Máster en Ingeniería de Sistemas y Servicios para la Sociedad de la Información

Trabajo Fin de Máster		
Título	Energy and Performance Modeling of NVIDIA Jetson TX1 Embedded GPU in Hyperspectral Image Classification Tasks for Cancer detection Using Machine Learning	
Autor	Jaime Sancho Aragón	
Tutor	Rubén Salvador Perea	VºBº
Ponente		
Tribunal		
Presidente	Juana Arriola Gutiérrez	
Secretario	Miguel Chavarrías Lapastora	
Vocal	César Sanz Álvaro	
Fecha de lectura	20-07-2018	
Calificación		

El Secretario:

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación

MSc in Systems and Services Engineering for the Information Society (MSSEIS)

Master Thesis

Energy and Performance Modeling of NVIDIA Jetson TX1 Embedded GPU in Hyperspectral Image Classification Tasks for Cancer detection Using Machine Learning

Jaime Sancho Aragón

2018

For all those who ever believed in me

Resumen

En los últimos años la inteligencia artificial se ha convertido en una de las ramas más importantes de la ciencia de la computación. Su uso se ha extendido no solo a diversos campos de investigación sino que también se encuentra ya como componente fundamental de multitud de aplicaciones y servicios tanto en la nube como cada vez más en sistemas empotrados. Un campo de aplicación con grandes repercusiones sociales es el de la medicina, donde se pueden explotar los algoritmos de inteligencia artificial para encontrar patrones de enfermedades y detectarlas automáticamente mediante el procesamiento de diversos tipos de imágenes y pruebas clínicas.

Este trabajo está enmarcado en el proyecto europeo HELICoiD, en el que se hace uso de imágenes híper espectrales junto con algoritmos de aprendizaje automático para detectar tejido canceroso en cerebros humanos durante operaciones de resección de tumores. La gran cantidad de información disponible en este tipo de imágenes ofrece la posibilidad de clasificar sus píxeles entre tejido sano o tejido canceroso.

Sin embargo, precisamente ese gran volumen de datos contenido en las imágenes híper espectrales así como la carga computacional de los algoritmos utilizados en el proyecto: *Principal Component Analysis (PCA)*, *Support Vector Machine (SVM)* y *K-Nearest Neighbours (KNN)*, requieren una capacidad de cómputo que impide el procesamiento en tiempo real con un procesador de propósito general. Por este motivo, se hace uso de un acelerador basado en una GPU empotrada: el sistema Jetson TX1 de NVIDIA.

En este documento se recogen el análisis de los algoritmos, la plataforma y el lenguaje de programación de NVIDIA CUDA C/C++ para proponer una implementación que cumpla los requerimientos de la aplicación. Además, se incluye una implementación alternativa a la original que obtiene resultados funcionales semejantes (con cierto error) a los originales y un tiempo de ejecución mucho menor. Esta idea, basada en el algoritmo SVM y un filtro espacial, está recogida en el trabajo fin de grado de Guillermo Bermejo.

Este trabajo tiene como objetivo no solo la implementación de la cadena de procesamiento de HELICoiD sino también evaluar la idoneidad de este tipo de plataformas para el desarrolo futuro de sistemas de clasificación de bajo coste en aplicaciones de medicina personalizada. Por ello, se recoge también una caracterización de la plataforma en términos de rendimiento y eficiencia energética para este tipo de aplicaciones.

Para ello se ha desarrollado un método para medir la energía consumida por la plataforma basado en el medidor de voltaje integrado INA3221. Utilizando este método, se desarrollan dos aplicaciones: un monitor gráfico para medir la potencia consumida en tiempo real y una biblioteca de funciones de C para medir la energía consumida durante la ejecución

de una aplicación (basada en C).

Finalmente, utilizando las bibliotecas proporcionadas por CUDA y la desarrollada para medir energía, se aborda la caracterización de la plataforma y se ofrecen resultados de rendimiento y eficiencia energética para los dos proyectos y los algoritmos que los forman. Por último se concluye con una discusión de los resultados donde se extraen las conclusiones de este trabajo y se plantean algunas posibles líneas futuras de trabajo.

Abstract

The last few years the artificial intelligence has emerged as one of the most important branches in computing science. It is being currently used not only in very different research fields but also as a key component in a lot of cloud applications, services and embedded systems. Medicine is an area of application with high social impact where artificial intelligence can contribute to automatic medical diagnosis through the processing of diverse images and clinical tests.

This work is framed within the European project HELICoiD, in which hyperspectral images and machine learning algorithms are used to detect cancerous tissue in human brains during tumour resection operations. The huge amount of information available in this type of images enables automatic pixel classification between healthy or cancerous tissue.

Nevertheless, it is indeed this huge data volume and the computational load required by the involved algorithms; *Principal Component Analysis (PCA)*, *Support Vector Machine (SVM)* and *K-Nearest Neighbours (KNN)* that require a processing power unattainable in real time by a general purpose processor. As a consequence, an embedded GPU-based accelerator is used in this work: the NVIDIA Jetson TX1 system.

This document tackles the algorithm, platform and NVIDIA CUDA C/C++ programming language analysis and an implementation proposal that achieves the application requirements. In addition, an alternative implementation, which obtains similar functional results (with a certain level of error) to the original, is included. This idea, based on the SVM and an spatial filter, is proposed in the final degree project of Guillermo Bermejo.

This work aims not only at implementing the HELICoiD processing chain but also at evaluating the suitability of this type of platforms for future embedded, low-cost classification systems in personalised medicine applications. Therefore, this work also tackles the platform characterisation in terms of throughput and energy efficiency for this kind of applications.

To do so, a method to measure the energy consumed by the platform based on the integrated voltage sensor INA3221, is developed in this work. With the method, two applications are created: a graphical monitor for real time power monitoring and a C library for energy consumption measurement of C-based applications.

Finally, using CUDA libraries and the energy measurement library developed in this work, the platform is characterised obtaining throughput and energy efficiency results for both projects and the algorithms involved. Lastly, a final discussion on the obtained results is included, and possible future lines of work are sketched.

Contents

List of Figures

List of Tables

List of Equations

Acronyms

AI Artificial Intelligence.

CSI Camera Serial Interface.

CUDA Compute Unified Device Architecture.

DP Display Port.

DSI Digital Serial Interface.

eDP embedded Display Port.

FPGA Field-Programmable Gate Arrays.

GDEM by its initials in Spanish Electronic and Microelectronic Design Group.

GPC Graphics Processing Cluster.

GPGPU General Purpose on Graphics Processing Units.

GPIO General Purpose Input Output.

GPP General Purpose Processor.

GPU Graphics Processing Units.

HDMI High Definition Multimedia Interface.

HELICoiD HypErspectraL Imaging Cancer Detection.

HEVC High Efficiency Video Coding.

HS Hyperspectral.

HSI Hyperspectral Imaging.

I/O Input/Output.

I2C Inter Integrated Circuit.

I2S Integrated Interchip Sound.

JetPack Jetson installer Package.

KNN K-Nearest Neighbours.

LD/ST Load Store Units.

LUTs Look-Up Tables.

MFCC Mel-Frequency Cepstral Coefficients.

MIMD Multiple Instruction Multiple Data.

ML Machine Learning.

MPEG Moving Pictures Expert Group.

MPPA Massively Parallel Processor Array.

NIPALS Non-Linear Iterative Partial Least Squares.

NIR Near infrarred.

NoC Network-on-Chip.

PCA Principal Components Analysis.

ROPs Raster Operation Pipelines.

SFU Special Function Units.

SIMD Single Instruction Multiple Data.

SM Streaming Multiprocessor.

SMM Streaming Multiprocessor Maxwell.

SoC System on Chip.

SoM System on Module.

SPI Serial Peripheral Interface.

SVM Support Vector Machines.

UART Universal Asynchronous Receiver-Transmitter.

UPM Universidad Politécnica de Madrid.

VNIR Visible and near-infrared.

1. Introduction

1.1. Motivation

In the last fifteen years, the growth in the maximum processors's frequency has been reduced dramatically due to the impossibility of dissipating the generated heat by the more and more transistors in a chip. This problem encouraged manufacturers to search for alternatives so that the compute capability could be improved. The solution was the use of more than a single core to increment the number of operations per second.

Nowadays, it is common that the commercial computers have four, eight or even sixteen cores to speed up applications processing. Nevertheless, to achieve higher performance, some applications use hardware accelerators such as Graphics Processing Units (GPU) or Field-Programmable Gate Arrays (FPGA).

The paradigm shift to parallel computing has enabled the implantation of applications with a huge amount of data and computational loads such as Artificial Intelligence (AI) algorithms. These algorithms are currently used in lots of different applications and constitute a hot research topic for the computer science community.

This master thesis follows the previous idea framed within the European project HypErspectraL Imaging Cancer Detection (HELICoiD) [47] funded by the Research Executive Agency, through the Future and Emerging Technologies (FET-Open) programme. The purpose of the project was the detection of cancerous tissue in the human brain using Hyperspectral Imaging (HSI) and Machine Learning (ML) algorithms. The essence of detection using HSI lies in the different behaviour of the reflecting light of healthy and cancerous tissue: with an adequate processing it is possible to differentiate between them given the spectral signatures are distinct enough.

Hyperspectral (HS) images contain the spectral information necessary to create the spectral signature: a curve representing the reflectance for each one of the acquired bands for one pixel (or the average pixel of a region). With the spectral signature of each type of tissue, it is possible to classify the different type of tissue using models based on the spectral signatures of the different types of cells. Nevertheless, this process involves a large computational load due to the amount of information in HS images. This fact represents a problem since the images need to be acquired and processed during the surgical operation: the acquisition must be done directly over the brain and the resulting classification image must be computed so that the surgeon can use it to decide the amount of cancerous tissue. As a consequence, the time constraint stipulated during the project by surgeons was approximately 120 seconds.

This issue leaded HELICoiD researchers to accelerate the processing part using a many-core platform called Massively Parallel Processor Array (MPPA) from Kalray Corporation. This work was carried out by the Electronic and Microelectronic Design Group (GDEM by its initials in Spanish) at the Universidad Politécnica de Madrid (UPM) and the work

done in this master thesis replicates this functionality in a different platform, to evaluate the suitability of embedded GPUs for this task.

In this case, the processing part of the HELICoiD application is accelerated in the NVIDIA Jetson TX1 platform, which stands for a low-power, SoC-based GPU embedded system that uses CUDA C/C++ as programming language. The algorithm chain considered is composed of three ML algorithms: Principal Components Analysis (PCA), Support Vector Machines (SVM) and K-Nearest Neighbours (KNN) and the core of this work is related with the implementation and optimisation of the PCA. Although existing versions of the other two algorithms made by other students are used, their integration into the final application and the complete system optimisation are tackled in this project too.

In addition, apart from the original processing chain developed in HELICoiD, another one proposed by Guillermo Bermejo [3] is evaluated in the platform. This alternative chain is compared with the original in terms of throughput and energy efficiency in order to characterise the NVIDIA Jetson TX1 system.

1.2. Objectives

The purpose of this work is the implementation of a case study based on ML using the GPU as an accelerator to measure the energy used by the platform and the throughput achieved for the application.

The objectives of this work project are the following:

- To accelerate the PCA algorithm used in the project:
  - To implement a functional PCA for HS images classification tasks.
  - To optimise the implementation accomplishing the time constraints of the project.
  - To study common CUDA libraries.
  - To search for implementation alternatives.
  - To implement and optimise the alternatives.
- To integrate the entire application using the three original algorithms (PCA, SVM, KNN).
- To optimise the final system implementation.
- To accelerate the alternative chain.
- To find a proper method to measure the processing time.
- To model the energy used by the platform:

  - To find a proper method to measure the energy consumed.
  - To provide an efficient and simple method to monitor the energy used in the platform in real time.
  - To provide a method to measure the energy consumed by an specific application.

- To characterise the platform in terms of performance and energy efficiency for this type of applications.

1.3. Structure

Chapter 2 explains in detail the basis of both projects analysing their structure. Then, HS images and algorithms involved (PCA, SVM, KNN and Spatial Filter) are first theoretically introduced and later framed in this project. Finally, different accelerators are briefly described along with some related works.

Chapter 3 introduces Jetson TX1 platform and its environment. Not only Jetson's hardware capabilities are explained but also its GPU microarchitecture (Maxwell). In addition, NVIDIA programming language CUDA C/C++ is described taking into consideration the microarchitecture features. To conclude, several best practices are described to provide insight into the optimisation stage.

Chapter 4 combines the algorithm definitions in Chapter 2 with the platform knowledge in Chapter 3. This way, all required algorithm implementations in NVIDIA Jetson TX1 are explained. While two implementations are done and optimised in this work (PCA and Spatial Filter), the other two (SVM and KNN) are implemented by other students. In the case of the first group, different implementation options are included (CUDA C/C++ code is represented in Appendix A for PCA and Appendix B for the Spatial Filter).

Chapter 5 uses algorithm implementations to integrate the two different projects. It describes the required inputs and outputs for each algorithm and how to connect them to generate the whole application.

Chapter 6 explains the energy characterisation in the platform. Describing how the Voltage Monitor INA3221, integrated in Jetson TX1, works, the chapter introduces two applications developed in this work: a graphical monitor to measure the power in real time and a C library to measure the energy consumed by C-based applications.

Chapter 7 includes the time and energy results for all the algorithm implementations and for the two whole applications. Partial results from PCA and the Spatial Filter are also considered. In addition, functional and processing results of both projects are compared with reference results (from HELICoiD).

Chapter 8 present the conclusions extracted from the results and related future lines.

2. Background

2.1. Introduction

This chapter explains the background in which this project is framed. It is divided into four sections.

The first briefly introduces the context of the project, describing the general structure and mentioning the algorithms involved. The second describes HS images in depth, presents some other applications and introduces their acquisition and pre-processing steps in HELICoiD. The third, explains theoretically the algorithms used in the project and then frame them in the context of HELICoiD. Finally, the architecture of different accelerators is introduced along with related works.

2.2. Project context

This master thesis is framed within the European project HELICoiD. Its purpose is the detection of cancerous tissue in the human brain making use of image processing ML algorithms in a *short period* (it will be explained) of time. The huge amount of information present in HS images allows the differentiation between healthy and cancerous tissue by using a chain of ML algorithms, as HELICoiD demonstrated. Nevertheless, these algorithms involve such a high computational load that it is necessary to accelerate the process in order to achieve the required time constraints. In this case, these constraints are imposed by surgeons, who need that the process does not take longer than 2 minutes approximately [7].

In Figure 2.1, the complete algorithm developed in HELICoiD is shown. It is divided into two main parts, the off-line process **(a)** and the in-situ process **(b)**.

The former part is in charge of creating a model able to differentiate between healthy and cancerous tissue. In order to obtain the model, experienced neurosurgeons identified the spatial regions corresponding with the cancerous tissue in a RGB representation of the captured HS images. This way, used as training set, the pixels representing the cancerous tissue are recognised and used to create a model. The generation of the model does not need to accomplish the time constraints since it is made in advance and used during the in-situ process.

The latter part generates a final image called classification map representing the regions with healthy and cancerous tissue using only the new HS images and the aforementioned model. In this case, the time constraints need to be fulfilled since the acquisition of the HS images is done in the surgery room and the result is needed by the surgeon to continue the operation. As it can be seen in Figure 2.1, the process is composed of a pre-processing step which enhances the HS image and two different processes using it as an input.

Processes included in the in-situ part are represented in Figure 2.2 along with the

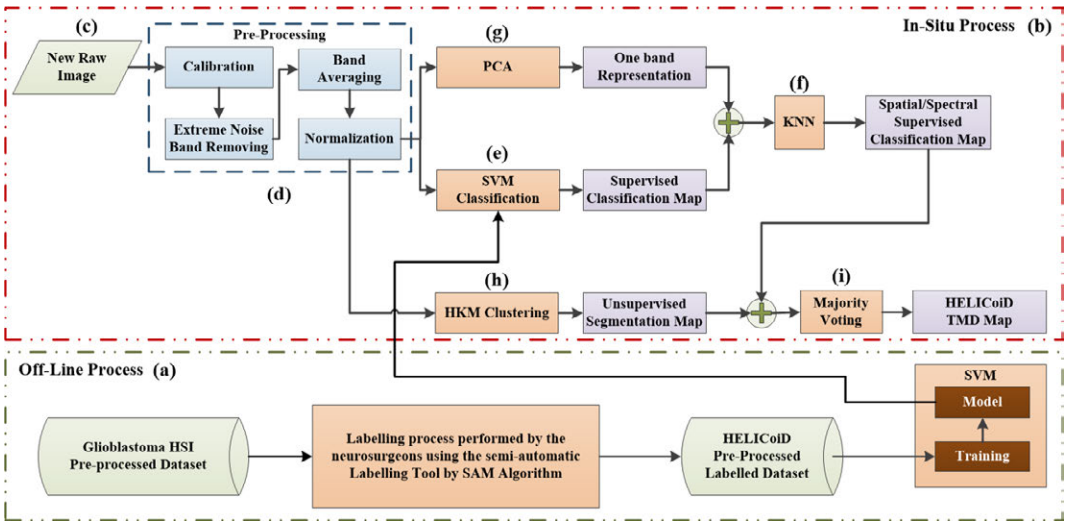


Figure 2.1: HELICoiD scheme [8].

system in which they are implemented in HELICoiD. While the pre-processing of the HS image and the clustering algorithms are computed in a GPP, the PCA, SVM and KNN algorithms are processed in a manycore platform, the MPPA from Kalray. The reason for that is the necessity of accelerating the mentioned algorithms in order to fulfil the time constraints.

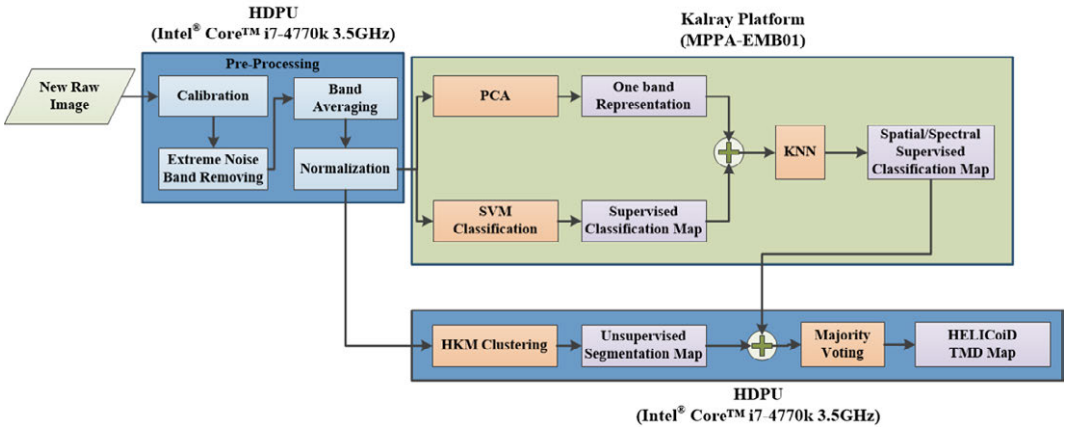
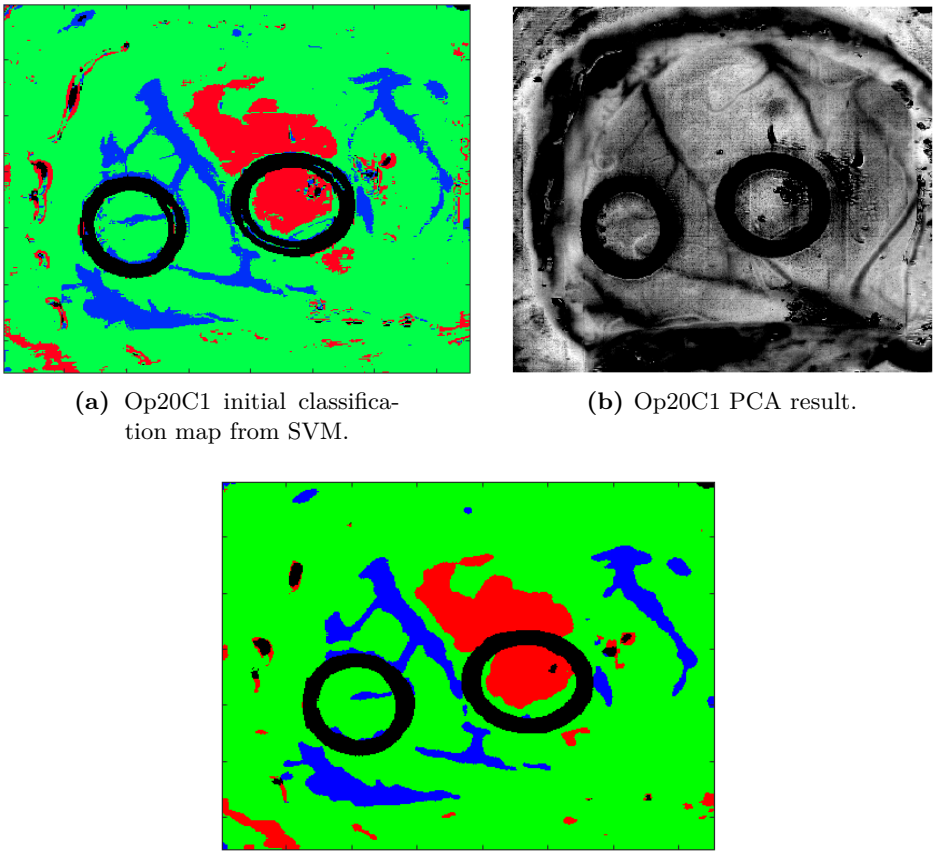


Figure 2.2: HELICoiD processing part in-situ scheme [8].

This master thesis focuses on the processing part of HELICoiD that is accelerated in the manycore platform. This is composed of three ML algorithms: PCA, SVM and KNN. Its purpose is the creation of a supervised classification map from the pre-processed HS image that is combined with the unsupervised map created with the clustering algorithm.

The original HELICoiD processing chain (referred to this way from now on in this manuscript) uses the SVM algorithm to obtain an initial pixel-wise classification map. This means that each pixel is classified independently, without taking into account near pixels.

The purpose of PCA and KNN algorithms is the generation of a refined classification map using the spatial and the spectral information of the initial classification map and the result from PCA, respectively. The results for the reference HS image Op20C1 are shown in Figure 2.3.



(a) Op20C1 initial classification map from SVM.

(b) Op20C1 PCA result.

(c) Op20C1 final classification map after KNN.

Figure 2.3: Images generated in the original HELICoiD chain for the HS image Op20C1. The colors in the classification maps represent the following: Green, healthy tissue; red, cancerous tissue; blue, hypervascularised tissue; and black, external material (rubber circles).

Apart from the original HELICoiD chain, this work also includes the acceleration of an alternative project described by Guillermo Bermejo [3]. In that case, the SVM algorithm is used to generate the initial classification map, however, only a spatial filter is used to refine it. The results of this simplification are compared in the terms defined in this work.

2.3. Hyperspectral Images

Before describing the involved ML algorithms, it is important to understand which is the input data used for them. For this reason, this section introduces what an HS image is and the reasons for their use in the context of this work. In addition, the process of acquiring and pre-processing the HS images used in HELICoiD is introduced.

2.3.1. Description

The field of spectral imaging was born in the final years of the XX century as an evolution of spectroscopy [18]. These two terms share part of their definition but they are not the same: while spectroscopy studies the spectral information of a particular point of the space, spectral imaging creates a complete image composed of many of these points. Thus, the former only contributes with spectral information but the latter also includes spacial information [9] [23].

The underlying idea of spectral techniques is that different types of matter behave differently under the incidence of electromagnetic radiation over its surface. The energy of the incident waves can be absorbed, transmitted, scattered or reflected differently by the surface of the objects. Moreover, this behaviour also depends on the wavelength of the electromagnetic wave. As a consequence, it is possible to obtain a curve representing the level of the energy reflected across the wavelength by a specific material. This is called *spectral signature*. This forms the basis of material classification depending on their behaviour under electromagnetic radiation [18] [9].

Figure 2.4 shows the spectral signature of water with different levels of Suspended Particulate Matter (SPM). With the information contained in these curves, it is possible to measure the mg/L of SPM contained in the water by using satellite data instead of analysing the water in a laboratory [6].

Another observation can be made regarding Figure 2.4: the wavelength information begins in 400 nm and finishes in 900 nm. It is common for spectral signatures to include wavelengths from the ultraviolet to the infrared spectrum, representing information invisible for the human being (Figure 2.5 shows the electromagnetic spectrum). These bands usually contain important information about the materials analysed [21].

Spectral imaging extends the previous concept of spectral signature to a matrix of points, resulting in a set of images standing for the reflectance in the different wavelengths. Depending on the number of bands included, the images take different names: with tens of bands, the image is called *multispectral image*, with hundred of bands, *hyperspectral image* and with thousands, it is called *ultraspectral image* [43] [21]. This contrast with the typical idea of images only composed of three bands situated within the visible range of

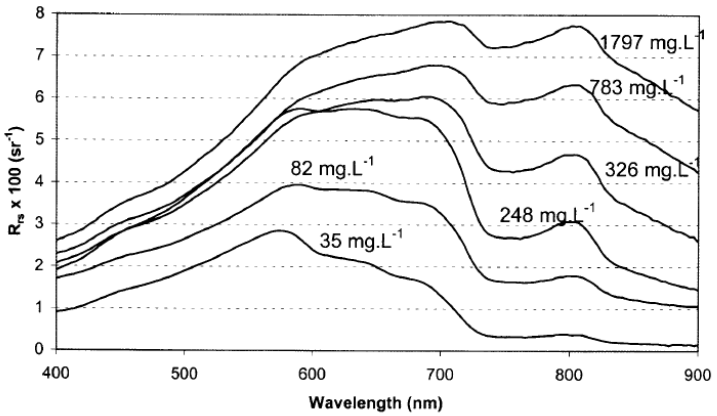


Figure 2.4: Different spectral signatures of water with Suspended Particulate Matter in the Gironde estuary [6].

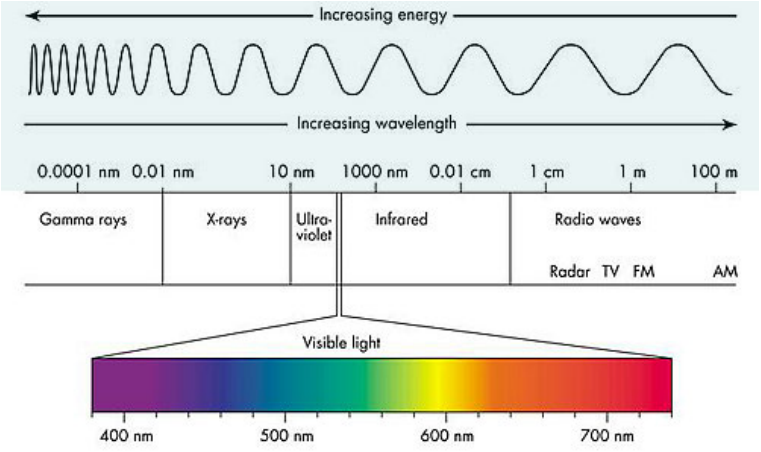


Figure 2.5: Electromagnetic spectrum. Spectrometry focuses on the spectrum between ultraviolet and infrared. Source: http://www.cyberphysics.co.uk/topics/light/emspect.htm

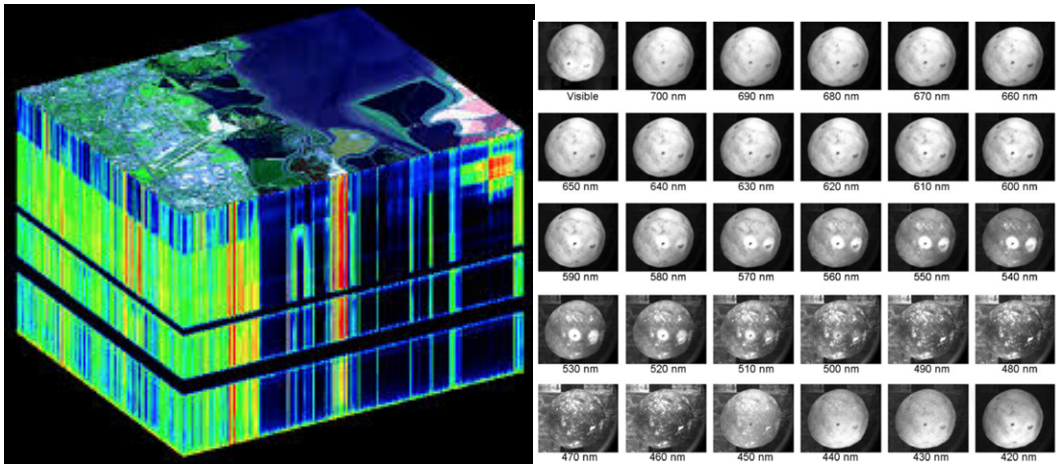
wavelengths corresponding with red, green and blue lights.

The typical representation of a HS image is shown in Figure 2.6 **(a)**. Furthermore, another HS image is included in Figure 2.6 **(b)**, in which the reflectances for different wavelenghts are represented as a set of grey-scale images.

Applications

As said before, applications for spectral imaging are usually focused on the identification of different materials in the spatial region of interest captured by the hyperspectral camera. This section introduces some examples found in the literature.

The first example uses spectral imaging in the field of art conservation and restoration [9]. This research area needs a non-invasive method in order to find out under-drawings



(a) Representation of an hyperspectral image. Source: http://large.stanford.edu/courses/2015/ph240/islam1/

(b) Representation of the different bands in grey-scale of an orange with external defects [23].

Figure 2.6: Different representation of hyperspectral images.

and pentimenti[1] in paintings, among others. Figure 2.7 shows how spectral imaging can help this research area by revealing the painterly pentimenti present in the hand of the picture. While the left image represents the grey-scale image which corresponds with the 640 nm band of the HS image, the right one shows the band corresponding with 1000 nm, in which the painterly pentimenti becomes visible.

Figure 2.7: Monochromatic spectral images at 640 nm and 1000 nm. In the second one, the painterly pentimenti in the hand becomes visible [9].

Spectral imaging can also be used to determine the quality of foods such as fruits and vegetables [23]. The work analyses their external features in different wavelengths and poses a relationship between the appearance and the internal condition for each one of the set of fruits and vegetables studied. This way it is possible to recognise their condition in order to offer a better quality product and to detect an adequate timing for harvesting.

One last example is related with tumor tongue detection [22]. The system presented uses HS images, like the one represented in Figure 2.8, to detect tongue cancer with a

[1] A visible trace of earlier painting beneath a layer or layers of paint on a canvas.

recognition rate of 96.5%. For that, authors use an algorithm based on Sparse Representation (SR).

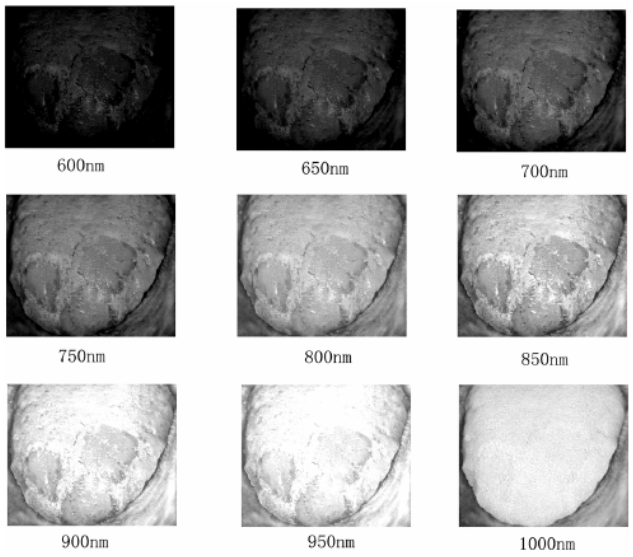


Figure 2.8: Hyperspectral image of tongue tumor [22].

There exist lots of other applications for hyperspectral imaging not covered in this document like earth observation, precision agriculture or security as some surveys cover [43] [21].

2.3.2. Hyperspectral images in HELICoiD

In this subsection the information regarding the HS images used in the HELICoiD project and in this work is introduced.

The acquisition system used is composed of a Visible and near-infrared (VNIR) (range between 400 and 1400 nm) camera, the Hyperspec VNIR A-Series model and a Near infrarred (NIR) (range between 750 and 2500 nm) camera, the Hyperspec NIR 100/U model, both manufactured by HeadWall Phothonics [7]. Figure 2.9 shows the set up used in HELICoiD [8].

The former is capable of capturing a spectral range from 400 nm to 1000 nm with a spectral resolution of 2-3 nm and a spatial resolution of 1004 pixels in just one dimension. As a consequence, it is necessary to move the camera so that a two-dimensional image is acquired. This is done by attaching the camera to a scanning platform which uses a stepper motor to cover an area of 230 mm. This type of HS cameras are known as pushbroom.

The latter captures a spectral range from 900 nm to 1700 nm with a spectral resolution of 5 nm and a spatial resolution of 320 pixels. Since the spatial resolution in this camera is one-dimensional too, the scanning process is similar to the previous one. However, as

found in practice during the project, the information obtained in these wavelengths did not contribute much to the final classification results.

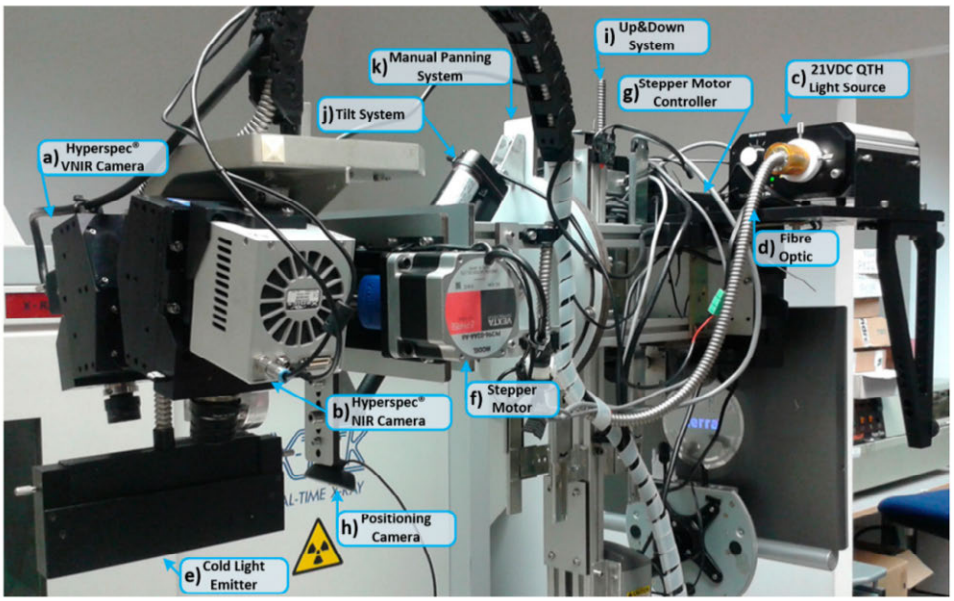


Figure 2.9: HELICoiD acquisition system [7].

Once captured, the HS images are preprocessed. This process, shown in Figure 2.10, needs to define the maximum and minimum reflectance values for each pixel and for each band. Consequently, a hyperspectral reference image for the maximum (special white tile) and for the minimum (shutter of the camera closed) is used. With this calibration made, the image is filtered with a special noise filter and then the highest and lowest bands which have low SNR are removed. Finally, the remaining bands are averaged and normalized in order to obtain 129 normalized spectral bands (in practice just the first 128 bands are used in order to facilitate calculations that need power of two values) [8].

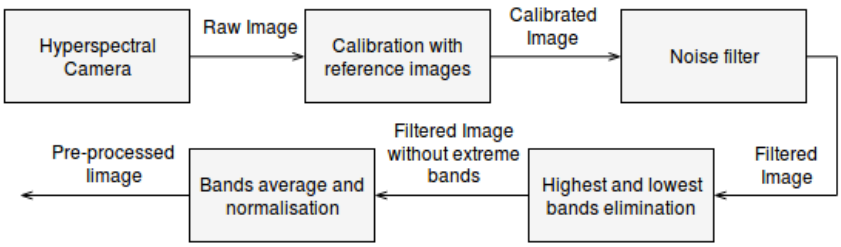
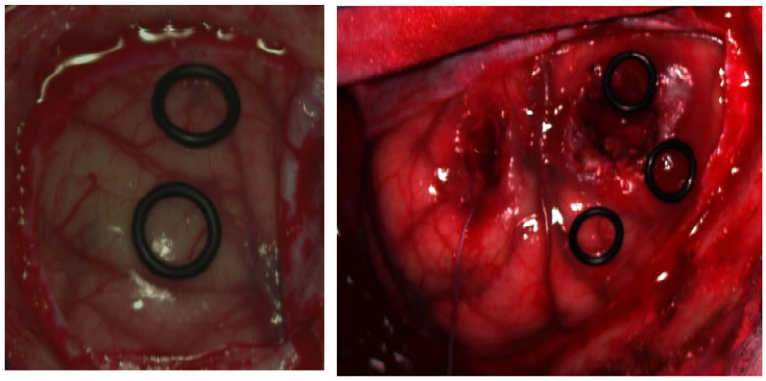


Figure 2.10: HS images pre-processing used in HELICoiD.

In this work, the HS image database from operations generated in HELICoiD is used[2] [8]. Table 2.1 shows the name of all the images along with their dimensions. Figure 2.11 **(a)** and 2.11 **(b)** show the RGB representation of the Op20C1 and Op8C2 HS images, respectively.



(a) Op20C1. **(b)** Op8C2.

Figure 2.11: RGB representation of two HS images from the HELICoiD project.

Table 2.1: Hyperspectral Images included in the HELICoiD database and their dimensions.

Name	Rows	Columns	Bands
Op8C1	459	548	128
Op8C2	479	552	128
Op12C1	442	496	128
Op12C2	444	497	128
Op15C1	375	493	128
Op20C1	377	329	128

2.4. Algorithms

2.4.1. Principal Components Analysis (PCA)

The Principal Component Analysis is a common ML algorithm used to reduce the data dimensionality of the vectors included in a dataset. The aim of the algorithm is to concentrate the maximum information possible but reducing the number of features used to describe a vector.

Notation in this document follows previous generally accepted notations [30], in which one vector of the dataset is expressed as $x \in \mathbb{R}^n$. This means that each vector of the dataset is defined by n features, hence, the vector space has n dimensions. In addition,

[2] https://hsibraindatabase.iuma.ulpgc.es/

the different vectors belonging to the dataset are noted as $x^{(i)}$ with $\{i = 1, \ldots, m\}$ being m the number of vectors.

The goal of the PCA algorithm is to find a subspace $x \in \mathbb{R}^{\mathrm{k}}/\ \mathrm{k} \ll \mathrm{n}$ in which the data can be expressed with the maximum variance [30]. For example, in Figure 2.12 a set of data represented depending on its two features is observed, i.e., the vector space of this data is $\mathbb{R}^2$. The new axes u_1 and u_2 represent the new vector space in which the data can be expressed with the maximum and minimum variance. In this example, if only the u_1 axis is considered and the data is projected onto it, the loss of information is negligible (for classification) and hence the dimensionality of the problem has been reduced. This example is used due to the ease of representing a 2 dimension vector space, but the reduction of one dimension is not, in general, useful enough.

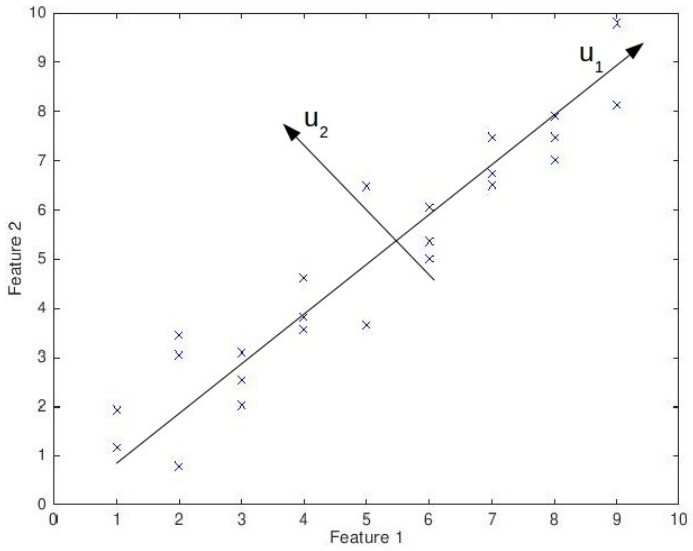


Figure 2.12: Representation of a data set with two features and the axes generated by PCA.

The process for obtaining the vector subspace begins with the subtraction of the mean of each of the features in order to obtain a dataset with zero mean:

$$\bar{x}_n = \frac{1}{m}(x_n^{(1)} + \cdots + x_n^{(m)}) = \frac{1}{m}\sum_{i=1}^{m} x_n^{(i)} \tag{2.1}$$

$$x_n^{'(i)} = x_n^{(i)} - \bar{x}_n \tag{2.2}$$

Now, a method to calculate the maximum variance in the new axis (vector u) is needed. This is accomplished by maximizing the variance of the projections $x^T u$ [30] (where Σ is

the covariance matrix of the data):

$$\begin{aligned}\frac{1}{m}\sum_{i=1}^{m}(x'^{(i)^T}u)^2 &= \frac{1}{m}\sum_{i=1}^{m}u^T x'^{(i)}x'^{(i)T}u \\ &= u^T\left(\frac{1}{m}\sum_{i=1}^{m}x'^{(i)}x'^{(i)T}\right)u \\ &= u^T\Sigma u\end{aligned} \tag{2.3}$$

Equalling the previous result to an undetermined constant λ it is immediate to recognise that what it is looked for are the eigenvectors of the covariance matrix:

$$u^T\Sigma u = \lambda \rightarrow \Sigma u = \lambda u \tag{2.4}$$

The constant λ is the eigenvalue associated to the eigenvector u and its value represent how much the eigenvector is capable of representing the data. As a consequence, all the eigen-pairs need to be found and then select those with the highest eigenvalues, since these represent the best vector space in which the dimensionality reduced dataset data can lie [50].

Finally, in order to express the initial data onto the created subspace, it is necessary to project the data in the k selected eigenvectors to obtain each one of the k components of the vector:

$$y^{(i)} = \left[u_1^T x^{(i)}, \ldots, u_k^T x^{(i)}\right] \tag{2.5}$$

For further information regarding the PCA algorithm please check Andrew Ng.'s work [30].

PCA in HELICoiD

The PCA algorithm in HELICoiD is applied exactly as described previously. In the case of the European project, HS images stand for a set of $m = imSize = rows \times cols$ vectors (pixels) with $n = 128$ dimensions which are dimensionality reduced to a single dimension $k = 1$.

The method used to calculate the eigenvectors and eigenvalues is called Jacobi [49]. It is an iterative algorithm in which the stop condition is determined by a fixed constant used to control the percentage of error made and the number of necessary iterations. Therefore it derives that the number of iterations is not only dependent on the size of data but also on the data itself, i.e., for two different $n \times n$ sized matrices, the number of iterations can

vary heavily.

The basic algorithm is illustrated in Figure 2.13. The process consists of a loop that accesses serially to all non-diagonal elements of the top part of the covariance matrix. In every iteration, the pair of elements (i, j) and (j, i) from the covariance matrix A are eliminated (set to 0) by rotating the matrix. As it can be seen in the line 9, a rotation matrix P depending on the element row and column is created. Multiplying the covariance matrix with P^t and P, the aforementioned elements are eliminated. At the end, the eigenvalue matrix E is composed of the elements in the diagonal when all the non-diagonal ones are lower than ε. The eigenvector matrix V is obtained as the matrix multiplication of all the rotation matrices used. Nevertheless, this algorithm needs more than $n \times n$ iterations because whenever the matrix is rotated to eliminate a pair, it is likely that during the process some previously eliminated (zeroed) elements are modified. For this reason, the process is performed more than once per element of the matrix. In this project, this number is called *global iterations*.

Input A (matrix), ε (error parameter), size
Output E (eigenvalue matrix), V (eigenvector matrix)

```
1:  procedure JACOBI METHOD
2:      A_0 = A
3:      E_0 = I_size
4:      k = 0
5:      while |some non-diagonal element of A_k| > ε do
6:          for i = 0 → size do
7:              for j = i + 1 → size do
8:                  if |A_{k-1}(i, j)| > ε then
9:                      create P_k(i, j)
10:                     A_k = P_k^t · A_{k-1}^t · P_k
11:                     E_k = E_{k-1} · P_k
12:                     k = k + 1
13:         N_global = N_global + 1
14:     V = A_k
15:     E = E_k
```

Figure 2.13: Pseudo-code for the Jacobi method.

In addition to the original algorithm presented, Jacobi allows certain level of inherent parallelism. It is possible to generate a rotation matrix that eliminates more than one element per iteration. Since the rotation matrix used to remove the element (i, j) is composed of only four elements placed in $(i, i), (i, j), (j, i), (j, j)$, to eliminate the element (i, j) only the rows i, j and the cols i, j from the matrix are changed. This allows to create a rotation matrix capable of removing up to $\frac{n}{2}$ (n is one dimension of the covariance

matrix) elements that do not interfere between themselves.

Nevertheless, since in this work only the first principal component is needed, the calculation of all the eigenvectors and eigenvalues is unnecessary. This fact enables the use of a different method to calculate the eigenvectors and eigenvalues iteratively from the most important to the least, i.e, only the ones needed for the given application are computed. This algorithm is called Non-Linear Iterative Partial Least Squares (NIPALS) [20], and it is used in the work as an alternative to the original method.

NIPALS algorithm replaces the calculation of the covariance matrix, the obtention of the eigenvectors and eigenvalues and the projection. As a consequence, only the preprocessing part and the NIPALS algorithm are required in order to obtain the first principal component. The algorithm is shown in Figure 2.14.

Input A (matrix), ε (error parameter), N (number of principal components), i_{max} (maximum number of iterations permitted)

Output T (principal components), P (associated eigenvectors), R (residual matrix)

1: **procedure** NIPALS METHOD
2: $R = A$
3: **for** $k = 0 \rightarrow N$ **do**
4: $\lambda = 0$
5: $T_k = R_k$
6: **while** $i < i_{max}$ **do**
7: $P_k = R^T \cdot T_k$
8: $P_k = P_k / ||P_k||$
9: $T_k = R \cdot P_k$
10: $\lambda' = ||T_k||$
11: $i = i + 1$
12: **if** $|\lambda' - \lambda| > \varepsilon$ **then**
13: **break**
14: **else**
15: $\lambda = \lambda'$
16: $R = R - T_k \cdot (P_k)^T$

Figure 2.14: Pseudo-code for the Nipals method.

2.4.2. Support Vector Machine (SVM)

The SVM is a classifier, an algorithm capable of differentiating input vectors based on the information given by a certain database. The main aim is the creation of models that can extract patterns from the examples in the database in order to classify the new samples.

This specific classifier is included in the supervised learning category, which means that the examples used for training the model are analysed and categorised into a priori known set of classes. Taking this into account, and using the aforementioned notation, the dataset can be represented by a number I of vectors with n dimensions, where each one is associated with a c_k class: $(x^{(i)}, c_k^{(i)}) \mid x \in \mathbb{R}^n,\ \mathrm{k} = \{\mathrm{class}_0, \ldots, \mathrm{class}_K\}$.

The model generated by the SVM algorithm is calculated by finding an hyperplane (n-dimensional) capable of separating the examples belonging to different classes. A common example is represented in Figure 2.15, where it can be observed that a two-dimensional hyperplane (a line) is separating the vectors associated with the class circle and with the class x [26].

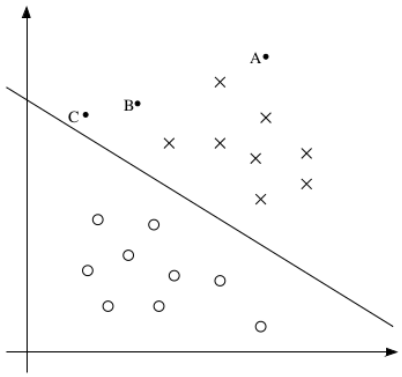


Figure 2.15: Representation of a data set with two features in which the circles represent the first class and the x's the second class. The line separates the examples from different classes as the SVM classifier would do [31].

Since it is possible to find infinite hyperplanes achieving the goal, the concept of margins is introduced. The algorithm not only finds an hyperplane that separates the examples but also tries to maximise the margins, defined as the distance between the hyperplane and the closest example of each class. This is done to separate the classes between them as much as possible in order to improve classification in the regions near to the decision hyperplane [4].

Once the model is generated, the location in the vector space of the new examples is taken into account in order to determine if the examples belong to a region or to another. Logically, the nearest examples to the hyperplane are more likely to be wrong than the farthest ones. Hence, it is also necessary a way to determine the grade of reliability of the classification, which is usually a percentage. Going back to Figure 2.15, it is easy to

assume that the probability of belonging to class x for the new example A is much greater than for C [51] [31].

The usage of hyperplanes limits the classifier to be used with datasets that can be separated linearly. However, some pattern recognition systems needs to determine non-linear regions. As a consequence, SVMs need to be able to deal with this, so with different kernels can be used depending on the type of input data. The method using hyperplanes, known as Linear Kernel Method, was the first to appear. Others are the Gaussian Radial Basis or the Hyperbolic Tangent Kernel [26].

This flexibility makes it possible to use the classifier in several applications with different classification needs such as remote sensing, in which SVMs are used to differentiate materials in HS images captured from satellites [26].

SVM in HELICoiD

The SVM model used in HELICoiD is based on the information from the RGB images classified manually by the neurosurgeons. The difference from the theory is the number of classes. Since the project defines four classes: healthy tissue, cancerous tissue, hypervascularised tissue and external materials, there is not just one hyperplane separating the examples from one class an another. In contrast, the model is composed of six hyperplanes that separates classes in pairs. The idea is represented in Table 2.2.

Table 2.2: Assignation of SVM binary classifiers (hyperplanes) using four classes.

	Class 0	**Class 1**	**Class 2**	**Class 3**
Class 0	-	Classifier 0	Classifier 1	Classifier 2
Class 1	Classifier 0	-	Classifier 3	Classifier 4
Class 2	Classifier 1	Classifier 3	-	Classifier 5
Class 3	Classifier 2	Classifier 4	Classifier 5	-

With the hyperplanes from the model, the first step to classify a new vector (a pixel with 128 dimensions) is the calculation of the distance between itself and the six hyperplanes. By this way, six distances are calculated for each pixel using the hyperplanes defined by a weight vector w and a constant ρ [55]:

$$f_c^{(i)} = \sum_{c=0}^{5} w_c \cdot x^{(i)} + \rho_c \tag{2.6}$$

With the distance to each hyperplane, a probability is calculated in order to give a meaning to the results. The probability calculation is made using the sigmoid function, which uses two constants A and B depending on the hyperplane [55]:

$$r_c^{(i)} = \frac{1}{1 + e^{A_c \cdot f_c^{(i)} + B_c}} \tag{2.7}$$

At this point, six probabilities for each pixel are calculated, however, the final purpose of SVM is to get four probabilities for each pixel: a probability of belonging to each one of the defined classes [26]. This is a difficult task with several solutions. In the case of HELICoiD, an iterative method introduced by Ting-Fan Wu [55] (**Algorithm 2**) is used.

2.4.3. K-Nearest Neighbours (KNN)

The KNN algorithm is another example of a supervised learning classifier. In contrast with SVM, this algorithm is called *lazy* and it is one of the simplest machine learning algorithms. The meaning of *lazy* in this context refers to the fact that the classifier does not create a model before classification, since all the calculations are done the moment a new vector arrives to the classifier [16].

In order to perform classification, the KNN algorithm calculates the distance between the new example and all the others within the dataset. It is important to recall that, since the dataset is modelled as a vector space of n dimensions, it is possible to define a distance operator between its elements (commonly, the euclidean distance). Once the distances are computed, the classifier chooses the K samples from the dataset closer to the new example. Finally, the class of the new example will be determined taking into account the classes of the K nearest neighbours, by majority voting [16].

Figure 2.16 shows an easy example of the process with $K = 3$ and a two-dimensional vector space with 2 different classes: circle and square. The new example, represented with a question mark, is located in the vector space and then the three nearest neighbours are found (the ones inside the circle). Since two neighbours belong to the circle class and just one to the class square, the classifier sets the class of the new example as circle.

KNN in HELICoiD

The previous algorithm is the basis for the algorithm included in HELICoiD, however, the idea and the process is different. The algorithm is a spatial-spectral classifier with the purpose of filtering the initial classification map from the SVM algorithm. For doing that, the PCA image, which contains the spectral information, is used [16] [46].

In spatial-spectral KNN, the pixels of the initial classification map are modelled as vectors with their normalised spatial dimension in the image (l, m) and a normalised spectral dimension obtained from the PCA image $s = PCA(l, m)$. Hence, each pixel of the image counts with three dimensions $x^{(i)} = (x_l^{(i)}, x_m^{(i)}, x_s^{(i)})$. In addition, in order to give different importance to the spatial or spectral part of the vector, the parameter λ is

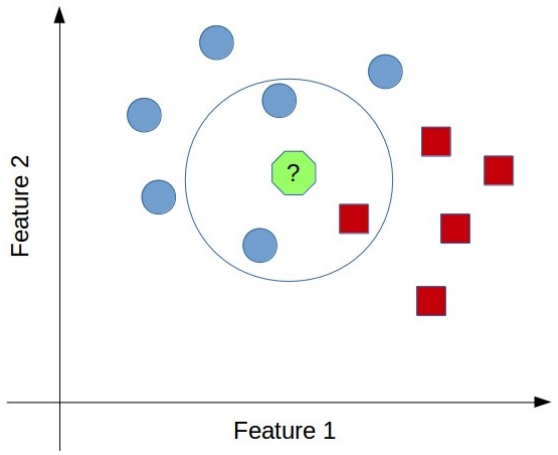


Figure 2.16: Representation of a data set with two features in which the circles represent the first class and squares the second class. A new example with an indeterminate class is classified based on the distance to the other examples.

introduced to the vector definition [45]:

$$x^{(i)} = (\lambda x_l^{(i)}, \lambda x_m^{(i)}, x_s^{(i)}) \tag{2.8}$$

The process compares all the pixels from the initial classification map with all the others in the image using an Euclidean distance. The distance between one pixel $x^{(i)}$ and another $x^{(n)}$ is expressed as [46]:

$$d(x^{(i)}, x^{(n)}) = \sqrt{(\lambda x_l^{(i)} - \lambda x_l^{(n)})^2 + (\lambda x_m^{(i)} - \lambda x_m^{(n)})^2 + (x_s^{(i)} - x_s^{(n)})^2} \tag{2.9}$$

With all the distances calculated for every pixel in the image, the next step consists in selecting the K nearest neighbours for each one of them. Their spatial indexes in the image are used to filter the initial classification map composed of 4 probabilities images (one per class). An average per class per pixel is calculated taking into account the probability values of the pixels corresponding with the K nearest neighbours [46]:

$$P_{class}^{(i)} = \frac{\sum_{k=0}^{K-1} p_{class}^{(i,k)}}{K} \tag{2.10}$$

With the filtered map for each class, the last step only consist in a majority voting for each pixel, obtaining the final classification map.

Finally, it is important to mention that in HELICoiD, the parameters are set as $\lambda = 1$ and $K = 40$ [46].

2.4.4. Spatial Filter

The spatial filter is the algorithm included in the alternative processing chain apart from SVM. The idea is similar to the spatial-spectral KNN if λ is set to a huge number. By this way, the distance calculation would be generated based only on the spatial information. In this situation, the K nearest neighbours would be always the K spatial neighbours.

Although the idea is similar, the process using the filter is totally different. In this case, a mask composed of $n \times n$ pixels is applied to every pixel in an image, depending on its type, the operation performed to the image is different. The process is showed in Figure 2.17.

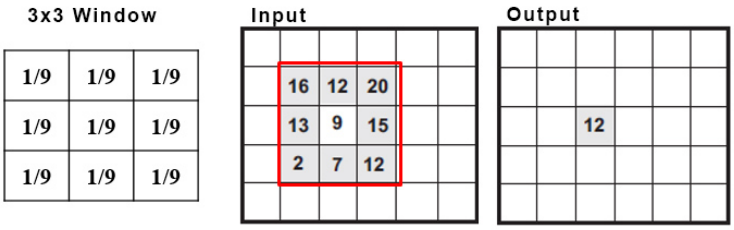


Figure 2.17: Example of a spatial filter with a gaussian mask [53].

In the case of the project, the spatial filter using a Gaussian mask is applied to each one of the four (one per class) initial classification maps generated by the SVM. Finally, the final classification map is generated using a majority voting for each pixel.

2.5. Embedded accelerator platforms

According to Moore's Law, the number of transistors included in a single chip doubles each two years, approximately. However, the well-known prediction started to fail during the previous decade due the impossibility of dissipating levels of heat comparable to nuclear reactors (called end of Dennard scaling). This is shown in Figure 2.18. This fact encouraged manufacturers to include more than a single core per chip.

Nowadays, it is common to use accelerators to increment computing performance. In this section, three different type of accelerators are covered: GPUs, FPGAs and manycore platforms.

GPU

Graphics Processing Units (GPU) are coprocessors born during the 80s with the purpose of easing the main core graphics computations [48]. This aim explains the characteristic architecture of GPUs: high number of simple cores which run instructions in parallel. The

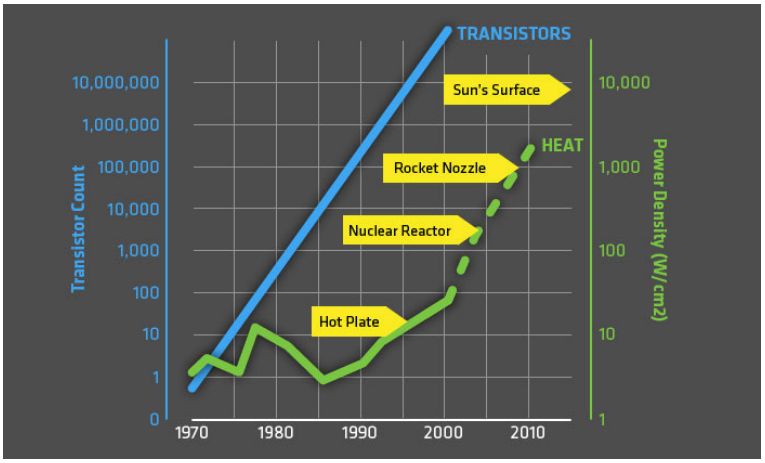


Figure 2.18: Moore's Law and heat comparison. Source: https://news.utexas.edu/sites/news.utexas.edu/files/moores-law-and-heat_700_0.png

main idea is to perform the same operation independently for all the pixels within an image. GPUs features Single Instruction Multiple Data (SIMD) parallelism.

GPUs are composed of several Streaming Multiprocessors (SM) which contain a fixed number of cores and a shared memory (among others features). Further details about these devices are included in chapter 3. In Figure 2.19[3], the NVIDIA Fermi microarchitecture is shown (the concepts included in the image will be explained in Chapter 3).

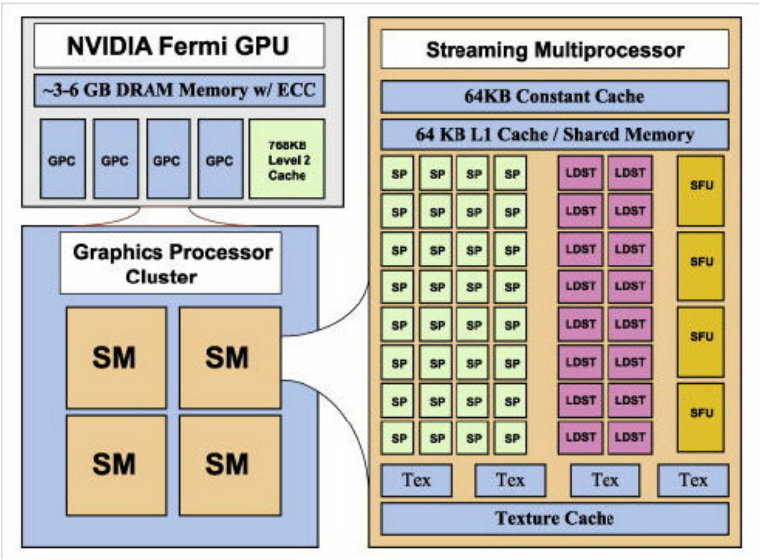


Figure 2.19: NVIDIA Fermi microarchitecture.

[3]Source: https://www.researchgate.net/profile/Vladlen_Skvortsov/publication/261690187/figure/fig10/AS:296870208393219@1447790781347/A-simplified-hardware-block-diagram-for-the-NVIDIA-Fermi-GPU-architecture-22.png

Literature includes many examples using GPUs as accelerators in the field of ML. The work done by A. Ruiz [44] introduces the usage of GPUs to segment neuroblastoma images acquired from modern microscopy scanners. The computational costs of processing this type of images make GPUs obtain speed-up results around 5.6 times compared with GPPs.

GPUs are widely used in Neural Networks, existing several frameworks. One of the most common is Caffe [19], which offers a complete toolkit for training, testing and deploying models efficiently. Caffe can be used to identify objects, to extract semantic features or to detects objects, for example. Real-time constraints are achieved using GPUs.

FPGA

Field Programmable Gate Arrays (FPGA) are devices which include configurable logic blocks and configurable interconnections between them, as shown in Figure 2.20. This way it is possible to create any circuit defining logic blocks and connections configuration. This flexibility and its inherent parallelism allows the FPGA's to be one of the most common accelerators [27].

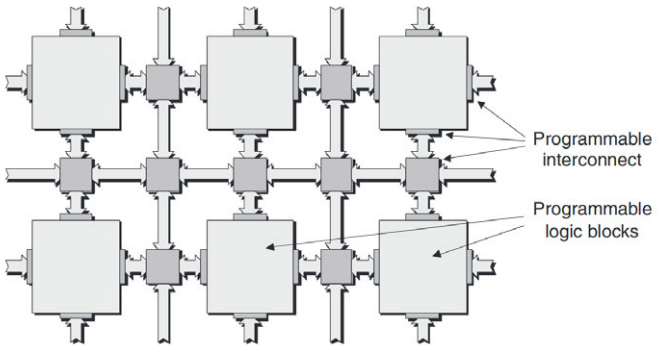


Figure 2.20: Generic FPGA architecture [27].

The basic idea of FPGAs is shown in Figure 2.21, in which the interconnections and logic blocks are represented as well as the input/output connections.

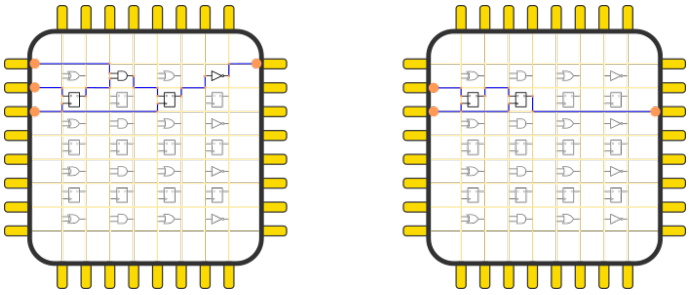


Figure 2.21: Representation of a simple FPGA. Source: https://cdn-images-1.medium.com/max/800/0*pQfEq_OM5iC3Iuqd.png.

Logic blocks are usually implemented as Look-Up Tables (LUTs). These elements are programmable small memories which implement a truth table. This way, it is possible to define the output for each input possibility (without using logic gates). Furthermore, LUTs are defined by the number of bits used as input (the output is one bit). It is common to use 4-input LUTs or 6-input LUTs. For further information refers to Clive Maxfield book [27].

FPGAs are common platforms used to accelerate machine learning applications. O. Boujelben's work [5] implements an automatic detector of asthma using lung sounds. The application employs Mel-Frequency Cepstral Coefficients (MFCC) (to generate characteristic vectors) and SVM (to classify) to differentiate between normal respiratory sounds and respiratory sounds containing wheezes. Both stages are implemented in an Virtex-6 FPGA ML605[4] in order to obtain real-time results.

Another example is the M.A Tahir's work [52], which uses FPGAs to accelerate cancer recognition applications. The work uses KNN classifier to identify breast and prostate cancer from two online data-sets. Comparing classification accuracy, FPGAs achieve same results as microprocessors, however, the performance of FPGAs are highly superior.

Manycore

A manycore platform is a system which includes hundreds or thousands processors in order to accelerate operations. However, these processors are different to the GPUs cores. Each of them is more complex, being able to execute different applications in parallel (not only simple operations), featuring Multiple Instruction Multiple Data (MIMD) parallelism. Figure 2.22 shows the architecture of a manycore platform: the MPPA from Kalray (used in HELICoiD) [18].

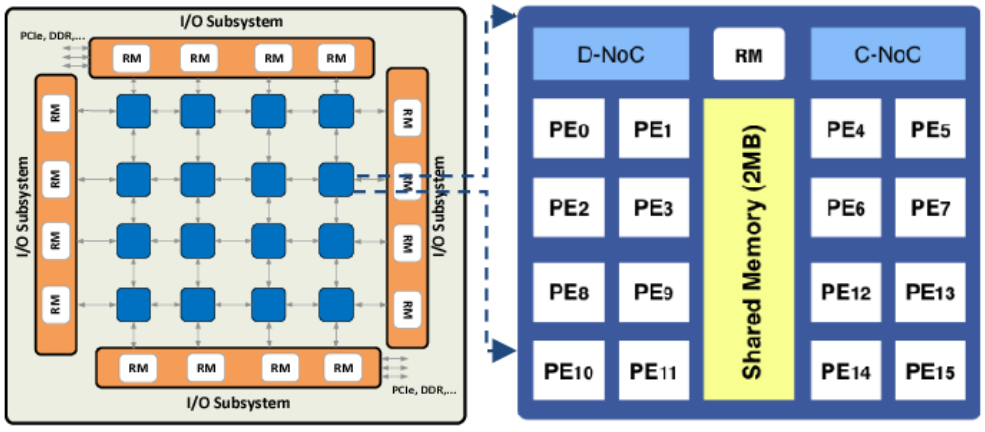


Figure 2.22: Architecture of MPPA from Kalray, a manycore platform [20]. In left, clusters composing the manycore platform, in right, the cluster composition.

[4] https://www.xilinx.com/publications/prod_mktg/ml605_product_brief.pdf

Manycore platforms are composed of several processor clusters and Input/Output (I/O) subsystems, which are in charge of communicating the external memory with clusters. Clusters are divided into several processors, a shared memory and a certain number of Network-on-Chip (NoC). Finally, NoC are used to manage the communication and synchronisation between clusters and I/O subsystems [25].

High Efficiency Video Coding (HEVC) is a video compression standard developed by Moving Pictures Expert Group (MPEG) which takes advantage of parallelism to accelerate the coding/decoding stages. Manycore platforms play an important role accelerating these operations. For example, C. Yan introduces the acceleration of two different sections in HEVC: intra-prediction [56] and parallel deblocking filter [57].

3. NVIDIA Jetson TX1

3.1. Introduction

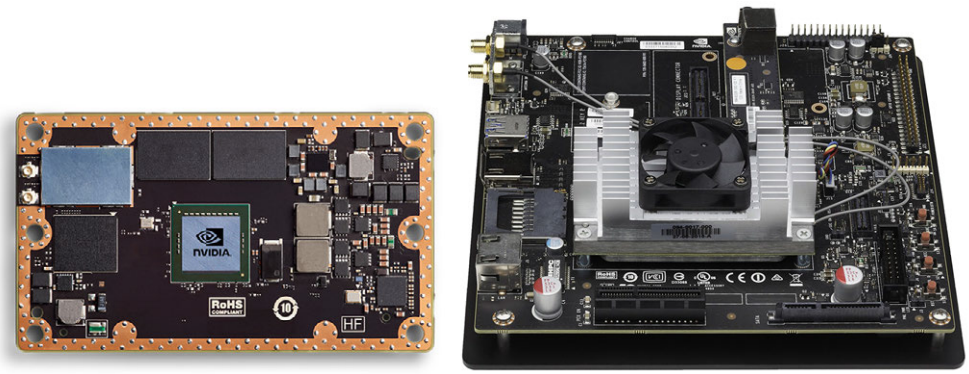
The huge amount of data involved and the time constraints imposed by the application force a parallel implementation approach for this application. The solution adopted in this master thesis is using an heterogeneous accelerator, the NVIDIA Jetson TX1, which contains a System on Chip (SoC) composed of a multicore ARM processor and an embedded GPU.

This chapter includes a detailed description of the board in terms of hardware and processing capabilities, focusing in the microarchitecture used by the GPU. Furthermore, it describes the computing framework Compute Unified Device Architecture (CUDA), some programming best practices, the method used measure execution time and the most common libraries available for CUDA.

3.2. Jetson TX1 Board

The NVIDIA Jetson TX1 System on Module (SoM) is a reduced-size supercomputer designed for GPU computing, computer graphics and artificial intelligence applications with low energy consumption. The SoC called Tegra X1 is composed of a Quad-core ARM Cortex-A57 processor and a NVIDIA Maxwell GPU with 256 CUDA cores, both sharing a principal 4 GB 64 bit LPDDR4 25.6 GB/s memory located in the SoM. Moreover, it includes a 16 GB eMMC 5.1 flash storage and the possibility of connecting a SATA device or an SD card [40].

The SoM, which has the approximate size of a credit card, is shown in Figure 3.1 **(a)** and the developer kit, which is used for application development, in Figure 3.1 **(b)**.



(a) Jetson TX1 Module. **(b)** Jetson TX1 Developer Kit.

Figure 3.1: Jetson TX1.

One of the purposes of the Jetson TX1 is computer vision applications, one of the reasons for the board counting with several interfaces for displaying graphical information such as Digital Serial Interface (DSI), embedded Display Port (eDP), Display Port (DP) and High Definition Multimedia Interface (HDMI). Furthermore, it also includes the interface called Camera Serial Interface (CSI) in order to provide the possibility of using up to six cameras [40].

In addition, the system counts with common interfaces included in embedded systems: Universal Asynchronous Receiver-Transmitter (UART), Serial Peripheral Interface (SPI), Inter Integrated Circuit (I2C), Integrated Interchip Sound (I2S), General Purpose Input Output (GPIO), USB 3.0 + USB 2.0, Ethernet and Bluetooth [40]. They all are all shown in Figure 3.2.

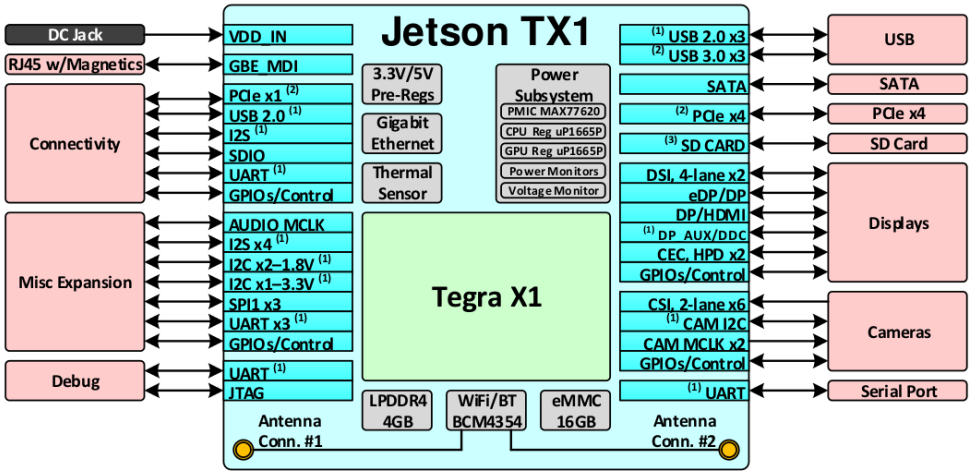


Figure 3.2: Jetson TX1 block diagram [37].

For the software part, there is a tool provided by NVIDIA called Jetson installer Package (JetPack) which is run in a host PC connected to the Jetson Board. JetPack installs an operating system derived from Ubuntu and several libraries (if they are chosen) among which the CUDA toolkit stands out. It can be located both in the host computer and in Jetson and it is necessary for making use of the GPU. Other available libraries provide tools for advanced computer vision, image processing, machine learning and deep learning [36].

Finally, it is important to mention that the compute capability [1] of the platform is 5.3 and the CUDA toolkit used during the project is CUDA 8.0.33.

[1]This number is provided by NVIDIA and it is used to distinguish the compute capabilities of the different GPUs.

3.3. Tegra X1 GPU. Maxwell Microarchitecture

As said before, the SoC included in the Jetson TX1 system is called Tegra X1 and it includes an ARM processor and an NVIDIA GPU. Since this work uses the GPU as the principal component for accelerating the application, it will be the only one explained in detail. To begin with, this section will study what a GPU is and then, the microarchitecture of the GPU included in Jetson TX1, which is called Maxwell.

3.3.1. GPU description

During the 80s, the newly born personal computers began to increment the complexity of their applications and as a consequence, some coprocessors arose to ease their computations. Moreover, this fact boosted the optimisation of these coprocessors for specific applications. This is the origin of some well-known devices in a computer, for instance, the sound card, the network card or the graphics card, also called GPU [48].

As a result, the GPU counts with an specific functioning intended for optimising the operations with graphics, which in most cases means operations with pixels. Taking this into account, it is straight-forward to realise that the architecture paradigm of the GPU is parallel due to the elevated number of pixels processed for each image and the existing independence -in most cases- among them. For this reason, GPUs include from hundreds to thousands of cores working in parallel.

During the last 10–15 years, the use of this device began to go beyond this scope. The concept is called General Purpose on Graphics Processing Units (GPGPU) and, as its name suggests, makes use of the GPU for accelerating non-graphics applications like the one developed in this project.

Finally, it is important to recall that since the GPU was born as a coprocessor, it needs a general purpose processor which launches its operation. The implications of this model will be explained in Section 3.4.

3.3.2. Maxwell Microarchitecture in Tegra X1

In the first place, it is important to define what the microarchitecture is. The term, also called μarch, is referred to the way in which the physical elements of the processor are arranged.

Although NVIDIA released its first GPU in 1999, it was not until 2010 when the Fermi microarchitecture introduced the concept of Graphics Processing Cluster (GPC) and Streaming Multiprocessor (SM) [35], which meant the basis for the GPUs from then

until now. In Figure 3.3, the chronological flowchart of NVIDIA's microarchitectures along with the size of the transistors used are shown.

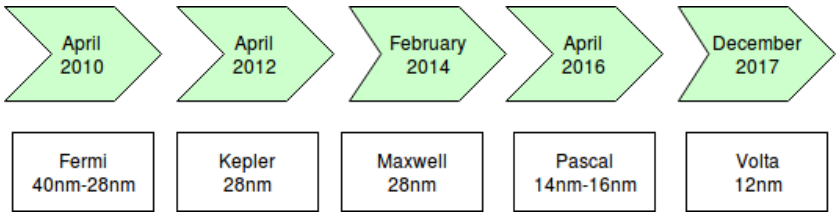


Figure 3.3: NVIDIA microarchitectures chronological flowchart and the size of the transistors in each one.

The first element mentioned, the GPC, stands for the first division of the GPU. This element contains the SMs, which in this microarchitecture are called Streaming Multiprocessor Maxwell (SMM). The GPU also includes the Raster Operation Pipelines (ROPs), a L2 cache and the memory interface. Although the most important elements from the previous ones are the SMMs, the L2 cache and the memory interface are important too and they will be mentioned in the memory hierarchy subsection. The only thing left to introduce are the specific elements in the Tegra X1 SoC as they are not fixed in Maxwell and depends on the specific GPU. The Tegra X1 GPU contains one GPC, two SMMs and 16 ROPs. In addition, the size of the L2 cache is 256 KB [38].

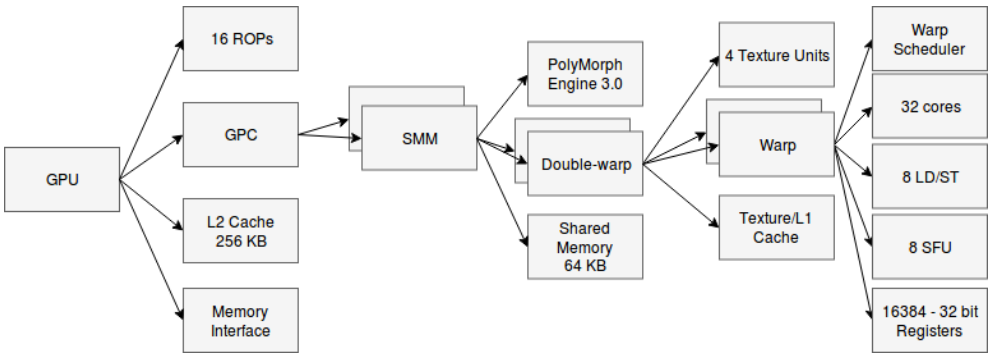


Figure 3.4: Scheme of the Maxwell microarchitecture in Tegra X1.

The SMM is composed of two double-warps, a PolyMorph Engine 3.0 and a shared memory. The most important part to be considered for now is the double-warp, hence, the details are explained in the next paragraph. For the other two components, the PolyMorph Engine 3.0 is used for tessellation in OpenGL [42] and the shared memory included in Tegra X1 has 64 KB (this will be explained in the memory hierarchy section).

The double-warp structure is, at its name suggests, composed of two *warps* (concept explained in the next subsection), but it also includes 4 Texture units and a Texture/L1 Cache memory. Finally, as seen in the scheme of Figure 3.4, each warp contains a warp scheduler, 32 cores, 8 Load Store Units (LD/ST), 8 Special Function Units (SFU) and

16384 registers of 32 bits. The cores contain just one Floating Point Unit (FPU) and an Integer Unit (IU) working at a frequency of 1 GHz.

According to the previous structure, it can be easily deducted that the GPU included in the Tegra X1 SoC contains 256 cores. This is shown in the summary of the GPC and the SMM represented in Figure 3.5.

(a) Graphics Processing Cluster (GPC).

(b) Streaming Multiprocessor Maxwell (SMM).

Figure 3.5: Maxwell microarchitecture in Tegra X1 [38].

From the introduced structure of the microarchitecture, two important concepts are extracted and further analysed: the *warp* and the memory hierarchy. They are addressed below.

Warp concept

The warp includes the necessary processing elements to carry out the computations. Taking into account the 32 cores included in a warp, it is possible to execute 32 instructions in parallel within one warp. However, the architecture features Single Instruction Multiple Data (SIMD) parallelism, which means that the instruction being executed in all the cores of a warp must be the same. Being more accurate, every time a warp is dispatched, the

32 cores execute the instruction issued in parallel even if less operations are required (the not-required data is dismissed). The importance of this concept is explained in Section 3.4.

Memory Hierarchy

This is a key element (probably the most important) to take into account when optimising performance because of the differences between the type of memory within the GPU and especially because of their location (and hence their performances). As seen in Figure 3.5, there exist four different physical memories: the L2 Cache, the Shared Memory, the Texture/L1 Cache and the registers. Furthermore, the memory interface is used to communicate the GPU with the DRAM included in the SoC [28] [29].

Figure 3.5 also specifies their positions: memories located nearer to the cores mean a faster access, less energy usage and logically, a smaller size due to a larger cost. The memory hierarchy is included in Table 3.1 along with the physical location, the type of memory and whether it is cached or not. Moreover, the order of the rows is not fortuitous, the speed of the different memories is sorted from the fastest (register) to the slowest (DRAM).

Table 3.1: Memory hierarchy in Maxwell microarchitecture.

Physical memory	Location	Cached	Type
Registers	Warp	No	R/W
Shared	SMM	No	R/W
Texture/L1 Cache [1]	Double Warp	Yes (L2)	R (cache)
L2 cache	GPU	-	Cache
DRAM (through memory interface)	GPU	Yes (L1/L2)	R/W

[1] The physical memory is the same for texture and L1 cache. While the former is usable explicitly by the user, the latter works as a common cache.

3.4. CUDA C/C++. Programming best practices

This section is divided into five subsections in order to cover all the dimensions of the programming language included in CUDA. In the first one, the way to program the physical elements using the language is explained, with the second one covering the structure of a common CUDA C/C++ program. Third subsection contains several examples of how to optimise the code, the best practices. Fourth includes the method to measure time and the last one several libraries included in CUDA.

3.4.1. Relationship with the microarchitecture

CUDA introduces three basic elements in order to organise the parallelism in the GPU: threads, blocks and grids. For each function executed in the GPU, which is called *kernel*, it is possible to define the number of threads per block and the number of blocks per grid which can be up to three dimensional. Figure 3.6 shows a two-dimensional grid composed of 6 blocks, each one composed of 12 threads arranged in a two-dimensional way. With this grid, it would be possible, for example, to access to every element of a 6×12 matrix using a thread for each and therefore, in parallel [41].

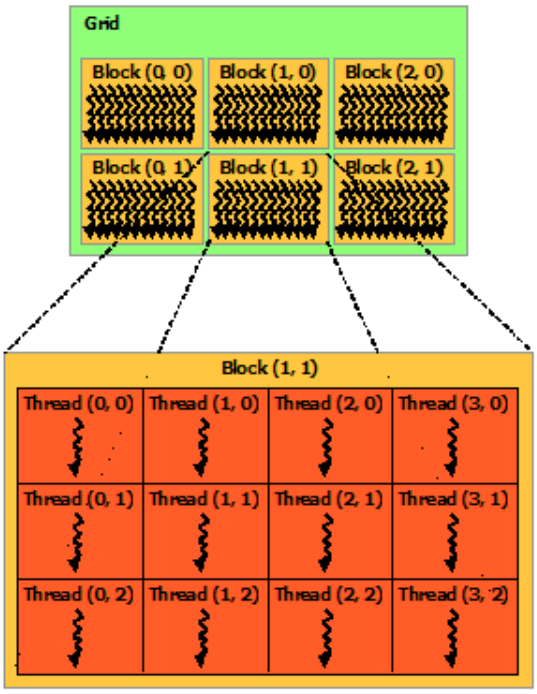


Figure 3.6: Structure of threads, blocks and grids in CUDA C/C++ [41].

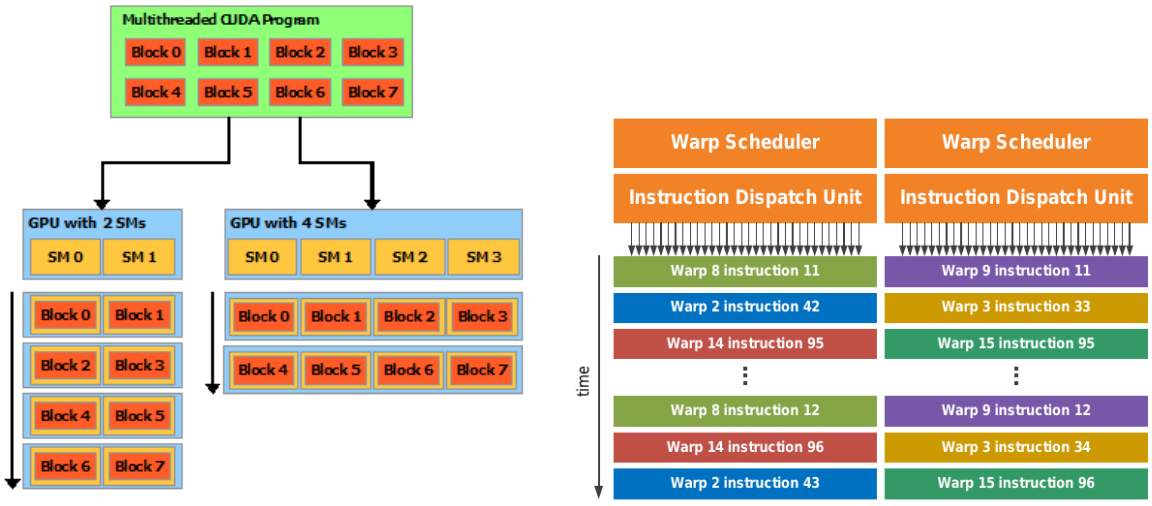
How are these elements related with the structures introduced in the Maxwell microarchitecture? The process that the GPU carries out in order to divide the work using the parameters *blocksPerGrid* and *threadsPerBlock* is the following [41]:

1. An identifier is assigned to each of the blocks (defined by the parameter *blocksPerGrid*) of the grid and to each thread (defined by the parameter *threadsPerBlock*) of each block.

2. Each of the blocks is assigned to an SMM. It allows scalability to the different GPUs, shown in Figure 3.7 **(a)**.

3. The threads within each block are arranged into groups of 32, also called warps. An identifier is assigned to each one.

4. The instructions of each warp are dispatched by the four warp schedulers in each SMM concurrently (the dispatching order is defined by the GPU depending on the available resources, resulting in possible different dispatching orders in each execution). This can be seen in Figure 3.7 **(b)**. The computations are carried out by the cores, the LD/STs or the SFUs.



(a) Block to SM assignment [41].

(b) Warp scheduling in Fermi. Maxwell counts with 4 warp schedulers per SMM [35].

Figure 3.7: Relationship between physical elements and CUDA C/C++ parameters.

Taking the previous assignment into consideration, it is important to update Table 3.1, which focuses on the physical elements. As seen in Table 3.2, the scope is now focused on the concepts used in CUDA C/C++.

Table 3.2: CUDA C/C++ memory scope in Maxwell microarchitecture.

CUDA memory	Accessibility Scope	Type	Note
Register	Thread	R/W	
Shared	Block	R/W	
Texture	Grid	R	Optimised for 2D spatial access pattern
Constant	Grid	R	Optimised when a warp access the same address
Local	Thread	R/W	Used when cores run out of registers. Similar to global memory
Global	Grid	R/W	Accesible even for different kernels

3.4.2. GPU programming model

In section 3.3.1 it was mentioned that the GPU execution needs to be launched by a General Purpose Processor (GPP). This means that the execution of the program is carried out in the GPP and only some parts (interestingly the ones with large computational load) are accelerated using the GPU. This process is shown in Figure 3.8 where a common notation can also be observed: the host is referred to as the GPP and the device as the GPU, both conforming an heterogeneous architecture.

The separation between the two processors has an important implication when programming: the memory for the host and for the device are different. In order to use the GPU, it is necessary to allocate memory in it and then copy the data from the host to the device. Once the information is in the GPU, the kernel can be launched, and, conversely, it is necessary to send the processed data back to the host when the GPU is done.

In Figure 3.9, a simple example of the program needed to run a single kernel is shown. Moreover, in Figure 3.10 the code of the kernel is represented.

A mechanism to identify threads within the grid and blocks is needed. This is shown in line 5 included in Figure 3.10, where each thread is defined by its position within the block and the position of the block within the grid. In the example of Figure 3.10, the identifier i is used so each thread can access to a different memory position in order to calculate the addition of two vectors.

3.4.3. Best practices

This section provides several best practices guidelines when optimising CUDA C/C++ application code. Most of them are derived from the microarchitecture and the aforementioned concepts [33].

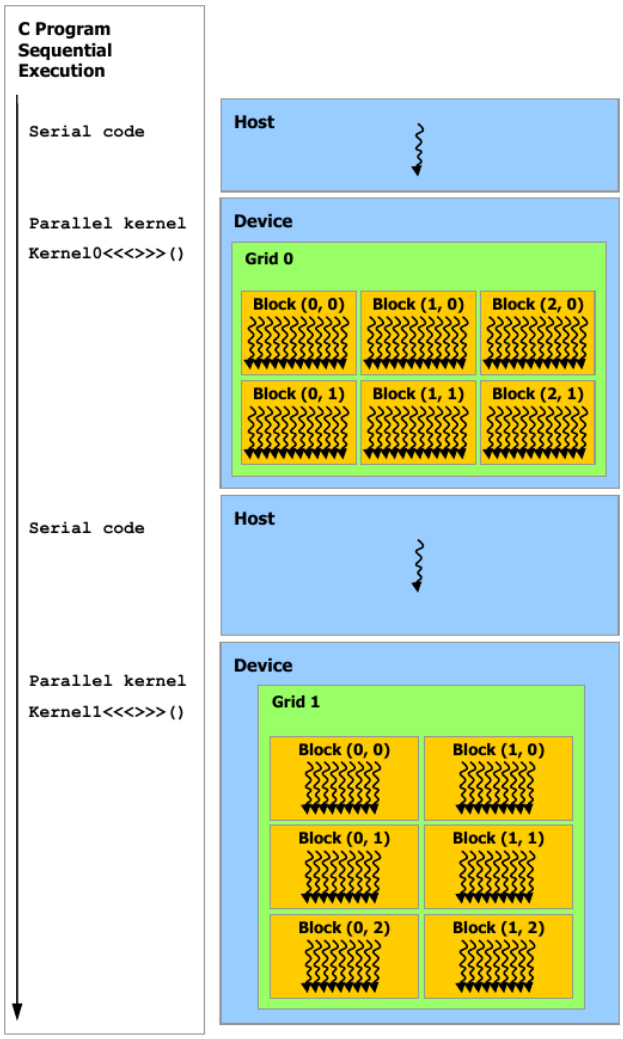


Figure 3.8: GPU programming model [41].

Memory Transfers. Pinned memory & Unified memory

The necessity of transferring the data from the host to the device and backwards adds a new latency issue. However, the property mentioned in Table 3.2 related with the global memory (the global memory is shared between different kernels) can help addressing this problem. The data transferred to the device can exist there as long as the device has enough memory. For this reason, it is recommendable to send the initial information at the beginning and do not send the results back until the end. Thus, the only memory transfer required is the initial data sent and the data processed back (as long as there is enough internal memory in the accelerator).

This can seem obvious when only one kernel is required, but normally, an application is composed of several kernels and the output of one is usually the input for the next

```
1  // Allocate device memory
2  float *A_d, *B_d, *C_d;
3  cudaMalloc((void **)&A_d, size);
4  cudaMalloc((void **)&B_d, size);
5  cudaMalloc((void **)&C_d, size);
6
7  // Copy host memory to device
8  cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
9  cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);
10
11 // Parallelism definition
12 int threadsPerBlock = 256;
13 int blocksPerGrid =
14         (numElements + threadsPerBlock - 1) / threadsPerBlock;
15
16 // Kernel launch
17 vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(A_d, B_d, C_d, numElements);
18
19 // Copy the processed data to host
20 cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);
21
22 // Free the device variables
23 cudaFree(A_d);
24 cudaFree(B_d);
25 cudaFree(C_d);
```

Figure 3.9: Example code for the GPU part.

```
1  __global__ void
2  vectorAdd(const float *A, const float *B, float *C, int numElements)
3  {
4      int i = blockDim.x * blockIdx.x + threadIdx.x;
5      if (i < numElements)
6          C[i] = A[i] + B[i];
7  }
```

Figure 3.10: Kernel used in the 3.9 figure code.

(this is usually the case in embedded, streaming processing systems). In the case of, for example, the PCA algorithm in this project, only the hyperspectral image is required as input data and it counts with more than 7 kernels. Hence, it is possible to transfer the hyperspectral image at the beginning and then, keep all the intermediate data within the GPU until obtaining the final result.

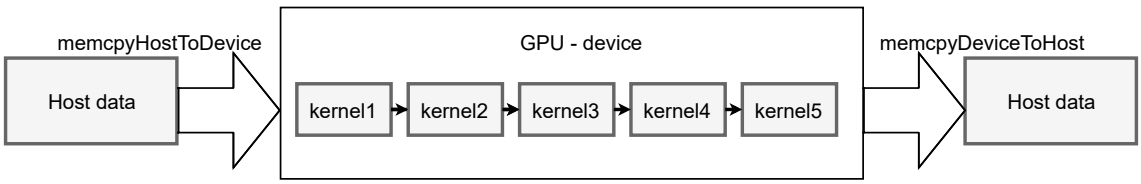


Figure 3.11: Example of the transfer at the beginning and at the end of the application.

Apart from this solution, which also introduces latency at the beginning and at the end, it is possible to use the so-called *pinned memory*. The normal allocation of memory in the host (the standard case in personal computers working under an standard OS) is pageable[2], which is a problem for transfers; for moving pageable memory to the device it is necessary to first allocate the required size in device's memory and then copy the data to a pinned array (this is accomplished by the CUDA driver) to complete the transfer. For this reason, if the memory is directly allocated as pinned from the host, the transfer time can be reduced. This situation is represented in Figure 3.12 [12].

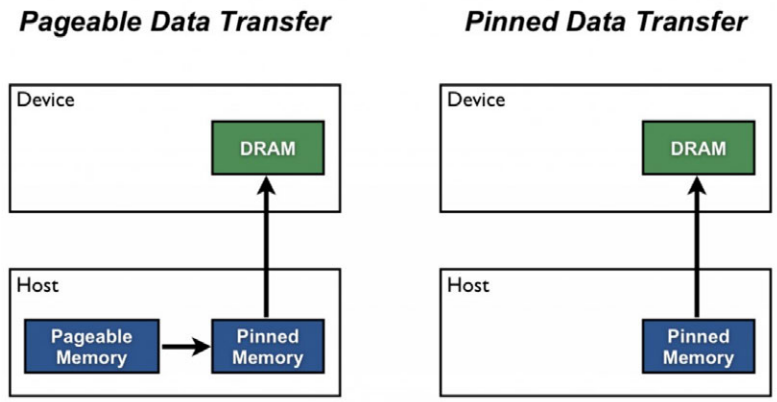


Figure 3.12: Problem with the pageable memory [12].

This leads to the concept of *zero-copy memory*, really useful in devices such as the Jetson TX1, whose main memory is shared between the host and device (as opposed to the case in PCs where the communication between host and device is done using PCI). If the data holding variables are defined as pinned memory, it is possible to just use the same pointer for the host and the device, resulting in the transfer being completely eliminated. Nevertheless, the use of too much pinned memory can be problematic for the system since it can run out of memory (if all the memory were allocated in pinned memory, the system would have problems to run the operating system). In addition, it is important to know that by using this method, the L1/L2 caches are bypassed, losing all the benefits that a cache structure can offer (fast access to repeated data reads) [41].

For sake of completeness, this section includes the concept of *unified memory* too. However, this is not considered as an optimisation. This term refers to the possibility of creating a variable that can be used both in host and device. In contrast with the zero-copy memory, this variable is defined in the host and, internally, the CUDA driver performs the transfer to the device. The advantage of this type of memory regarding the normal one (a pointer in the host and a different pointer in device) is that only one pointer

[2]This type of memory is divided into pages that may be moved to the disk in order to free the main memory.

is needed; even so, the transfer is not avoided. The caches in this case are enabled [15].

Memory in the GPU

One of the most important best practices consists in using the appropriate memory in each situation. As seen in Table 3.2, the different types of memory provide different access speeds, being the difference in some cases of up to two orders of magnitude. Most of the latency issues in GPU applications are due to external memory accesses (in general, in computing applications).

In general, all the variables used locally in threads must be declared as register memory. However, if the limit of registers (defined by the total number of registers in the platform and the number of threads used) per thread is reached, the variables will be allocated in the local memory (the same physical memory as the global memory and with the same properties), reducing the processing speed drastically due to slower memory accesses. This problem can also occur when defining arrays within a thread, so this should be avoided whenever possible.

Shared memory provides an efficient way to communicate data between threads inside a block. The speed is considerably faster than using global memory and in some applications, just communication among threads within a block is all that is needed.

Global memory is the slowest in most computing systems, but it is a common necessity of all threads from a grid to access to global variables to retrieve input data. As a consequence, this memory is widely used. Accesses to it have to be greatly controlled and optimised, which usually involves some changes to how an algorithm is specified or coded.

Global memory provides the functionality of the two caches located into the GPU (L1/L2) in order to accelerate the unpredictable repeated memory accesses as well as a method to accelerate the known ones with the constant memory. This memory can improve the speed when all the threads within a warp accesses to the same address.

Finally, the texture memory is optimised for two-dimensional accesses and as long as it is cached, the speed improvement compared with the global memory is large [28] [29].

Global memory. Coalesced & Aligned Accesses

The access to global memory is the slowest one, as seen before. Nevertheless, it is possible to optimise it following two basic rules: the accesses must be coalesced and aligned. A normal access to the global memory is cached through the L1 cache, which has 128 byte-lines, hence, all the accesses move 128 aligned bytes from the global memory to the cache, even if only one thread requests a single byte.

For example, Figure 3.13 shows how a warp is accessing to memory addresses from 96 to 224. This operation must be done in two transactions due to the misalignment: the first one accesses the memory positions between 0 and 127 and the second one between

128 and 255. If the access to memory was from positions 128 to 255, only one transaction would be needed and the same amount of data would be moved. Obviously, incrementing the number of transactions decrease performance [33] [13]. This problem could be solved in some situations arranging the data so that each warp accesses only one time to the global memory.

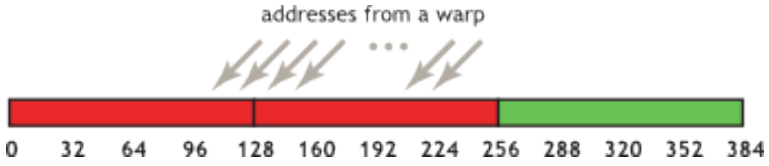


Figure 3.13: Unaligned global memory access [33].

The previous example introduces the misalignment issue. Nevertheless, it is easy to think about the problem produced by a non coalesced access pattern, i.e, a memory access to two or more separated positions when only one could do it. In the worst case, it would be possible to access to memory positions separated 128 bytes. By doing this, the number of transactions would be the same number as the number of words required, drastically reducing performance [33] [13]. This example is shown in Figure 3.14.

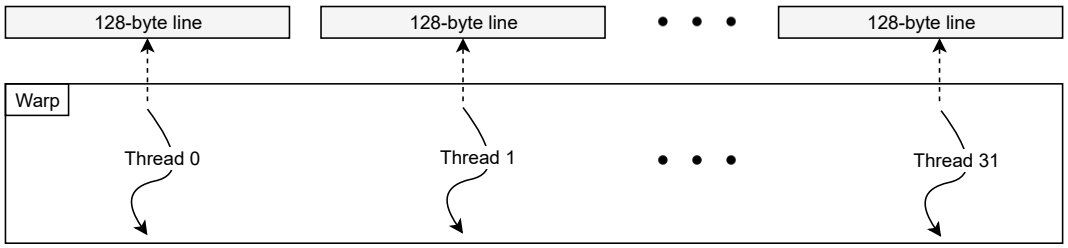


Figure 3.14: Worst case in the non-coalescing global memory access.

As opposite, Figure 3.15 shows a coalesced access. All threads within a warp are accessing to the same 128-byte line so only one transaction is needed.

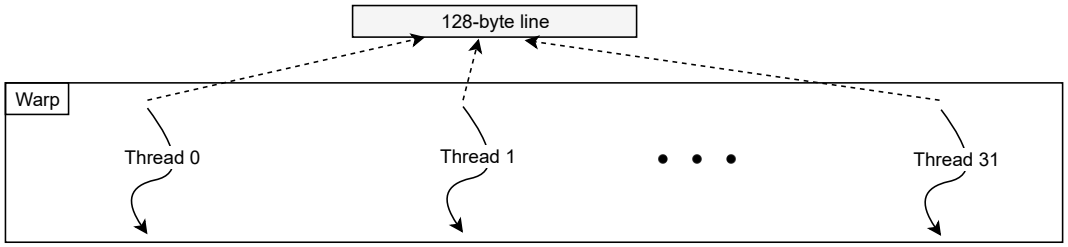


Figure 3.15: Coalescing global memory accesses.

Shared memory. Bank conflicts

Access to shared memory is significantly faster than access to global memory. Because of this, its use is recommended when the threads within a block only need to communicate among them or when data accesses to global memory are repeated many times. However, this memory might introduce a latency issue due to its physical design.

Shared memory is divided into 32 memory banks so that 32-bit words are allocated to successive banks. This reduced number of banks causes that sometimes, several threads within a warp try to access the same bank (but different address). In this case, the bank cannot return the requested words concurrently so it serialises the accesses [41].

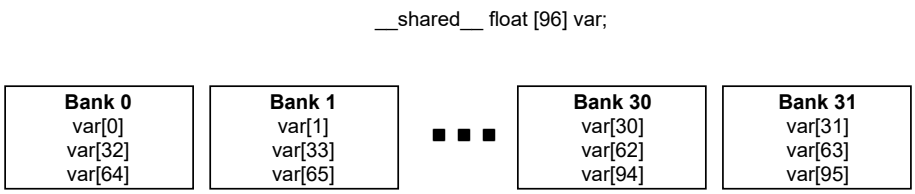


Figure 3.16: Allocation of 96 float words (32 bits) in shared memory.

In Figure 3.16 the allocation of a shared variable is represented. In this example, each bank stores three different 32 bit words. If a warp tried to access more than one address in a bank, a bank conflict would appear, causing the previously mentioned serialisation.

Restrict & Const keyword

These two keywords can be applied to pointers in order to improve the code generation.

On one hand, the `restrict` keyword is used to inform the compiler that the pointer is not aliased, i.e., the pointer does not share memory with other restricted pointers. As a consequence, the compiler can optimize code generation by eliminating the aliasing check. Actually, this is not only an optimisation in CUDA C/C++ programming since it is allowed in, for instance C/C++ or OpenCL.

On the other hand, the `const` keyword increments the probability of the compiler for defining a pointer as constant (without the keyword the compiler is capable of doing it, but it can fail). If the compiler detects a pointer as constant, the memory is stored in constant memory, improving access speed [41] [2].

```
1 __global__ void kernel(const float* __restrict__ a,
2  const float* __restrict__ b, float* c){ ... }
```

Figure 3.17: Example of kernel using the keywords *const* and *restrict*.

Pragma unroll

This is another directive used to improve code generation by the compiler. In this case, the loops are unrolled in order to not use the iteration variable, the comparison and the addition inherent to the loop. This way, the number of instructions is reduced at compile time, thus, improving the run time. Figure 3.18 features an example code showing this situation.

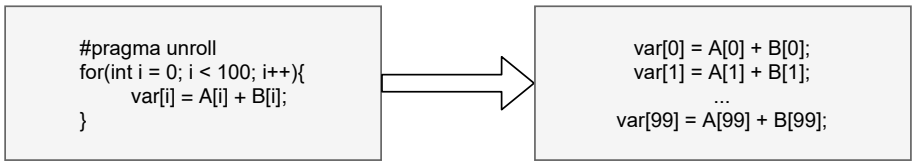


Figure 3.18: Conversion made by the compiler when the pragma unroll directive is written.

Race conditions. Atomics & Reduction

As in every system using parallel programming, race conditions are present. This concept refers to the problem that appears when a thread tries to read a memory position while others are writing it. For example, some basic operations with vectors like *sum*, *minimum* or *maximum* are affected by this issue.

In Figure 3.19, the serial code for a sum function is represented. In line 4 it can be observed that the `sum` variable is changed every iteration by reading the previous value and then writing the accumulation of this value with the current value of the array. The equivalent code in the CUDA C/C++ programming could be like the one shown in Figure 3.20.

```
1 float sum(float* A, int size){
2         float sum = 0.0;
3         for(int i = 0; i < size;i++){
4                 sum += A[i];
5         }
6         return sum;
7 }
```

Figure 3.19: Example of a serial code in C for a sum function.

```
1 __global__ void sum(float* A, float *sum){
2         int i = blockDim.x * blockIdx.x + threadIdx.x;
3         sum[0] += A[i];
4 }
```

Figure 3.20: Example of a parallel code in CUDA C/C++ for a sum function (incorrect).

However, in this example, race conditions appear because there are threads reading the value of `sum[0]` at the same time other threads are writing it.

A solution for this problem could be the atomic operations provided by NVIDIA. These operations assure that in the moment a thread is writing a value, there are no threads reading it. Although the problem is solved because the final results are correct, this solution is quite slow due to the serialization of memory accesses. The performance obtained with atomic operations is comparable to a serial code.

```
1 __global__ void sum(float* A, float *sum){
2         int i = blockDim.x * blockIdx.x + threadIdx.x;
3         atomicAdd(&sum[0], A[i]);
4 }
```

Figure 3.21: Example of a parallel code in CUDA C/C++ for a sum function using atomic functions.

Another widely adopted solution is the use of *reductions*. The problem of race conditions is solved by using the aforementioned memory hierarchy. The basic reduction takes advantage of the shared memory structure in order to perform the required operation in each of the blocks. When the block reductions are performed, the operation is done between them.

But how is the operation done within the blocks? As a guiding example, the sum reduction example is used. The idea is to divide the block by two n times, so that in every iteration, threads just change one value of the block.

Figure 3.22 shows a block composed of 16 threads that is reduced in $log_2 16 = 4$ iterations. For each iteration, the upper half threads add their corresponding value to the lower half thread's values, reducing the problem half the original size and avoiding the race conditions (only an addition between two values is performed in each thread). At the end, the first thread of each block will have the value of the sum. The last step is to add the values in all the blocks, which can be carried out serially (because the number of blocks is normally enough) or with another reduction. The equivalent code is shown in Figure 3.23 [10].

As said before, this is the simplest reduction approach, although it is possible to optimise more this kind of operations. Further details about the use of reductions not using blocks but warps are introduced by Justin Luitjens [24].

Warp divergence

This issue is directly related with the *warp concept* introduced in subsection 3.3.2. As explained there, all threads within a warp execute the same instruction but for different memory accesses. The divergence case is represented in Figure 3.24.

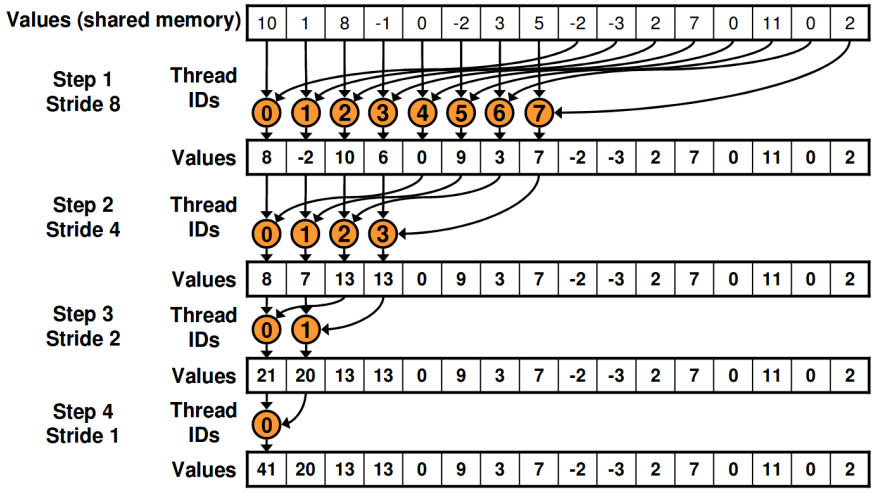


Figure 3.22: Example of reduction within a block [10].

```
__global__ void sum(float *A, float *sum, int nBlock){
        __shared__ float sdata[1024];
        int i = threadIdx.x + blockDim.x *blockIdx.x;
        int tid = threadIdx.x;
        if(tid < nBlock)
                sdata[tid] = A[i];
        else
                sdata[tid] = 0.0;
        __syncthreads();
        for(int s = blockDim.x / 2; s > 0; s = s/2){
                if(tid < s)
                        sdata[tid] += sdata[tid + s];
        __syncthreads();
        }
        if(tid == 0)
                sum[blockIdx.x] = sdata[tid];
}
```

Figure 3.23: Example of a parallel code in CUDA C/C++ for a sum function using reductions.

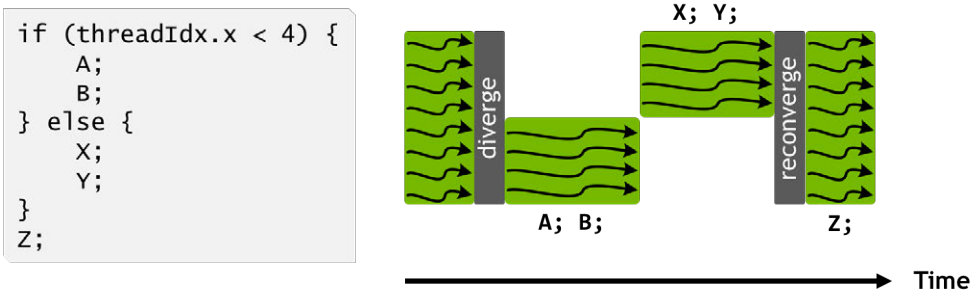


Figure 3.24: Warp divergence. Source: https://devblogs.nvidia.com/inside-volta/

It can be observed that all the threads within the warp wait for the rest to finish its branch execution. The figure is not exactly accurate because what the warp is actually doing is executing the two branches for all the threads. At the end, in the reconverge phase, each thread dismisses the results from its non-active branch.

As a consequence, it is important to take into account that for optimisation purposes, it is necessary to eliminate the divergence within warps as much as possible[3].

Streams

The fact that host and device are different processors, offers the possibility of using them concurrently. CUDA C/C++ leverages on this heterogenity, enabling the host launching kernels that run in the device while the host continues its execution, which means that kernel execution is asynchronous.

In contrast, memory transfers between host and device are synchronous, forcing the main thread in host to wait until the transfer is completed. This is so because the data transfer is presumably needed before continuing the execution in host or in device. It is shown in Figure 3.25. However, this is not true in every occasion, so a new concept is required, *streams* [14].

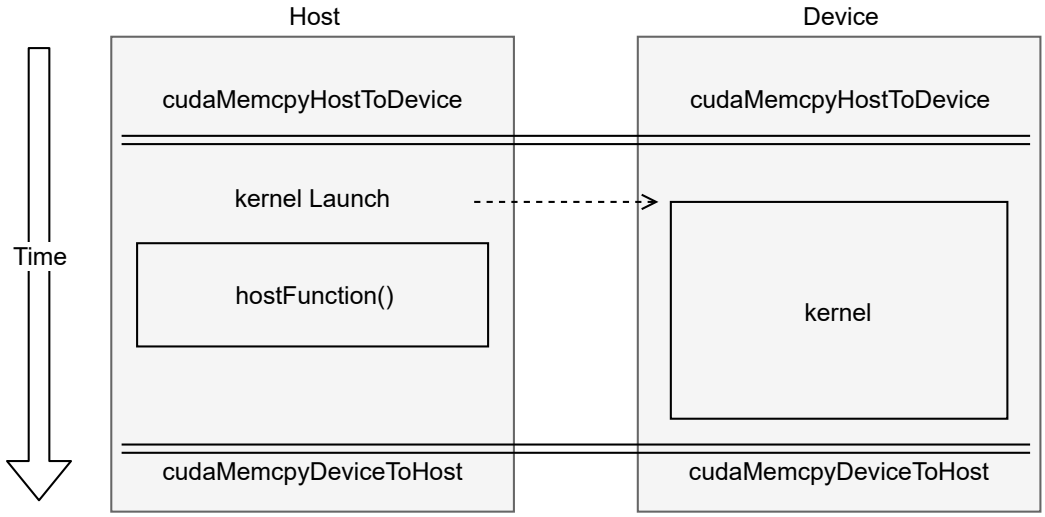


Figure 3.25: Representation of the default synchronization in CUDA C/C++. The double lines represent the synchronization points between host and device (in memory transfers). In addition, it can be observed how the host function and the kernel run in parallel.

With the use of streams, it is possible to define data transfers as asynchronous so that the host only launches transferences and continues with the main thread instead of waiting for transfers completion. Using this method it is possible to overlap memory

[3]This does not mean that divergence reduces the performance in all cases. Imagine a grid with one block of 128 threads: there are 4 warps of 32 threads. If the kernel used a switch structure with 4 cases and each one includes 32 aligned threads, the divergence would be null.

transfers and kernel executions in order to increase parallelism. This model can be observed in Figure 3.26. Boxes with different colours represent different kernels, all of which are decomposed into 3 equally-colored boxes: host-to-device (H2D) and device-to-host (D2H) data transfers, and kernel execution itself (numbered boxes). The kernel execution is happening at the same moment as the memory transfer [14].

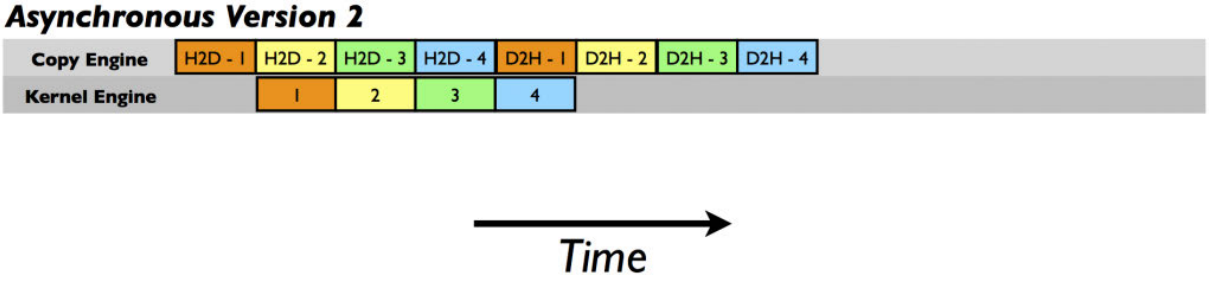


Figure 3.26: Representation of the parallelism achieved with streams regarding memory transfers and kernel executions [14].

As seen, a kind of transfer-execution pipeline is created so once $H2D_i$ ($kernel_i$ H2D transfer) is done, $H2D_{i+1}$ can start, overlapping with $kernel_i$ execution. The same idea holds for $D2H_i$ transfers once $kernel_i$ execution is done.

3.4.4. Time measurement

Since the project aims to optimise an application and characterise it in terms of throughput and energy, it is important to define a proper measurement method. This subsection is focused on the former element while the latter is widely explained in Chapter 6. The reason is that CUDA C/C++ provide tools so that time can be measured easily, while the energy measurements are not that straight forward to measure. Therefore, these will be made once the application is optimised in time, for which the method to measure time is analysed before tackling optimisations.

CUDA includes all the libraries from C and C++, hence, it is possible to use their time measurement methods taking care of the asynchronous functions from the GPU. However, CUDA provide its own method to measure CPU/GPU times based on events using the CUDA event API [11]. Events allow the programmer to not use synchronization barriers to measure time so the execution time is not affected by metrics measurement. A code example is shown in Figure 3.27.

For that reason, the event approach is used in this project in order to measure time. For verification purposes, events are compared with the function `gettimeofday`[4](synchronising host and device), obtaining the same results.

[4]This function makes use of the system clock to set two time stamps and compare them to obtain the time passed with microseconds precision.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);

kernel1<<<blocksPerGrid, threadsPerBlock>>>();
...
kernelN<<<blocksPerGrid, threadsPerBlock>>>();

cudaEventRecord(stop);
cudaEventSynchronize(stop);

float timeEvent = 0;
cudaEventElapsedTime(&timeEvent, start, stop);
```

Figure 3.27: Example of CUDA events to measure time.

CUDA events offer many options. Using *Visual Profiler* or *nvprof* (the command line version of Visual Profiler) [34] it is possible to obtain a timeline of CUDA activities during the execution of an application. It includes kernel executions, data transfers and CUDA API calls. Each element in timeline is defined by the time event, however, other events such as the number of instructions executed per cycle or the number of global memory transactions (among many others) can be measured. This way, the application can be fully analysed.

3.4.5. Libraries

In addition to the programming language, NVIDIA also offers a high number of libraries in order to perform different operations easily. These ones are centred in deep learning, linear algebra, mathematics, signal, image and video processing, and parallel algorithms. The following list shows their focus [32]:

- cuDNN: deep neural networks.
- cuBLAS: standard basic linear algebra subroutines.
- CUDA Math Library: mathematical functions.
- cuSPARSE: basic linear algebra subroutines for sparse matrices.
- cuRAND: random number generator.
- cuSOLVER: dense and sparse direct solvers.
- cuFFT: Fast Fourier Transform.
- nvGRAPH: parallel algorithms for high performance analytics on graphs.

- Thrust: parallel algorithms and data structures.

Although in this project only cuBLAS and Thrust are explored, it is a good idea to have these libraries in mind when programming in CUDA C/C++ to leverage on them for optimisation purposes or for reducing design time.

4. Algorithm implementation

4.1. Introduction

This chapter includes the implementation of the four algorithms studied in this work. In the first place, the three acceleration of the algorithms from the original HELICoiD project (PCA, SVM and KNN) is explained. Then, the spatial filter implementation from the alternative chain is described.

Since the PCA and the spatial filter implementation is part of this work, they are widely explained as opposed to the SVM and KNN, where a light description is included, referring the reader to the original works where they are further described. In addition, the proposed kernel codes for both algorithms are included in Appendix A and Appendix B, respectively.

4.2. PCA

This section includes a detailed description of the acceleration of the PCA algorithm. For this a, combination of the algorithm described in Chapter 2 with the platform insights from Chapter 3 is needed.

In order to implement the PCA algorithm, two alternatives (explained in Chapter 2) are found: the Jacobi chain and the NIPALS chain. While the former is composed of the pre-processing step, the covariance matrix calculation, the Jacobi method and the projection, the latter only uses the pre-processing step and the NIPALS method. This is depicted in Figure 4.1. The upper branch shows the Jacobi chain and the lower the NIPALS chain. It is important to note that they are two different methods (not complementary) to obtain the same result, the principal component image.

Consequently, this section is divided first into the pre-processing part and then into the two different ways to perform the PCA. Within the sections, a detailed explanation of the process along with different versions of the kernels implemented is included.

An important observation regarding the PCA acceleration is memory transfers. As seen in the best practices section, there are a few methods for transferring data. In this case, the HS image is sent to the device at the beginning of execution and the results are sent back to the host at the end. All the intermediate results are stored within the GPU. The zerol-copy option is not used due to the bypassing caches, which worsen the results heavily.

4.2.1. PCA Data Pre-processing

Before going into the pre-processing details, it is important to know the file format used to store HS images. These are binary files composed of real-valued numbers between 0

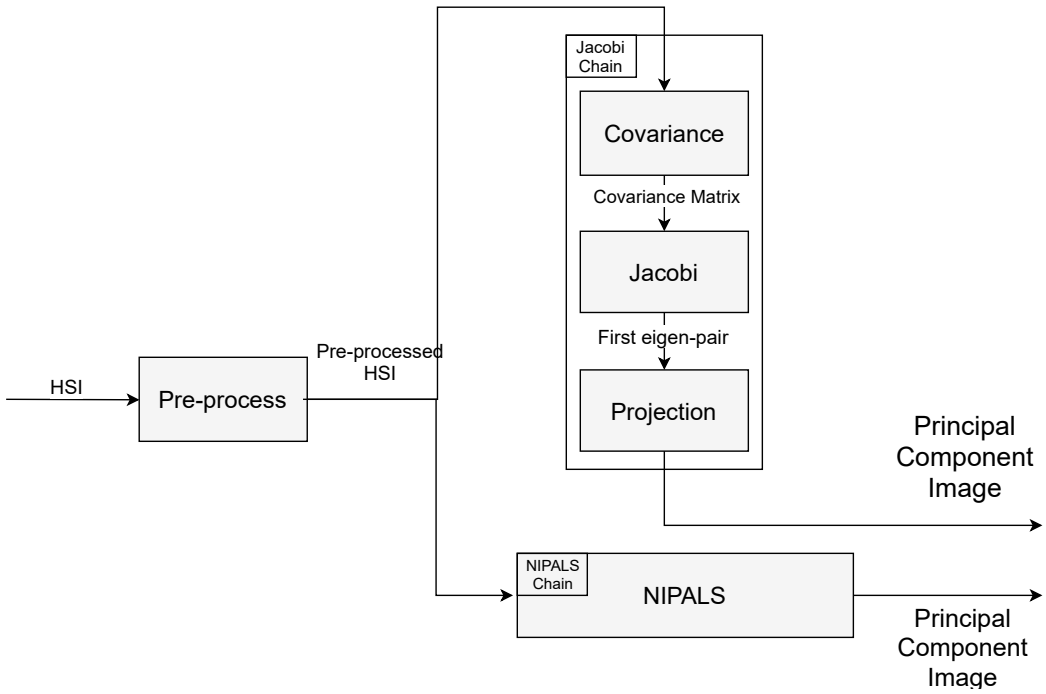


Figure 4.1: PCA diagram. The upper branch represents the Jacobi chain and the lower branch the NIPALS chain. Only one branch can be used to perform the PCA.

and 1 encoded using 32-bit floats (*float32*). Taking into account that the memory is one dimensional, the three-dimensionality of the hyperspectral data is represented as follows: images are stored band by band and the images are stored row by row. This is depicted in Figure 4.2.

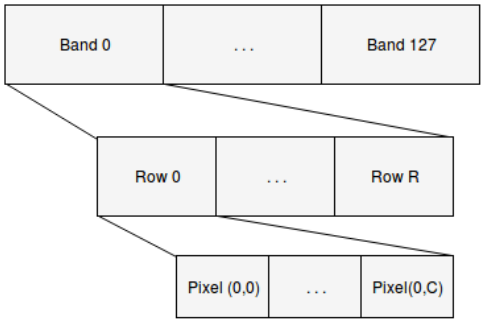


Figure 4.2: Hyperspectral image in memory with 128 bands, R rows and C columns.

As said before, the pre-processing stage is required for subtracting the average of every feature, i.e, the average of each spectral band. Therefore, the average of each band needs to be calculated first and then, subtracted from to all the pixels band by band. The process is depicted in Figure 4.3.

As a consequence, this part is divided into two different kernels, one to compute the

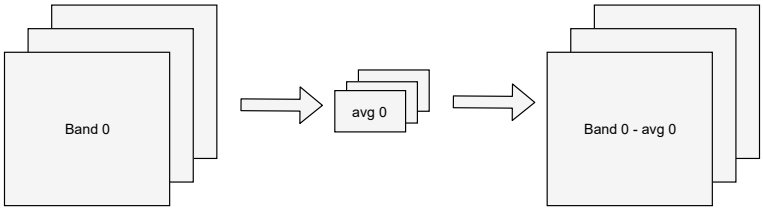


Figure 4.3: Pre-process scheme.

average per band and the other for subtracting these averages from each band. However, the former is much more difficult to program than the latter because it involves a reduction per band.

The first approach proposed for this reduction is a naive average per band using the GPU. The idea is getting advantage of the GPU in order to calculate the average in each one of the bands in parallel. This results in a grid composed of only 128 threads in which each one does $rows \times columns$ summations serially (kernel shown in Figure A.1). As it can be thought, this approach yields a low level of parallelism which could be improved using another implementation.

Nevertheless, this is a problem in which race conditions appear. This problem has two solutions as analysed in Chapter 2: atomic operations or reductions. With the first idea it is possible to use a thread per pixel ($rows \times columns \times bands$ pixels) but as mentioned, atomic operations serialise operations producing a low optimised solution (kernel shown in Figure A.2). For this reason, the reduction seems to be the proper solution.

The problem is that in fact, not only a reduction of the entire vector is required, but also a reduction per band, i.e., 128 different band reductions. As a consequence, the approach is slightly different. The solution adopted uses a block per spectral band in order to make a reduction per block. These blocks are defined with the maximum number of threads possible for the Maxwell architecture: 1024. Figure 4.4 shows an example reduction of a single band composed of 4096 pixels.

The operation is divided into other two: on one hand, all the threads within a block reduce the image from $rows \times columns$ to 1024 serially in $\frac{imSize}{1024} - 1$ iterations; on the other hand, the 1024 resultant values are reduced using a standard reduction to a single value that is scaled with the averaging factor. In addition, the second part can be carried out using the shared memory or the shuffle warp operations (kernels shown in Figure A.3 and Figure A.4).

With the average values calculated, it is only missing to subtract them from the proper band. This is a simple task, performed using a kernel with the same number of threads as pixels in the hyperspectral image (kernel shown in Figure A.5) in which each of them subtracts the average independently.

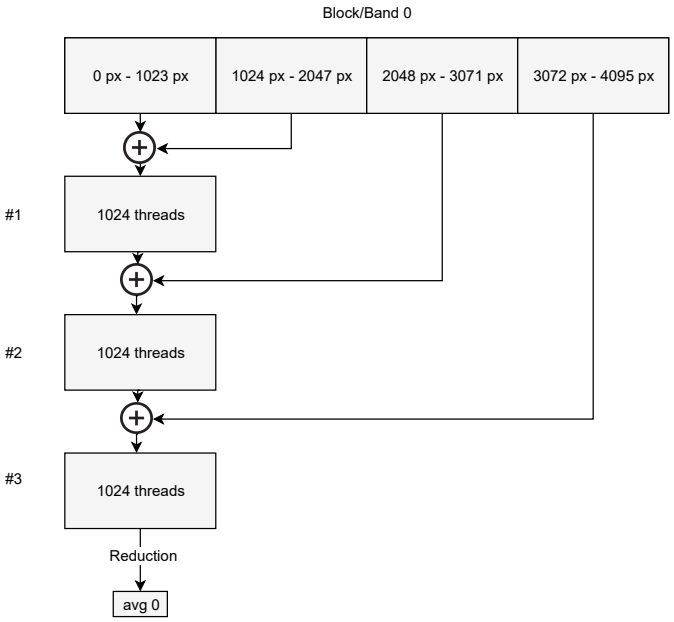


Figure 4.4: Reduction scheme used for calculating the average band by band. This figure represents the reduction of one band composed of 4096 pixels.

4.2.2. Jacobi chain

This is the first algorithm introduced for computing the PCA process. It involves the creation of the covariance matrix using the preprocessed data, the calculation of all the eigenvalues and eigenvectors and the projection of the first component onto the initial data. In this case, the eigenvalues and eigenvectors calculation is carried out using the Jacobi method [49], an iterative algorithm which diagonalises the covariance matrix in an approximate way. Figure 4.5 shows the process.

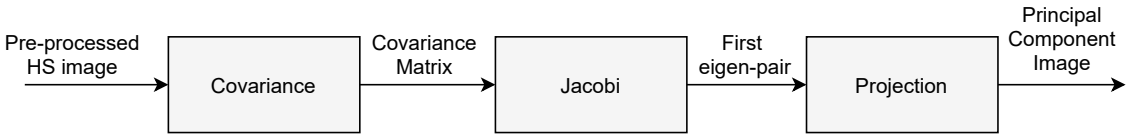


Figure 4.5: Jacobi chain process.

Covariance matrix

Theoretically, a covariance matrix is composed of (i, j) elements representing the covariance value between a variance vector i and j. As a consequence, for its creation, it is necessary to multiply each variance vector with all the others including itself. Nevertheless, in the case of this project, the covariance matrix is used to know the covariance value between the bands. Using the same notation as in Chapter 2, the matrix can be denoted as:

$$\Sigma = \begin{pmatrix} x_1^{'(i)} \cdot x_1^{'(i)} & \cdots & x_1^{'(i)} \cdot x_n^{'(i)} \\ \vdots & \ddots & \vdots \\ x_n^{'(i)} \cdot x_1^{'(i)} & \cdots & x_n^{'(i)} \cdot x_n^{'(i)} \end{pmatrix} \tag{4.1}$$

Taking into account that every band is composed of m pixels, the previous matrix can be expressed as a matrix multiplication in order to explain the operations (V denotes the matrix composed of the vectors without average as rows from the previous step):

$$\Sigma' = \begin{pmatrix} x_1^{'(1)} & \cdots & x_1^{'(m)} \\ \vdots & \ddots & \vdots \\ x_n^{'(1)} & \cdots & x_n^{'(m)} \end{pmatrix} \cdot \begin{pmatrix} x_1^{'(1)} & \cdots & x_n^{'(1)} \\ \vdots & \ddots & \vdots \\ x_1^{'(m)} & \cdots & x_n^{'(m)} \end{pmatrix} = V^t \cdot V \tag{4.2}$$

Having a look to the calculations in equation 4.2, it can be extracted that the covariance matrix is symmetric and also that its dimensions are $n \times n = 128 \times 128$ for all the HELICoiD matrices. In addition, in order to obtain this matrix it is necessary to do 128×128 vector multiplications (each vector composed of m pixels). This operation is a bottleneck in sequential platforms, hence, GPU's are likely to obtain great acceleration.

Beginning with the proposed solution, it is important to recall the fact that the hyperspectral image with zero average is stored in the same way as the original hyperspectral image: bands by bands. This corresponds not with V but with V^t so as a consequence, the first step needed is transposing V^t in order to perform the multiplication.

The transposition kernel uses the same number of threads as pixels so that each of them accesses an element of the HS image (without average). The only operation needed then is the storage of the accessed element in a different array in which all the bands of a pixel are placed together. In this case, the first 128 values are the 128 bands of the pixel $(0, 0)$ and so on. The code simplicity of this kernel does not imply bad results (the kernel is shown in Figure A.6), however, it is possible to optimise the memory accesses using the shared memory. Nevertheless, in this work, this is not done because a better optimisation is made by completely removing the matrix transposition.

Before continuing with the removal of the transposition kernel, the matrix multiplication is introduced. This is a well-known problem that benefits from a GPU implementation.

The main problem with this operation is the large number of elements accessed and also the need for accumulating multiplied elements for each result element (which generates race conditions as already analysed). For this reason and taking advantage of the fact that every element is used several times for the calculations, the usage of shared memory and registers enables a good solution through the so-called tile approach (kernel shown in Figure A.7). The steps to perform a tile-based matrix multiplication, shown in Figure 4.6

are as follows:

1. Divide the result matrix into square blocks associating their threads with matrix elements. In the example, 4 blocks with 4 threads each one.

2. For a block (k, n) in C, fill a shared block $(k, 1)$ from matrix A and a shared block $(1, n)$ from matrix B. In the example shown, focusing on block $C_{1,1}$, block $A_{1,1}$ and block $B_{1,1}$ are moved to shared memory.

3. With the blocks in shared memory, each thread calculates the part of the multiplication for its element (i, j) taking i row from matrix A and column j from matrix B. Finally, they save the calculation to a register. In the example, the thread (1,1) is calculating $sum_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21}$, the thread (1,2) $sum_{12} = a_{11} \cdot b_{12} + a_{12} \cdot b_{22}$, etc.

4. Steps 2 and 3 are repeated changing the blocks moved to the shared memory iteratively. While A blocks move horizontally to the right, B blocks move vertically to the bottom. The calculations are added to the ones from the previous iterations for each thread. In the example, the blocks of the second iteration are represented with a darker gray ($A_{1,2}$ and $B_{2,1}$). Focusing in the thread $(1, 1)$ again, it is calculating $sum_{11} = sum_{11} + a_{13} \cdot b_{31} + a_{14} \cdot b_{41}$

5. The result for each element, sum_{ij}, corresponds with the elements of matrix C. To conclude, these values are moved to the global memory.

$$\begin{pmatrix} \overbrace{a_{11} & a_{12}}^{A_{1,1}} & \overbrace{a_{13} & a_{14}}^{A_{1,2}} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \cdot \begin{pmatrix} \overbrace{b_{11} & b_{12}}^{B_{1,1}} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ \underbrace{b_{41} & b_{42}}_{B_{2,1}} & b_{43} & b_{44} \end{pmatrix} = \begin{pmatrix} \overbrace{c_{11} & c_{12}}^{C_{1,1}} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{pmatrix}$$

Figure 4.6: Matrix tile multiplication $A \cdot B = C$. The rectangles over the result matrix represent the blocks used in the grid. The rectangles in the other two matrices represent the shared memory blocks stored for the block C_{11} in the iteration 1 in light gray and in the iteration 2 in dark gray.

With the previous idea in mind and the structure of a matrix multiplied by its transposed (equation 4.3), it is possible to create a kernel multiplication that combines transposition and multiplication "at the same time".

$$A \cdot A^t = \begin{pmatrix} a_{row1} \cdot a_{row1} & \cdots & a_{row1} \cdot a_{rowR} \\ \vdots & \ddots & \vdots \\ a_{rowR} \cdot a_{row1} & \cdots & a_{rowR} \cdot a_{rowR} \end{pmatrix} \tag{4.3}$$

The process is similar to the one described for the standard tile-based matrix multiplication, but in turn, in this case, the shared blocks are always taken from the same matrix and the movement is always from right to left. In addition, the block (k, n) of the result matrix creates the blocks of shared memory out of blocks $(k, 1)$ and $(n, 1)$ of the initial matrix. With this change, the blocks are always moving across the rows and the index of the final element determines only the necessary rows (kernel is shown in Figure A.8). Figure 4.7 shows the same example as before. Since the block calculated now is C_{12}, blocks moved to shared memory are A_{11} and A_{21} in first iteration. The next iteration blocks A_{12} and A_{22} are moved to complete the values of block C_{12}. As a consequence, the transposition matrix process is removed while the multiplication time is not affected, improving total computation time and hence throughput.

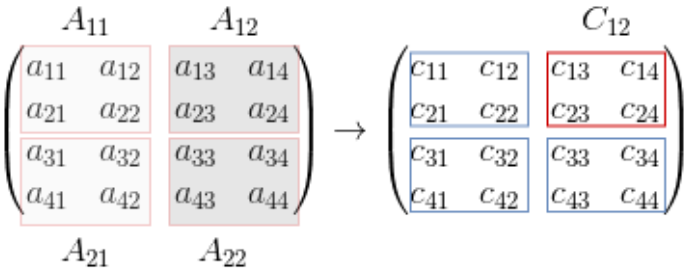


Figure 4.7: Matrix tile multiplication $A \cdot A^t = C$. The rectangles over the result matrix represent the blocks used in the grid. The rectangles in the other matrix represent the shared memory blocks stored for the block C_{12} in the iteration 1 in light gray and in the iteration 2 in dark gray.

Although this kernel seems to be a good solution, the cuBLAS library offers an optimised function that fits perfectly with this operation (*cublasSgemm*) (function is shown in Figure A.9). The implications of using one or the other are considered in the results in Chapter 7.

Jacobi

In order to follow the implementation explanation, Figure 2.13 is repeated in this section as Figure 4.8 for reader's convenience. In this case, line 9 is added. This means that it is possible to find all the removable elements (as explained in Chapter 2) in order to eliminate up to $n/2$ elements (being n one dimension of the square covariance matrix).

```
   Input A (matrix), ε (error parameter), size
   Output E (eigenvalue matrix), V (eigenvector matrix)
 1: procedure JACOBI METHOD
 2:     A_0 = A
 3:     E_0 = I_size
 4:     k = 0
 5:     while |some non-diagonal element of A_k| > ε  do
 6:         for i = 0 → size do
 7:             for j = i + 1 → size do
 8:                 if |A_{k-1}(i, j)| > ε then
 9:                     (find removable elements)
10:                     create P_k(i, j)
11:                     A_k = P_k^t · A_{k-1}^t · P_k
12:                     E_k = E_{k-1} · P_k
13:                     k = k + 1
14:         N_global = N_global + 1
15:     V = A_k
16:     E = E_k
```

Figure 4.8: Pseudo-code for the Jacobi method.

The first idea for implementing this algorithm in the GPU was using a kernel for every operation inside the loops and getting advantage of the inherent parallelism: find all the removable elements in a single iteration (line 9), creation of the rotation matrix P (line 10), rotation of the covariance matrix $A' = P^t \cdot A \cdot P$ (line 11) and update of the eigenvalues in matrix E (line 12). The process of finding all the removable elements of the matrix for a rotation matrix is iterative, it is impossible to find the next element without knowing the previous one. This fact pushed to add a different kernel in order to find all the suitable elements. With this implementation, the algorithm worked properly but, in contrast, the time constraints imposed by the project were not met due to the latency introduced when launching the kernels[1].

This result led to a solution in which the loops were inside the threads in order to avoid the kernel launches. However, the problem of synchronisation[2] appeared. The issue

[1]The latency of each kernel launch is low but the number of them is huge: $\frac{128 \times 128}{2} \times N_{global} \times 4$.

[2]Before CUDA 9.0 (the project uses CUDA 8.0), only synchronisation between threads within a same

is originated because of the parallel part of Jacobi: it is necessary to know all the removable elements before creating the rotation matrix. It is impossible to block the execution of all the threads at that point. As a consequence, the parallel approach of Jacobi is removed from the project and the sequential version is adapted. Both options are depicted in Figure 4.9.

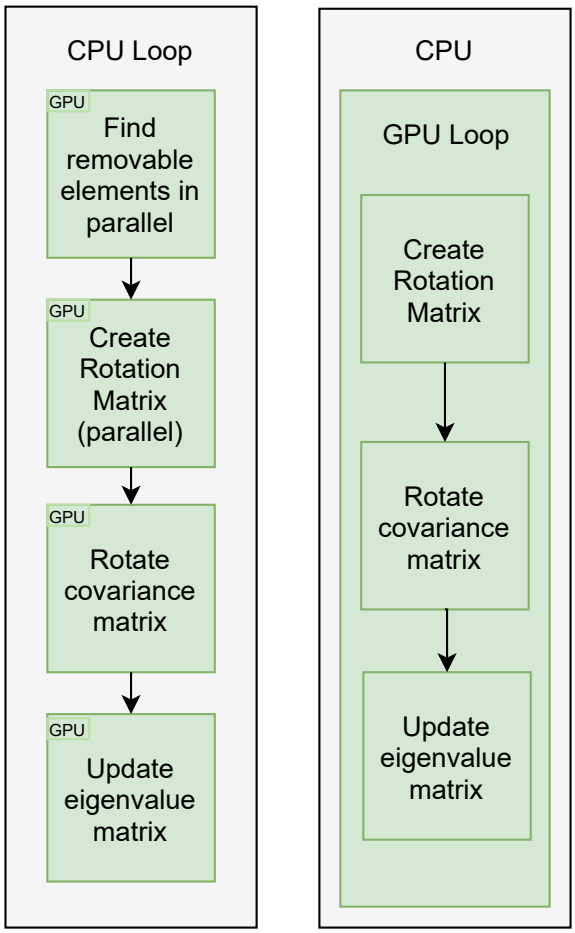


Figure 4.9: Jacobi versions. On left, version getting advantage of the inherent parallelism of the method (four different kernels launched from a CPU loop). On right, version without getting advantage of the inherent parallelism of the method (a single kernel with the loop inside).

As said before, the Jacobi kernel using the sequential version of the algorithm (kernels shown in Figure A.12 and Figure A.13) introduces the loops inside all the threads in the grid. It is composed of only 128 threads in order to improve matrix multiplications because this is the only parallelizable part, as the rest of the operations are performed serially. It is important to note that the multiplications are optimised so that only the columns and rows (p, q) from the eigenvectors and eigenvalues matrices are changed each iteration In

block exists, hence, it is impossible to synchronise the different blocks (the only possibility is the creation of different kernels).

contrast, this solution implies a low parallelism degree and it is not likely to be a better solution than the one in the host. For this reason, computing this function in the host is also considered a valid option (function shown in Figure A.10 and Figure A.11).

Projection

The final part of the process involves projecting the first eigenvalue onto the original data. To do it, the first eigenvector is selected by choosing the greater eigenvalue (the one that better represents the data) and its associated eigenvector. Once found, only a multiplication between the HS image without average and the eigenvector is required:

$$PROJ = \begin{pmatrix} var_1 & \cdots & var_{128} \\ \vdots & \ddots & \vdots \\ var_{imSize} & \cdots & var_{128} \end{pmatrix} \cdot \begin{pmatrix} eV_1 \\ \vdots \\ eV_{128} \end{pmatrix} \tag{4.4}$$

This operation allows for a large level of parallelism: it is possible to use the same number of threads as the number of spatial pixels ($imSize$), carrying out 128 operations each one. This seems to be a simple and effective solution (kernel shown in Figure A.14).

4.2.3. NIPALS chain

This algorithm only needs the preprocessed HS image in order to obtain a certain number of eigenvalues projected onto the original data; in this case only the first one is needed. Similar to Jacobi, this is an iterative method which adjusts the level of error with a constant ε and whose number of iterations are dependant on the data. The algorithm [1] [20], counts with four main steps within a loop for each component calculated. The algorithm, shown in Figure 2.14, is repeated in this section in Figure 4.10 for reader's convenience.

Since the iteration count in this algorithm is low[3], the possibility of using different kernels for each operation is the solution adopted. In addition, as only the first principal component is required, the final step of the algorithm (Figure 4.10 line 16) can be omitted because the R matrix is used for the following iteration. As a consequence, the method is divided into 4 kernels that perform the four main steps of the algorithm within a loop.

Before going into the details of the kernels, it is important to note that as the loop is placed in the host part, the condition to break the while loop must be done in the host. This fact forces the communication between the variable λ' from the device to the host, adding some latency in each iteration, which in cases with an elevated number of iterations can be a problem. For this reason, in order to avoid the specific transfer in each iteration,

[3] As demostrated by the tests done with the project images, the maximum number of iterations is below 100 iterations in the worst case. For the smallest image, for instance, only 5 iterations are required.

Input A (matrix), ε (error parameter), N (number of principal components), i_{max} (maximum number of iterations permitted)

Output T (principal components), P (associated eigenvectors), R (residual matrix)

```
1:  procedure NIPALS METHOD
2:      R = A
3:      for k = 0 → N do
4:          λ= 0
5:          T_k = R_k
6:          while i < i_max do
7:              P_k = R^T · T_k
8:              P_k = P_k/||P_k||
9:              T_k = R · P_k
10:             λ' = ||T_k||
11:             i = i + 1
12:             if |λ' − λ| > ε then
13:                 break
14:             else
15:                 λ = λ'
16:         R = R − T_k · (P_k)^T
```

Figure 4.10: Pseudo-code for the Nipals method.

the variable used for λ' is allocated as pinned memory using the zero copy approach. In this case, the lack of caches is not a problem.

The operation (line 7) in the first kernel ($P_k = R^T \cdot T_k$) is similar to the one in the projection stage, a matrix with dimensions $bands \times imSize$ multiplied by a vector with $imSize$ elements. Nevertheless, it is not the same. In this case, the dimensions are the opposite, which means that if the same process as in the projection stage was done, there would only be 128 threads doing $imSize$ operations serially each one. This stands for a solution with low parallelism and with a large number of operations required in each thread, which does not seem to be a proper solution (kernel in Figure A.15). Consequently, a solution based on the same idea of the average in the pre-processing stage is used. For each row of the matrix, a block with 1024 threads multiplies and adds its row by the vector serially until 1024 results inside each thread are obtained (kernel in Figure A.16). At this point, the results are added using a reduction. The result is a vector with 128 elements.

The second operation (line 8) ($P_k = P_k/||P_k||$) is divided into two kernels: one for obtaining the norm of the vector and the other one to compute the division of all the elements in the vector. Computing the norm is almost the same as computing an average. For this reason, only a reduction adding all the elements of the squared vector is used. The second stage only consists of dividing the square root of the previous value by all

the elements, thus being possible to accomplish this operation in parallel for all of them (kernel shown in Figure A.17).

The third operation (line 9) ($T_k = R \cdot P_k$) is the same as the first but with a matrix sized $imSize \times bands$ and a vector sized $bands$ elements. In this case, the kernel uses a number of threads equal to $imSize$, carrying out 128 operations each one (kernel shown in Figure A.18).

For the last operation ($\lambda' = ||T_k||$), the idea is the same as for the second. The kernel performs a reduction of the whole vector with the values previously squared. Then, the square root is calculated (kernel shown in Figure A.19).

Although it is not mentioned, in the matrix-vector multiplication kernels the idea of constant and restrict pointers from the best practices is followed. Since in this operation the vector is accessed many times, it is possible to improve the throughput by storing it in read-only memory.

4.3. SVM

The implementation of the SVM, done by Sergio Sánchez [51], is explained in this section. Recalling the SVM algorithm in Chapter 2, the SVM in-situ classification part (the one needed to be accelerated) is composed of three main steps: the distance calculation to the six hyperplanes for new vectors, the transformation from distance to probabilities and the generation of the four probability maps (one per class). Figure 4.11 shows the diagram.

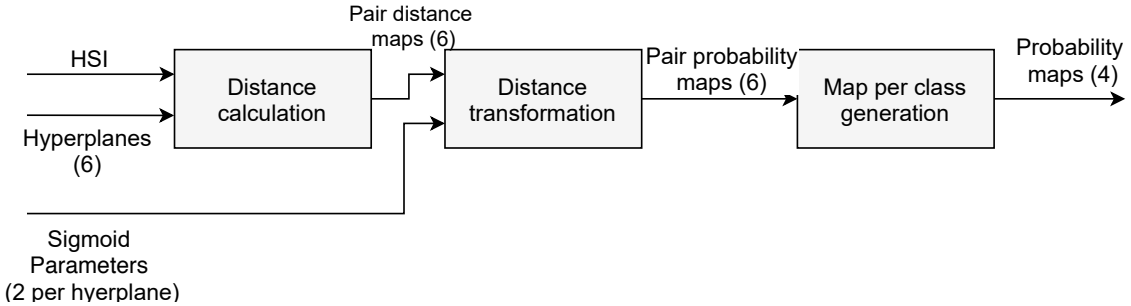


Figure 4.11: SVM diagram.

The first part of the process is implemented as a single kernel which uses a 2 dimensional grid composed of $imSize \times CLASSIFIERS$ threads. While the size of the image corresponds with the x dimension, six classifiers are associated with the y dimension. This way each thread is calculating the distance between the vector associated with the spatial image (a vector with 128 features) and the hyperplane associated with the classifier. The arrangement of the grid is shown in Figure 4.12.

The second and the third stages are calculated in the second kernel. In this case, the

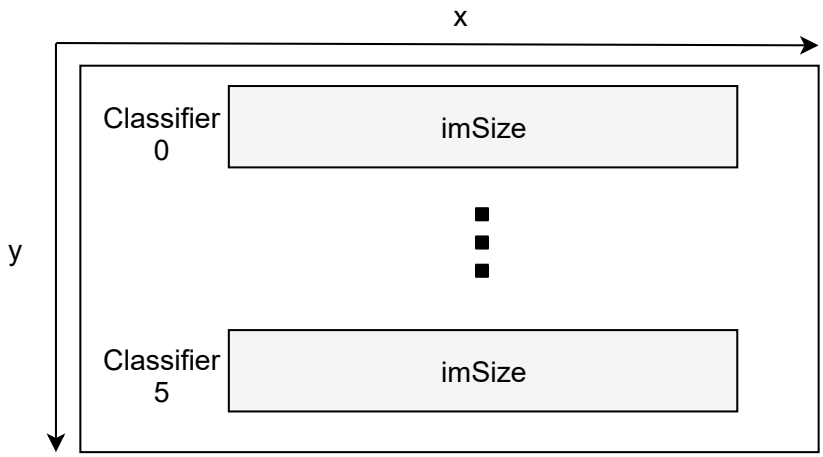


Figure 4.12: SVM grid in the first kernel: distance calculation.

kernel uses a grid similar to the previous one, however, the number of elements in y axis is the number of classes: 4. In the first place, the kernel transforms the distances into probabilities making use of the sigmoid function and then, using a different method from the used in HELICoiD, the pairwise probabilities are converted into the probabilities for each class [51].

4.4. KNN

The implementation of the KNN, done by Adolfo Vara de Rey [54], is composed of three stages: initialisation, neighbours search and filtering. In this case, the three steps are included in a single kernel which uses the same number of threads as the spatial size of the image. The diagram is shown in Figure 4.13.

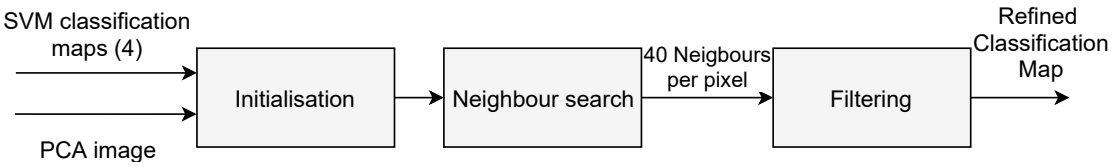


Figure 4.13: KNN diagram.

The initialisation part assigns a row, column, pixel and PCA component identifier to each pixel in order to generate the vectors needed. It is important to mention that since the value of the PCA component is used many times by different threads, it is stored in texture memory, obtaining this way faster accesses.

The neighbour search uses a *dynamic window* to obtain the 40 nearest neighbours. In Chapter 2 it is mentioned how every other pixel in the image is compared with the one

under evaluation but this process is too expensive. Fortunately, experimental results show that it is only necessary to calculate the distance with pixels several lines above and under the pixel because the spatial component of the distance with other pixels is always larger. As a consequence, the *dynamic window* concept mentioned appears. In Figure 4.14, the movement of the window is shown. Since this process is made serially (each pixel performs this process in parallel), whenever a distance is calculated, it is sorted at the same time. At the end, an array with the index of the 40 neighbours with the smallest distance is obtained (for each pixel).

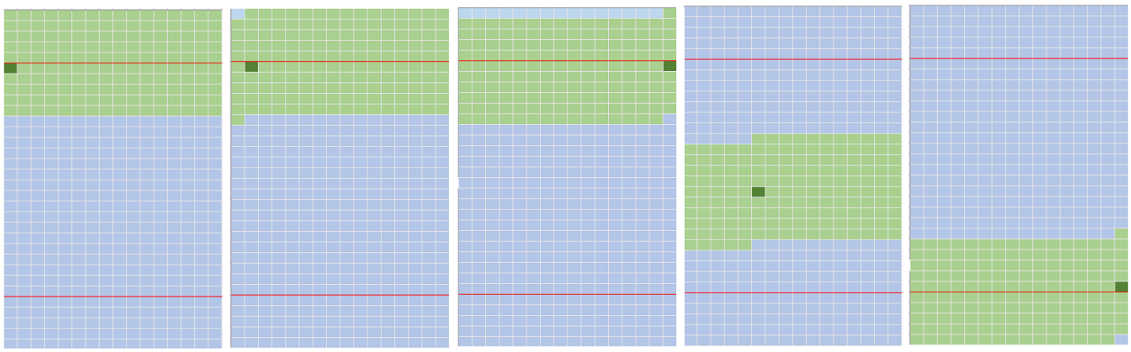


Figure 4.14: KNN dynamic window [54].

The filtering stage takes the 40 indexes for each pixel as input. Retrieving the probability values from the SVM map for those indexes, an average probability per class is calculated selecting the highest value of the four classes as the final class [54].

4.5. Spatial Filter

A spatial filter is a well-known tool for morphological operations in the field of image processing. It is used to change the value of all the pixels in an image taking into account a specific number of nearest pixels within a moving window. The number of nearest pixels and the operation performed is defined with a mask; normally with a square shape. The type of operation performed by the filter depends on the values of the mask and there are lots of them: smoothing, sharpening, border detection, etc. An example of the filtering step is shown in Figure 4.15.

In this project, the mask used is a Gaussian mask, normally used to blur an image by averaging the pixels within the mask. Nevertheless, the use of the filtering step in this work is not directly applied to an standard grayscale image. Recalling Chapter 2, the output of the SVM algorithm are four probability images representing the probability of each pixel to belong to each class. With this information, the initial classification map can be obtained by just selecting the class with the maximum probability for each pixel.

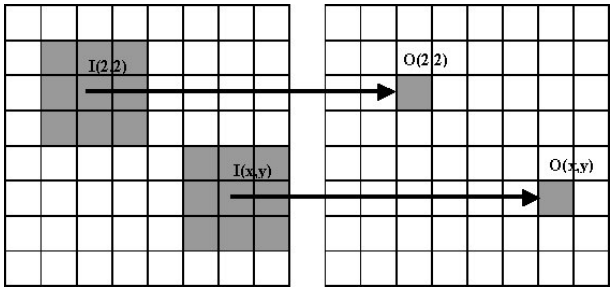


Figure 4.15: Example of a spatial filter with a 3×3 mask. Source: http://tracer.lcc.uma.es/problems/mfp/SpatialFiltering.jpg

Hence, in order to filter the initial classification map, each one of the four probability maps class are filtered and then, for each pixel the class with highest probability is set as the pixel class.

In this operation, all the pixels are independent, so the operations for each of them can be performed in parallel. For this reason, the filtering is one of the most amenable computation to be accelerated in a GPU. Its implementation in the GPU is simply a grid with the same number of threads as the size of the image. Each thread calculates in parallel the average between the pixels included in its window for each class.

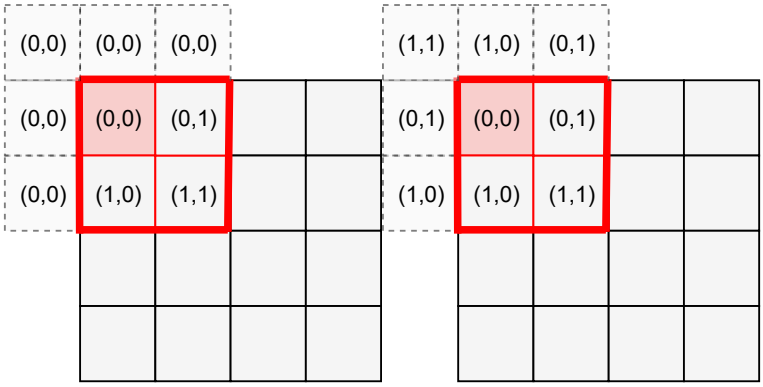
However, a problem occurs when applying the mask to the borders of the image. As it can be seen in Figure 4.15, if the mask is moved to a pixel near to the border (less than $\frac{n}{2}$ with n the size of the mask), the number of pixels available to compute the average is different. For this reason, it is common to follow a different approach for them. In this work, two of the most common solutions are evaluated: (1) replicating the border pixels and (2) mirroring the border pixels.

The first solution simply applies the filter to all the pixels of the image replicating the value of the pixel in those threads dealing with border pixels (kernel shown in Figure B.1). Figure 4.16 **(a)** represents the solution. When applying the filter, for example, to the corner of an image, all the non-existing pixels are considered to have the same value as the one in the corner.

The second solution tries to give borders a context meaning. For doing it, the non-existing pixels are replaced with its mirrored version, thus, the average takes more into account the context of the pixel than its own value (kernel shown in Figure B.2). In Figure 4.16 **(b)**, the process is simulated for the same corner.

Although the value of the pixels in the borders can change heavily depending on the selected approach, it is important to recall that this problem only affects to the pixels near to the border, which usually do not contain useful information. It means that only few pixels compared with the whole image will be changing.

Finally, the last observation to be considered is the size of the window. Since the spatial filter intends to achieve the same results as the original chain, the number of pixels within



(a) Solution 1: replicating the evaluated pixel in the non-existing ones.

(b) Solution 2: mirroring the window pixels for the non-existing ones.

Figure 4.16: Solutions for filtering the image borders with an example 3×3 mask.

the window must be similar to the window used for the KNN: 40 nearest neighbours. Hence, the window used for the mask of the filter is 7×7 [4].

[4]The size of the windows are usually odd in order to maintain symmetry with the evaluated pixel in the center. For this reason, the square mask nearest to 40 pixels is 7×7.

5. Project Integration

5.1. Introduction

This chapter includes details about the integration of the original HELICoiD algorithm chain as well as the alternative chain integration. As said before, SVM and KNN have been accelerated in the Jetson TX1 by Sergio Sánchez [51] and Adolfo Vara de Rey [54] respectively.

5.2. HELICoiD Original project

It is important to recall the global structure of the project in order to understand the decisions taken for the integration. Figure 5.1 depicts the scheme of the original project using the three algorithms and its inputs and outputs. As it can be observed, the process can be divided into two parts: the first one corresponds with the execution of the PCA and SVM algorithms in parallel and the second with the KNN.

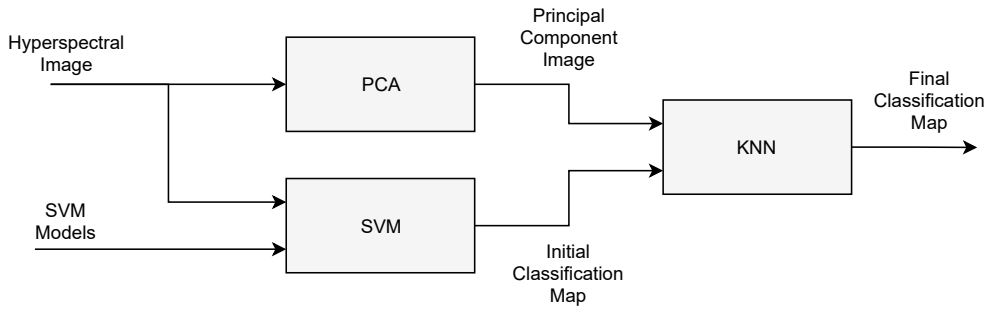


Figure 5.1: Original process scheme.

The entire project is devised as a main program which calls the three algorithm functions and connects the required inputs. For that, the program needs to load in memory (from disk) the selected HS image and the SVM models. Logically, the HS image is transferred only one time to the GPU, but it is used to feed both the PCA and SVM kernels. With the principal component from the PCA and the initial classification map (a probability image for each class) from the SVM as inputs, the KNN can proceed with the generation of the final classification map. Nevertheless, there are some observations to take into account about the SVM and KNN implementations.

In the case of the SVM, the algorithm is already included in a function which using device pointer inputs, returns a device pointer to the SVM image. However, the output arrangement is not the same as the KNN input uses. The former arranges pixels class by class whereas the latter arranges the entire images class by class. The idea is represented in Figure 5.2: on the top, the SVM output is represented, and on the bottom, the input of the KNN. For this reason, the output of the SVM is modified in order to obtain the pixels

arranged class by class so the KNN can benefit from it instead of rearranging SVM output data from inside the KNN. For doing that, and with the idea of not introducing more latency to the chain, the second kernel of the SVM function must be changed. This kernel uses a grid composed of a number of threads equal to the size of the image multiplied by the number of classes. In addition, each of the threads calculate the value of the probability for its associated pixel and class and stores it in a register. Then, the result is copied to the global memory class by class. Hence, by simply changing the thread access pattern to the global memory, the result classification map arrangement is changed.

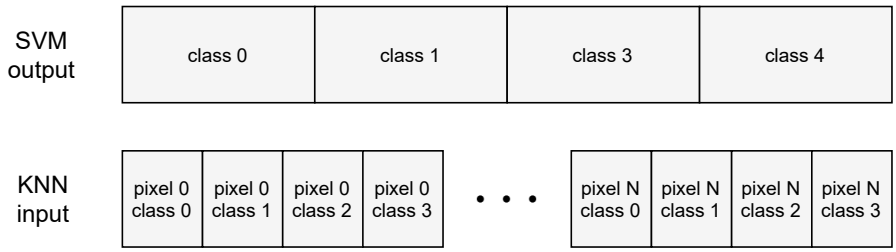


Figure 5.2: Different classification map arrangements. On the top, the output of the SVM, on the bottom the input of the KNN.

In the case of the KNN, the implementation available [54] is done as a main function that read files from disk and performs the required operations. Because of this, it needs to be changed to be a function that uses device pointers to the required resources and that returns a pointer. The idea is to continue using all the available memory inside the GPU. This way, no memory transfers are required and the transfer time is avoided. Consequently, the device pointers to the PCA and SVM outputs are used as the input to this one. The only remaining problem in this case is the use of a KNN host function used to crop the SVM images to a maximum size of 512×512 (due to the implementation of the algorithm received). This issue would force to move the SVM images to the host to crop them and the result would be sent again to the device. Nevertheless, implementing the cropping function in the device would allow to avoid the two memory transfers. Because of it, a version of the cropping function is implemented in the device in order to avoid the data movements.

With the three algorithms now working as a complete processing chain, only a final remark is needed. Although the PCA and SVM algorithms can be executed in parallel, the only way in which it can be achieved is by executing one in the host and the other in the device (recalling Chapter 2, the best practices about the Streams). This can be achieved using Jacobi solution in host (from PCA). Hence, while the host is calculating the Jacobi method for the PCA, the device is free to execute the SVM. This can be considered an optimisation, but it only can be achieved using the Jacobi chain. In addition, the processing time of SVM is one magnitude order lower than PCA and two than KNN so the optimisation degree is negligible.

5.3. Alternative project

As in the previous chapter, the project is structured as a main program that loads the necessary data and call the SVM and the filter functions, connecting their inputs and outputs. In this case, the scheme is easier and it is depicted in Figure 5.3.

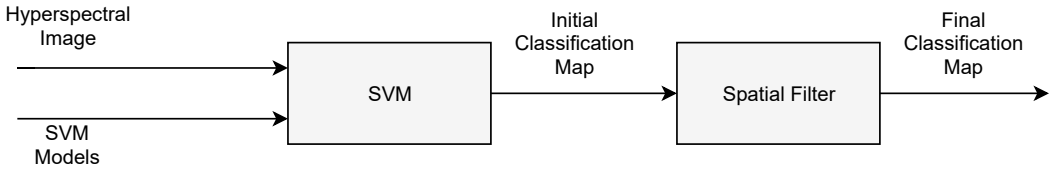


Figure 5.3: Scheme of the alternative project.

On one hand, the same implementation of the SVM algorithm is used whereas the change in the output arrangement is not needed. For the spatial filter, it is better to organise the probability images class by class.

On the other hand, the filter kernel is included in a function using device pointers. In addition, another kernel is included at the output in order to select the class with the largest probability for each pixel.

This way, the final classification map is obtained. The comparison in terms of error and performance/efficiency is included in Chapter 7.

6. Energy characterisation

6.1. Introduction

This chapter explains in detail the method used for measuring the energy consumption in the Jetson TX1 platform. At the beginning, before knowing the platform had an internal sensor, measuring the board consumption with a computer-controlled external power supply was the first approach to the problem. Measuring current in a shunt resistor was also considered. Nevertheless, the board has an integrated internal sensor capable of measuring voltage called INA3221.

The chapter is divided into three sections: the first explains the way in which the INA3221 monitor can be used within the platform, section 6.2. Then in section 6.3 a graphical monitor developed in this work to visualise the power consumption of the platform is described. Finally, section 6.4 introduces a C library created in order to simplify the measurement of the energy consumption in C-based applications executed in the platform.

6.2. Voltage Monitor INA3221

This sensor placed in the the Tegra X1 module is able to measure the voltage drop from three shunt resistors corresponding with the supply monitor of the main input of the module, of the GPU and of the CPU. As it can be seen in Figure 6.1, the power monitor channels are ordered exactly this way: main input (1), GPU (2) and CPU (3). The value of the shunt resistors in each channel are shown. This value is going to be used for calculating the current consumed by each channel [37].

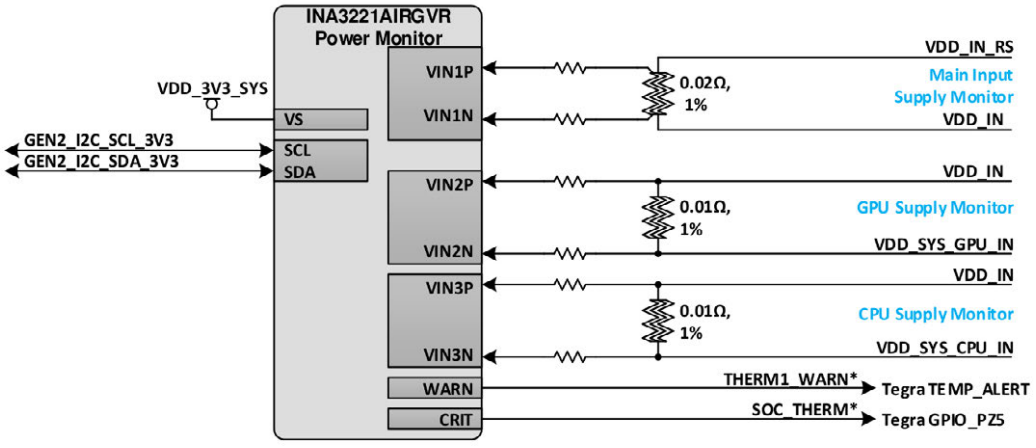


Figure 6.1: Block diagram of the INA3221 sensor in Jetson TX1 [37].

In the figure, it can also be observed that the device has an I2C interface used for the connection with the Jetson TX1. This allows to stablish communications with the

sensor by using `sysfs`[1], i.e, files that contain the value of the voltage read from the sensor in every moment. These files also include the conversion to current and to power as shown in Figure 6.2. Although the figure only shows channel 0 (main input), the other two channels are accessible in files with the same name but with a different number, for instance, `in_current1_input`.

```
1 $ cat /sys/devices/platform/7000c400.i2c/i2c-1/1-0040/iio_device/in_current0_input
2 $ cat /sys/devices/platform/7000c400.i2c/i2c-1/1-0040/iio_device/in_voltage0_input
3 $ cat /sys/devices/platform/7000c400.i2c/i2c-1/1-0040/iio_device/in_power0_input
```

Figure 6.2: Files with the information of voltage, current and power for the first INA3221 channel.

This is a good solution to measure the power in the platform in real time, however, the sysfs feature is included in Jetson TX1 revisions above 300 but the one used in this work is revision 100. As a consequence, in order to measure the voltage, current or power, the voltage information must be retrieved from the sensor using the I2C interface.

Taking into consideration the location of the `sysfs` files, it can be deduced that the I2C interface used is the one called `i2c-1` and the identifier of the slave sensor is `0x40`. Referring to the INA3221 data sheet [17], registers from 1 to 6 in Figure 6.3 are the ones which contain the information needed. As it can be seen, not only the shunt voltage is measured by the sensor, but also the bus voltage, which is used to calculate the power following the standard equation $P = VI$.

POINTER ADDRESS (Hex)	REGISTER NAME	DESCRIPTION	POWER-ON RESET		TYPE(1)
			BINARY	HEX	
0	Configuration	All-register reset, shunt and bus voltage ADC conversion times and averaging, operating mode.	01110001 00100111	7127	R/$\overline{W}$
1	Channel-1 Shunt Voltage	Averaged shunt voltage value.	00000000 00000000	0000	R
2	Channel-1 Bus Voltage	Averaged bus voltage value.	00000000 00000000	0000	R
3	Channel-2 Shunt Voltage	Averaged shunt voltage value.	00000000 00000000	0000	R
4	Channel-2 Bus Voltage	Averaged bus voltage value.	00000000 00000000	0000	R
5	Channel-3 Shunt Voltage	Averaged shunt voltage value.	00000000 00000000	0000	R
6	Channel-3 Bus Voltage	Averaged bus voltage value.	00000000 00000000	0000	R

Figure 6.3: Shunt and bus voltage registers in INA3221 [17].

The values obtained from the registers need to be transformed in order to be converted into a voltage value. The process is the following:

1. Transform the register value from little endian to big endian (if needed).
2. Divide the value by a factor depending on the type of register: shunt voltage register or bus voltage register. Factors are the following: $f_{shunt} = 40\ \mu V$ and $f_{bus} = 8\ mV$.
3. As the three least significant bits of the register are not part of the measure, the number must be shifted three positions to the right, i.e., divided by eight.

[1]The sysfs filesystem is a pseudo-filesystem which provides an interface to kernel data structures.

In our case, the measurement is always positive, so there is no need to do any sign conversions. However, it is possible for the INA3221 sensor to measure negative voltages. The process in this situation is described in the data-sheet, but it is out of the scope of this work.

With the voltage information from every shunt and the shunt resistor value for each channel, it is possible to obtain the drained current by simply applying Ohm's Law. Multiplying the current value by the voltage bus for each channel results in the instantaneous power consumed by either the whole module, the GPU or the CPU, respectively.

6.3. Graphical Monitor

Once the power can be measured, the next step tackled the development of a visualisation tool based on Python and the module PyQtGraph. The graphical monitor developed is depicted in Figure 6.4.

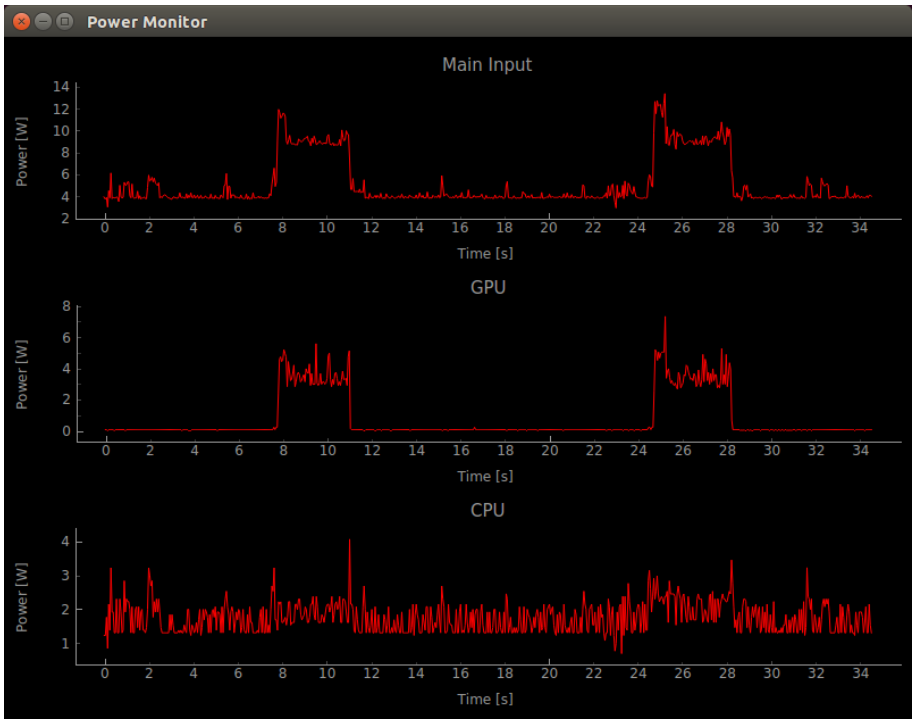


Figure 6.4: Screenshot of the graphical monitor when measuring the execution of original project implementation with the Op20C1 HS image.

The period used to measure the information takes into account the time needed for Python to capture the I2C data, transform it and show it in the graph. The last part represents most of the time consumed, which oscillates between 50 and 100 milliseconds. Consequently, the graphical representation in this implementation can not show faster variations than $1/50\ ms = 20\ Hz$ in real time.

This application is devised as a qualitative tool used just to distinguish graphically the consumption in the three channels. However, it can not be used for accurate energy measurements (moreover, it adds a non-negligible power consumption to the module). For that purpose, a C library has been developed and is introduced in the following section.

6.4. C Library

This library introduces a set of functions to measure the energy between two time stamps in a C code for each channel. After initialisation, two time events in the code are set and finally, the energy value for this interval is returned. In this case, the precise timing is of paramount importance since the period needs to be perfectly known to integrate the power to compute energy.

The library creates two threads: one to run the main C application and another to read and process the voltage from the sensor in every period. This second thread is created when the start event is called and it is removed with the stop event call. As it can be seen in Figure 6.5, the energy thread executes a `while` loop in which first, the instantaneous power is calculated from the voltage information from the sensor and then, the energy is calculated using a rectangle integration approach (multiplying the instantaneous power by the period and accumulating it).

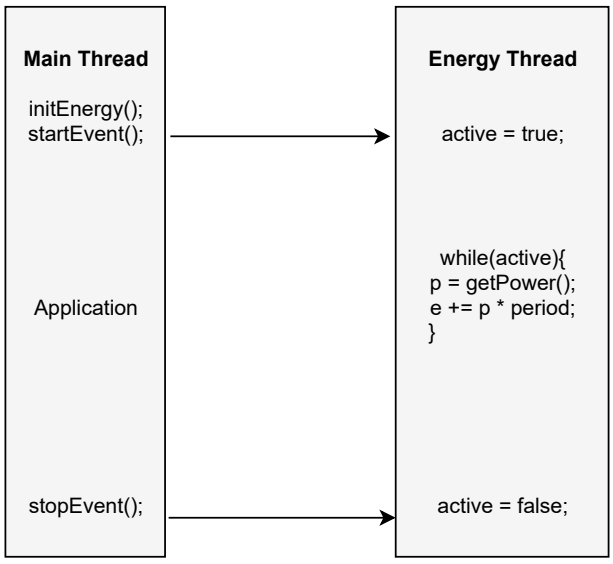


Figure 6.5: C library process.

As said before, the timing is important for the last operation. For this reason, in every iteration of the loop, the time between the previous and the current iteration is calculated and set as the period. This way, the period is the actual time between one measure and the following. The only problem with this process is that the sensor measurement period

is between 1 ms and 1.4 ms and, sometimes, the loop time is lower. In these cases, the measure from the sensor is the same as the previous one, which is similar to a naive interpolation.

Using the library to measure the static power consumption of the Jetson TX1 module, the following results were obtained: mean power consumption of 2.3 Watts while the module is on energy-save mode and 3 Watts with the module in its maximum performance mode. The energy-save mode refers to a frequency clock for the CPU of 1.326 GHz and for the GPU of 768 MHz. The maximum performance mode refers to a frequency clock for the CPU of 1.734 GHz and for the GPU of 998.4 MHz. This values are set using the script called `jetson_clocks.sh`.

7. Results

7.1. Introduction

This chapter includes all the results extracted from the implementation of the original project (HELICoiD) as well as from the alternative project based on the spatial filter.

Consequently, the chapter is structured in four main sections: one for describing the methodology followed, two for comparing the energy and time consumption of both projects and the last one that compares both the resulting classification maps and the timing/energy results themselves[1]. The time-energy results are many times given as number of pixels per image, obtaining results of throughput and energy efficiency (in this chapter the term *efficiency* always refer to energy efficiency). Moreover, the first and second sections introduce partial time results from the PCA algorithm and the spatial filter, respectively.

Furthermore, this chapter includes results from cuBLAS library for matrix multiplication operations. The intention was also to include results from the Thrust library. However, only reductions could be accelerated using it (the only algorithm used in this project and available in Thrust), obtaining very bad results in general. As a consequence, its results are not included.

7.2. Methodology

In the first place, the PCA was implemented in MATLAB, obtaining the same images as in the MPPA platform. Results from this implementation were used as a reference in order to compare the results step by step with the implementation in the Jetson TX1 system. Once a functionally correct version was obtained, PCA optimisation process began. This task was performed mainly using the NVIDIA Visual Profiler, shown in Figure 7.1. Visual Profiler allows programmers to visualise the time consumed by every element in the application execution. This includes kernel executions, memory transfers and operations performed by CUDA (kernel launches, synchronisation, allocations...). In addition, the tool provides an exhaustive analysis of the executed kernels (achieved bandwidth, occupancy...), which is very useful optimising [39]. The same process was performed for the spatial filter.

The project integration, in both cases, was performed checking the results and comparing them with the MPPA results. When all results were equal (admitting certain level of error which was also characterised as shown in Section 7.5), the time and energy measurements and optimisation began.

[1] Although results were first compared with the MPPA implementation developed in HELICoiD, in this work it was decided to combine the functional and time/energy comparison in a single section. For this reason, the comparison is after the time/energy characterisation.

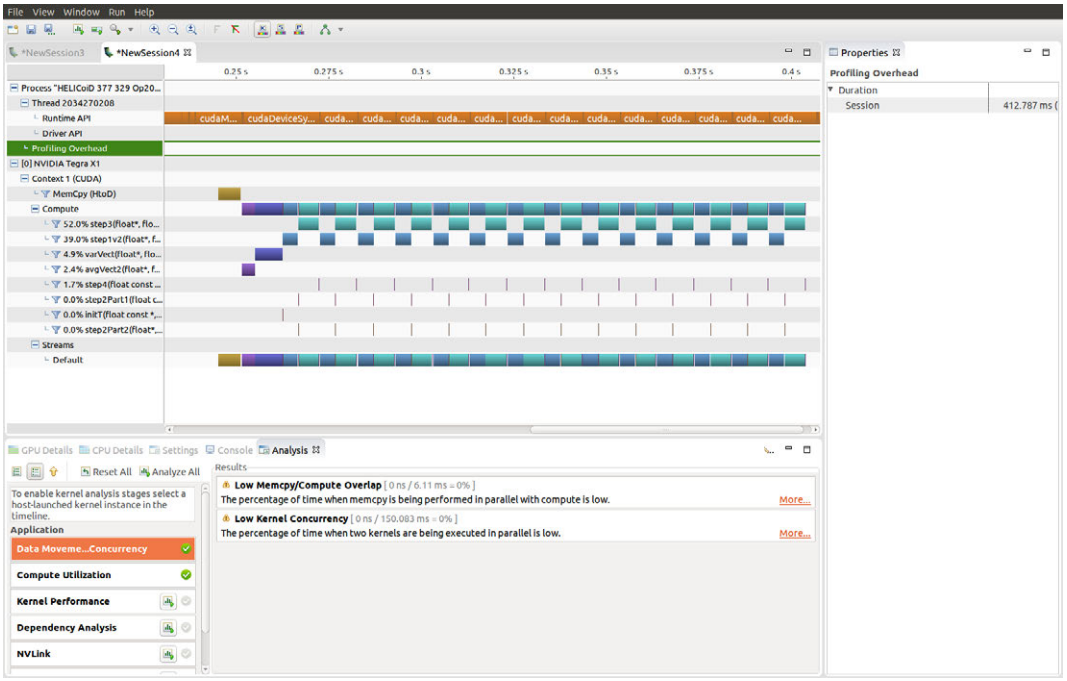


Figure 7.1: NVIDIA Visual Profiler. Profile of a PCA (NIPALS) execution.

All the images from the HELICoiD database (included in Table 2.1) were used in order to measure the processing time and the energy consumed by the platform. For the partial PCA results and the spatial filter, the images used were only the smallest (Op20C1) and the largest (Op8C2), being compared with a naive sequential host version in order to show the speed-up factor achieved in the platform using the different kernels.

Time and energy results are calculated as an average of ten application executions, dismissing two outliers from twelve executions. In order to measure the error, Op20C1 and Op8C2 final classification maps are compared with the MPPA results. The different pixels from the reference are counted and divided by the total amount of spatial pixels in the image. This way, the percentage of error is calculated:

$$f(x,y) = \begin{cases} 1, & x \neq y \\ 0 & x = y \end{cases} \tag{7.1}$$

$$e(\%) = \frac{\sum_{i=1}^{imSize} f(i_{\text{ref}}, i)}{imSize} \cdot 100 \tag{7.2}$$

Finally, tests have been performed with the system NVIDIA Jetson TX1 revision 100 in the maximum performance mode: CPU frequency clock 1.734 GHz and GPU frequency clock 998.4 MHz. The software used is Jetpack 2.3.1 and CUDA 8.0.33.

7.3. HELICoiD Original Project

7.3.1. PCA

Pre-processing

The first operation required in the PCA algorithm is the average per band which is part of the pre-processing step. For this operation, four different kernels were proposed and its results are shown in Figure 7.2:

- The first proposal, the naive kernel, performs 128 average operations in 128 different threads. This low level of parallelism does not achieve better results than the implementation in the CPU.
- The second one, the atomic kernel, also obtains results comparable to the ones in the host side. As said in Chapter 3, atomic operations serialise memory accesses, yielding worse time results.
- Finally the last two kernels are based on reductions, one using shared memory and the other shuffle warp operations. As it can be seen, the two kernels achieve similar results, being by far the best solutions. For this reason, the shuffle warp reduction kernel is selected for the final PCA implementation.

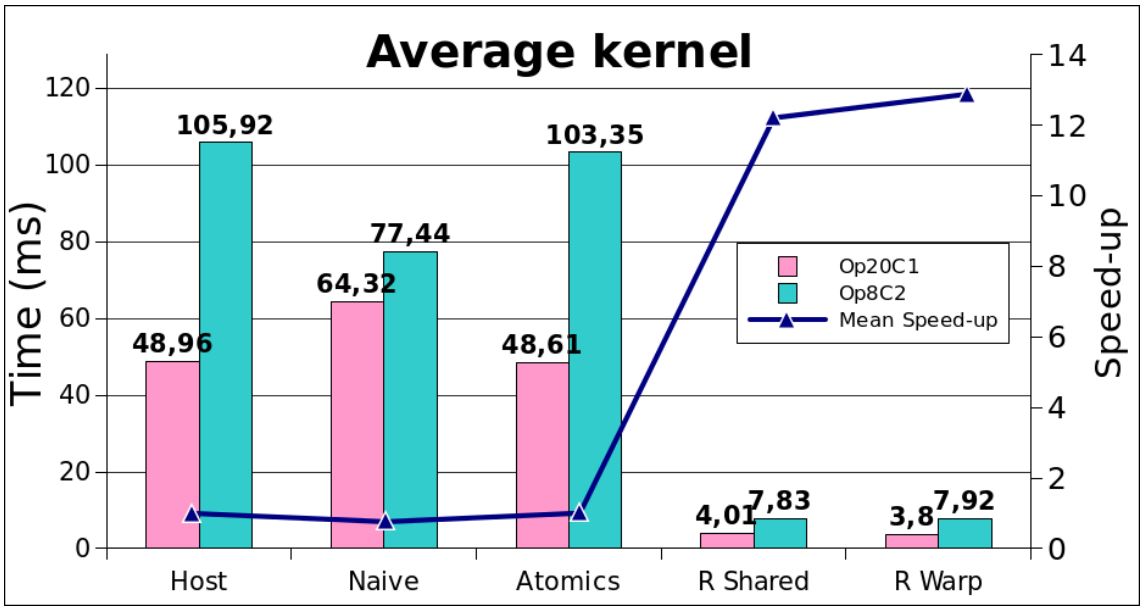


Figure 7.2: Results of the different average per band kernels. The mean speed-up line includes the speed-up information from HS images Op20C1 and Op8C2. The speed-up reference is the host execution time.

The second operation of the pre-processing step is the average subtraction from the HS image band by band. Results are depicted in Table 7.1. In this case, the proposed kernel achieves a huge parallelism level since every thread is accessing to a different pixel in the hyperspectral image, obtaining good results compared with the host.

Table 7.1: Results for the CPU and GPU version of the average subtraction kernel.

Image	Metric	Host	GPU
Op20C1	**Time (ms)**	58.31	6.93
Op20C1	**Speed-up**	x1	x8.14
Op8C2	**Time (ms)**	126.21	14.64
Op8C2	**Speed-up**	x1	x8.62

Jacobi chain

The first option to calculate the principal component of the HS image is the Jacobi chain. As explained before, this chain is composed of three steps: creation of the covariance matrix, Jacobi method computation and projection of the first eigenvector.

For the **covariance matrix**, two solutions were proposed in Chapter 4:

- On one hand, to separate the transposition and the multiplication so they can be performed in two different kernels. The results for this solution are collected in Figure 7.4 in the first and fourth column (HS image Op20C1 and Op8C2). Compared with results in the host, in Figure 7.3, the transposition kernel stands for a good solution in spite of its naive implementation. In addition, the matrix multiplication kernel yields very good results but it has to be taken into account the facts that the multiplication in the host is totally naive and the matrix multiplication is a perfect operation to be accelerated in a GPU.

- On the other hand, since the required multiplication is between a matrix and its transpose, it is possible to directly read the transposed matrix from memory in order to avoid a dedicated transposition operation. This way, columns two and five in Figure 7.4 show the time results for a custom kernel using this approach and columns three and six show an alternative kernel from cuBLAS library. In the figure, it can be observed that the custom kernel achieves similar time results as the multiplication kernel. This is the expected result since the multiplication is done using the same parallelisation approach but changing the order in which the data is introduced. Furthermore, the time results for the cuBLAS kernel are almost three times better in terms of throughput. The principal drawback of this library is the initialisation time introduced, which discards its use for just one execution of the application (as is the case right now in HELICoiD).

Taking into account the obtained results, the custom kernel which combines the transposition and the multiplication is the one selected for the covariance matrix computation.

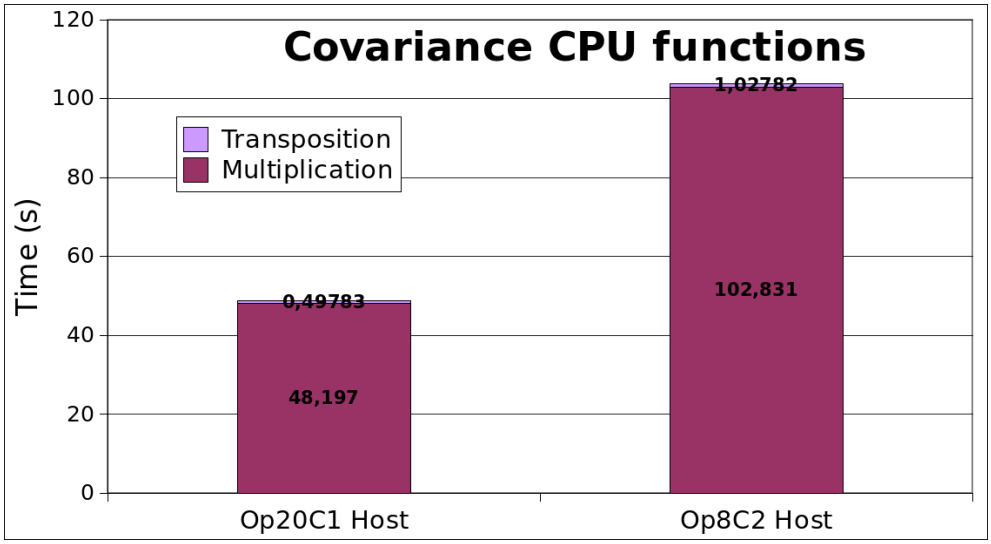


Figure 7.3: Results for the covariance stage in the host.

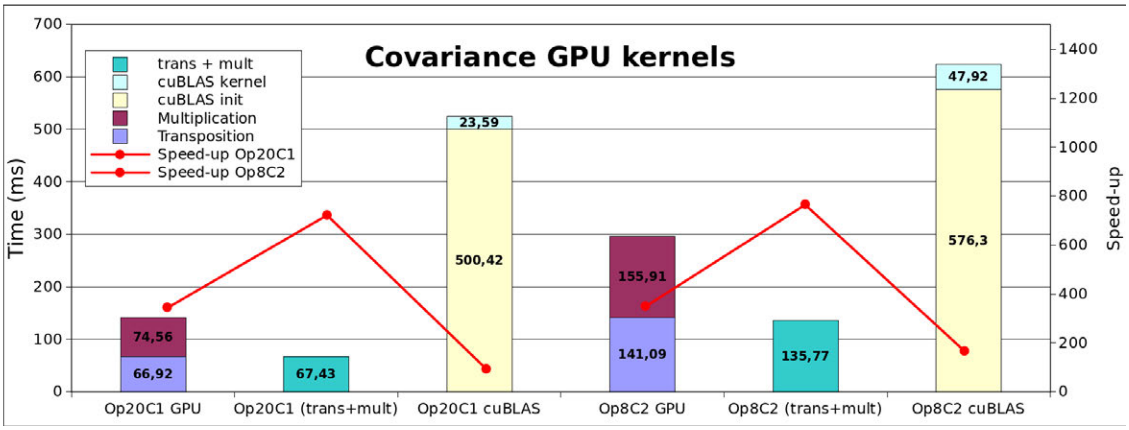


Figure 7.4: Results for the covariance stage in GPU. The speed-up line is referred to the sequential implementation in the host.

The next step is the calculation of **the eigenvalues and eigenvectors using the Jacobi method**. As mentioned in Chapter 4, this method is iterative and it is difficult to take advantage of a parallel implementation for acceleration. Looking at Table 7.2 it can be observed that GPU implementation does not achieve better results than in the host. In fact, the time obtained is almost half in the host side. For this reason, the host implementation is selected. The table also includes the time per iteration, which will be used for comparison with the Nipals method.

The last part in this chain is the **projection**, which involves a matrix-vector multiplication. Results are showed in Table 7.3. Although the kernel code used for this part is

Table 7.2: Partial results for the CPU and GPU version of the Jacobi method with $\varepsilon = 0.001$. The part of the CPU is taking into account the memory transfer of the covariance matrix and the first eigenvalue.

Image	Metric	Host	GPU	Host/iteration	GPU/iteration
Op20C1	**Time (ms)**	268.81	426.66	26.88	42.67
Op20C1	**Speed-up**	x1	x0.63	x1	x0.63
Op8C2	**Time (ms)**	262.23	389.54	29.13	43.28
Op8C2	**Speed-up**	x1	x0.67	x1	x0.67

quite simple, the results obtained are satisfactory.

Table 7.3: Results for the CPU and GPU version of the projection.

Image	Metric	Host	GPU
Op20C1	**Time (ms)**	182.47	3.66
Op20C1	**Speed-up**	x1	x49.9
Op8C2	**Time (ms)**	246.47	7.62
Op8C2	**Speed-up**	x1	x32.34

NIPALS chain

With the NIPALS method, the second proposal to obtain the eigenvectors and eigenvalues in this work, only the first eigen-pair is calculated. The results depicted in Figure 7.5 (note the logarithmic y left axis) correspond with the execution of one iteration for every step of the algorithm (explained in Chapter 2). Recalling the four steps, the first and third carry out a matrix-vector multiplication and the second and fourth a sum reduction. In the case of the matrix-vector multiplications, the GPU obtains good results specially for the third step, in which there are $imSize$ threads performing 128 multiplications-additions in parallel. In the case of the sum reductions, it is clear that reductions with a low quantity of values like the second step (128 values) obtain better results in the host side. Nevertheless, as the amount of data increases, like in the fourth step ($imSize$ values), the reduction in the device side is worth in this case.

Taking into account the results, the first, third and fourth steps are implemented in the GPU for the final PCA implementation. The second step was finally also included in the GPU since the time difference between the host and GPU implementation is negligible compared with all the other times of the system.

Entire PCA

Finally, the results for the entire algorithm using both methods are analysed in this section in order to show the differences between them. It is important to mention that the constant defining the error for the iterative methods is in both cases $\varepsilon = 0.001$.

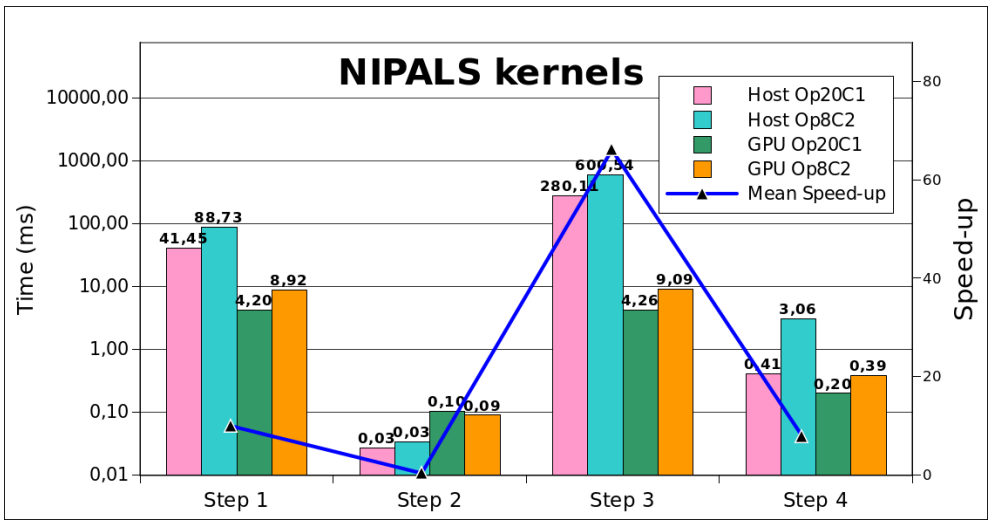


Figure 7.5: Results for the different NIPALS steps in the host and the GPU. The speed-up is in relation to the sequential implementation in the host. Note the logarithmic left y axis (representing time).

Figure 7.6 introduces the time and energy consumed by the PCA using the Jacobi and the NIPALS chain. In these results the HS image transfer, the pre-processing step and each chain are included (CUDA initialisation, file reading from disk and the memory allocation/deallocation are not considered). In addition, Table 7.4 introduces the number of iterations required for both chains. From the Jacobi chain results, some remarks can be made:

- The time results obtained are under the time constraints imposed by HELICoiD requirements by far (around 2 minutes for the entire application). However, these are partial results as the times constraints are imposed for the entire application.

- The variation between the time/energy across the different images analysed is really low taking into account that the largest image has around the double amount of pixels as the smallest one.

- The time consumed by the algorithm is not really dependant on the number of global iterations. In addition, the number of iterations is very stable and depend on the data of the image, not on the image size (remember that Jacobi is always applied to a $bands \times bands$ matrix).

- The mean power consumed by the platform while the PCA algorithm is being performed is around 9 Watts[2], only 6 Watts over the mean power while the platform

[2]The mean power for each image is calculated dividing the energy by the time required. With all the mean power values, a mean power value for all the images is calculated.

is idle in maximum performance mode. This means that the mean power needed to execute the algorithm in the platform is around 6 Watts.

- In contrast with an iterative method implemented in the GPU, the number of iterations for each image in different executions is constant because the iterative part of the algorithm, Jacobi, is done in the CPU. The case of the iterative algorithm in the GPU is explained in the NIPALS observations.

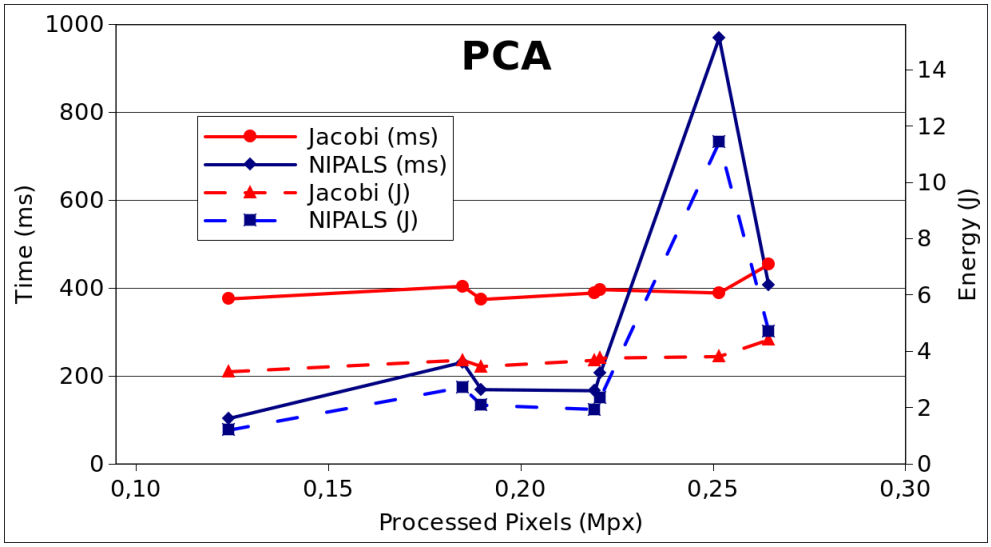


Figure 7.6: PCA results for the two chains. The x axis refers to the number of pixels, the left y axis to the processing time and the right y axis to the energy consumed.

Table 7.4: Iterations for both the Jacobi and NIPALS chains.

Image	Iterations Jacobi	Iterations NIPALS
Op20C1	10	9.5
Op15C1	9	16
Op25C2	8	11.75
Op12C1	8	9.25
Op12C2	8	11.5
Op8C1	7	54
Op8C2	9	20

The analysis of NIPALS processing chain is the following:

- Again, the time required to process all the images in the database fulfils with the time constraints of the project even better. As it can be seen, the processing time is lower in general compared with the Jacobi chain.

- The range of variation in the results is considerable. Not only among different images but also among different executions of the application for the same image. This is shown by the number of iterations, which in some cases is a not an integer number because it refers to the average of several executions. Although the input data is the same, the order in which the GPU performs the operations may vary in different executions (this issue was explained in Chapter 2). This fact causes that some additions produce different results in different executions due to rounding effects of floating point numbers. The rounding error grows as the number of iterations increases and the precision required to reach the converge is $\varepsilon = 0.001$ so the number of iterations can vary.

- The number of iterations affects the processing time heavily. This makes sense since most of the algorithm is iterative (except for the pre-processing stage).

- Calculating the mean power for all the images processed through the NIPALS chain results in almost 12 Watts, which is 9 Watts over the idle power consumption and 3 Watts more than the Jacobi chain. However, although the mean power is larger, the total energy used is lower than with Jacobi (since computing time is lower).

- For the last two images, which are the largest ones, the number of iterations is larger than in all the others. This may suggest a tendency for the the number of iterations to increase with the size of the image.

It is important to analyse both methods within their context. While Jacobi is calculating 128 eigen-pairs, NIPALS is calculating only the first one. Even so, against all odds, the time and energy required for both chains is quite similar, being the NIPALS chain even worse in some cases. Figure 7.7 shows the behaviour of the algorithm taking into account the number of processed pixels (the spatial dimension) per second and the number of pixels processed per Joule. As explained before, the Jacobi chain stands for a stable method which slowly follows the increment of processed pixels and with a worse use of the energy, in general, compared with the NIPALS chain. The second chain (NIPALS) is able to obtain a greater number of processed pixels per Joule, however, its behaviour is more erratic and for larger images it seems to lose efficiency compared to Jacobi.

7.3.2. SVM

The SVM implementation, used for the HS image pixel-wise classification, has been tested and measured in terms of time and energy for all HELICoiD database images. Results are represented in Figure 7.8 and Figure 7.9.

In this case, the algorithm achieves better results than PCA in terms of processed pixels per second and it seems to improve as the number of pixels increases. Nevertheless, it is

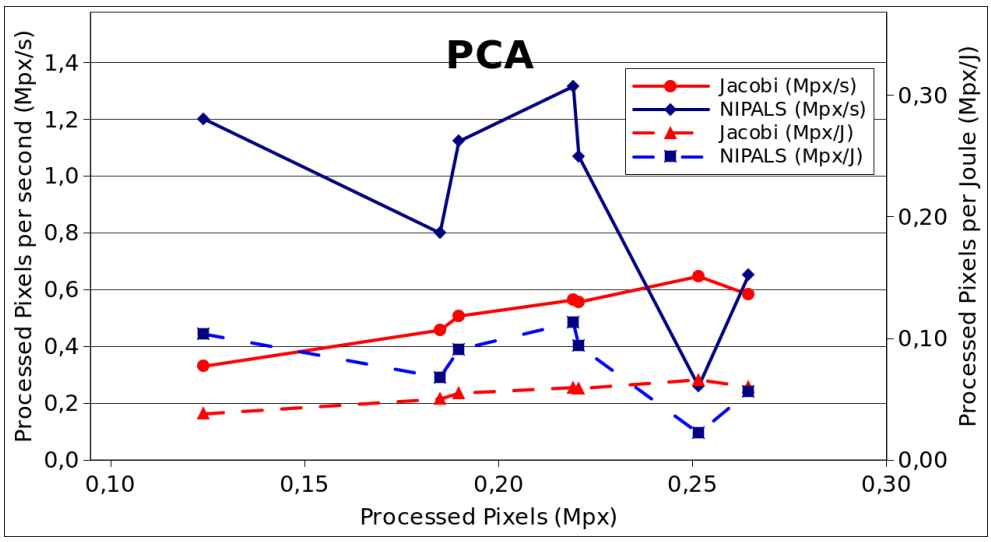


Figure 7.7: PCA results for the two chains. The x axis refers to the number of pixels, the left y axis to the processed pixels per second and the right y axis to the processed axis per Joule.

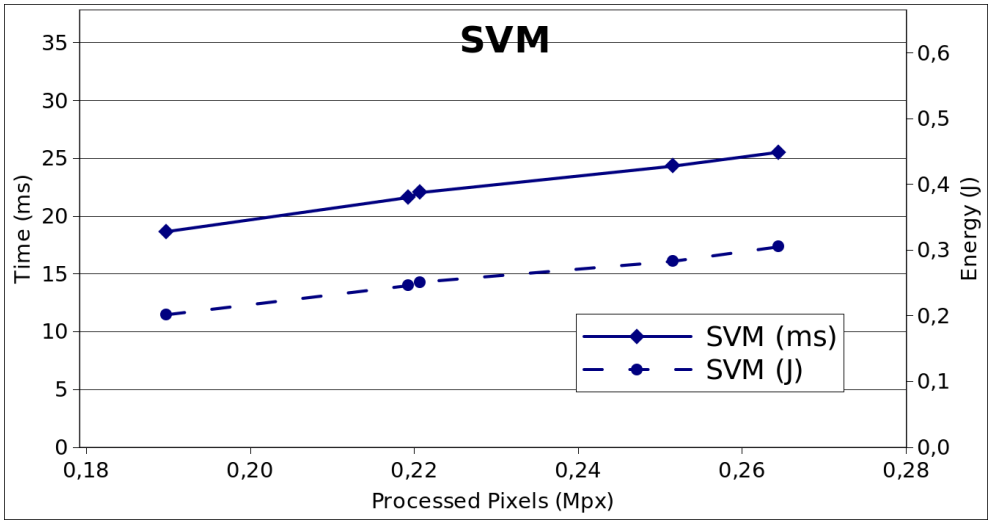


Figure 7.8: SVM results. The x axis refers to the number of pixels, the left y axis to the time and the right y axis to energy consumed.

totally the opposite in terms of energy consumption, as the execution gains throughput, energy efficiency is lost.

7.3.3. KNN

The last algorithm results are depicted in Figure 7.10 and Figure 7.11. Looking at the results, it can be seen that the KNN features the highest computational load. In addition, throughput and efficiency scale well with the number of pixels, which is the desired behaviour. This means that neither a compute or communication bound has been reached.

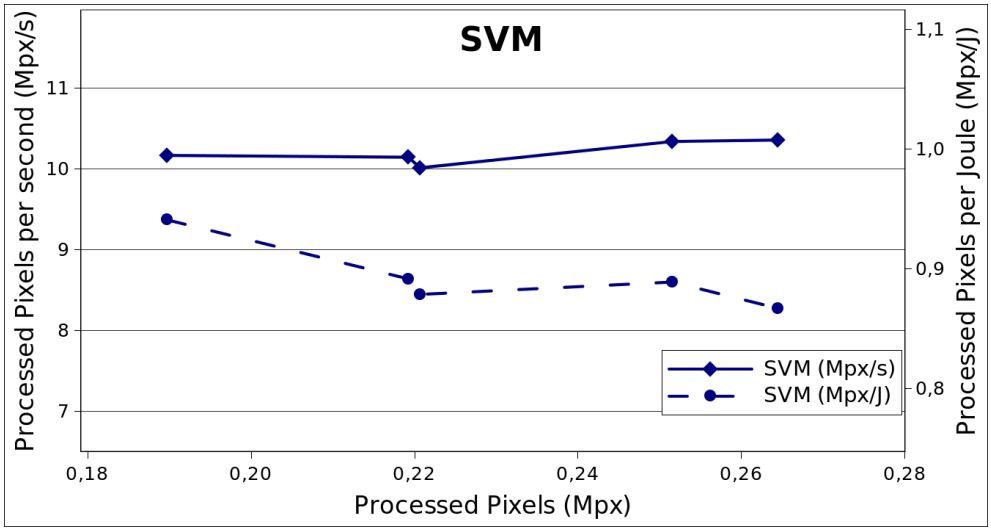


Figure 7.9: SVM results. The x axis refers to the number of pixels, the left y axis to the processed pixels per second and the right y axis to the processed axis per Joule.

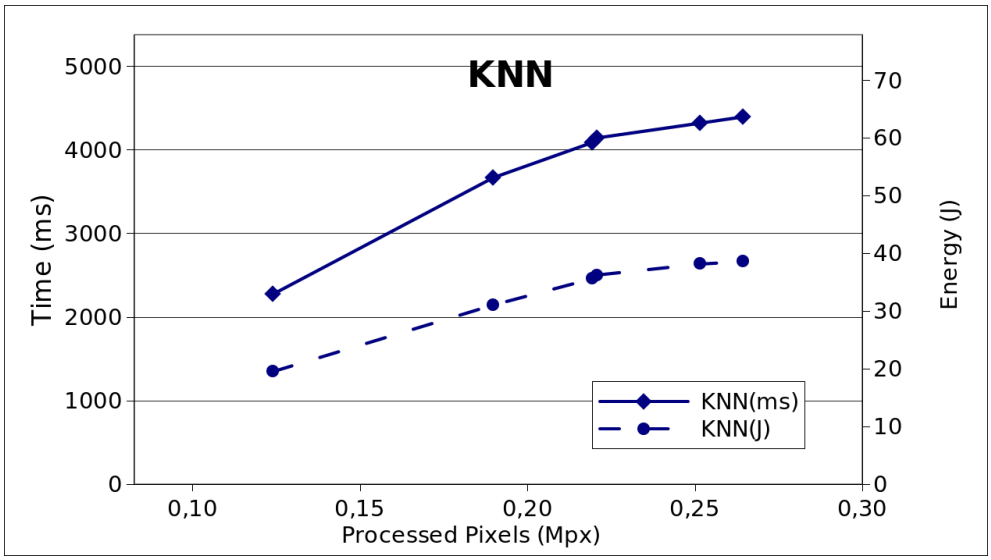


Figure 7.10: KNN results. The x axis refers to the number of pixels, the left y axis to the time and the right y axis to the energy consumed.

7.3.4. Global chain: PCA + SVM + KNN

In this section, the whole application is characterised in terms of throughput and efficiency. The three algorithm implementations are first compared in two charts in Figures 7.12 and 7.13 in terms of throughput (px/s) and efficiency (px/J), respectively. It is clearly seen how the SVM throughput and efficiency are the best among the three, followed by the PCA and then by the KNN. This was to be expected as the SVM computation patterns are the most regular of the three, and hence the available resources are more and better efficiently exploited. In addition, the three algorithms are quite stable in the case of using

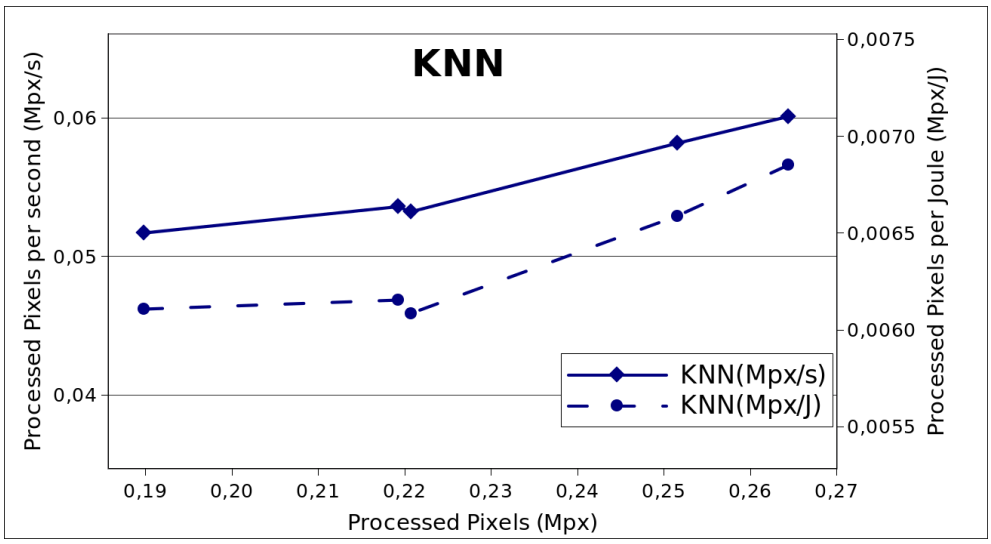


Figure 7.11: KNN results. The x axis refers to the number of pixels, the left y axis to the processed pixels per second and the right y axis to the processed axis per Joule.

Jacobi in the PCA algorithm.

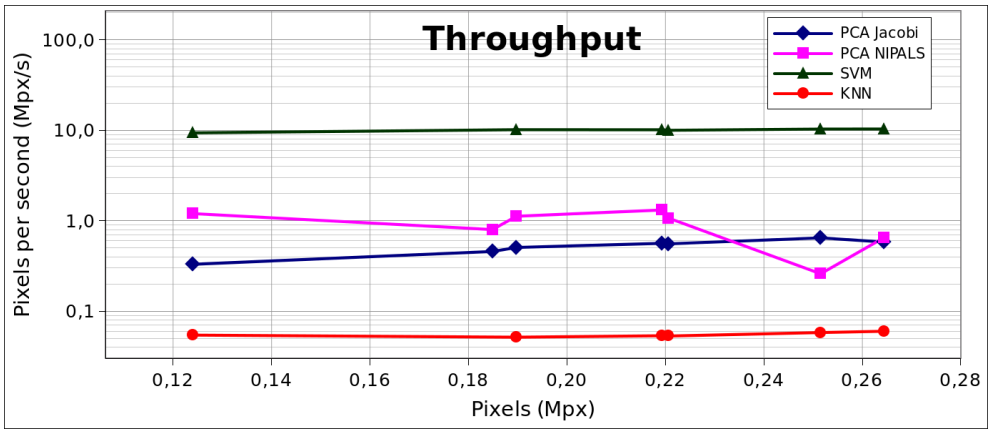


Figure 7.12: Comparison of the three algorithms in terms of throughput.

The charts in Figures 7.14 and 7.15 introduce the global results after the integration of the 3 algorithms into the complete chain without and with normalisation, respectively. These times include variable allocation, processing, memory transfers and memory deallocation.

From Figure 7.14, it can be extracted that the time needed to process all the images in the HELICoiD database goes from 2 to 6 seconds, accomplishing the time constraints of the project. Regarding energy, results vary from 20 to 50 Joules, increasing with the number of processed pixels. As it can be seen, the difference between using Jacobi or NIPALS in the PCA stage are not very relevant in terms of energy or time as they are quite similar for most images.

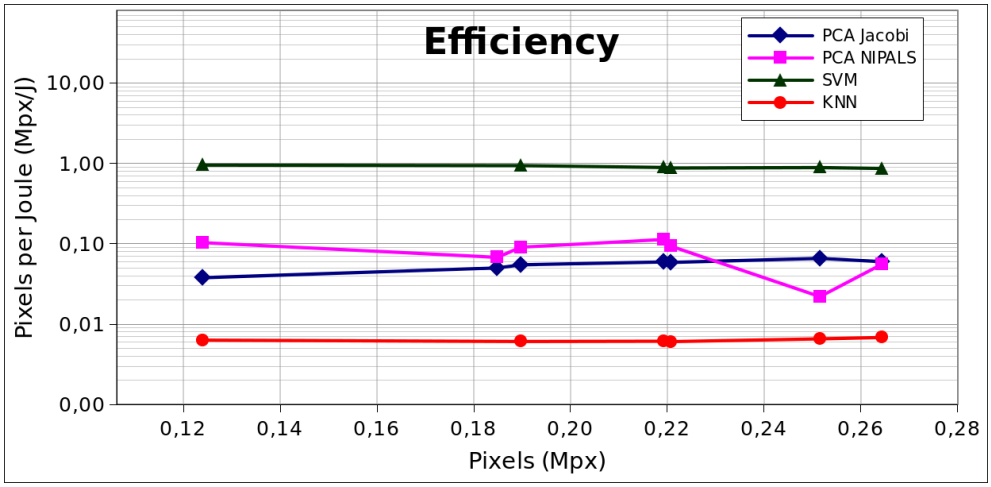


Figure 7.13: Comparison of the three algorithms in terms of efficiency.

In the case of Figure 7.15, which contains normalised time and energy, it can be observed that results in both time and energy are similar to those obtained in the KNN stage in Figure 7.11 (note the y axes in Kpx/s Kpx/J, not in Mpx/s Mpx/J). From this result, it can be concluded that the KNN algorithm is the bottleneck of the application.

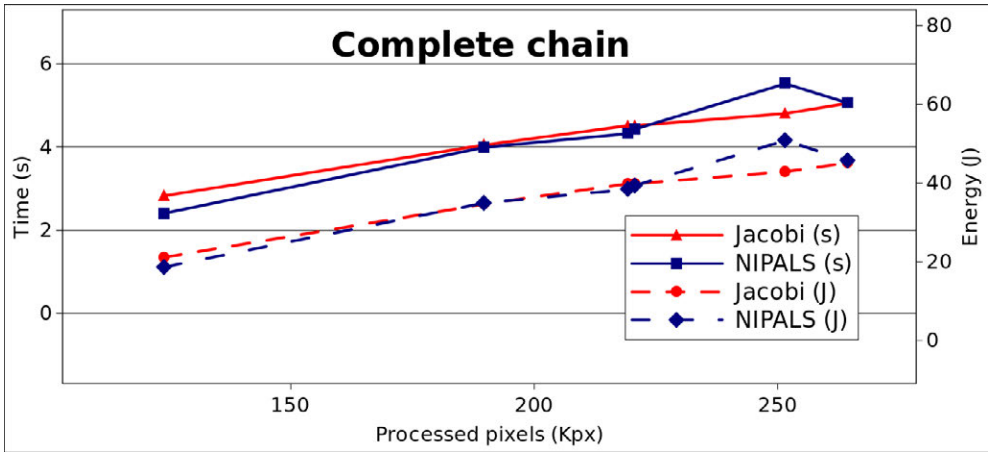


Figure 7.14: Time and energy results for the original project.

Finally, an average for the normalised throughput and efficiency results is included in Table 7.5. As discussed, the difference between Jacobi and Nipals is negligible.

Table 7.5: Throughput and efficiency results for the complete chain of the original algorithm, for both the Jacobi and Nipals chains.

PCA Chain	Throughput (Kpx/s)	Efficiency (Kpx/J)
Jacobi	47.87	6.12
Nipals	48.94	6.06

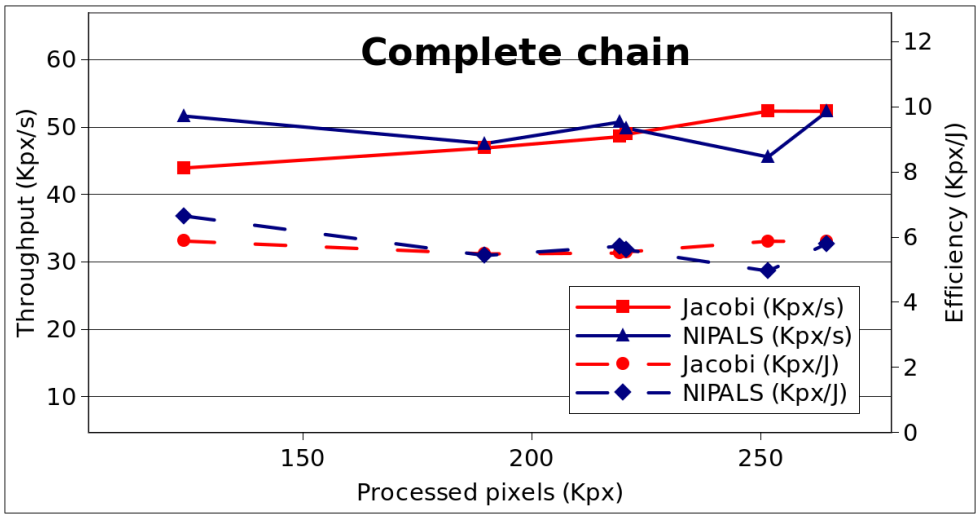


Figure 7.15: Time and energy normalised results for the original project.

7.4. Alternative Project

7.4.1. Filter

This section contains the results for the two kernels proposed for the spatial filter as shown in Table 7.6. It can be seen that the filtering process is an amenable operation to be accelerated in a GPU, giving the good speed-up factors obtained. Nevertheless, the time needed for the mirror borders approach is around the double as the one needed for the naive solution without considering the borders. Since the process is the same but introducing several `if` cases, results show how the latency introduced by the divergent branches affects performance (even when the wrap divergence only occurs on the image borders).

Table 7.6: Partial results for the two different filter approaches.

Image	Metric	Host No Borders	GPU No borders	Host Mirror	GPU Mirror
Op20C1	**Time (ms)**	68.95	3.53	257.53	6.56
Op20C1	**Speed-up**	x1	x19.53	x1	x39.26
Op8C2	**Time (ms)**	145.29	6.87	554.10	13.014
Op8C2	**Speed-up**	x1	x21.75	x1	x42.58

From the window size of the filter set to 7×7 pixels as introduced in Chapter 4, it can be extracted that the mirror filter is doubling the processing time for just improving a 3-pixel ($floor(7/2)$) frame of an image with more than 300 pixels in each dimension. As a consequence, the mirror filter is dismissed.

Taking this into account, the results for the spatial filter algorithm implementation are included in Figure 7.16 and Figure 7.17. From them, it can be observed how a good throughput and efficiency compared with all the other algorithms in the project is obtained.

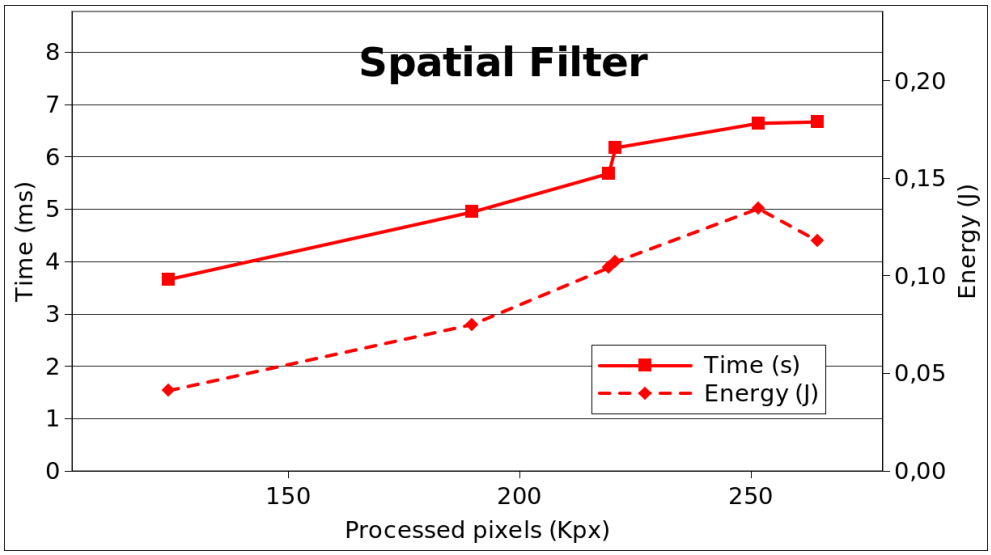


Figure 7.16: Spatial filter results. The x axis refers to the number of pixels, the left y axis to the time and the right y axis to the energy.

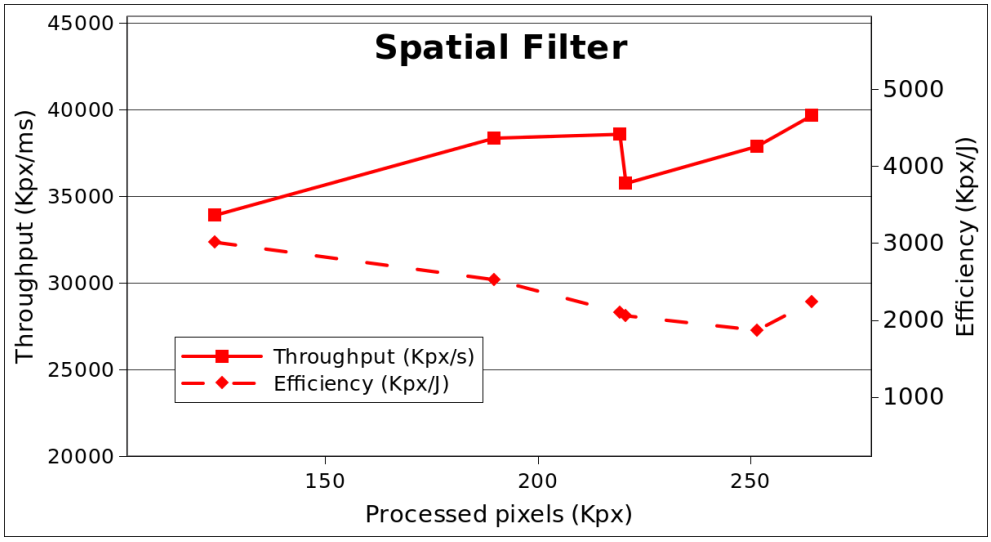


Figure 7.17: Spatial filter results. The x axis refers to the number of pixels, the left y axis to the processed pixels per second and the right y axis to the processed axis per Joule.

7.4.2. Global chain: SVM + Spatial filter

As seen in Chapter 2 the global alternative project is composed of the SVM algorithm, whose results are the same as these in subsection 7.3.2 and the spatial filter. Like in the

previous section, the charts for the global alternative project in Figure 7.18 and Figure 7.19 show the time/energy and throughput/efficiency results for the entire application, i.e., allocating variables, algorithm processing, transferring memory and deallocating variables.

From the figures it can be extracted that this alternative project implementation do accomplishes the time constraints imposed and that the processing time is totally dependent on the number of pixels, as the almost linear increase in the curve in Figure 7.18 shows. Due to this fact, the efficiency and throughput are represented by a mostly flat curve, indicating also that the algorithm is working as expected in the range of values shown.

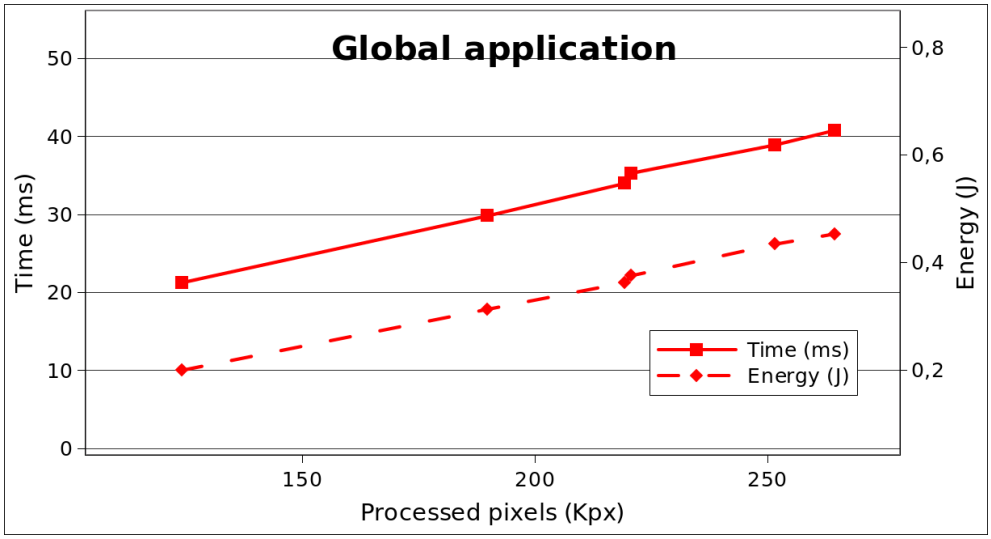


Figure 7.18: Time and energy results for the alternative project.

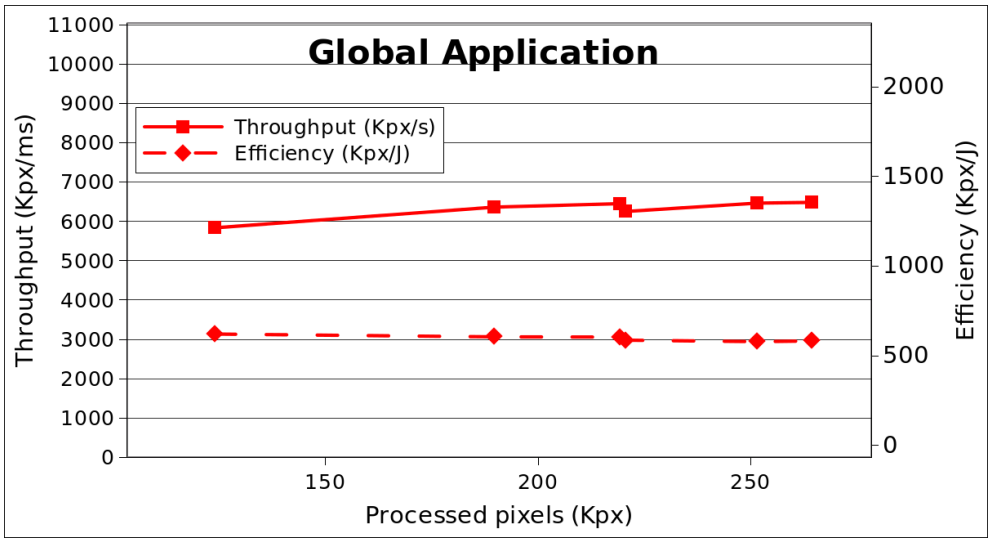


Figure 7.19: Normalised time and energy results for the alternative project.

Finally, the average results of throughput and efficiency are shown in Table 7.7.

Table 7.7: Throughput and efficiency results for the alternative algorithm.

	Throughput (Kpx/s)	**Efficiency (Kpx/J)**
Alternative Project	5902	697

7.5. Comparison between chains

First of all, in this section, the classification maps obatained for the two implementations are compared with the original HELICoiD results which use the MPPA from Kalray. Table 7.8 reports the percentage of wrong pixels for the two reference images, Op20C1 and Op8C2. As it can be seen, the three different methods achieve a very low error rate for the two images. The different classification maps are introduced in Figure 7.20 and Figure 7.21. Those results demonstrate that even with an 8% of wrong pixels, the subjective quality of the image is still good enough. These results prove the success of the acceleration in the Jetson TX1 for the two processing chains considered.

Table 7.8: Percentage of wrong pixels compared with the MPPA final classification maps for the Op20C1 and Op8C2 images.

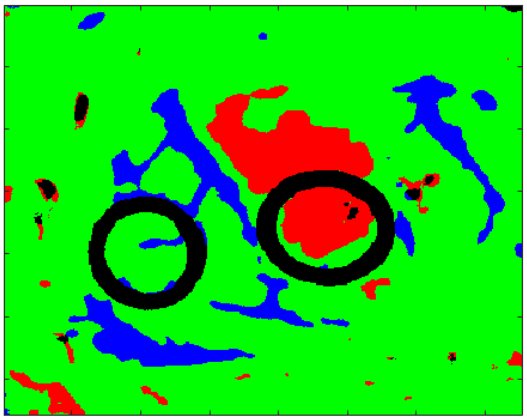
	Original: Jacobi	**Original: Nipals**	**Alternative**
Op20C1	0.82 %	0.77 %	1.06 %
Op8C2	8.05 %	8.00 %	8.11%

In order to compare the performance with the MPPA, Table 7.9 shows results for the Op20C1 HS image processing time (in seconds). The table includes processing times for each HELICoiD algorithm (PCA, SVM and KNN) and for the entire chain executed in the MPPA [25] and in the Jetson TX1. Results show that the Jetson TX1 implementations obtain similar results in the case of SVM and KNN, however, PCA is processed around ten times faster with Jacobi and thirty with NIPALS. This can be explained taking into account the high amount of linear algebra computations needed in PCA, which fit perfectly with the GPU architecture.

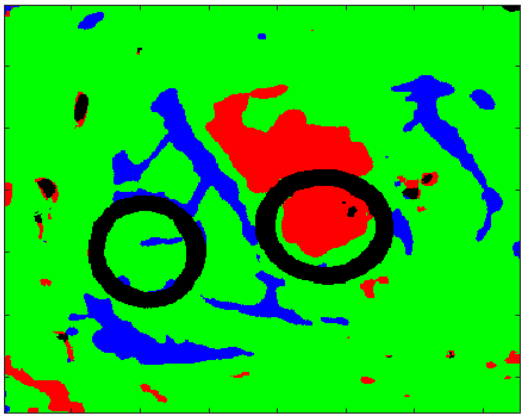
Table 7.9: Comparison between platforms (MPPA and Jetson TX1) for Op20C1 HS image (in seconds) [25].

Platform	**PCA**	**SVM**	**KNN**	**Global**
MPPA	3.78 (Jacobi)	0.06	2.46	6.24
Jetson TX1	0.37 (Jacobi) 0.10 (NIPALS)	0.01	2.28	2.82 (Jacobi) 2.40 (NIPALS)

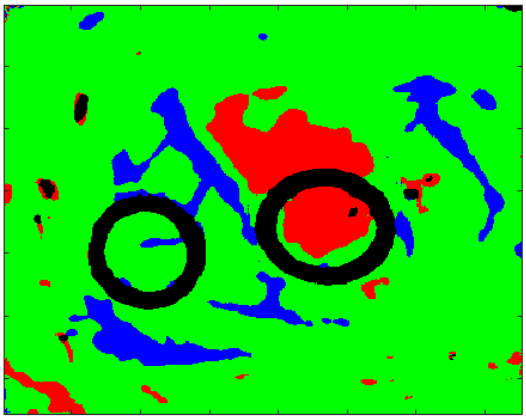
(a) Op20C1 final classification map from MPPA.



(b) Op20C1 final classification map from the original project with Jacobi.



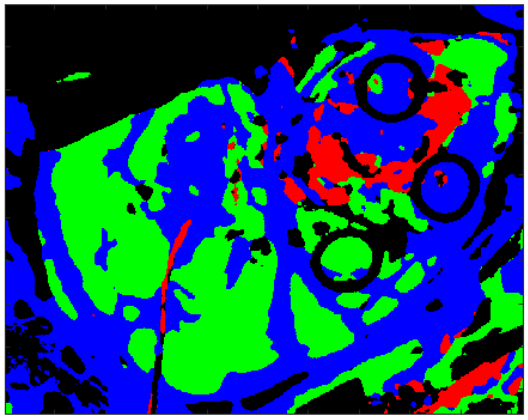
(c) Op20C1 final classification map from the original project with Nipals.



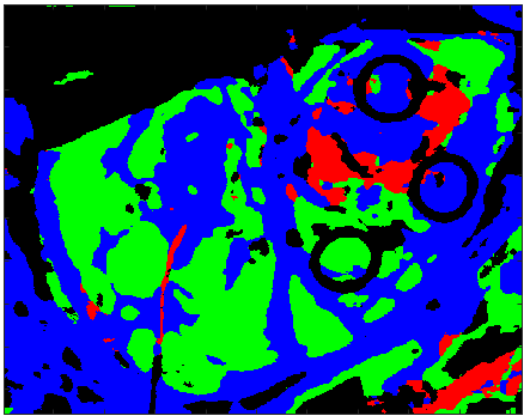
(d) Op20C1 final classification map from the alternative project based on the spatial filter.

Figure 7.20: Op20C1 final classification map for the different platforms/projects.

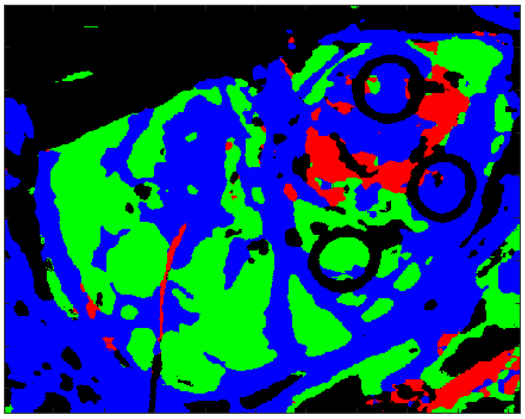
Finally, the throughput and efficiency results for both alternatives are depicted in Table 7.10. The table shows results two orders of magnitude better in terms of throughput and efficiency for the alternative algorithm. It proves that it is possible to improve the processing part of the HELICoiD project using an spatial filter.



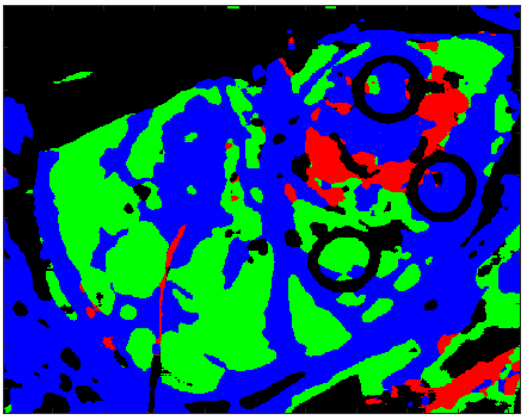
(a) Op8C2 final classification map from MPPA.



(b) Op8C2 final classification map from the original project with Jacobi.



(c) Op8C2 final classification map from the original project with Nipals.



(d) Op8C2 final classification map from the alternative project based on the spatial filter.

Figure 7.21: Op8C2 final classification map for the different platforms/projects.

Table 7.10: Throughput and efficiency comparison between the results in the original and the alternative project implementations.

	Throughput (Kpx/s)	**Efficiency (Kpx/J)**
Original Project Jacobi	47.87	6.12
Original Project NIPALS	48.94	6.06
Alternative Project	5902	697

8. Conclusions

8.1. Conclusions

This work addressed the NVIDIA Jetson TX1 characterisation in terms of throughput and energy efficiency for the European H2020 project processing chain: a set of machine learning algorithms for the automatic detection of cancerous tissue in hyperspectral images during tumour resection operations. First, the structure and algorithms included in both projects were analysed before tackling their subsequent implementation in an accelerated platform, an NVIDIA embedded GPU in a Jetson TX1 board. Then, the platform was deeply studied to understand its internal functioning and programming using the CUDA C/C++ language. With the ideas from the previous analysis, the acceleration of the PCA algorithm and the spatial filter were made. These implementations along with the SVM and KNN made by other students previously, were used to integrate the two final applications: the original processing chain from HELICoiD and an alternative chain based on a spatial filter. Finally, the implementations were evaluated in terms of execution time and energy. While time measurement was made using a ready to use CUDA library, however, energy measurement was done through a library specifically developed in this work along with a real time power graphic monitor.

This section contains the conclusions extracted from the aforementioned work:

- It is possible to obtain a good acceleration in a GPU when performing massive independent parallel operations. The results showed that operations which involve a low parallelism (in the order of hundreds of threads) are not worth to be parallelised.
- Reductions are a very good and highly used method to get advantage of the GPU parallelism when accelerating operations in which race conditions appear.
- Most of the linear algebra operators such as matrix-vector multiplication or matrix-matrix multiplication are good candidates to be parallelised using a GPU.
- The low parallelism intrinsic to the Jacobi method makes its acceleration in a GPU-based platform really challenging, up to a point that probably a different method is needed.
- It is advisable the study of optimised libraries to perform some of the involved operations, however, some times a custom solution can offer better results.
- Since the execution order within the GPU is determined by the GPU runtime, the results in calculations that involve multiple floating point operations can be different in different executions due to the rounding effects.
- The range of power used by the Jetson TX1 platform is 3–18 Watts according to NVIDIA, which has been confirmed by the tests made in this work. This means that

an application using the maximum power per second only consumes 6 times more than the platform in idle state. This low variation between the power consumption range increases the importance of improving the other factor involved in energy consumption: time. Therefore, it can be concluded that the processing time is a really important factor in the Jetson TX1 platform and that it is highly probable that energy optimisation means time optimisation.

- For the PCA, both the Jacobi chain and the NIPALS chain offer a good solution for the HELICoiD project, fulfilling the time constraints. While the former stands for a stable implementation which obtains a processing time with low variability through the pixels processed, the latter introduces a high variability which principally depends on the data of the image. In terms of throughput and efficiency, NIPALS only obtains worse results in the case of the largest images.

- The operations involved in the SVM are mainly linear algebra operators, moreover, most of them are independent. For this reason, the results obtained are the best among the three algorithms in the original chain.

- The high number of operations required by the KNN causes that the algorithm stands for the processing bottleneck of the original chain.

- The time results obtained in this work implementation are 2.6 times faster than the implementation from HELICoiD in MPPA. This result proves that embedded GPUs and particularly the Jetson TX1 system are suitable for this application.

- The original project implementation meets the time requirements and consumes a mean power of 8 Watts during its execution. This is a good result that enables the use of fewer energy resources for embedded processing scenarios, probably at the cost of sacrificing real-time (unless more optimised implementations are obtained).

- The results in this work verify that it is possible to obtain similar results to the ones obtained in HELICoiD without taking into account the spectral information from the PCA algorithm. In addition, the throughput and efficiency results overcome in two orders of magnitude those from the original processing chain, introducing the possibility of classifying hyperspectral images in real time with a low energy cost. Although the mean power consumption of the platform while processing is 10 Watts (2 Watts above the original processing chain), the low time required for the operation implies a tiny energy use. This situation can lead to an embedded system which uses a battery in order to classify the hyperspectral images.

8.2. Future lines of work

The development and results obtained in this work open different future lines of work that are sketched in the following paragraph:

- Regarding the PCA, a future line could tackle the acceleration of the Jacobi method making use of the GPU. With the release in CUDA 9 of the so-called *cooperative groups*, a more flexible synchronization is possible and it could allow for the creation of a better solution.

- The optimisation efforts in this work did not take into account energy consumption along with the execution time optimisation. An important future line should address a paper Design Space Exploration (DSE) to find good time-energy efficiency trade-offs in the implementation. The number of threads per block and the frequency of the CPU and the GPU, among others, play an important role in the energy consumption in the Jetson TX1 platform.

- It could be possible to create an application which performs the entire HELICoiD processing chain from the acquisition to the final classification map visualisation in the NVIDIA Jetson TX1. In this case, using the alternative processing chain, the complete algorithm could achieve real time, getting this way an implementation suitable even for an embedded, portable device.

- The ability to measure the energy in real time allows for the creation of platform models that can be used to predict the energy usage. The creation of these performance-energy models is a current hot topic in the community, and the work tackled in this MSc. project situates us in a good position to follow this research direction.

- In this regard, characterisation of the platform in terms of performance can be completed using the *Roofline Model*. This tool allows to create a visual model in which every application can be compared with the maximum performance achievable by the platform. This way, it would be possible to optimise applications taking into account the hardware limitations, combined with the energy information obtained along the way during the DSE.

A. PCA: Proposed Kernels

```
__global__ void avgNaive(const float *hyperImage,
                         float *avg, int imSize, int bands) {

        int i = blockDim.x * blockIdx.x + threadIdx.x;

        for (int j = 0; j < imSize; j++) {
                avg[i] += hyperImage[j + i * imSize];
        }
        avg[i] = avg[i] / (imSize);
}
```

Figure A.1: Naive average.

```
__global__ void avgAtomics(float *hyperImage,
                         float *avg, int imSize, int bands){

        int i = blockDim.x * blockIdx.x + threadIdx.x;

        atomicAdd(&avg[i / (imSize)], hyperImage[i] / (imSize));
}
```

Figure A.2: Atomics average.

```
__global__ void avgReductionShared(float* vect,
                        float* avg, int imSize, int bands) {

        int tid = threadIdx.x;
        int warpSize = 32;

        float sum = 0.0;
        __shared__ float sdata[1024];

        for(int j = threadIdx.x;j<imSize;j+=blockDim.x){
                sum += vect[j+blockIdx.x*imSize];
        }

        __syncthreads();

  if(tid < blockDim.x)
          sdata[tid] = sum;
  else
          sdata[tid] = 0.0;

 __syncthreads();

 for(int s = blockDim.x / 2; s > 0; s = s/2){

          if(tid < s)
                  sdata[tid] += sdata[tid + s];

          __syncthreads();
  }

  if(tid == 0){
          avg[blockIdx.x] = sdata[tid];
  }
}
```

Figure A.3: Shared memory reduction average.

```
__global__ void avgReductionWarp(float* vect,
                        float* avg, int imSize, int bands) {

        int warpSize = 32;

        float sum = 0.0;

        for(int j = threadIdx.x;j<imSize;j+=blockDim.x){
                sum += vect[j+blockIdx.x*imSize];
        }

        __syncthreads();

        for(int j = warpSize/2; j>0; j/=2){
                sum += __shfl_down(sum,j);
        }

        if(threadIdx.x % warpSize == 0)
                atomicAdd(&avg[blockIdx.x], sum);

        __syncthreads();

        if(threadIdx.x == 0)
                avg[blockIdx.x] /= imSize;

}
```

Figure A.4: Reduction warp shuffle average.

```
__global__ void varVectDevice(float* const  hyperImage,
         float* const __restrict__  avg,
         float*  __restrict__ var,int imSize, int bands) {

        int i = blockDim.x * blockIdx.x + threadIdx.x;

        var[i] = hyperImage[i] - avg[i / (imSize)];

}
```

Figure A.5: Device variance.

```
__global__ void transDevice(float* const __restrict__ var,
         float*  __restrict__ vart, int imSize, int bands) {

        int i = blockDim.x * blockIdx.x + threadIdx.x;
        int width = imSize;
        int widtht = bands;

        if (i < width * widtht) {
                At[(i % width) * widtht + i / width] = A[i];
        }
}
```

Figure A.6: Device matrix transposition.

```
__global__ void multGPU(float* const __restrict__ A,
        float* const __restrict__ B, float*  __restrict__  C,
        int aRows, int aCols, int bRows, int bCols,
        int cRows, int cCols) {

        float cValue = 0;
        int row = blockIdx.y * TILE_DIM + threadIdx.y;
        int col = blockIdx.x * TILE_DIM + threadIdx.x;

        __shared__ float As[TILE_DIM][TILE_DIM];
        __shared__ float Bs[TILE_DIM][TILE_DIM];

        for (int k = 0; k < (TILE_DIM + aCols - 1) / TILE_DIM; k++) {

                if (k * TILE_DIM + threadIdx.x < aCols && row < aRows)
                        As[threadIdx.y][threadIdx.x] =
                                A[row * aCols + k * TILE_DIM
                                + threadIdx.x];
                else
                        As[threadIdx.y][threadIdx.x] = 0.0;

                if (k * TILE_DIM + threadIdx.y < bRows && col < bCols)
                        Bs[threadIdx.y][threadIdx.x] =
                                B[(k * TILE_DIM + threadIdx.y) * bCols
                                + col];
                else
                        Bs[threadIdx.y][threadIdx.x] = 0.0;

                __syncthreads();

                for (int n = 0; n < TILE_DIM; ++n)
                        cValue +=
                        As[threadIdx.y][n] * Bs[n][threadIdx.x];

                __syncthreads();

        }

        if (row < cRows && col < cCols)
                C[((blockIdx.y * blockDim.y + threadIdx.y) * cCols)
                 + (blockIdx.x * blockDim.x) + threadIdx.x] = cValue;

}
```

Figure A.7: Device matrix multiplication.

```
__global__ void transMultGPU(float* const __restrict__ var,
                             float * __restrict__ cov,
                              int imSize, int bands) {

        float cValue = 0;
        int row = blockIdx.y * TILE_DIM + threadIdx.y;
        int col = blockIdx.x * TILE_DIM + threadIdx.x;

        __shared__ float As[TILE_DIM + 1][TILE_DIM + 1];
        __shared__ float Bs[TILE_DIM + 1][TILE_DIM + 1];

        for (int k = 0; k < (TILE_DIM + imSize - 1) / TILE_DIM; k++) {

                if (k * TILE_DIM + threadIdx.x < imSize && row < bands)
                        As[threadIdx.y][threadIdx.x] =
                                var[threadIdx.x
                                + threadIdx.y * imSize
                                + k * TILE_DIM
                                + blockIdx.x * imSize * TILE_DIM];
                else
                        As[threadIdx.y][threadIdx.x] = 0.0;

                if (k * TILE_DIM + threadIdx.y < imSize && row < bands)
                        Bs[threadIdx.y][threadIdx.x] =
                                var[threadIdx.x
                                + threadIdx.y * imSize
                                + k * TILE_DIM
                                + blockIdx.y * imSize * TILE_DIM];
                else
                        Bs[threadIdx.y][threadIdx.x] = 0.0;

                __syncthreads();

                for (int n = 0; n < TILE_DIM; ++n)
                        cValue +=
                        As[threadIdx.x][n] * Bs[threadIdx.y][n];

                __syncthreads();

        }

        if (row < bands && col < bands)
                cov[((blockIdx.y * blockDim.y + threadIdx.y) * bands)
                + (blockIdx.x * blockDim.x) + threadIdx.x] = cValue;
}
```

Figure A.8: Device transposition + multiplication.

```
1  void transMultiplycuBLAS(cublasHandle_t cublas_handle, float *var_d,
2          float *covM_d, int imSize, int bands ){
3
4          float alpha = 1.0f;
5          float beta = 0;
6          /* C = alpha * op(A) * op(B) + beta * C */
7          cublas_status = cublasSgemm(
8                  cublas_handle,  // library context
9                  CUBLAS_OP_T,    // operation op(A)
10                 CUBLAS_OP_N,    // operation op(B)
11                 bands,          // m : #rows of A and C
12                 bands,          // n : #columns of B and C
13                 imSize,         // k : #columns of A and #rows B
14                 &alpha,         // alpha : scalar
15                 var_d,          // A
16                 imSize,         // lda : leading dimension  matrix A
17                 var_d,          // B
18                 imSize,         // ldb : leading dimension  matrix B
19                 &beta,          // beta : scalar
20                 covM_d,         // C
21                 bands           // ldc : leading dimension matrix C
22                 );
23 }
```

Figure A.9: CuBLAS transposition + multiplication.

```
1  void jacobiHost(float *covM_h,
2          float *rotMatrix_h, int imSize, int bands){
3
4          float theta,t,c,s,sign,wiz;
5          int element,p,q;
6          int counter = 0;
7          int width = bands;
8
9          for(int n = 0;n<MAXIT;n++){
10           counter = 0;
11           for(int j = 0;j<bands;j++){
12             for(int k = j+1;k<bands;k++){
13
14               // First iteration diagonal of rot set to 1
15                   if (n == 0 && j == 0 && k == 1){
16                     for(int i = 0;i<width;i++)
17                           rotMatrix_h[i * width + i] = 1;
18                   }
19
20                   // Equal symmetric elements
21                   element = j * width + k;
22
23                   // Rotation parameters
24                   p = element / width;
25                   q = element % width;
26
27                   covM_h[q * width + p] = covM_h[p * width + q];
28
29                   theta = (covM_h[q * width + q]
30                                 - covM_h[p * width + p])
31                                  / (2 * covM_h[p * width + q]);
32                   if (theta > 0)
33                           sign = 1.0;
34                   else
35                           sign = -1.0;
36
37                   t = sign /
38                   (fabsf(theta) + sqrtf((powf(theta, 2) + 1)));
39                   c = 1 / (sqrtf((powf(t, 2) + 1)));
40                   s = c * t;
41
42                   // The first iteration the rotation matrix is just the
43                   // first rotation matrix with the ones in the diagonal
44                   if (n == 0 && j == 0 && k == 1) {
45                     rotMatrix_h[p * width + p] = c;
46                         rotMatrix_h[q * width + q] = c;
47                         rotMatrix_h[p * width + q] = s;
48                         rotMatrix_h[q * width + p] = -s;
49
50                   }
51                   // If the absolute value of the element is lower
52                   // than the threshold the counter adds one.
53                   // This is used to stop the jacobi Algorithm when
54                   // every non-diagonal element in the matrix is
55                   //lower than the threshold
56                   if (fabsf(covM_h[element]) < THRESHOLD) {
57                         counter++;
58                   }
```

Figure A.10: Jacobi host function. Part 1.

```
1
2
3                   /* EIGENVECTORS */
4                   // If the iteration is not the first, the rotation
5                   // matrix is calculated just changing
6                   // the p and q column.
7                   if (!(n == 0 && j == 0 && k == 1)) {
8                   for(int i = 0;i<width;i++){
9                         wiz = rotMatrix_h[i * width + p];
10                        rotMatrix_h[i * width + p] =
11                                rotMatrix_h[i * width + p] * c
12                                + rotMatrix_h[i * width + q] * (-s);
13                         rotMatrix_h[i * width + q] = wiz * s
14                                + rotMatrix_h[i * width + q] * c;
15                          }
16                  }
17
18                  /* EIGENVALUES AND ITERATION */
19                  // First multiplication of the matrix:
20                  // AUX = T' * COV;
21                  // Only the p and q rows changes
22                  for(int i = 0;i<width;i++){
23                        wiz = covM_h[p * width + i];
24                        covM_h[p * width + i] = c * covM_h[p * width + i]
25                                + (-s) * covM_h[q* width + i];
26                        covM_h[q * width + i] = s * wiz
27                                + c * covM_h[q * width + i];
28                  }
29
30                  // Second multiplication of the matrix:
31                  //     COV2 = AUX * T;
32                  // Only the p and q columns changes
33                  for(int i = 0;i<width;i++){
34                        wiz = covM_h[i * width + p];
35                        covM_h[i * width + p] = c * covM_h[i * width + p]
36                                + (-s) * covM_h[i * width + q];
37                        covM_h[i * width + q] = s * wiz
38                                + c * covM_h[i * width + q];
39                  }
40
41
42                // If the counter reaches the number of the elements
43                // on the upper part of the matrix jacobi stops.
44                if (counter == width / 2 * (width - 1)) {
45                        printf("Jacobi stopped at %d iteration\n", n);
46                        break;
47                }
48
49          }
50  }
```

Figure A.11: Jacobi host function. Part 2.

```
__global__ void jacobiKernel(float*  __restrict__ cov,
         float*  __restrict__ aux,
         float*  __restrict__ rotMatrix, int width) {

        int i = blockDim.x * blockIdx.x + threadIdx.x;
        // Since the kernel only uses one block, the shared memory is used
        __shared__ int element[1], p[1], q[1], counter[1];
        __shared__ float t[1], c[1], s[1], theta[1], sign[1];
        float wiz;
        for (int n = 0; n < MAXIT; n++) {
         if (i == 0)
          counter[0] = 0;
         for (int j = 0; j < width; j++) {
          for (int k = j + 1; k < width; k++) {

                  if (n == 0 && j == 0 && k == 1)
                          rotMatrix[i * width + i] = 1;

                 // The first thread calculates the parameters
                 // of the rotation matrix and things that only
              // are required one time
                 // the rest of the threads are
                 // used to multiply efficiently
                  if (i == 0) {

                          element[0] = j * width + k;

                          p[0] = element[0] / width;
                          q[0] = element[0] % width;

                         cov[q[0] * width + p[0]] =
                                 cov[p[0] * width + q[0]];


                         theta[0] = (cov[q[0] * width + q[0]]
                                  - cov[p[0] * width + p[0]])
                                   / (2 * cov[p[0] * width + q[0]]);
                         if (theta[0] > 0)
                                  sign[0] = 1.0;
                     else
                                  sign[0] = -1.0;

                        t[0] = sign[0] /
                        (fabsf(theta[0])
                        + sqrtf((powf(theta[0], 2) + 1)));

                        c[0] = 1 / (sqrtf((powf(t[0], 2) + 1)));
                        s[0] = c[0] * t[0];

                        if (n == 0 && j == 0 && k == 1) {
                                rotMatrix[p[0] * width + p[0]] = c[0];
                                rotMatrix[q[0] * width + q[0]] = c[0];
                                rotMatrix[p[0] * width + q[0]] = s[0];
                                rotMatrix[q[0] * width + p[0]] = -s[0];

                         }
```

Figure A.12: Jacobi device function. Part 1.

```
                        // If the absolute value of the element
                        // is lower than the threshold
                        // the counter adds one. Stop condition
                        if (fabsf(cov[element[0]]) < THRESHOLD) {
                                counter[0]++;
                        }
                }

                __syncthreads();

                /* EIGENVECTORS */
                // If the iteration is not the first,
                // the rotation matrix
                // is calculated just changing the p and q column.
                if (!(n == 0 && j == 0 && k == 1)) {
                        wiz = rotMatrix[i * width + p[0]];
                        rotMatrix[i * width + p[0]] =
                        rotMatrix[i * width + p[0]] * c[0]
                               + rotMatrix[i * width + q[0]] * (-s[0]);
                        rotMatrix[i * width + q[0]] = wiz * s[0]
                                + rotMatrix[i * width + q[0]] * c[0];
                }

                /* EIGENVALUES AND ITERATION */
                // First multiplication of the matrix:
                // AUX = T' * COV;    Only the p and q rows changes
                wiz = cov[p[0] * width + i];
                cov[p[0] * width + i] = c[0] * cov[p[0] * width + i]
                        + (-s[0]) * cov[q[0] * width + i];
                cov[q[0] * width + i] = s[0] * wiz
                        + c[0] * cov[q[0] * width + i];

                __syncthreads();

                // Second multiplication of the matrix:
                // COV2 = AUX * T; Only the p and q columns changes
                wiz = cov[i * width + p[0]];
                cov[i * width + p[0]] = c[0] * cov[i * width + p[0]]
                        + (-s[0]) * cov[i * width + q[0]];
                cov[i * width + q[0]] = s[0] * wiz
                        + c[0] * cov[i * width + q[0]];

                __syncthreads();
          }
        }

        // If the counter reaches the number  of the elements on
        // the upper part of the matrix Jacobi stops
        if (counter[0] == width / 2 * (width - 1)) {
                if (i == 0)
                        printf("Jacobi stopped at %d iteration\n", n);
                break;
         }
        }
}
```

Figure A.13: Jacobi device function. Part 2.

```
__global__ void projection(float *var, float *eigV,
 float *projection, int colEigv, int imSize, int bands) {

        int i = blockDim.x * blockIdx.x + threadIdx.x;

        float sum = 0;
        if (i < (imSize)) {
                #pragma unroll
                for (int j = 0; j < bands; j++)
                        sum += var[i + j * imSize] * eigV[j * bands
                                                + colEigv];
                projection[i] = sum;
        }

}
```

Figure A.14: Projection device function.

```
__global__ void step1(float* const __restrict__ R,
   float* const __restrict__ T, float* const __restrict__ P,
     int width, int bands) {

        int i = blockDim.x * blockIdx.x + threadIdx.x;

        float sum = 0;
        if (i < imSize) {
        #pragma unroll
                for (int j = 0; j < imSize; j++)
                        sum += R[j + i * imSize] * T[j];
                P[i] = sum;
        }
}
```

Figure A.15: Nipals Step 1 naive device function.

```
__global__ void step1v2(float* const __restrict__ R,
 float* const __restrict__ T, float* const __restrict__ P,
  int imSize, int bands) {

        int band = blockIdx.x;
        int offset = threadIdx.x;
        int size = imSize;

        float sum = 0.;

        // Loop 1 - Iterate block through image
        for (int px = offset; px < size; px += blockDim.x)
                sum += R[band * size + px] * T[px];

        // Copy to shared memory
        __shared__ float to_reduce[1024];
        to_reduce[offset] = sum;
        __syncthreads();

        // Loop 2 - Reduce block
        for (int i = blockDim.x/2; i > 0; i /= 2) {
                if (offset > i)
                        return;

                to_reduce[offset] += to_reduce[offset + i];
                __syncthreads();
        }

        if (offset == 0) {
                P[band] = to_reduce[0];
        }
}
```

Figure A.16: Nipals Step 1 device function.

```
__global__ void step2Part1(const float *P, float *sum) {

        int i = blockDim.x * blockIdx.x + threadIdx.x;
        int tid = threadIdx.x;

    __shared__ float sdata[BANDS];

    if(i < BANDS){
        if(tid < 1024)
                sdata[tid] = powf(P[i], 2);
        else
                sdata[tid] = 0.0;

          __syncthreads();

          for(int s = blockDim.x / 2; s > 0; s = s/2){

                  if(tid < s)
                          sdata[tid] += sdata[tid + s];

                  __syncthreads();
          }

          if(tid == 0){
                  sum[0] = sdata[tid];
          }
  }
}

__global__ void step2Part2(float *P, float *sum) {

        int i = blockDim.x * blockIdx.x + threadIdx.x;

        P[i] = P[i] / sqrtf(sum[0]);

}
```

Figure A.17: Nipals Step 2 device function.

```
__global__ void step3(float* const __restrict__ R,
 float* const __restrict__ P, float* const __restrict__ T,
   int imSize, int bands) {

        int i = blockDim.x * blockIdx.x + threadIdx.x;

        float sum = 0;
        if (i < imSize) {
                #pragma unroll
                for (int j = 0; j < bands; j++)
                        sum += R[i + j * imSize] * P[j];
                T[i] = sum;
        }
}
```

Figure A.18: Nipals Step 3 device function.

```
__global__ void step4(const float *T, float *auxSum, int imSize) {

        int i = blockDim.x * blockIdx.x + threadIdx.x;
        int tid = threadIdx.x;
    __shared__ float sdata[1024];

    if(i < imSize){
        if(tid < 1024)
                sdata[tid] = powf(T[i], 2);
        else
                sdata[tid] = 0.0;

          __syncthreads();

          for(int s = blockDim.x / 2; s > 0; s = s/2){

                  if(tid < s)
                          sdata[tid] += sdata[tid + s];

                  __syncthreads();
          }

          if(tid == 0){
                  auxSum[blockIdx.x] = sdata[tid];
          }
  }
}
```

Figure A.19: Nipals Step 4 device function.

B. Spatial Filter: Proposed Kernels

```
__global__ void filter(float * const __restrict__ SVM,
 float * __restrict__ filtered, int lines, int samples,
  int imSize, int win) {

        int i = threadIdx.x + blockIdx.x*blockDim.x;
        int x = i % samples;
        int y = i / samples;
        int N = win/2;
        float sum;
        #pragma unroll
        for(int n = 0; n < CLASSES;n++){
          sum = 0.0;
          if(i < imSize){
                for(int j = x - N ; j <= (x+N) ; j++){
                        for(int k = y - N ; k <= (y+N) ; k++){
                          if(j<0 || j >= samples || k < 0 || k>= lines)
                                  sum += SVM[i + n*imSize];
                          else
                                  sum += SVM[j+k*samples + n*imSize];
                        }
                }
                filtered[i + n*imSize] = sum / (win*win);
          }
        }
}
```

Figure B.1: Filter replicating the value in the non-existing elements.

```
__global__ void filterMirror(float * const __restrict__ SVM,
 float * __restrict__ filtered, int lines,
  int samples, int imSize, int win) {

        int i = threadIdx.x + blockIdx.x*blockDim.x;
        int x = i % samples;
        int y = i / samples;

        int N = win/2;
        float sum;
        for(int n = 0; n < CLASSES;n++){
          sum = 0.0;
          if(i < imSize){
                for(int j = x - N ; j <= (x+N) ; j++){
                  for(int k = y - N ; k <= (y+N) ; k++){
                  // Right upper corner
                  if(j >= samples && k < 0)
                        sum +=
                         SVM[(samples-j+x)+(-k+y)*samples + n*imSize];
                  // Right lower corner
                  else if(j >= samples && k >= lines)
                        sum +=
                        SVM[(samples-j+x)+(lines-k+y)*samples + n*imSize];
                  // Left lower corner
                  else if(j < 0 && k >= lines)
                        sum +=
                         SVM[(-j+x)+(lines-k+y)*samples + n*imSize];
                  // Left upper corner
                  else if(j < 0 && k < 0)
                        sum += SVM[(-j+x)+(-k+y)*samples + n*imSize];
                  // Right side
                  else if(j >= samples)
                        sum +=
                        SVM[(samples-j+x)+(k)*samples + n*imSize];
                  // Down side
                  else if(k >= lines)
                        sum += SVM[(j)+(lines-k+y)*samples + n*imSize];
                  // Left side
                  else if(j < 0)
                        sum += SVM[(-j+x)+(k)*samples + n*imSize];
                  // Up side
                  else if(k < 0)
                        sum += SVM[(j)+(-k+y)*samples + n*imSize];
                  // Normal case
                  else
                        sum += SVM[j+k*samples + n*imSize];
                  }
                 }
                 filtered[i + n*imSize] = sum / (win*win);
          }
    }
}
```

Figure B.2: Filter mirroring the window values in the non-existing elements.

Bibliography

[1] Mircea Andrecut. Parallel GPU implementation of iterative PCA algorithms. *Journal of Computational Biology*, 16(11):1593–1599, 2009.

[2] Jeremy Appleyard. CUDA Pro Tip: Optimize for Pointer Aliasing. August 2014. URL https://devblogs.nvidia.com/cuda-pro-tip-optimize-pointer-aliasing/.

[3] Guillermo Bermejo Casla. Diseño de un algoritmo KNN aplicado a la detección de cáncer cerebral mediante imágenes hiperespectrales, 2017. Final Degree Project, Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación (ETSIST), Universidad Politécnica de Madrid.

[4] Robert Berwick. An Idiots guide to Support vector machines (SVMs). 21:2011, 2003.

[5] O. Boujelben and M. Bahoura. FPGA implementation of an automatic wheezes detector based on MFCC and SVM. In *2016 2nd International Conference on Advanced Technologies for Signal and Image Processing (ATSIP)*, pages 647–650, March 2016.

[6] David Doxaran, Jean-Marie Froidefond, Samantha Lavender, and Patrice Castaing. Spectral signature of highly turbid waters: Application with SPOT data to quantify suspended particulate matter concentrations. *Remote Sensing of Environment*, 81 (1):149 – 161, 2002. URL http://www.sciencedirect.com/science/article/pii/S0034425701003418.

[7] Himar Fabelo, Samuel Ortega, Raquel Lazcano, Daniel Madroñal, Gustavo M. Callicó, Eduardo Juárez, Rubén Salvador, and Diederik Bulters. An Intraoperative Visualization System UsingHyperspectral Imaging to Aid in Brain Tumor Delineation. *MDPI sensors*, 2018.

[8] Himar Fabelo, Samuel Ortega, Daniele Ravi, B. Ravi Kiran, Coralia Sosa, Diederik Bulters, Gustavo M. Callicó, Harry Bulstrode, Adam Szolna, Juan F. Piñeiro, Silvester Kabwama, Daniel Madroñal, Raquel Lazcano, Aruma J-OShanahan, Sara Bisshopp, María Hernández, Abelardo Báez, Guang-Zhong Yang, Bogdan Stanciulescu, Rubén Salvador, Eduardo Juárez, and Roberto Sarmiento. Spatio-spectral classification of hyperspectral images for brain cancer detection during surgical operations. *PLOS ONE*, 13(3):1–27, 03 2018. URL https://doi.org/10.1371/journal.pone.0193721.

[9] Christian Fischer and Ioanna Kakoulli. Multispectral and hyperspectral imaging technologies in conservation: current research and potential applications. *Studies in Conservation*, 51(sup1):3–16, 2006. URL http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0193721.

[10] Mark Harris. Optimizing Parallel Reduction in CUDA. 2007. URL http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

[11] Mark Harris. How to Implement Performance Metrics in CUDA C/C++. 2012. URL https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/.

[12] Mark Harris. How to optimize data transfers in CUDA C/C++. December 2012. URL https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/.

[13] Mark Harris. How to Access Global Memory Efficiently in CUDA C/C++ Kernels. January 2013. URL https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/.

[14] Mark Harris. How to Overlap Data Transfers in CUDA C/C++. 2013. URL https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/.

[15] Mark Harris. Unified Memory in CUDA 6. November 2013. URL https://devblogs.nvidia.com/unified-memory-in-cuda-6/.

[16] Kunshan Huang, Shutao Li, Xudong Kang, and Leyuan Fang. Spectral–Spatial Hyperspectral Image Classification Based on KNN. *Sensing and Imaging*, 17(1):1, Dec 2015.

[17] Texas instruments. INA 3221 Data-Sheet. http://www.ti.com/lit/ds/symlink/ina3221.pdf.

[18] Jaime Jarrin Valencia. Manycore real time constraint achievementusing OpenMP and OpenCL to detect cancer cells, 2016. Master Thesis, Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación (ETSIST), Universidad Politécnica de Madrid (UPM).

[19] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[20] R. Lazcano, D. Madroñal, H. Fabelo, S. Ortega, R. Salvador, G. M. Callicó, E. Juárez, and C. Sanz. Parallel implementation of an iterative PCA algorithm for hyperspectral images on a manycore platform. In *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 1–6, Sept 2017.

[21] Raquel Lazcano López. Parallelization of the PCA algorithm on the Kalray MPPA-256 platform, 2015. Master thesis, Institut d'Électronique et de Télécommunications de Rennes (IETR), Institut National des Sciences Appliquées (INSA) de Rennes.

[22] Zhi Liu, Hongjun Wang, and Qingli Li. Tongue Tumor Detection in Medical Hyperspectral Images. 12:162–74, January 2012.

[23] D. Lorente, N. Aleixos, J. Gómez-Sanchis, S. Cubero, O. L. García-Navarrete, and J. Blasco. Recent Advances and Applications of Hyperspectral Imaging for Fruit and Vegetable Quality Assessment. *Food and Bioprocess Technology*, 5(4):1121–1142, May 2012. URL https://doi.org/10.1007/s11947-011-0725-1.

[24] Justin Luitjens. Faster Parallel Reductions on Kepler. February 2013. URL https://devblogs.nvidia.com/faster-parallel-reductions-kepler/.

[25] D. Madroñal, R. Lazcano, S. Ortega, H. Fabelo, R. Salvador, G. M. Callicó, E. Juárez, and C. Sanz. Implementation of a spatial-spectral classification algorithm using medical hyperspectral images. In *Proceedings of XXXII Conference on Design of Circuits and Integrated Systems (DCIS 2017)*, 2017.

[26] Daniel Madroñal Quintín. Implementación de una Support Vector Machine en RVC CAL para imágenes hiperespectrales, 2015. Master Thesis, Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación (ETSIST), Universidad Politécnica de Madrid (UPM).

[27] Clive Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. Newnes, 1 edition, 2004.

[28] Xinxin Mei and Xiaowen Chu. Dissecting GPU Memory Hierarchy Through Microbenchmarking. 28, October 2015.

[29] Xinxin Mei, Kaiyong Zhao, Chengjian Liu, and Xiaowen Chu. Benchmarking the memory hierarchy of modern gpus. In Ching-Hsien Hsu, Xuanhua Shi, and Valentina Salapura, editors, *Network and Parallel Computing*, pages 144–156. Springer Berlin Heidelberg, 2014.

[30] Andrew Ng. Machine Learning Course. CS229 Lecture notes:Part XI Principal Component Analysis, 2008. URL http://cs229.stanford.edu/syllabus.html.

[31] Andrew Ng. Machine Learning Course. CS229 Lecture notes:Part V Support Vector Machine, 2008. URL http://cs229.stanford.edu/syllabus.html.

[32] NVIDIA. GPU-accelerated Libraries for Computing. https://developer.nvidia.com/gpu-accelerated-libraries, .

[33] NVIDIA. NVIDIA CUDA Best practices guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, .

[34] NVIDIA. Profiler's user Guide. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview, .

[35] NVIDIA. NVIDIA CUDA Compute Architecture: Fermi Whitepaper. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, .

[36] NVIDIA. Jetson JetPack documentation. http://docs.nvidia.com/jetpack-l4t/index.html#developertools/mobile/jetpack/l4t/3.2/introduction.htm%3FTocPath%3D_____1, .

[37] NVIDIA. Jetson TX1 OEM PRODUCT DESIGN GUIDE. http://developer.download.nvidia.com/assets/embedded/secure/jetson/TX1/docs/JetsonTX1_OEM_Product_DesignGuide_20170912.pdf?ox_IIZA_uK2vi5eMc-3kkO7RHu00KxKszQi2nv-COOMRQn8BbLygQWFNc761IPjEx-Md3yPGaTxwpEsRmUzOEn0S3TaUUNp0X80cEnS4zb9UEK64iKt1tdRKM3-Z8UUGgeIlKwNK6dmiV8uL4rlA4lr9Sl-GHqiTovg7ZIyIfYlIwhg21ldqgRhcZZHgeEHoj-IKuA, .

[38] NVIDIA. NVIDIA Tegra X1 Whitepaper. https://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf, .

[39] NVIDIA. Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler, .

[40] NVIDIA. Jetson TX1 System on Module Product Sheet. http://images.nvidia.com/content/tegra/embedded-systems/pdf/JTX1-Module-Product-sheet.pdf, 2008.

[41] NVIDIA. *NVIDIA CUDA C Programming Guide*, 2012. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[42] Philippe Rollin NVIDIA. OpenGL 4.x Tesellation. http://www.nvidia.com/content/siggraph/Rollin_Oster_OpenGL_CUDA.pdf, .

[43] Antonio Plaza, Jon Atli Benediktsson, Joseph W. Boardman, Jason Brazile, Lorenzo Bruzzone, Gustavo Camps-Valls, Jocelyn Chanussot, Mathieu Fauvel, Paolo Gamba, Anthony Gualtieri, Mattia Marconcini, James C. Tilton, and Giovanna Trianni. Recent advances in techniques for hyperspectral image processing. *Remote Sensing of Environment*, 113:S110 – S122, 2009. URL http://www.sciencedirect.com/science/article/pii/S0034425709000807. Imaging Spectroscopy Special Issue.

[44] Antonio Ruiz, Jun Kong, Manuel Ujaldon, Kim Boyer, Joel Saltz, and Metin Gurcan. Pathological image segmentation for neuroblastoma using the GPU. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pages 296–299. IEEE, 2008.

[45] R. Salvador, H. Fabelo, R. Lazcano, S. Ortega, D. Madroñal, G. M. Callicó, E. Juárez, and C. Sanz. HELICoiD Tool Demonstrator for Real-Time Brain Cancer Detection. In *Proceedings of Design and Architectures for Signal and Image Processing (DASIP)*, 2016.

[46] R. Salvador, H. Fabelo, R. Lazcano, S. Ortega, D. Madroñal, G. M. Callicó, E. Juárez, and C. Sanz. Toward Efficient Embedded Implementations of KNN-based Spatial-Spectral Classification In Hyperspectral Imaging for Cancer Detection. In *Proceedings of XXXI Conference on Design of Circuits and Integrated Systems (DCIS 2016)*, 2016.

[47] Rubén Salvador, Samuel Ortega, Daniel Madroñal, Himar Fabelo, Raquel Lazcano, Gustavo Marrero, Eduardo Juárez, Roberto Sarmiento, and César Sanz. HELICoiD: Interdisciplinary and Collaborative Project for Real-time Brain Cancer Detection: Invited Paper. In *Proceedings of the Computing Frontiers Conference*, pages 313–318, 2017. URL http://doi.acm.org/10.1145/3075564.3076262.

[48] Jaime Sancho Aragón. Desarrollo de un visor de "Free-Viewpoint Video" sobre gafas de realidad virtual, 2017. Final degree project, Escuela Técnica Superior de Ingeniería de Telecomunicaciones (ETSIT), Universidad Politécnica de Madrid (UPM).

[49] Sam Sinayoko. Eigenvalue problems I: Introduction and Jacobi Method. URL http://docplayer.net/43336795-Eigenvalue-problems-i-introduction-and-jacobi-method.html.

[50] Lindsay I Smith. A tutorial on principal components analysis. Technical report, Cornell University, USA, February 26 2002. URL http://www.iro.umontreal.ca/~pift6080/H09/documents/papers/pca_tutorial.pdf.

[51] Sergio Sánchez Ramírez. Embedded GPU based Accelerator Implementation of a Suport Vector Machine for Brain Tumour Detection, 2018. Final Degree Project,

Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación (ETSIST), Universidad Politécnica de Madrid.

[52] M. A. Tahir and A. Bouridane. An Fpga Based Coprocessor for Cancer Classification Using Nearest Neighbour Classifier. In *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, volume 3, pages III–III, May 2006.

[53] Humboldt State University. Spatial Enhancements. 2015. URL http://gsp.humboldt.edu/olm_2015/Courses/GSP_216_Online/lesson4-2/spatial.html.

[54] Adolfo Vara de Rey Suárez. Implementación en una GPU empotrada usando CUDA de un Filtro Espacio-Espectral de Mapas de Clasificación Hiperespectrales para Detección de Tumores Cerebrales, 2018. Final degree project, Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación (ETSIST), Universidad Politécnica de Madrid (UPM).

[55] Ting-Fan Wu and Chih-Jen Lin. Probability estimates for multi-class classification by pairwise coupling. URL http://www.jmlr.org/papers/volume5/wu04a/wu04a.pdf.

[56] C. Yan, Y. Zhang, F. Dai, J. Zhang, L. Li, and Q. Dai. Efficient parallel HEVC intra-prediction on many-core processor. *Electronics Letters*, 50(11):805–806, May 22 2014. URL https://search.proquest.com/docview/1541993291?accountid=14712.

[57] Chenggang Yan, Yongdong Zhang, Feng Dai, Xi Wang, Liang Li, and Qionghai Dai. Parallel deblocking filter for HEVC on many-core processor. *Electronics Letters*, 50(5): 367–368, Feb 27 2014. URL https://search.proquest.com/docview/1525958810?accountid=14712.