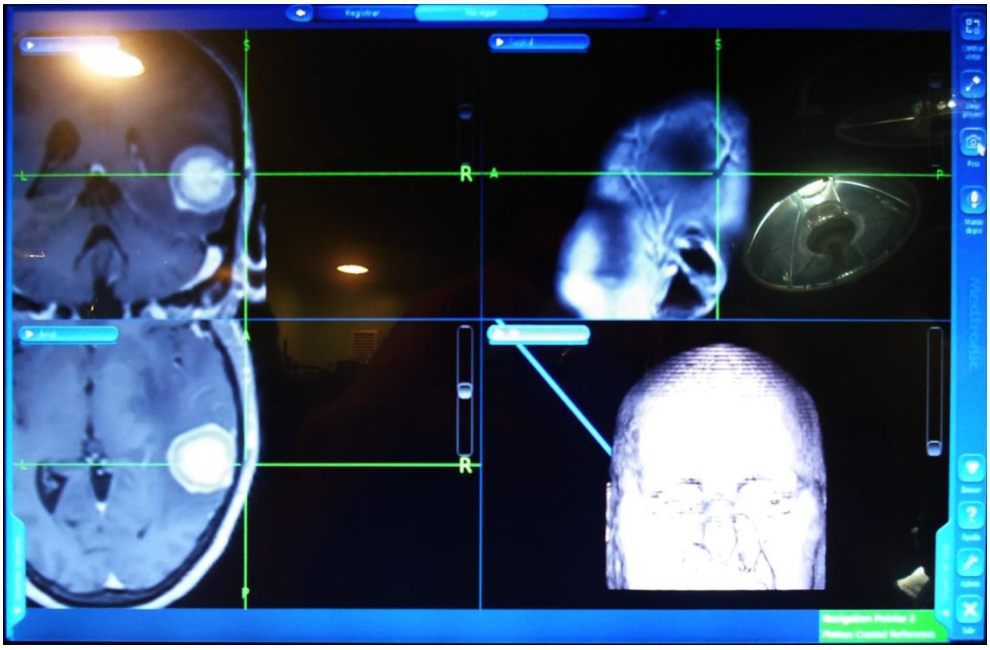


Figure 3-5: IGS system screen capture with the coordinates of the tumour marker in the MRI.

3.3.1.2 Tissue resection

After the first hyperspectral image capturing, the HELICoiD demonstrator is moved out from the surgical zone and neurosurgeons start the tumour resection. They take a sample of the tissue that is inside of the tumour marker. The sample is stored in a container with the HELICoiD label and it is assigned a number corresponding with the marker. These samples will be sent to pathology and the result will be used as the ground truth reference for the algorithm development. The numbers assigned will help in the labelling process.

The second set of images is captured while the tumour is gradually resected. At a time when there is macroscopically normal brain and tumour exposed and when the surgeon feels it is safe to halt surgery, the surgeon ensures perfect haemostasis then washes the field thoroughly with warm saline to wash away any residual blood while ensuring no significant temperature change (and resultant blood flow change). Following this, the field is sucked dry by application of a cottonoid to the parenchyma and applying suction to this. The operating surgeon then identifies the most suitable location to capture the images. Markers are again placed on to the tumour and on to another area, which the operating surgeon judges as normal brain. Hyperspectral images are then obtained again with and without markers in-situ. Again, tissue samples are obtained from the position of the markers and sent to pathology laboratory for tissue diagnosis. The number of markers using in each surgery is variable and depends on the nature and characteristics of the tumour.

3.3.1.3 Experts evaluation

All resected tissue is sent to the neuropathology laboratory where it is formalin fixed and undergoes standard H&E staining and any further required staining to establish a definitive histopathological diagnosis. Neuropathologists are the only ones that can determine if a tissue inside of the marker is or not tumour by using off-line histopathologic techniques. Samples are classified as tumour or brain and for tumour samples, these are further subdivided into tumour type and grade while normal brain samples are classified as white or grey matter.

3.3.1.4 Samples labelling

In the last step, by using the information provided by the pathologists, some pixels of the image are labelled, using a labelling tool developed in MATLAB, in order to generate the ground truth for the training of the supervised algorithm. This labelling tool is used by the neurosurgeon that performs the correspondent neurosurgical operation. The neurosurgeon selects a pixel from a RGB image to be labelled based on the visual appearance. Once the reference pixel has been selected, due to the difficulty of assigning a pixel to a certain class with certain degree of assurance, the Spectral Angle (SA) between the selected pixel and the other pixels in the hypercube is calculated. Applying a threshold, selected by the specialist, a binary mask is obtained. Adjusting this threshold dynamically, a new RGB image is generated, containing only the pixels for which their spectral angle with respect to the reference pixel is lower than the threshold. Then, the specialist selects the region of interest and assigns a label to the pixels inside this region. The specialist will adjust the threshold until the displayed area will match with the expected type of tissue. Figure 3-6 shows a screenshot of the HELICoiD Labelling Tool. The spectral signatures of the pixels labelled as ground truth will be used as inputs in the supervised classification algorithm scheme in order to generate the model and determine its goodness using quantitative metrics.

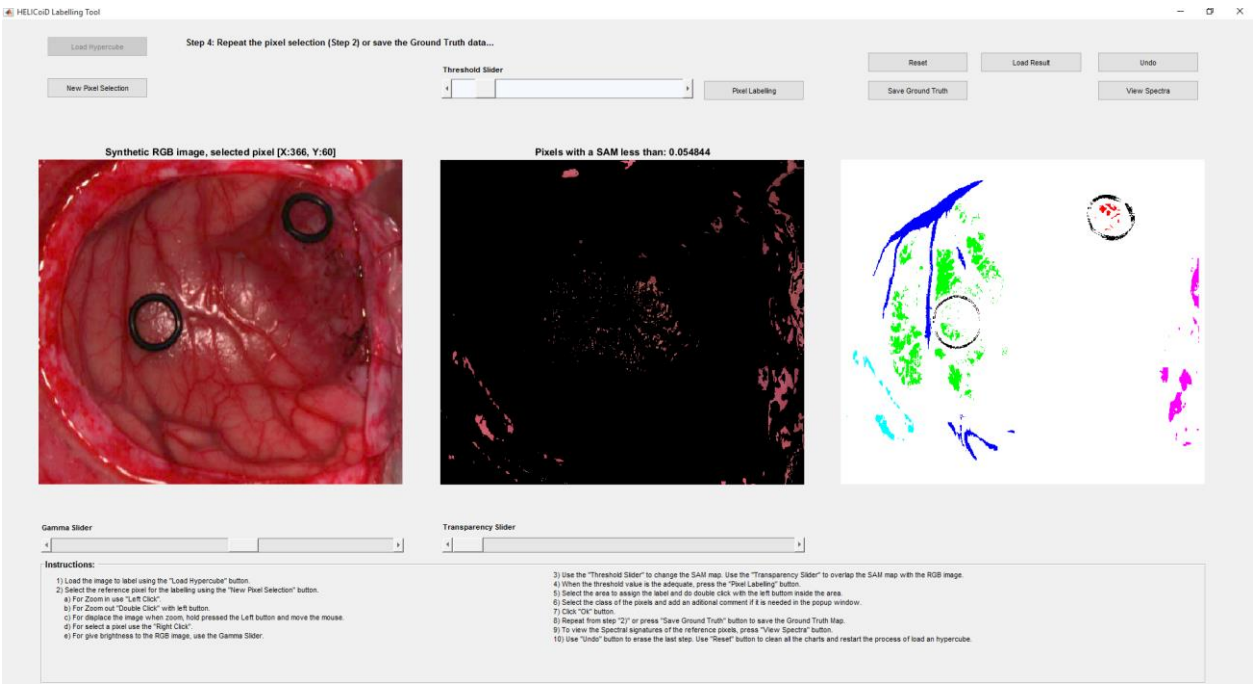


Figure 3-6: Screenshot of the HELICoiD Labelling Tool.

3.3.2 Hyperspectral Brain Image Dataset

This study uses only the VNIR images captured by the acquisition system. The dataset employed in this Master Thesis is composed by 6 different images from 4 different patients. These images have been selected as a reduced dataset since they are the images with the highest quality that allows the labelling process with a high confidence from the specialists. This dataset is focused only on the primary grade IV glioblastoma (GBM) tumours. Table 3-2 summarizes the operations and captures selected from the complete database and Table 3-3 illustrates the synthetic RGB images, generated from the captured hypercubes, of each capture.

3.3.3 Hyperspectral Brain Labelled Sample Dataset

The labelled dataset generated using the HELICoiD Labelling Tool employed in this master thesis is obtained from the previously described images of glioblastoma tumours. Table 3-4 illustrates the ground truth maps generated by the neurosurgeons using the HELICoiD Labelling Tool. Table 3-5 details the total number of pixels labelled. The dataset has been reduced to 4 different classes (**Normal Tissue**, **Tumour Tissue**, **Blood Vessels** and **Background**).

Operation ID	Operation Number	Capture Number
1	8	1, 2
2	12	1, 2
3	15	1
4	20	1

Table 3-2: Hyperspectral Brain Image dataset of GBM tumour selected

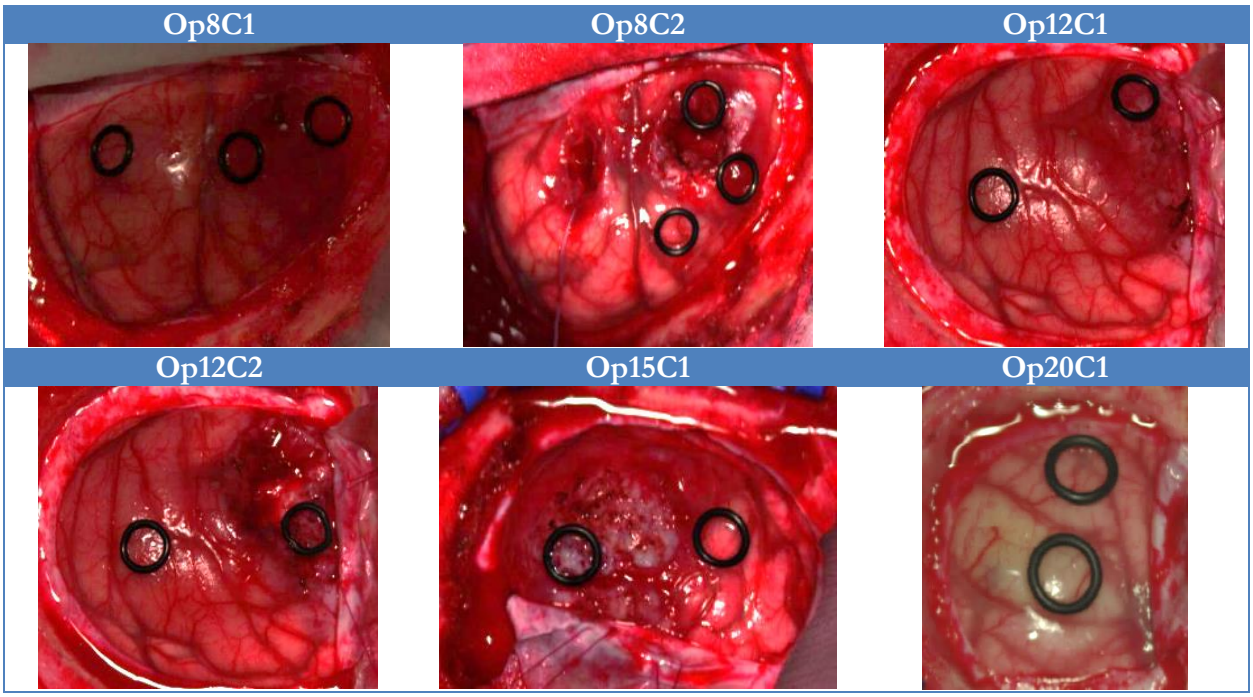


Table 3-3: Hyperspectral Brain Image Dataset

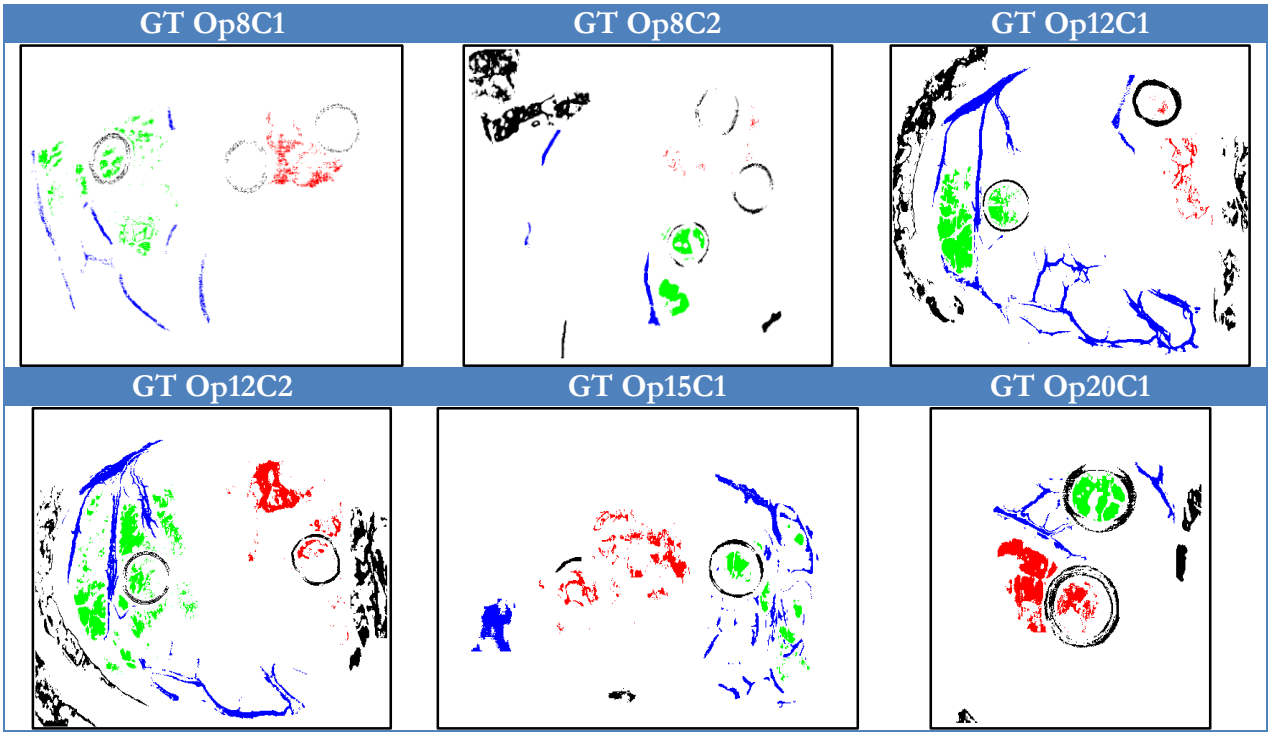


Table 3-4: HELICoiD ground truth maps of VNIR images

Class			#Labelled pixel dataset
Normal			18,644
Tumour	**Primary (G-IV)**	**GBM**	11,054
Blood Vessel			22,671
Background	**Meninges**	**Dura Mater**	10,888
		Skull Bone	9,941
	Generic Background		10,524
Total:			**87,722**

Table 3-5: HELICoiD labelled pixel dataset of VNIR images

3.4 SUMMARY

To obtain the hyperspectral images of the in-vivo human brain surface during the neurosurgical operations, the HELICoiD project has built a demonstrator capable of simultaneously obtaining two hyperspectral cubes. The two hyperspectral cameras selected are a VNIR camera that ranges between 400 nm to 1000 nm and a NIR camera that ranges between 900 nm to 1700 nm.

Data pre-processing system is composed by a computer which manages the entire system and the processing sub-system platform that has the goal of performing the hyperspectral classification in order to provide the results in real-time (see Figure 3-1).

An in-vivo human brain hyperspectral image database has been created using the HELICoiD demonstrator. In order to obtain the samples correctly labelled, the four steps flowchart presented in Figure 3-3 has been followed.

In the hyperspectral image capture process, if the tumour can be seen on the surface, two sterilised rubber ring markers are placed to identify the position of the tumour and that of normal brain as judged by the neurosurgeon where he/she can be quite confident that the brain tissue is normal. Later, the operator of the HELICoiD demonstrator captures the hyperspectral image of the exposed brain surface with and without markers. The markers offer an area of the image where the pixels can be labelled with the surgeon prior evaluation and then, this prior evaluation is contrasted with the pathology results.

Neurosurgeons take a sample of the tissue that is inside of the tumour marker. These samples are sent to pathology and the result are used as the ground truth reference for the algorithm development.

The expert evaluation is made by neuropathologists which are the only ones that can determine if a tissue inside of the marker is or not a tumour by using off-line histopathologic techniques.

In the last step, by using the information provided by the pathologists, some pixels of the image are labelled, using a labelling tool developed in MATLAB, in order to generate the ground truth for the training of the supervised algorithm. This labelling tool is used by the neurosurgeon that performs the correspondent neurosurgical operation.

CHAPTER 4: PARALLEL IMPLEMENTATION OF BRAIN CANCER DETECTION ALGORITHM

4.1 INTRODUCTION

This section is focused in the Random Forest implementation used as a supervised pixel-wise algorithm for brain cancer detection. In chapter 2.3.1.1 was explained how RF works and the different phases of RF training process.

In the next chapter 4.2, the RF CPU implementation that is used as starting point for the GPU implementation, presented in this work, is explained. This implementation has been tested using the previously explained datasets. These datasets will be used for the comparison between the CPU and the GPU implementation. Chapter 4.3 will present an analysis of the CPU implementation in order to identify the possible bottlenecks that can be parallelized using a GPU for the acceleration of the hyperspectral image processing. The last chapter, section 4.4, will explain how to implement the RF algorithm in the GPU.

4.2 CPU IMPLEMENTATION

The RF CPU implementation has been based on an existent implementation performed by Marvin N. Wright and Andreas Ziegles called Ranger (RANdom Forest GeneRator) [63]. This implementation has been selected due to it has been tested for the Git Hub community. Wright and Ziegles implemented Ranger for the necessity of having a RF algorithm optimized for high dimensional data and large number of features without a license for commercial use.

One of the main reasons why Ranger was selected is that the core of Ranger is implemented in C++ and uses only standard libraries, using the version C++ 11 of the standard for the programming language C++. Ranger is an algorithm where the authors have identified the bottlenecks and optimized the algorithm, having a demonstrated computational and memory efficiency.

Ranger is a good option to start our implementation in GPU for the reasons previously commented. Figure 4-1 shows the runtimes of five different RF algorithms with different number of trees, features, samples and *mtry* values (the percentage of features tried at each split) [63] where it can be appreciated that Ranger is faster than the other options.

Ranger has realized a great job with the CPU parallelization splitting the training phases between the different CPU threads. However, the data structure guarantees a good performance in the data access, avoiding cache failures. In our dataset, each row represents an instance and the different columns are the features. Each feature represents a band of the hyperspectral image. Ranger represents the rows like a feature and the columns are the instances because the usual operation in the algorithm is to read a feature and compare the different instances. For this reason, Ranger improves cache failures changing the memory data structure. Ranger can work with three different data types: double, float and char. This work is only focused in double data because the employed dataset is represented with double information.

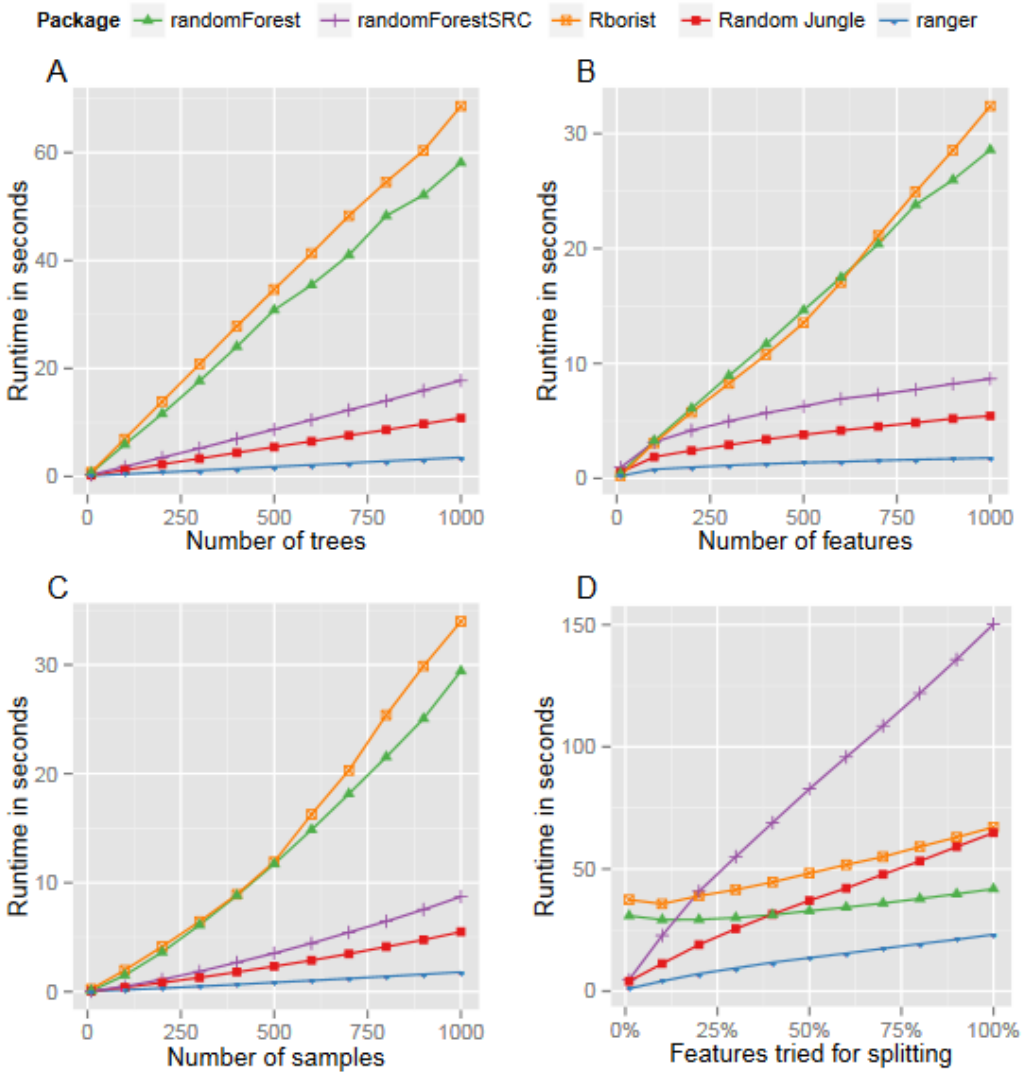


Figure 4-1: Figure extracted from [63] where different RF implementation are compared with Ranger. A) runtime analysis with variation of the number of trees, B) variation of the number of features, C) variation of the number of samples, D) variation of the percentage of features tried for splitting (mtry value).

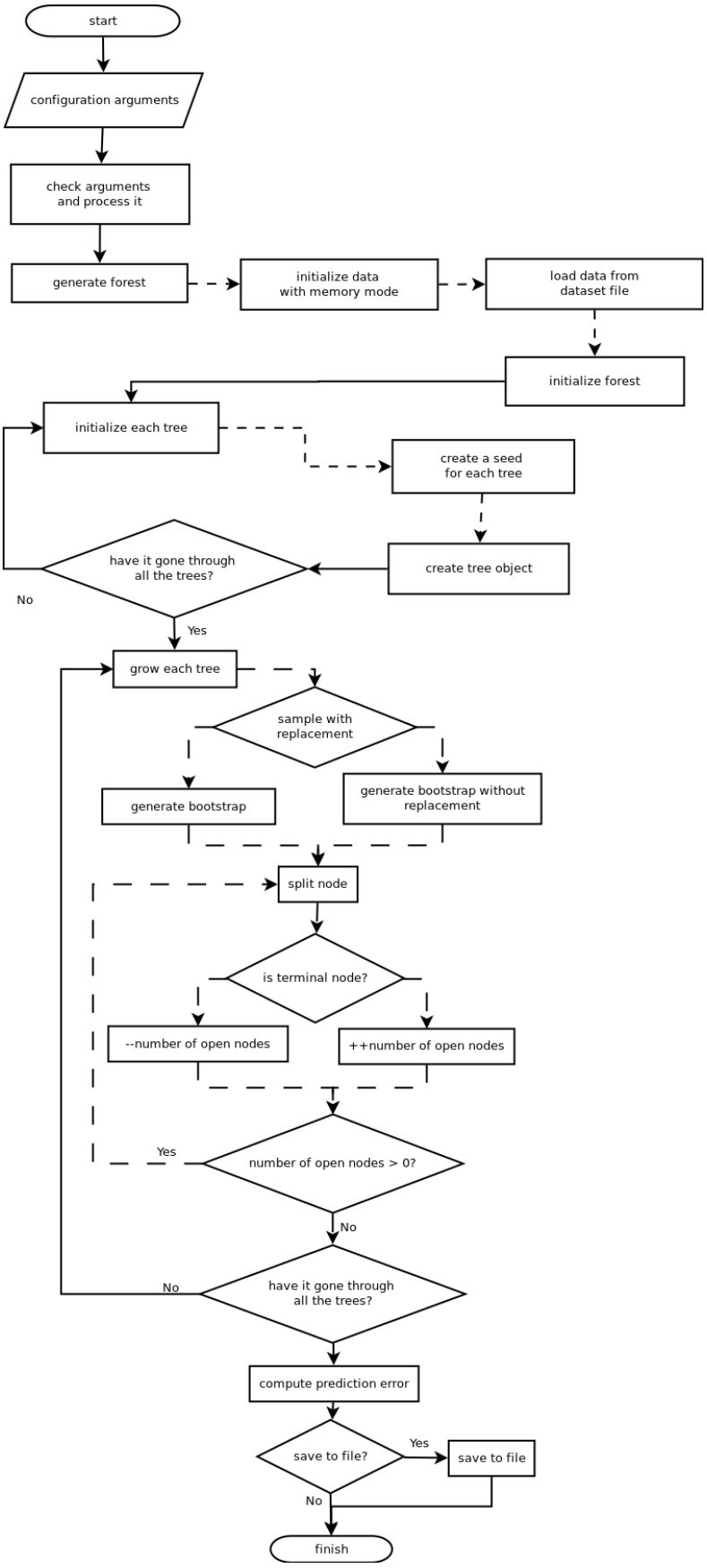


Figure 4-2: Ranger training phase flow diagram. Dashed lines represent the internal process of the previous process box.

CPU parallelization in the training phase is implemented in the growing tree phase (see Figure 4-2). The previous phase, like importing the dataset or to initialize the forest, only uses the main thread. During the tree growing, the main thread is used to show information to the user and the user does not have the impression that Ranger is inactive during a hard training. About the number of threads in CPU implementation, the user can indicate the number of threads using the argument *–nthreads* in the terminal, but it is not necessary to indicate the number of threads used to get a good performance in Ranger, at least in C++ implementation. Ranger automatically uses the number of max threads that the CPU can throw.

Regarding to the splitting node process, implemented split criteria is the decrease of node impurity for classification and regression RF, and the log-rank test for survival RF. In classification trees, node impurity is measured with the Gini index (see chapter 2.3.1.1.1) but in regression trees Ranger uses the estimated response variance [63].

4.2.1 Experimental Results

This chapter presents the evaluation of Ranger using the dataset generated within the HELICoiD project. For this evaluation, only the labelled dataset of the operation number 12 capture 2 has been employed in order to simplify the evaluation process. The K-Fold Cross-Validation method has been employed for the evaluation. In this method, the dataset is partitioned into K disjoint folds and each fold has the same class proportion.

The basis of Cross-Validation consists in using $K-1$ folds for training a classifier and the remaining for assessing its performance. This procedure is repeated K times varying the test set in each iteration until all folds have been used to evaluate the model performance. Figure 4-3 shows an example of this method using 10 folds, it means $= 10$, where the fold used to evaluate the model in each iteration is highlighted in red colour.

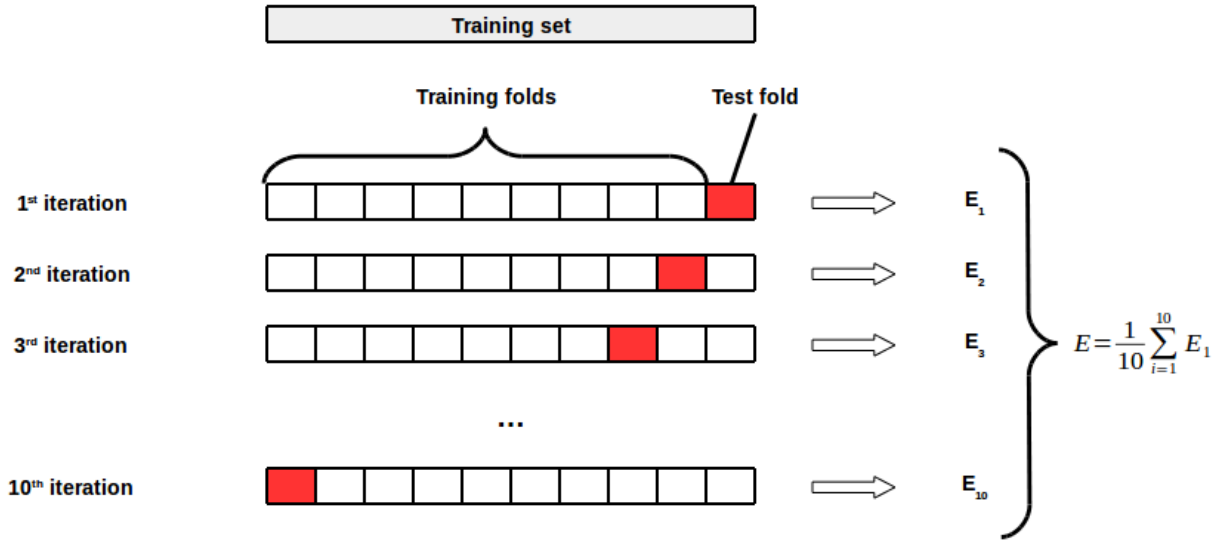


Figure 4-3: K-Fold Cross Validation example

The metrics chosen for estimating the classifier performance in this study are sensitivity, specificity and overall accuracy. These metrics summarize the information supplied by a confusion matrix (Table 4-1).

		Prediction			
		Normal Tissue	**Tumour Tissue**	**Blood Vessels**	**Background**
Ground Truth	**Normal Tissue**	650	0	0	0
	Tumour Tissue	0	333	0	0
	Blood Vessels	0	2	588	0
	Background	1	0	0	872

Table 4-1: Example of confusion matrix of the results obtained from one of the iterations during the cross-validation process.

In our test, we used ten folds and the previous matrix is just one example out of ten confusion matrices generated during the test, where various were perfect (the prediction matched all the samples).

Before providing the definitions of the sensitivity, specificity and overall accuracy, some terms must be defined:

- **True Positive (TP)**: Correctly detected conditions. It means that the result of the test is positive and the actual value of the classification is positive.
- **False Positive (FP)**: Incorrectly detected conditions. The result of the test is negative and the actual value of the classification is positive.
- **True Negative (TN)**: Correctly rejected conditions. The result of the test is negative and the actual value of the classification is negative.
- **False Negative (FN)**: Incorrectly rejected conditions. The result of the is positive and the actual value of the classification is negative.

The standard classification metrics employed in this study can be summarized as follows:

- **Sensitivity:** Is the proportion of actual positives that are correctly identified as positives by the classifier. It is expressed as follows:

$$Sensitivity = \frac{TP}{TP + FN} \tag{4-1}$$

- **Specificity**: Is the proportion of the actual negatives that the classifier successfully tests negative for it. It is computed as follows:

$$Specificity = \frac{TN}{TN + FP} \tag{4-2}$$

- **Overall Accuracy**: Refers to the ability of the model to correctly predict the class label of new or previously unseen data. The next equation shows formula of the overall accuracy metric.

$$Overall\ Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{4-3}$$

Finally, the previous metrics was used and computed the average to each class. Figure 4-4 and Table 4-2 show the average of each metric to the different classes performed to the whole dataset. These results outperform 99% of overall accuracy for every class. In terms of specificity and sensitivity, these results show a good discrimination rate between the different classes.

As conclusion, seeing these results, we have demonstrated that Ranger can behave correctly with our dataset and, therefore, we can continue with the acceleration process of the RF algorithm.

	Normal Tissue	Tumour Tissue	Blood Vessel	Background
Sensitivity Average (%)	99.95	99.94	99.80	99.95
Specificity Average (%)	99.98	99.95	99.98	99.98
Overall Accuracy Average (%)	99.97	99.95	99.94	99.97

Table 4-2: Cross-validation result of the Ranger CPU implementation

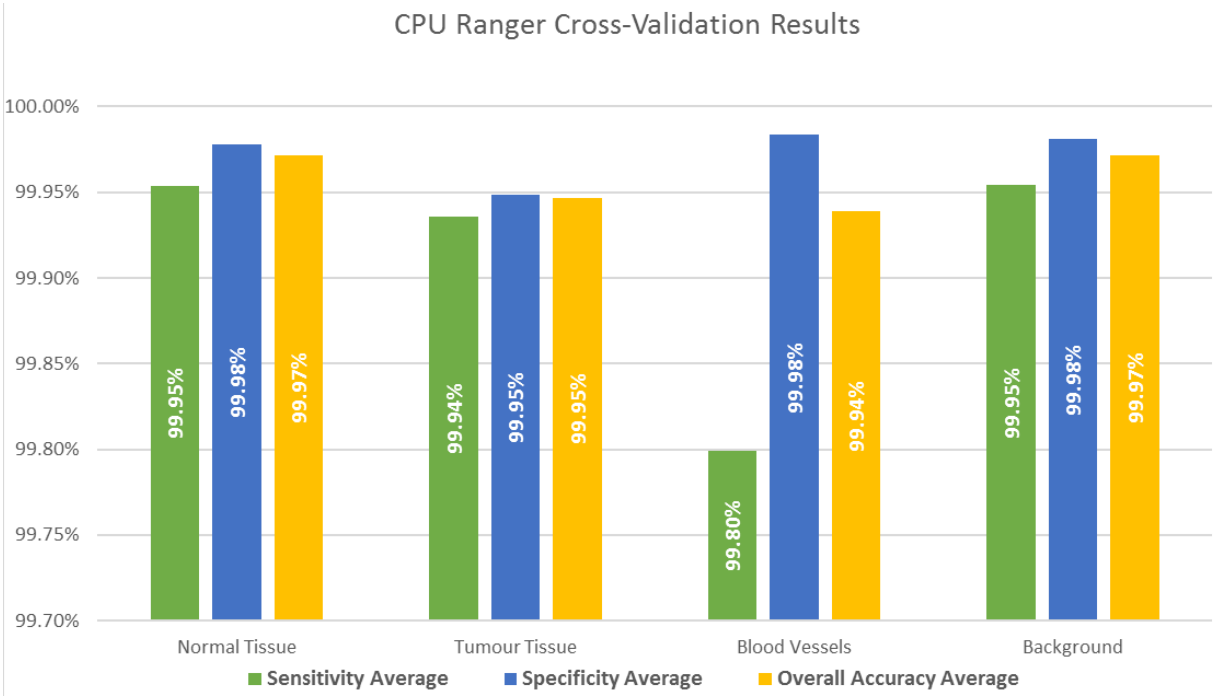


Figure 4-4: Cross-validation result of the Ranger CPU implementation with a forest of 500 trees

4.3 PARALLELIZATION ANALYSIS

Ranger has a good performance analysis in the different bottleneck of RF algorithm, for this reason the analysis of Ranger is a challenge but we are going to focus our efforts in the different parts where Ranger performs the most time-consuming tasks in the training phase.

The first process of Ranger is dedicated to generate the forest, reading the dataset, preparing it for the treatment, structuring the information in data objects and generating the trees. Although we know that data objects functionalities can be accelerated using GPUs, such as the data object sort function that could be achieved using a GPU sort implementation (for example parallel Radix implementation, or the use of Thrust CUDA library), in this work, we will not focus our effort in this task. However, we will concentrate our efforts in the hardest part of a RF implementation that it is within the growing trees process.

The authors of Ranger made a good decision when they decided to implement the CPU parallelization in each growing tree process and not in the internal process. Thinking about

complementing Ranger, GPU is a good decision in order to accelerate the hardest internal process and the rest of the process is accelerated using CPU parallelization.

The aim of this analysis will be to identify where we can reduce the processing time and to get a better performance of Ranger implementation. The identified bottlenecks will be accelerated using CUDA kernels and using the context management to combine CPU and GPU parallelization.

4.3.1 Bottlenecks Identification

We will focus the acceleration development in the training phase of RF, the hardest part of this algorithm. Checking the Figure 4-2, a good starting point could be to accelerate the tree initialization process where it is necessary to generate the learning set and the random subset of features that tree will use for training. These operations consist of a loop that generates a random output which represents the sample in the bootstrap, the learning set, or the feature candidate for training. Both are made by the main CPU thread, there are not a CPU parallelization.

The first bottleneck is identified in order to accelerate the forest initialization. Initialization part is not critical; it take up some decimals per tree but this accumulated time could take several seconds in a RF with larger number of tree and a huge dataset.

The principal bottleneck of Ranger is within the splitting process, to compute node impurity. Terminal nodes does not needs to splitting and is necessary to identify which feature is better, between all possible candidate, and what value to use (see Figure 2-6). The possible values of a feature are determined by the different values that it possesses in the dataset. Splitting process just compute the decrease of node impurity with all possible feature and their possible values. The bigger decrease is chosen such as the better option.

Ranger knows this problem and for this reason, if is not selected the memory saving option, it optimizes the data structure in order to reduce the process time, for example to sort all possible values and eliminate the last one because to split with the biggest value is impossible and node will not split. Despite this optimization in data strut, to find the best split still is the hardest part in training phase and it take several minutes with a huge dataset.

Such as we just said, the problem of locating the best split scale proportionally to the number of samples and the number of possible features. In RF, the split value for a node is extracted from all possible values in the dataset. This means that when the dataset grows, it is likely that the number of possible values also grows. About the number of features, these features are randomly selected but the user indicates the number of features tried at each split (*mtry*). By default, Ranger works with eleven possible features, but in a dataset with hundreds of features, this number is very lower. For those reasons, finding the best split feature and value is the most critical process and where we can get a high performance using GPU.

4.4 GPU IMPLEMENTATION

In this section, we will explain about GPU kernels implemented in order to get a better performance in the RF algorithm. Although Ranger has implanted all type of forest (regression, classification, survival, etc.), we focus our work on classification forest training phase, due to it is the type of forest required by the HELICoiD application.

In chapter 4.3, we identified multiple bottlenecks and in this section, we will explain how we get to accelerate that processes.

4.4.1 Bootstrap kernel

As it can be seen above, the generation of bootstrap samples is not a hard process but if the number of tree increment sharply as result we may need several seconds. For this reason, with GPU implementation we could reduce the process time a little bit. This part is not critical but it has a simple solution.

Every tree needs his learning set and, for this reason, RF uses bootstrapping, i.e. random sampling with replacement. The number of bootstrap samples is a percentage of the number of instances of the dataset, in Ranger that percentage is represented by the *sampleFraction* attribute. For instance, if *sampleFraction* value is 1.0, then the number of bootstrap samples is equal to the number of instances of the dataset, albeit the bootstrap may not contain all samples of the dataset.

In the bootstrap kernel, the idea is that each block generates the training dataset for each tree. As is, can be seen in Figure 4-5, the solution is a simple one dimensional grid where each block generates the learning set of a tree. In this case, each thread randomly generates a sample ID between all possible samples. Since the number of threads is limited by the capability of the GPU, each thread must generate one or more sample IDs.

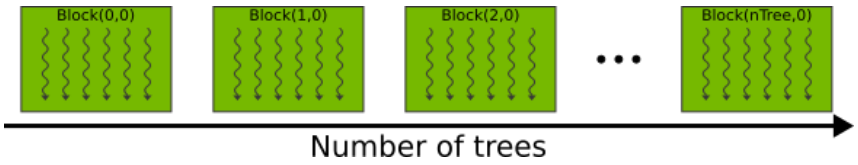


Figure 4-5: Bootstrap one-dimensional grid layout of the bootstrap kernel for n number of trees. The number of blocks generates in the X axis is equal to number of trees. Each block generates the learning set of a tree.

In the bootstrap kernel, the thread identification is represented by the *tid* variable and this identification represents the sample which will be generated. The thread must generate one or more sample IDs and, for this reason, the *tid* must increment as follows:

$$tid += blockDim.x \tag{4-4}$$

where *blockDim.x* represents the number of threads in X axis of the block.

In GPU bootstrap implementation, the kernel makes a histogram at the same time it is generating the learning set. This histogram is used in order to generate the Out-Of-Bag samples (see chapter 2.3.1.1.1) which will be used for estimation error.

Since we want the generation of the sample ids to be random, it is necessary to use a random generator. For this purpose, cuRAND library has a function which generate a uniform distribution between 0.0 and 1.0. In this case, we need to generate a specific range that contemplates the full range of samples. It is generated with the following sequence:

1) Use cuRAND to generate a uniform distribution between 0.0 and 1.0.
2) Multiply this by the desired range (largest value – smallest value + 0.999999).
3) Add the offset (+ smallest value).
4) Truncate to an integer.

The cuRAND library initialization needs a seed which will be used for random generator. In the bootstrap kernel, an input parameter is a vector containing a seed for each tree, better said, for each block.

Finally, CUDA bootstrap kernel gives us an array where each row represents the learning set of each tree and the histogram of the samples (Figure 4-6 shows a simple example). CPU thread will check the histogram in order to prepare the OOB set, verifying that samples have not been selected in the learning set and use them to estimate error.

This kernel implementation has some limitations. First, the number of trees cannot exceed the maximum x-dimension of a grid of thread blocks (determinate by the GPU capability, see [65]) because it generates a block per tree (Figure 4-5). The second limitation is about the global memory of the GPU: the max number of tree is limited by:

$$nTree = \frac{globalMemorySize}{(1 + sampleFraction) * nSamples * 8} \qquad (4\text{-}5)$$
$$\big((1 + sampleFraction) * nSamples * 8\big) \subset \mathbb{Z}$$

Where *nSamples* represents the number of samples in the dataset and *sampleFraction* is the number of bootstrap samples (percentage of the *nSamples*). In the product between *sampleFraction* and *nSamples* we are interested only in the integer part. Arrays are composed of elements *size_t* whose size are 8 bytes.

SamplesID =

9	0	2	2	5	7	3
0	1	0	9	8	4	9
8	1	5	2	6	7	0
4	4	1	0	1	4	5
7	2	0	3	6	8	9

InbagCount =

1	0	2	1	0	1	0	1	0	1
2	1	0	0	1	0	0	0	1	2
1	1	1	0	0	1	1	1	1	0
1	2	0	0	3	1	0	0	0	0
1	0	1	1	0	0	1	1	1	1

Figure 4-6: Bootstrap kernel result in order to generate a learning set of 5 samples from a dataset of 10 to 5 trees. SamplesID is the learning set per tree identifying the sample with a ID. InbagCount is a histogram of the samples used for training, necessary for to check with samples belong to the Out-Of-Bag set.

About the generation of feature candidate, the idea was to replicate the bootstrap solution where each block generates the possible feature candidate in one tree. However, during the implementation, we realize that this operation is not optimal for GPU parallelization for various reasons. In the first place, our dataset has been pre-processed using a pre-processing pipeline presented in 0 and it just use 129 band, it means 129 feature max, and we separated the tree initialization per block. It means that the max number of threads per block would be 129, wasting a lot of threads per block. Another reason, and more important, the number of possible features cannot be repeated. This implies a communication between different threads checking that they are not repeating a possible feature id. This communication between threads drastically affects to the GPU performance.

4.4.2 FindBestSplit kernel

In the previous chapter, we identify the most critical part in the training phase, the split phase (see Figure 4-2). In Ranger, classification tree implements the *findBestSplit()* method where it generates a new split node because it is not a terminal node, a pure node. In this method, Ranger looks for the best feature, between all possible features, and the best value for it, the threshold. In the bottleneck identification chapter (4.3.1), we explained why this is the critical zone in Ranger implementation.

The GPU implementation of this part is subdivided in three kernels, the first one generates an overall class count from the dataset (counting the number of instances that belongs to each class), the second kernel generates another overall class count of the possible right child nodes (counting the number of instances that belongs to each class in this right child node) and the last one computes the decrease of impurity of the node.

In split selection chapter (2.3.1.1.3), we introduced the Gini index and what is its purpose to evaluate the measure of impurity degree of the node. If the node is not a terminal node, it will split in a new level, and with the decrease of impurity we can determinate what is the best option for this not terminal node. In the GINI formula (see 2.3.1.1.3 chapter), $\sum P(t|c)^2$ represents the decrease of impurity and objective of this implementation is to identify what is the possible feature and the value of the threshold that maximize the summation.

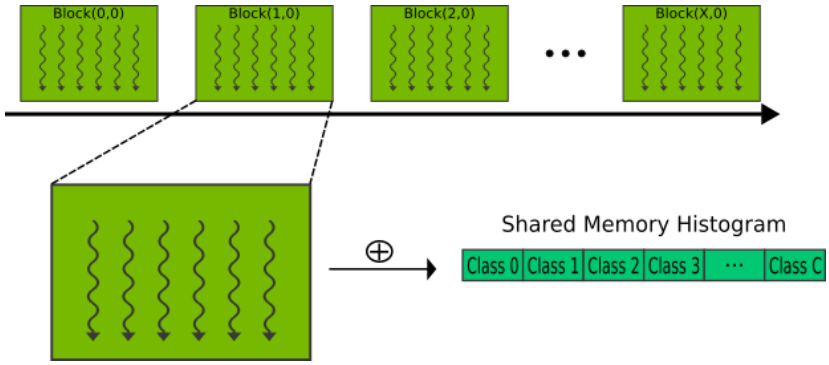


Figure 4-7: one-dimensional grid layout of the overall class count kernel. The number of blocks generates is X+1 and threads must run through all samples and generates a temporal histogram in the block's shared memory.

The first kernel, which generates an overall class count from the dataset, uses a one-dimensional grid layout and each threads access to a classification result of a sample and generates a temporal histogram per block in the shared memory, Figure 4-7 shows this operation. In this kernel, the *tid*, thread id, is defined as:

$$tid = threadIdx.x + blockIdx.x * blockDim.x \quad (4\text{-}6)$$

where *threadIdx.x* is the id of the thread in the block, *blockIdx.x* is the block id and *blockDim.x* is the number of threads in the block. If the number of threads between all blocks is less than the number of samples, is necessary that each thread gets sample result and for this reason we have to define the offset of the *tid*. This offset is defined as

$$offset = gridDim.x * blockDim.x \quad (4\text{-}7)$$

being *gridDim.x* the number of blocks.

Once kernels have processed all samples, and the histogram per block is finished, the next step is to generate the histogram in the global memory, adding all histograms of the blocks. In this case, only the first C + 1 threads, being C the number of classes, will operate in this process. In this way, the number of threads which will be access to the global memory is B*(C+1) and conflicting access will be B threads, where B is the number of blocks (see Figure 4-8). For this reason, it is not interesting to use an excessive number of blocks because the global memory access must be an atomic operation.

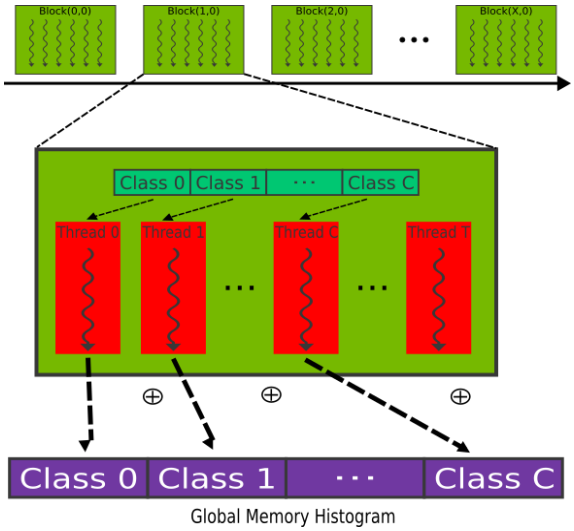


Figure 4-8: In the overall class count kernel, when it has run through all samples, it must to generate the total histogram in the global memory of the GPU. In this case, only the first C + 1 threads will operate, being C the number of classes.

At this time, the GPU has the overall class count of the dataset in the GPU memory, exactly in the global memory. How we explained above, the next process is to generate a class count of samples which will branch to the right child node. The way that this right child node is hypothesized depends on the feature selected for the bifurcation and the threshold value selected. This operation is more complex than the previous one because we need to compare all the possible combinations.

This second kernel will use a two-dimensional grid where the X axis of blocks will be used to process the possible split with a feature and the Y axis will indicate the possible value that it is processing such as threshold (see Figure 4-9). For instance, the *block(0,0)* will generate the overall class count of the right child node using the first possible feature of the possible split features and the first possible value and the *block(0,1)* will use the second possible value like threshold. On the threads of the blocks, they will go through all the samples to make the histogram in the shared memory of the block and, in turn, will count the samples that bifurcate towards the right child node. This sample count will also be performed in the shared memory, each thread having its own reserved memory for making its own count. The possible features and possible values will be generated in the host, using the data methods implemented in Ranger.

Once all threads have shifted through all the samples and they have finished the histogram and the count of samples in the right child node, the block must send the information to the global memory. For this process, only $C + 2$ threads are used, where $C + 1$ first threads are responsible for passing the histogram, in the shared memory, to the global memory and the other thread will count all samples in the right child node (it runs through all the accounts made by the different threads in the shared memory). This operation is identical to the bootstrap kernel.

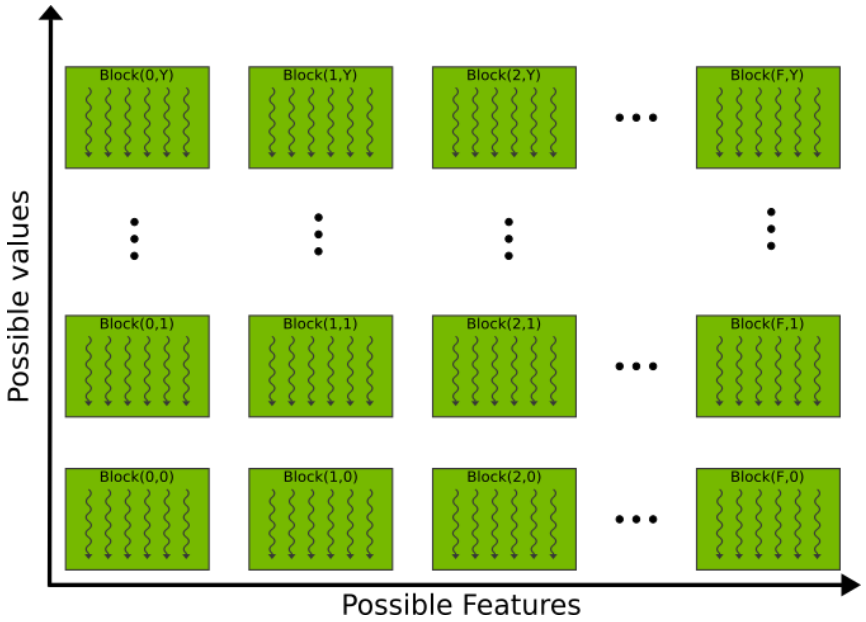


Figure 4-9: Two-dimensional grid layout of overall class count in the possible right node child. We use the X coordinate for the feature response and the Y coordinate assign a threshold. A row of blocks calculates all feature responses for a given threshold. F is the number of features and Y represents the number of blocks used in the Y axis.

Now, we are going to describe the memory distribution in this kernel, the second one. How it is foreseen, the number of possible values per feature has not to be the same for all feature and we have to allocate memory in a way that can be accessed systematically. As we have to extract the possible values for each feature from the host, because of distribution of the grid, we have opted for an option that slightly wastes part of memory. The possible values are sending to the GPU's memory using a two-dimensional array, where the rows represent the feature, and the columns represent the possible values of the feature (see Figure 4-10). We can appreciate that we are wasting memory but it is a sacrifice that we make to obtain a greater performance. This decision was made for one main reason: to avoid the dynamic allocation of the memory, as it takes a lot of performance out of the GPU.

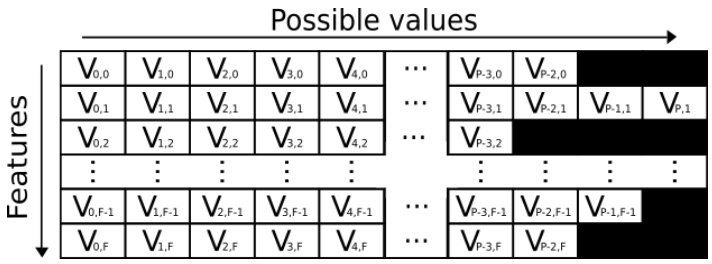


Figure 4-10: Array with possible values of threshold per feature. $P + 1$ is the max number of possible values, but all features do not have the same number of possible values. $F + 1$ is the number of features, been F the id of the last feature. Black cells represent values that do not belong to the possible values of the feature.

Due to the way we send data of possible values per feature, the result of this kernel, histogram of classes and the count of samples in the right child node, also waste some memory (see Figure 4-11). The arrays pitch is the max number of possible splits.

Finally, the last kernel is used for compute the decrease of impurity. In this kernel, we use a one-dimensional grid layout but using $F + 1$ (number of features) blocks, and each block in the X axis is used in order to generate the best decrease of impurity, the bigger one, of a feature (see Figure 4-12). The threads of a block calculate the decrease of impurity of each split threshold using the data generated in previous kernels. The decrease of impurity is calculated as follows:

$$\frac{\sum_{c \in y} class\ count\ right\ node[c]^2}{number\ of\ samples\ in\ right\ node} + \frac{\sum_{c \in y} class\ count\ left\ node[c]^2}{number\ of\ samples\ in\ left\ node} \quad (4\text{-}8)$$

where c belongs to set of classes $y = \{c_1, c_2, \dots, c_j\}$ and "class count right node" is the histogram generated in the previous kernel. The histogram "class count left node" is generated using the "class count right node" histogram and the current histogram, in the current node which is going to be split, generated in the first kernel.

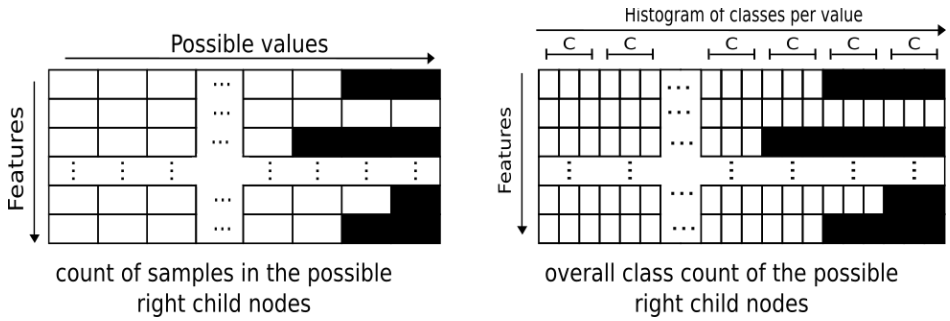


Figure 4-11: Arrays of results of the overall class count in right node child kernel. The first array represents the number of samples in the right child node and the second array is a histogram of classes in the right child node. C is the number of classes.

Each thread of a block in this kernel has his own memory space in the shared memory of block where it is going to keep the best decrease that it has been calculated. Finally, the first thread, thread with $tid = 0$, will access to the memory of all threads and it will save the best decrease of impurity in the global memory. The result of this last kernel is a one-dimensional array with the best decrease of each feature and another array with the corresponding threshold for the best decrease. This is the task of the host that locates the best feature, using the decrease of impurity generated in the kernel.

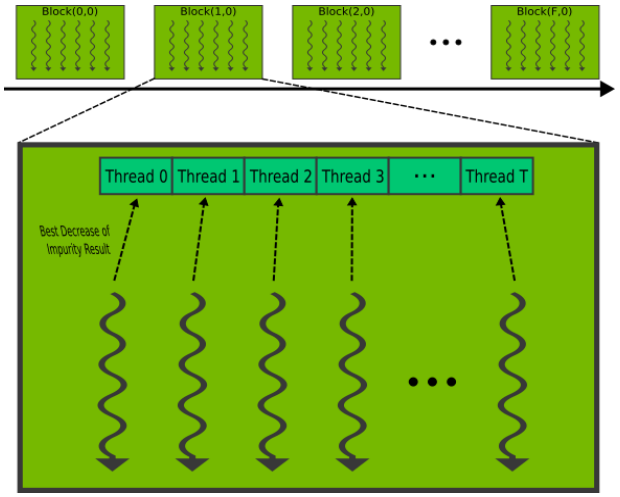


Figure 4-12: Grid layout of Compute decrease of impurity kernel. The number of blocks is $F + 1$ (number of possible split features). The threads of a block calculate the decrease of impurity of each split threshold and keep the best decrease of impurity calculated in the shared memory of the block (each thread has his own space).

4.5 SUMMARY

The RF CPU implementation has been based on an existent implementation called Ranger. Ranger's authors implemented it for the necessity of having a RF algorithm optimized for high dimensional data and a large number of features without a license for commercial use.

Ranger has realized a great job with the CPU parallelization, splitting the training phases between the different CPU threads. The trees grow is divided in the multiple CPU threads and we have accelerated the internal process using the CUDA GPU, complementing the CPU task within the GPU.

Regarding the bottleneck identification, we have identified that accelerating the tree initialization process, where it is necessary to generate the learning set and the random subset of features, would be a good start point. However, the main bottleneck is in the split node task where RF have to split a not terminal node in two child nodes. In this bottleneck, the non-terminal node needs to be split and is necessary to identify which feature is better, between all possible candidate, and which value have to be used.

Based on the identification of bottlenecks, we have proposed some GPU implementations that have optimized these tasks. The bootstrap GPU implementation is used in order to accelerate the learning set of the different trees which belongs to the forest. This learning set must be selected randomly using samples from the training dataset (repetition of samples in the learning set is allowed) and the cuRAND library to generate the learning set has been used. Finally, the split node task was implemented using three kernels: the first one generates an overall class count from the dataset (counting the number of instances that belongs to each class), the second kernel generates another overall class count of the possible right child nodes (counting the number of instances that belongs to each class in this right child node) and the last one computes the decrease of impurity of the node.

CHAPTER 5: EXPERIMENTAL RESULTS

5.1 INTRODUCTION

This section presents the results obtained using the accelerated GPU implementation previously explained in chapter 4.4. First, we test the acceleration with the developed *bootstrap kernel* versus *bootstrap sequential* of Ranger and later, we will test the *findBestSplit kernel* against its sequential implementation in Ranger. Finally, we have created a RF model using the GPU kernels and another one using Ranger without GPU implementations in order to compare results and processing times to generate the model.

In order to perform the experiments, the previously described HELICoiD database has been used. This database is divided into three different cases studies (CSs). These CSs differ in which patients are included as subject of study. The CS1 has the main goal of checking if the discrimination between healthy and tumour tissue can be performed using the available labelled data, and avoiding the inter-patient variability of data. It means that the datasets explored in this CS include hypercubes from surgical operations where both type of tissue, healthy and tumour, are present. In order to avoid the inter-patient variability of data, each surgical procedure is used independently for training and testing the classifier. In CS2 all the available labelled data are merged in a unified dataset. It means that a unique database is created by joining all single patient data, so the inter-patient variability is taken into account. Finally, CS3 is the most realistic one. In this CS, each surgical procedure datum is used as test set of a classification algorithm, and that classifier model is built using the information from the rest of hyperspectral labelled data (belonging to different patients). This case study represents the real case of a new operation, where the classification has to be performed with a classifier that has been trained with data from previous operations. Table 5-1 shows the total number of labelled pixels per each case study.

Regarding to the equipment used for these tests, we used a laptop and an equipment rack that belongs to IUMA. The laptop is composed by an Intel Processor i7-6700HQ and a NVIDIA GPU GTX 960M with 2GB GDDR5. The equipment rack is a server with an Intel Xeon Processor E3-1225 v3 and two NVIDIA GPU Tesla K40 with 12 Gb GDDR5. The laptop uses Ubuntu 16.04 with CUDA Toolkit 8 and IUMA's server uses Xubuntu 14.04 with CUDA Toolkit 7.5. Table 5-2 shows the specifications of each platform.

Case Study	Operation ID	#Operation	#Capture	# Pixels
CS1	1	8	1	5,477
			2	10,769
	2	12	1	15,753
			2	24,464
	3	15	1	8,082
	4	20	1	9,635
CS2	Complete Dataset			74,180
CS3	1	8	1, 2	57,934
	2	12	1, 2	33,963
	3	15	1	66,098
	4	20	1	64,545

Table 5-1: Total number of labelled pixels per each case study

Feature	GTX 960M	Tesla K40
CUDA capability	5.0	3.5
Global Memory (MB)	2,002	11,441
Multiprocessors (MP)	5	15
CUDA Cores/MP	128	192
GPU Max clock (MHz)	1,176	745
Memory clock rate (MHz)	2,505	3,004
Memory Bus Width (bit)	128	384
L2 Cache Size (Bytes)	2,097,152	1,572,864
Total amount of constant memory (Bytes)	65,536	65,536
Total amount of shared memory per block (Bytes)	49,152	49,152
Total number of registers available per block	65,536	65,536
Maximum number of threads per multiprocessor	2,048	2,048
Maximum number of threads per block	1,024	1,024
Feature	**I7-6700HQ**	**Xeon E31225**
Cores	Quad core	Quad core
Clock speed (GHz)	2.6	3.2
Turbo clock speed (GHz)	3.5	3.6
Threads	8	4
L2 cache (MB)	1	1
L3 cache (MB)	6	8
Manufacture process (nm)	14	22

Table 5-2: Specifications of the testing platforms

5.2 BOOTSTRAP ACCELERATION

In this chapter, we will compare the results obtained by the bootstrap implementation, both GPU and Ranger. For this test, we do not need the HELICoiD database because the bootstrap function only generates a random learning set where the result is an array with sample IDs. The number of samples selected are in a range between 7500 and 120000 samples (7500, 15000, 30000, 50000, 75000 and 120000). The number of trees selected covers the range between 250 to 1000 (250, 500, 750 and 1000).

As the GPU of the laptop is more limited in global memory than the GPU of the server, the first test is limited to a maximum of 120,000 samples and 1,000 trees. Table 5-3 shows the computational time results of the bootstrap test executed in the laptop, where an average speedup of 1.62 is obtained. Table 5-4 represents the test results executed in the IUMA's server, where an average speedup of 4.31 is obtained.

Figure 5-1 shows the comparison between the results generated using the GTX 960M and the Tesla K40 where it can be seen that the performance of the GTX 960M descends drastically if the number of trees is higher than 750. However, Tesla K40 does not share this behavior. This fact has an explanation: it depends of the number of Stream Multiprocessor (SMX) that the GPUs has. Such as we see in Table 2-2, the Tesla K40 has 16 SMX units and this is the main reason why it gets a better performance than the GTX 960M which has only 8 SMX units. As it has been seen previously, the SMX schedules threads in groups of 32 parallel threads called warps and each SMX executes one block at the same time.

In the bootstrap kernel, we generate as many blocks as trees have the RF, meaning that with 750 trees we generate 750 blocks in X axis. Such as we can see, the number of SMX is limited in both GPUs and for this reason the GPU must share resources for all blocks. When sharing resources, it is necessary a control for this task, which can have serious repercussion to the system. As we can see in Figure 5-1, GTX 960M has to invest too much computational time in the synchronization of the different blocks with the respective resources when the number of blocks is higher than 750. On the other hand, we can appreciate that the bootstrap kernel in GTX 960M is useful when we use a forest in the range of 250-500 trees and its optimal performance is achieved when huge datasets are used.

Tesla K40 does not suffer the same problems as the GTX960M. In addition to the higher number of SMX, Tesla K40 seems to perform better with the synchronization of a larger number of blocks.

nTree	nSamples	Sequential Time (C code) (s) CPUTime (i7-6700HQ 2.6 GHz)	Parallel Time (CUDA) (s) GPUTime (GTX 960M)	SpeedUp
250	7,500	0.093184	0.066360	1.40
250	15,000	0.164743	0.078874	2.09
250	30,000	0.321110	0.116051	2.77
250	50,000	0.522073	0.167799	3.11
250	75,000	0.786342	0.212565	**3.70**
250	120,000	1.259975	0.340994	**3.70**
500	7,500	0.157502	0.170702	0.92
500	15,000	0.312735	0.203628	1.54
500	30,000	0.633309	0.278684	2.27
500	50,000	1.085661	0.379697	2.86
500	75,000	1.586806	0.484595	3.27
500	120,000	2.525299	0.716861	3.52
750	7,500	0.235407	2.148555	0.11
750	15,000	0.468209	2.197597	0.21
750	30,000	0.928312	2.311009	0.40
750	50,000	1.551004	2.448726	0.63
750	75,000	2.332393	2.638155	0.88
750	120,000	3.782006	2.983854	1.27
1,000	7,500	0.319996	4.258782	0.08
1,000	15,000	0.624503	4.331570	0.14
1,000	30,000	1.238414	4.476360	0.28
1,000	50,000	2.057585	4.665802	0.44
1,000	75,000	3.108375	4.924184	0.63
1,000	120,000	5.040401	5.388656	0.94
			Average Speed-up:	**1.62**

Table 5-3: Testing *bootstrap kernel* in GTX 960M

nTree	nSamples	Sequential Time (C code) (s) CPUTime (Intel Xeon E31225 V3 3.2 GHz)	Parallel Time (CUDA) (s) GPUTime (Tesla K40)	SpeedUp
250	7,500	0.091130	0.023506	3.88
250	15,000	0.175958	0.043638	4.03
250	30,000	0.367556	0.083197	4.42
250	50,000	0.586500	0.143618	4.08
250	75,000	0.880229	0.211027	4.17
250	120,000	1.431666	0.347052	4.13
500	7,500	0.178603	0.044786	3.99
500	15,000	0.354499	0.079807	**4.44**
500	30,000	0.707062	0.168293	4.20
500	50,000	1.193473	0.286314	4.17
500	75,000	1.757869	0.432982	4.06
500	120,000	2.822436	0.690042	4.09
750	7,500	0.267183	0.065586	4.07
750	15,000	0.526066	0.126529	4.16
750	30,000	1.057218	0.257204	4.11
750	50,000	1.767521	0.432125	4.09
750	75,000	2.655239	0.642739	4.13
750	120,000	4.271583	1.027972	4.16
1,000	7,500	0.354081	0.083752	4.23
1,000	15,000	0.706979	0.169215	4.18
1,000	30,000	1.404332	0.343160	4.09
1,000	50,000	2.338026	0.574334	4.07
1,000	75,000	3.510987	0.864581	4.06
1,000	120,000	5.647008	1.369967	4.12
			Average Speed-up:	**4.31**

Table 5-4: Testing *bootstrap kernel* in Tesla K40

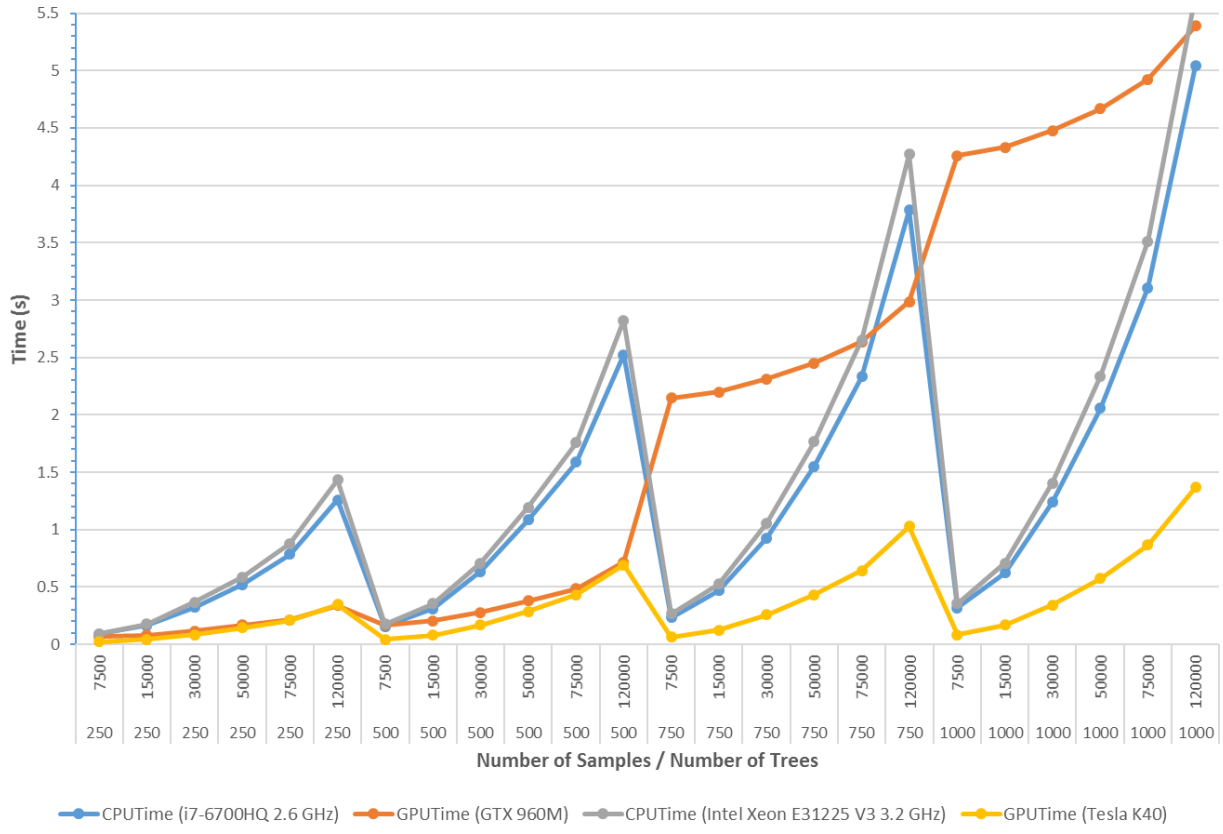


Figure 5-1: Testing *bootstrap kernel* comparison (GTX 960M vs. Tesla K40).

The first test was limited by the memory of GTX960M. For this reason, we generated a second test only focused in the Tesla K40 where we used a bigger dataset (between 50,000 and 500,000 samples) than the previous test to perform the simulations. The results of this test are presented in Table 5-5 and Figure 5-2. As we can see in Figure 5-2, the second test just confirm us that this kernel is optimal when using a huge dataset, in which we can achieve an average speed-up factor of 2.88.

nTree	nSamples	Sequential Time (C code) (s) CPUTime (Intel Xeon E31225 V3 3.2 GHz)	Parallel Time (CUDA) (s) GPUTime (Tesla K40)	SpeedUp
250	50,000	0.601531	0.193292	3.11
250	100,000	1.215109	0.353639	3.44
250	200,000	2.42959	0.695672	3.49
250	300,000	3.534034	1.062629	3.33
250	400,000	4.73703	1.386039	3.42
250	500,000	5.916897	1.659994	3.56
500	50,000	1.155681	0.416077	2.78
500	100,000	2.341504	0.734769	3.19
500	200,000	4.702039	1.412664	3.33
500	300,000	7.073178	2.140107	3.31
500	400,000	9.459429	2.781975	3.40
500	500,000	11.855447	3.335857	**3.55**
750	50,000	1.734538	1.448463	1.20
750	100,000	3.508167	1.925221	1.82
750	200,000	7.056159	2.921789	2.42
750	300,000	10.611689	4.014368	2.64
750	400,000	14.174089	4.975038	2.85
750	500,000	17.769304	5.794928	3.07
1,000	50,000	2.311743	2.585877	0.89
1,000	100,000	4.680634	3.213049	1.46
1,000	200,000	9.404986	4.527513	2.08
1,000	300,000	14.164496	5.972424	2.37
1,000	400,000	18.905916	7.243635	2.61
1,000	500,000	23.649113	8.333215	2.84
			Average Speed-up:	**2.88**

Table 5-5: Testing *bootstrap kernel* in Tesla K40 with huge datasets.

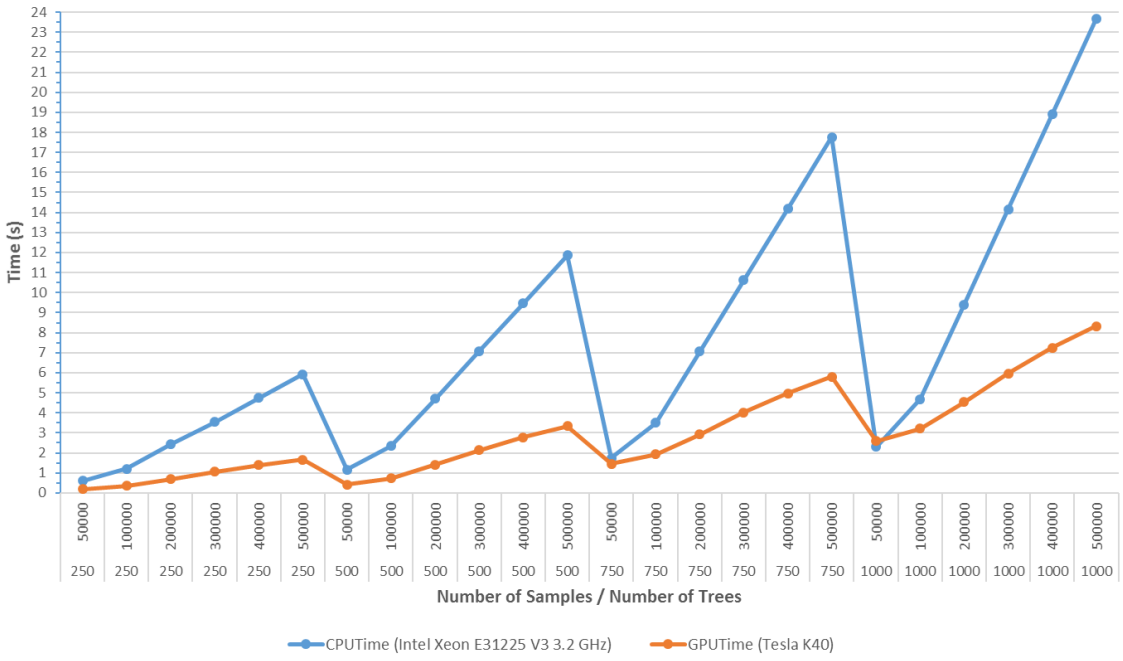


Figure 5-2: Testing *bootstrap kernel* in Tesla K40 with huge datasets.

5.3 FINDBESTSPLIT ACCELERATION

This section will present the results obtained in the acceleration of the *findBestSplit kernel*. In this test, we need information to be able to perform the calculation to find the best possible split. For this reason, we use the datasets of HELICoiD project in order to perform the experiments with a real dataset composed by hyperspectral samples. As we explained before, these datasets are divided in three different cases studies (CS) and our tests have been divided per CS. Likewise, these tests are divided into two tables, one for each platform where we are testing the kernel (laptop and IUMA's server).

5.3.1 Case Study 1

Table 5-6 and Table 5-7 present the time results and speedup factor obtained by the *findBestSplit kernel* using the sequential code in C and the parallel code employing the two platforms, laptop and IUMA's server respectively, using the CS1 datasets. Figure 5-3 shows the graphical comparison between the results obtained using the laptop versus the IUMA's server.

nTree	nSamples	Sequential Time (C code) (s) CPUTime (i7-6700HQ 2.6 GHz)	Parallel Time (CUDA) (s) GPUTime (GTX 960M)	SpeedUp
3	5,477	0.166216	0.028331	5.87
6	5,477	0.352264	0.054414	6.47
12	5,477	0.717855	0.107595	6.67
18	5,477	1.046782	0.158752	6.59
24	5,477	1.381691	0.214333	6.45
3	8,082	0.364301	0.045905	7.94
6	8,082	0.736120	0.092084	7.99
12	8,082	1.431020	0.184814	7.74
18	8,082	2.143636	0.278408	7.70
24	8,082	2.937000	0.371413	7.91
3	9,635	0.479328	0.054612	8.78
6	9,635	1.001172	0.111768	8.96
12	9,635	1.931148	0.223908	8.62
18	9,635	3.012326	0.344104	8.75
24	9,635	3.922742	0.458407	8.56
3	10,769	0.622280	0.074141	8.39
6	10,769	1.363768	0.153332	8.89
12	10,769	2.605362	0.289277	9.01
18	10,769	4.032816	0.424822	9.49
24	10,769	5.406466	0.576733	9.37
3	15,753	1.213798	0.122570	9.90
6	15,753	2.469423	0.246133	10.03
12	15,753	5.113555	0.478722	10.68
18	15,753	7.772578	0.765859	10.15
24	15,753	10.42862	1.028869	10.14
3	24,464	2.911491	0.261690	11.13
6	24,464	5.223429	0.498381	10.48
12	24,464	12.40471	1.043797	**11.88**
18	24,464	16.35122	1.606170	10.18
24	24,464	21.29328	2.172736	9.80
			Average Speed-up:	**8.82**

Table 5-6: Testing *findBestSplit kernel* with CS1 datasets in laptop (i7 6700HQ with GTX 960M)

nTree	nSamples	Sequential Time (C code) (s) CPUTime (Intel Xeon E31225 V3 3.2 GHz)	Parallel Time (CUDA) (s) GPUTime (Tesla K40)	SpeedUp
3	5,477	0.208499	0.079809	2.61
6	5,477	0.407510	0.159099	2.56
12	5,477	0.808518	0.251906	3.21
18	5,477	1.217964	0.395985	3.08
24	5,477	1.624718	0.499840	3.25
3	8,082	0.423988	0.145437	2.92
6	8,082	0.856488	0.299945	2.86
12	8,082	1.699788	0.475895	3.57
18	8,082	2.518180	0.747115	3.37
24	8,082	3.413396	0.942992	3.62
3	9,635	0.575933	0.200741	2.87
6	9,635	1.151098	0.393828	2.92
12	9,635	2.322173	0.656389	3.54
18	9,635	3.434160	0.983800	3.49
24	9,635	4.632244	1.272101	3.64
3	10,769	0.711774	0.271501	2.62
6	10,769	1.411162	0.574780	2.46
12	10,769	2.852185	0.927910	3.07
18	10,769	4.231031	1.396985	3.03
24	10,769	5.700045	1.797260	3.17
3	15,753	1.409402	0.467919	3.01
6	15,753	2.815423	0.993704	2.83
12	15,753	5.609884	1.587015	3.53
18	15,753	8.368161	2.424578	3.45
24	15,753	11.39263	3.131281	3.64
3	24,464	3.022051	0.972542	3.11
6	24,464	6.104473	1.963285	3.11
12	24,464	12.46993	3.377254	3.69
18	24,464	18.16799	4.750176	3.82
24	24,464	24.13288	6.112912	**3.95**
			Average Speed-up:	**3.20**

Table 5-7: Testing *findBestSplit kernel* with CS1 datasets in IUMA's server (Xeon E31225 with Tesla K40)

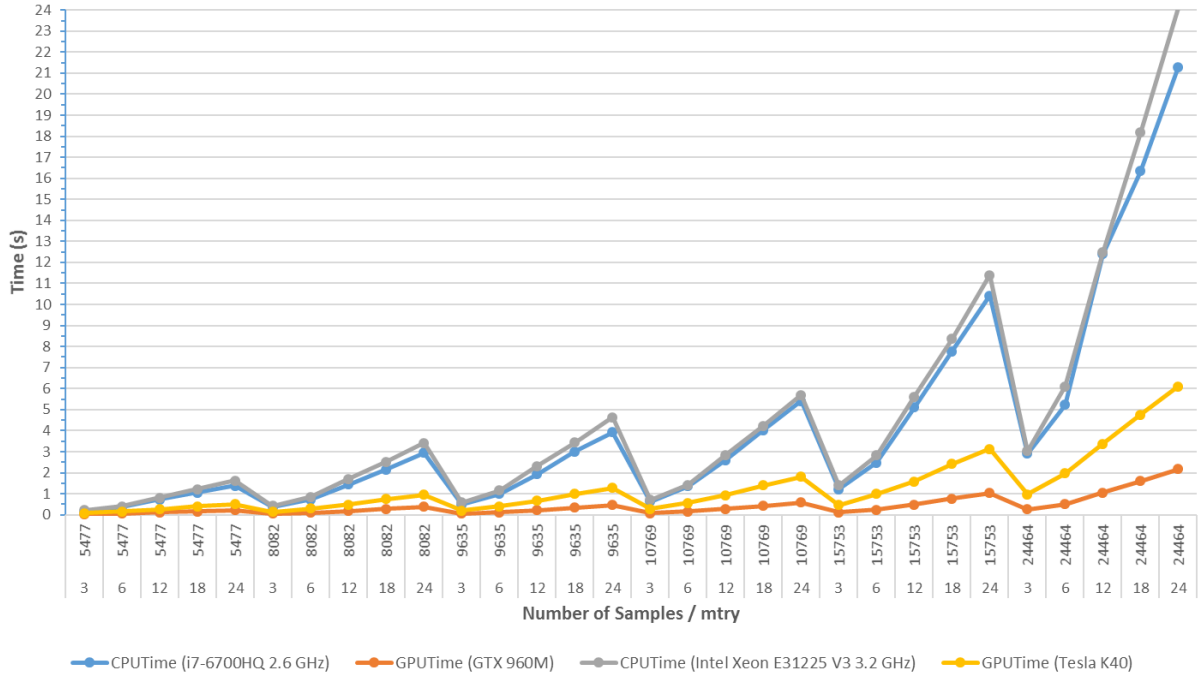


Figure 5-3: Testing *findBestSplit kernel* with CS1 datasets comparison (laptop vs. IUMA's server)

5.3.2 Case Study 2

Table 5-8 and Table 5-9 present the time results and speedup factor obtained by the *findBestSplit kernel* using the sequential code in C and the parallel code employing the two platforms, laptop and IUMA's server respectively, using the CS2 dataset. Figure 5-4 shows the graphical comparison between the results obtained using the laptop versus the IUMA's server.

nTree	nSamples	Sequential Time (C code) (s) CPUTime (i7-6700HQ 2.6 GHz)	Parallel Time (CUDA) (s) GPUTime (GTX 960M)	SpeedUp
3	74,180	19.535107	1.509968	**12.94**
6	74,180	40.331463	3.350480	12.04
12	74,180	76.312996	7.014941	10.88
18	74,180	117.095947	11.012700	10.63
24	74,180	156.574646	14.978607	10.45
			Average Speed-up:	**11.39**

Table 5-8: Testing *findBestSplit kernel* with CS2 datasets in laptop (i7 6700HQ with GTX 960M)

nTree	nSamples	Sequential Time (C code) (s) CPUTime (Intel Xeon E31225 V3 3.2 GHz)	Parallel Time (CUDA) (s) GPUTime (Tesla K40)	SpeedUp
3	74,180	23.693186	5.142533	4.61
6	74,180	46.965099	10.825226	4.34
12	74,180	91.503319	19.904064	4.60
18	74,180	136.435379	29.349756	4.65
24	74,180	180.134369	38.606590	**4.67**
			Average Speed-up:	**4.57**

Table 5-9: Testing *findBestSplit kernel* with CS2 datasets in IUMA's server (Xeon E31225 with Tesla K40)

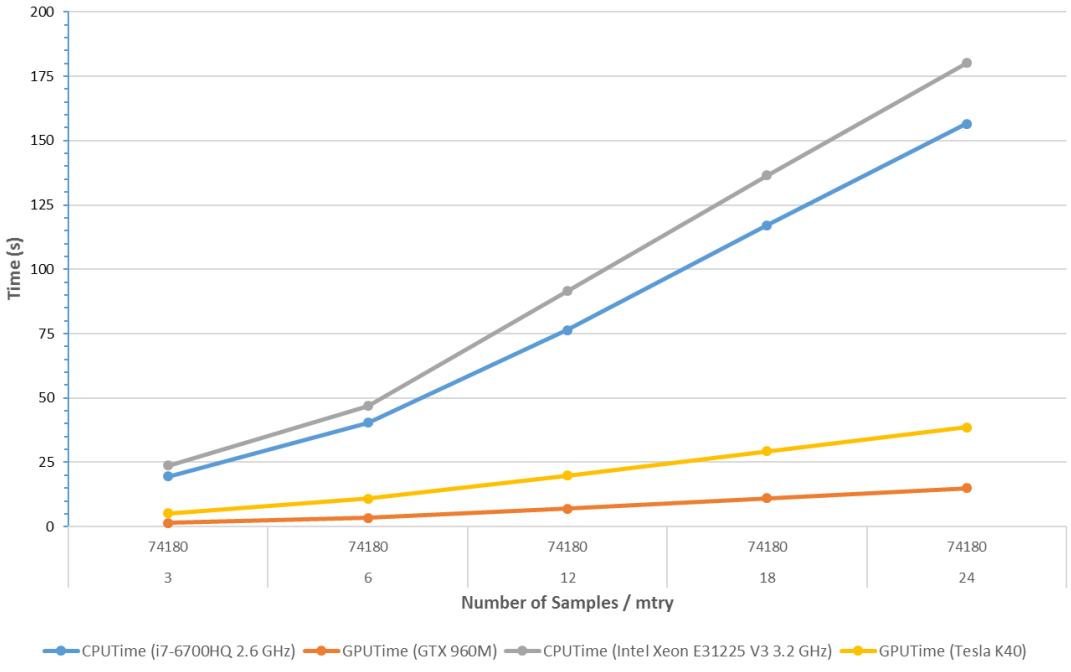


Figure 5-4: Testing *findBestSplit kernel* with CS2 dataset comparison (laptop vs. IUMA's server)

5.3.3 Case Study 3

Table 5-10 and Table 5-11 present the time results and speedup factor obtained by the *findBestSplit kernel* using the sequential code in C and the parallel code employing the two platforms, laptop and IUMA's server respectively, using the CS3 datasets. Figure 5-5 shows the graphical comparison between the results obtained using the laptop versus the IUMA's server.

nTree	nSamples	Sequential Time (C code) (s) CPUTime (i7-6700HQ 2.6 GHz)	Parallel Time (CUDA) (s) GPUTime (GTX 960M)	SpeedUp
3	33,963	5.184504	0.424273	**12.22**
6	33,963	10.809633	0.899532	12.02
12	33,963	21.073154	1.872147	11.26
18	33,963	32.202522	2.974596	10.83
24	33,963	42.188377	3.876581	10.88
3	33,963	5.211975	0.424135	11.44
6	33,963	10.736996	0.905671	11.28
12	33,963	21.392807	1.904726	11.14
18	33,963	31.994953	2.950963	11.11
24	33,963	43.393173	3.962756	11.17
3	66,098	16.634384	1.298497	11.23
6	66,098	32.384205	2.727051	11.19
12	66,098	68.413177	5.600777	11.17
18	66,098	108.289284	8.734106	11.17
24	66,098	135.899902	11.023503	11.19
3	64,545	14.938005	1.197955	11.19
6	64,545	30.173466	2.597006	11.18
12	64,545	58.758015	5.470393	11.18
18	64,545	98.624908	8.289091	11.18
24	64,545	118.805336	11.417547	11.18
3	57,934	11.672660	0.920275	11.18
6	57,934	25.465443	2.048625	11.18
12	57,934	53.526726	4.704114	11.18
18	57,934	73.213020	7.131975	11.18
24	57,934	98.803253	9.593641	11.18
3	57,934	12.070780	0.958238	11.18
6	57,934	23.657591	2.028787	11.18
12	57,934	47.059811	4.446221	11.18
18	57,934	73.401405	7.074694	11.18
24	57,934	97.153702	9.462690	11.18
			Average Speed-up:	**11.23**

Table 5-10: Testing *FindBestSplit kernel* with CS3 datasets in laptop (i7 6700HQ with GTX 960M)

nTree	nSamples	Sequential Time (C code) (s) CPUTime (Intel Xeon E31225 V3 3.2 GHz)	Parallel Time (CUDA) (s) GPUTime (Tesla K40)	SpeedUp
3	33,963	9.282999	1.687833	5.50
6	33,963	19.673382	2.896021	6.79
12	33,963	38.241749	15.756333	2.43
18	33,963	56.075256	7.850453	**7.14**
24	33,963	74.523392	10.498322	7.10
3	33,963	9.373720	1.761651	5.79
6	33,963	19.461369	2.843697	5.85
12	33,963	38.148643	18.761877	5.66
18	33,963	59.003113	8.459873	6.31
24	33,963	73.466690	20.490589	6.14
3	66,098	24.004589	3.887309	5.95
6	66,098	56.794464	15.451085	5.98
12	66,098	108.844879	28.516108	6.01
18	66,098	164.178741	22.498653	6.08
24	66,098	219.618484	29.230730	6.03
3	64,545	20.835800	12.303882	6.01
6	64,545	51.068466	7.987484	6.02
12	64,545	104.726059	16.761984	6.03
18	64,545	167.044647	31.439022	6.04
24	64,545	222.590393	33.750046	6.03
3	57,934	22.543921	4.031380	6.03
6	57,934	38.713554	5.726884	6.03
12	57,934	88.190231	11.805882	6.03
18	57,934	131.492386	17.413771	6.03
24	57,934	174.076813	25.439869	6.03
3	57,934	26.395317	10.724771	6.03
6	57,934	47.414055	6.773387	6.03
12	57,934	82.160934	24.553497	6.03
18	57,934	135.53183	36.214668	6.03
24	57,934	162.501053	40.027287	6.03
			Average Speed-up:	**5.97**

Table 5-11: Testing *FindBestSplit kernel* with CS3 datasets in IUMA's server (Xeon E31225 with Tesla K40)

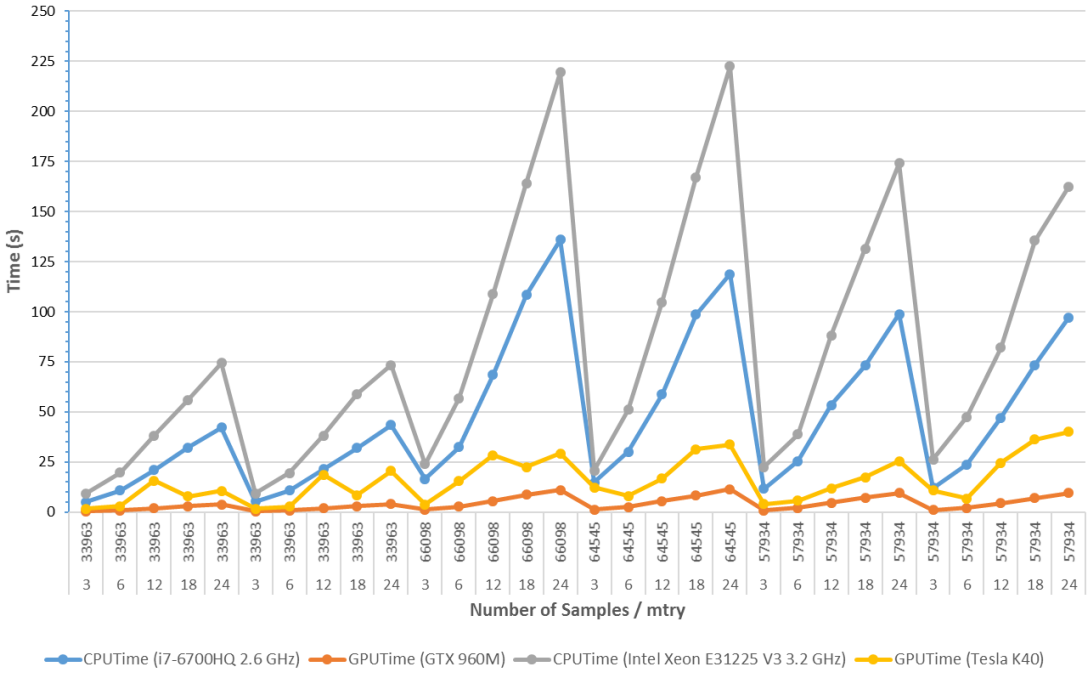


Figure 5-5: Testing *findBestSplit kernel* with CS3 datasets comparison (laptop vs. IUMA's server)

5.3.4 Summary of the *findBestSplit kernel* results

These three different tests have shown that the kernel works perfectly, accelerating the internal process of selecting the best candidate and the best threshold for the split node. All tests have been verified. The sequential code and the parallel code generate the same results.

Unlike the *bootstrap kernel*, the *findBestSplit kernel* has better performance in GTX 960M than Tesla K40 (see Figure 5-6). This fact is produced because this kernel has a grid with less variability than the *bootstrap kernel*, the number of block in Y axis is fixed and the range of blocks in X axis is much smaller than the range in the bootstrap kernel. As a conclusion, we can understand that we are misusing the resources of Tesla K40. GTX 960M, with its higher clock frequency but fewer resources (registers, global memory, CUDA cores, etc.), are being benefited in this comparison using this kernel.

The number of blocks in X axis can be higher than the ones selected for the experiments (3, 6, 12, 18 and 24), but this range of possible features (*mtry*) should be selected depending on the size of the dataset features (129 in our case). The possible features must be randomly selected. Using a too high *mtry* value could have negative repercussions on the randomness of the Random Forest algorithm.

Finally, we have demonstrated that the *findBestSplit kernel* works correctly and it has a better performance with a huge number of samples in the non-terminal node, reaching to achieve in the best case a speedup factor of 11. As it can be seen in Figure 5-6, where a summary of the results obtained in the whole dataset for all the case studies is presented. When the dataset increases the number of samples, the differences between the time consuming also increase.

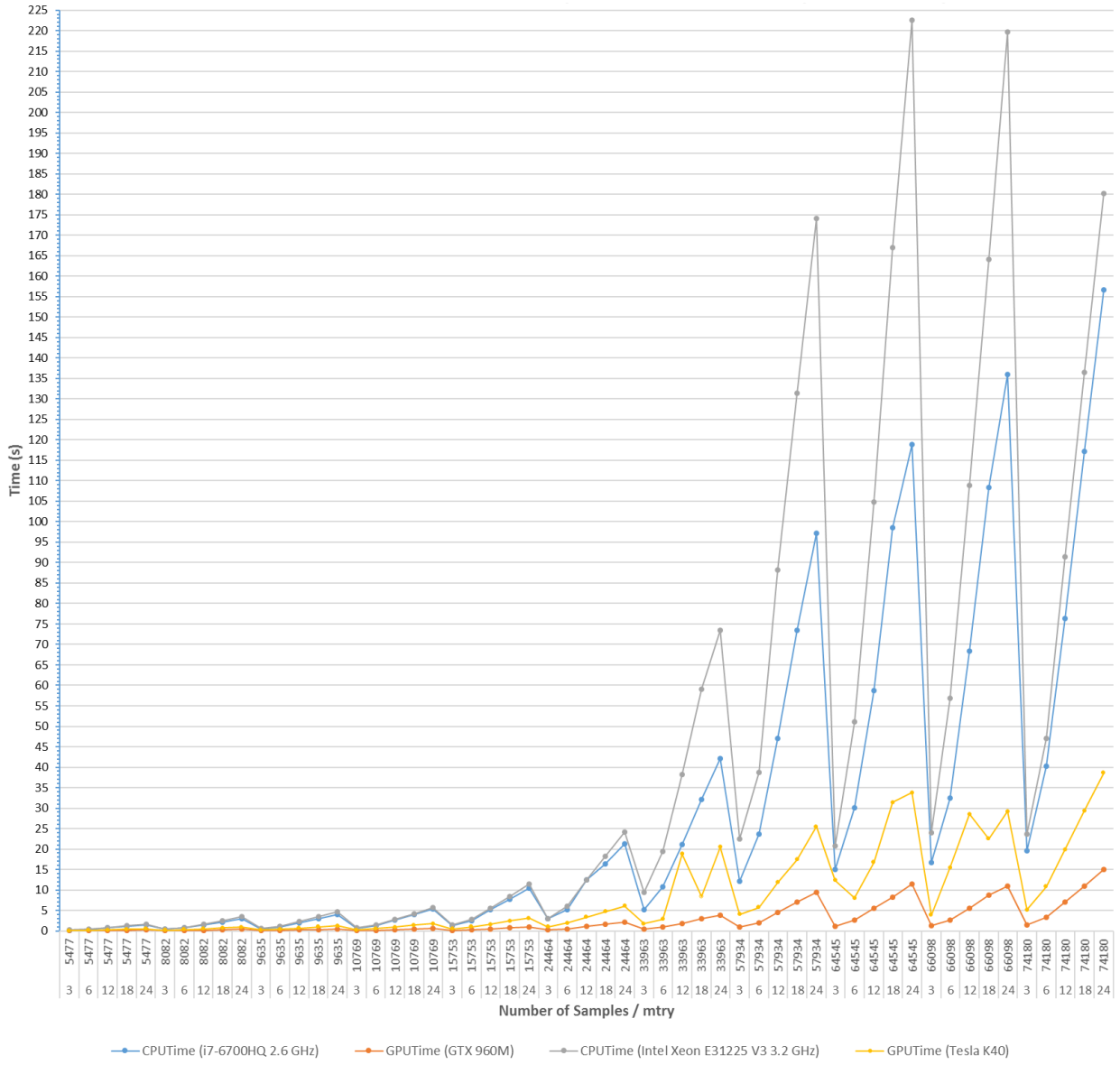


Figure 5-6: Summary of the results obtained in each CS with the *findBestSplit kernel* (laptop vs. IUMA's server).

5.4 SUMMARY

In this chapter, the *bootstrap kernel* and *findBestSplit kernel* implementations have been analysed achieving good results in both platform, laptop and IUMA's server.

The *bootstrap kernel* has showed excelent result versus the bootstrap implementation using Ranger. We proposed to generate the learning set of different trees during the initialization of the forest in order to construct multiple trees by setting the learning set. It can be only used if Ranger option generates a bootstrap learning set without replacement.

Figure 5-1 shows a higher difference between GTX 960M and Tesla K40 from 750 trees onwards. The main reason of this result is the number of SMXs of both GPUs. As the number of SMX is limited, and each SMX can only execute one block at the same time, it is necessary a control to share the SMX between all the blocks. This task can have serious negative repercussion to the system and, in this case, GTX 960M has to invest too much computational time in the

synchronization of the different blocks with the respective resources when the number of blocks is higher than 750.

Regarding *findBestSplit kernel*, GTX960M exhibits better performance than Tesla K40 (see Figure 5-6). As a conclusion, we can understand that we are mishandling the resources of Tesla K40. GTX 960M, with a higher clock frequency but fewer resources, produces better results. However, it could be improved by using multiple CUDA streams, because the number of blocks in X axis cannot be much higher than the test selection as using a too high *mtry* value could have negative repercussions on the randomness of the Random Forest algorithm.

CHAPTER 6: CONCLUSIONS AND ON-GOING WORK

6.1 CONCLUSIONS

In this master thesis, the main goal was the implementation of a classification algorithm in a GPU to accelerate the training and the classification process of hyperspectral images. For this purpose, we had to study hyperspectral images and the advantages of this technology compared to the current techniques employed for cancer detection. We focused our efforts on the supervised pixel-wise algorithms of machine learning, especially the Random Forest (RF) algorithm.

RF has demonstrated to be a good candidate due to the accurate results that this method generates in different scopes, especially in the computer vision community where RF is very popular. This method allows quick predictions but requires a long time-consuming for the training phase with huge datasets, due to the high computational cost of this task. In conclusion, the training phase must be the priority in the acceleration of the algorithm. Chapter 2.3.1.1.1 contains the information about the specific phases of the training parts of the RF algorithm, for instance the Split selection phase.

The acceleration platform selected for the implementation has been GPUs, which has become extremely popular in the high-performance computing area thanks to its massively parallel hardware architecture. Due to the availability in the IUMA infrastructures of a NVIDIA Tesla K40, a GPU development with the purpose of covering high performance parallel computing demand has been selected. For this reason, we have developed our kernels using the CUDA platform. The documentation about CUDA platform was essential for this research work. Chapter 2.5 is focused on the CUDA platform and chapter 2.6 introduces the characteristics of the Tesla K40 architecture.

The RF CPU implementation (C++ code) was obtained from GitHub. Multiple RF implementations from GitHub were tested and finally, the Ranger implementation was selected. Ranger is a RF algorithm optimized for high dimensional data and large number of features. The Ranger implementation was studied and verified in section 4.2. During the Ranger implementation study, different bottlenecks were identified (see section 4.3.1). As a result, three bottlenecks were recognized: two bottlenecks during forest initialization and one bottleneck in the tree growing phase.

In the GPU implementation, the bottleneck acceleration kernels were implemented. One bottleneck was discarded because the GPU is not an optimal option in this case. The use of the GPU implies a communication between different threads, checking that they are not repeating values, and thus having a serious negative repercussion in the algorithm performance. In chapter 4.4.1 and 4.4.2, the different kernels implementations were explained.

In conclusion, the *bootstrap kernel* has a great result versus the bootstrap in Ranger. We propose to generate the learning set of different trees during the initialization of the forest in order to construct multiple trees by setting the learning set. Testing the *bootstrap kernel*, we have obtained good results, achieving an average speedup factor of 1.62 and 4.31 using the GTX 960M GPU and Tesla K40 GPU respectively. The differences between GTX 960M and Tesla K40 depends of the number of Stream Multiprocessor (SMX) available in the GPUs. When sharing resources, a control for this task is necessary as it can have serious negative repercussion in the system. In this case, the GTX 960M has to invest too much computational time in the synchronization of the different blocks with the respective resources when the number of blocks is higher than 750. Tesla K40 does not suffer the same problem. In addition to the higher number of SMXs, Tesla K40 seems to perform better with the synchronization of a higher number of blocks.

Regarding to the *findBestSplit kernel* implementation, we have a better performance in GTX 960M, achieving an average speedup factor with the huge dataset of 11.39 and 4.57 using the GTX 960M GPU and Tesla K40 GPU respectively. For these tests, we used the HELICoiD database because the necessity of using actual information of our main application in order to generate the best split selection. In chapter 3.3, the database generation is explained. As a final conclusion, we can understand that we are misusing the resources of Tesla K40 and GTX960M, that has a higher clock frequency but fewer resources, generating better results. In any case, both GPU platforms (Tesla K40 and GTX960M) offer better performances than the sequential implementation in both CPUs platforms (Intel Xeon E31225 and i7-6700HQ).

6.2 ON-GOING WORK

In this Master Thesis, GPUs for general purpose and the CUDA platform were studied in order to generate a CUDA solution for a Random Forest algorithm. In this case, the Ranger implementation was analysed and different bottlenecks were identified. As conclusion, we were able to generate some solutions for the different bottlenecks identified in Ranger. However, currently we are working in introducing the developed CUDA kernels in the Ranger algorithm.

For handling the different CUDA kernels, a handler object was developed: the *CUDAUtility* object. This handler object uses the Singleton pattern, which restricts the instantiation of a class to one instance. Every class that uses CUDA functionalities must get an instance of the *CUDAUtility* object but the instance is always the same one and it cannot be copied.

At this moment, the *CUDAUtility* has been introduced in the Ranger implementation but it has demonstrated poor performance. The *bootstrap kernel* is properly working introduced in the algorithm, but the *findBestSplit kernel* does not exhibit the same behaviour as the multiple calls to this kernel is deteriorating the performance of the Ranger algorithm. Currently, CUDA kernels are working in a default stream and we are working on a solution using multiple streams that can solve the current problem. Using the multiple stream solution, we can take advantage of the efficiently multi-threading of the Ranger implementation.

CHAPTER 7: REFERENCES

[1] Li Q., He X., Wang Y., Liu H., Xu D., & Guo F. Review of spectral imaging technology in biomedical engineering: achievements and challenges. Journal of biomedical optics, 18(10), 100901-100901, 2013.

[2] Manolakis D., & Shaw G. Detection algorithms for hyperspectral imaging applications. Signal Processing Magazine, IEEE, 19(1), 29-43, 2002.

[3] Manolakis D., Marden D., & Shaw G. A. Hyperspectral image processing for automatic target detection applications. Lincoln Laboratory Journal, 14(1), 79-116, 2003.

[4] Wu D., & Sun D. W. Advanced applications of hyperspectral imaging technology for food quality and safety analysis and assessment: a review—part II: applications. Innovative Food Science & Emerging Technologies, 19, 15-28, 2013.

[5] Teke M., Deveci H. S., Haliloglu O., Gurbuz S. Z., & Sakarya U. A short survey of hyperspectral remote sensing applications in agriculture. In Recent Advances in Space Technologies (RAST), 171-176, 2013.

[6] Lu G., & Fei B. Medical hyperspectral imaging: a review. Journal of biomedical optics, 19(1), 010901-010901, 2014.

[7] A. Sahu et al., "Characterization of Mammary Tumors Using Noninvasive Tactile and Hyperspectral Sensors," in IEEE Sensors Journal, vol. 14, no. 10, 3337-3344, Oct. 2014.

[8] C. Dai, Q. Li, J. Liu, Y. Wang and C. Dai, "Evaluation of Erythropoietin Efficacy on Diabetic Rats Based on Microscopic Hyperspectral Imaging Technique," Bioinformatics and Biomedical Engineering (iCBBE), 2010 4th International Conference on, Chengdu, 1-4, 2010.

[9] T. Vo-Dinh, D. Stokes, M. Wabuyre, M. Martin, J. Song, R. Jagannathan, E. Michaud, R. Lee, and X. Pan. A hyperspectral imaging system for in vivo optical diagnostics. IEEE Engineering in Medicine and Biology Magazine, 23 (5): 40–49, 2004.

[10] B. Regeling, W. Laffers, A. O. H. Gerstner, S. Westermann, N. A. Müller, K. Schmidt, J. Bendix, and B. Thies. Development of an image pre-processor for operational hyperspectral laryngeal cancer detection. Journal of Biophotonics, 2015.

[11] M. Hohmann, A. Douplik, J. Varadhachari, A. Nasution, J. Mudter, M. Neurath, and M. Schmidt. Preliminary results for hyperspectral videoendoscopy diagnostics on the phantoms of normal and abnormal tissues: Towards gastrointestinal diagnostics. vol. 8087, 2011.

[12] S. Rathore, M. Hussain, A. Ali, and A. Khan. A recent survey on colon cancer detection techniques. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 10 (3): 545–563, 2013.

[13] Zhu, S., Su, K., Liu, Y., Yin, H., Li, Z., Huang, F., ... & Chen, Y. Identification of cancerous gastric cells based on common features extracted from hyperspectral microscopic images. *Biomedical optics express*, *6*(4), 1135-1145. 2015.

[14] M. Maggioni, G. Davis, F. Warner, F. Geshwind, A. Coppi, R. DeVerse, and R. Coifman. Hyperspectral microscopic analysis of normal, benign and carcinoma microarray tissue sections. Proc. SPIE 6091, Optical Biopsy VI. 2006.

[15] A. Chaddad, C. Tanougast, A. Dandache, A. Al Houseini, and A. Bouridane. Improving of colon cancer cells detection based on haralick's features on segmented histopathological images. Computer Applications and Industrial Electronics (ICCAIE), IEEE International Conference on. 87–90, 2011.

[16] H. Akbari, L. Halig, H. Zhang, D. Wang, Z. Chen, and B. Fei. Detection of cancer metastasis using a novel macroscopic hyperspectral method. Proc. SPIE 8317, Medical Imaging 2012: Biomedical Applications in Molecular, Structural, and Functional Imaging. 2012.

[17] J. Benavides, S. Chang, S. Park, R. Richards-Kortum, N. Mackinnon, C. MacAulay, A. Milbourne, A. Malpica, and M. Follen. Multispectral digital colposcopy for in vivo detection of cervical cancer. Optics Express, 11 (10): 1223–1236, 2003.

[18] D. Ferris, R. Lawhead, E. Dickman, N. Holtzapple, J. Miller, S. Grogan, S. Bambot, A. Agrawal, and M. Faupel. Multimodal hyperspectral imaging for the noninvasive diagnosis of cervical neoplasia. Journal of Lower Genital Tract Disease, 5 (2): 65–72, 2001.

[19] K. Zuzak, R. Francis, E. Wehner, J. Smith, M. Litorja, D. Allen, C. Tracy, J. Cadeddu, and E. Livingston. Hyperspectral imaging utilizing lctf and dlp technology for surgical and clinical applications. Proc. SPIE 7170, Design and Quality for Biomedical Technologies II. Vol. 7170, 2009.

[20] B. Sorg, B. Moeller, O. Donovan, Y. Cao, and M. Dewhirst. Hyperspectral imaging of hemoglobin saturation in tumor microvasculature and tumor hypoxia development. Journal of Biomedical Optics, 10 (4), 2005.

[21] L. Boucheron, Z. Bi, N. Harvey, B. Manjunath, and D. Rimm. Utility of multispectral imaging for nuclear classification of routine clinical histopathology imagery. BMC Cell Biology, 8 (SUPPL. 1), 2007.

[22] D. Hattery, M. Hassan, S. Demos, and A. Gandjbakhche. Hyperspectral imaging of kaposi's sarcoma for disease assessment and treatment monitoring. Proceedings 31st Applied Imagery Pattern Recognition Workshop from Color to Hyperspectral: Advancements in Spectral Imagery Exploitation, 2002.

[23] D. Dicker, J. Lerner, P. Van Belle, S. Barth, D. Guerry IV, M. Herlyn, D. Elder, and W. El-Deiry. Differentiation of normal skin and melanoma using high resolution hyperspectral imaging. Cancer Biology and Therapy, 5 (8): 1033–1038, 2006.

[24] D. Roblyer, R. Richards-Kortum, K. Sokolov, A. El-Naggar, M. Williams, C. Kurachi, and A. Gillenwater. Multispectral optical imaging device for in vivo detection of oral neoplasia. Journal of Biomedical Optics, 13 (2), 2008.

[25] D. Roblyer, C. Kurachi, A. Gillenwater, and R. Richards-Kortum. In vivo fluorescence hyperspectral imaging of oral neoplasia. Proc. SPIE 7169, Advanced Biomedical and Clinical Diagnostic Systems VII, 2009.

[26] J. Chin, E. Wang, and M. Kibbe. Evaluation of hyperspectral technology for assessing the presence and severity of peripheral artery disease. Journal of Vascular Surgery, 54 (6): 1679–1688, 2011.

[27] E. Larsen, L. Randeberg, E. Olstad, O. Haugen, A. Aksnes, and L. Svaasand. Hyperspectral imaging of atherosclerotic plaques in vitro. Journal of Biomedical Optics, 16 (2), 2011.

[28] J. Schweizer, J. Hollmach, G. Steiner, L. Knels, R. Funk, and E. Koch. Hyperspectral imaging - a new modality for eye diagnostics. Biomedizinische Technik, 57 suppl. 1, 293–296, 2012.

[29] S. Panasyuk, S. Yang, D. Faller, D. Ngo, R. Lew, J. Freeman, and A. Rogers. Medical hyperspectral imaging to facilitate residual tumor identification during surgery. Cancer Biology and Therapy, 6 (3): 439–446, 2007.

[30] K. J. Zuzak, S. C. Naik, G. Alexandrakis, D. Hawkins, K. Behbehani, and E. Livingston. Intraoperative bile duct visualization using near-infrared hyperspectral video imaging. The American Journal of Surgery, 195 (4): 491 – 497, 2008.

[31] E. Olweny, S. Faddegon, S. Best, N. Jackson, E. Wehner, Y. Tan, K. Zuzak, and J. Cadeddu. Renal oxygenation during robot-assisted laparoscopic partial nephrectomy: Characterization using laparoscopic digital light processing hyperspectral imaging. Journal of Endourology, 27 (3): 265–269, 2013.

[32] H. Akbari, Y. Kosugi, K. Kojima, and N. Tanaka. Blood vessel detection and artery-vein differentiation using hyperspectral imaging. Engineering in Medicine and Biology Society, (EMBC). 1461–1464, 2009.

[33] National Institute for Health and Clinical Excellence. Improving Outcomes for People with Brain and Other CNS Tumours. NICE Guidelines (CSG10), 2016.

[34] N. Sanai, M. S. Berger. Glioma extent of resection and its impact on patient outcome. Neurosurgery, 62, 753–766, 2008.

[35] N. Sanai, M. S. Berger. Operative techniques for gliomas and the value of extent of resection. Neurotherapeutics, 6, 478–486, 2009.

[36] K. Petrecca, M.-C. Guiot, V. Panet-Raymond, L. Souhami, Failure pattern following complete resection plus radiotherapy and temozolomide is at the resection margin in patients with glioblastoma. J. Neurooncol. 111, 19–23, 2013.

[37] W. Stummer, J.-C. Tonn, H. M. Mehdorn, U. Nestler, K. Franz, C. Goetz, A. Bink, U. Pichlmeier, ALA-Glioma Study Group, Counterbalancing risks and gains from extended resections in malignant glioma surgery: A supplemental analysis from the randomized 5-aminolevulinic acid glioma resection study. Clinical article. J. Neurosurg. 114, 613–623, 2011.

[38] R. E. Kast, G. W. Auner, M. L. Rosenblum, T. Mikkelsen, S. M. Yurgelevic, A. Raghunathan, L. M. Poisson and S. N. Kalkanis, Journal of Neuro-Oncology, 120, 55–62, 2014.

[39] M. H. T. Reinges, H.-H. Nguyen, T. Krings, B.-O. Hütter, V. Rohde, J. M. Gilsbach, Course of brain shift during microsurgical resection of supratentorial cerebral lesions: Limits of conventional neuronavigation. Acta Neurochir. 146, 369–377, 2004.

[40] K. A. Ganser, H. Dickhaus, A. Staubert, M. M. Bonsanto, C. R. Wirtz, V. M. Tronnier, S. Kunze, Quantification of brain shift effects in MRI images. Biomed. Tech. Suppl. 42, 247–248, 1997.

[41] W. Stummer, U. Pichlmeier, T. Meinel, O. D. Wiestler, F. Zanella and H.-J. Reulen, The Lancet Oncology, 7, 392–401, 2006.

[42] F. W. Floeth, M. Sabel, C. Ewelt, W. Stummer, J. Felsberg, G. Reifenberger, H. J. Steiger, G. Stoffels, H. H. Coenen and K.-J. Langen, European Journal of Nuclear Medicine and Molecular Imaging, 38, 731–741, 2011.

[43] Breiman, L. Random forests. Machine learning, 45(1), 5-32, 2001.

[44] Breiman, L. Bagging predictors. Machine learning, 24(2), 123-140, 1996.

[45] Cano, G., Garcia-Rodriguez, J., Garcia-Garcia, A., Perez-Sanchez, H., Benediktsson, J. A., Thapa, A., & Barr, A. Automatic selection of molecular descriptors using random forest: Application to drug discovery. Expert Systems with Applications, 72, 151-159, 2017.

[46] Louppe, G. Understanding random forests: From theory to practice. arXiv preprint arXiv:1407.7502, 2014.

[47] Waldvogel, B. Accelerating random forests on CPUs and GPUs for object-class image segmentation (Doctoral dissertation, Rheinische Friedrich-Wilhelms-Universität Bonn), 2013.

[48] N.H Praptono, Pahala Sirait, M. Ivan Fanany and Aniati Murni Arymurthy. An automatic detection method for high density slums based on regularity pattern of housing using Gabor filter and GINI index. International Conference on Advanced Computer Science and Information Systems (ICACSIS), 347–351, 2013.

[49] Perveen, N., Gupta, S., & Verma, K. Facial expression recognition using facial characteristic points and Gini index. In Engineering and Systems (SCES), 2012 Students Conference on (pp. 1-6). IEEE, 2012.