Aceleración mediante el Intel oneAPI Toolkit de algoritmos para la detección de cáncer de piel y tumores cerebrales a través de imágenes hiperespectrales

Intel OneAPI Toolkit acceleration of algorithms for the detection of skin cancer and brain tumors through hyperspectral images

Antonio Álvarez Sánchez
Eneko Retolaza Ardanaz
Fabrizio Nicolás Zeballos

GRADO EN INGENIERÍA INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería Informática

Madrid, septiembre de 2024

Directores:

González Calvo, Carlos
Bernabé García, Sergio

Agradecimientos

A nuestros directores, Sergio y Carlos por vuestra ayuda fundamental. No habría sido posible sin vosotros, gracias por enseñarnos tanto sobre este tema y por despertar en nosotros una profunda curiosidad y pasión por la ingeniería.

Antonio Álvarez Sánchez

Quiero expresar mi más profundo agradecimiento a mis padres por darme esta valiosa oportunidad. Aunque la distancia haya sido un desafío, siempre habéis estado a mi lado con vuestro amor y apoyo incondicional. A mi hermano, sigue con tu constancia, vas a conseguir todo lo que te propongas. A mi pareja, María, por acompañarme y confiar siempre en mí. A mis amigos, por las alegrías del día a día.

Eneko Retolaza Ardanaz

Quisiera expresar mi más profundo agradecimiento a todos los que me han ayudado y apoyado en este camino. A mis padres y familia, su constante confianza en mi ha sido mi mayor motivación y sin ellos no habría sido posible llegar hasta aquí. También estoy inmensamente agradecido a mis amigos, cuya compañía me ha ayudado a seguir adelante en los momentos más desafiantes. También a mi pareja, por ayudarme y ser uno de los apoyos más importantes.

Fabrizio Nicolás Zeballos

Quiero agradecer a todas las personas que me han apoyado en esta etapa. A mis padres, que se han sacrificado tanto para poder darme oportunidades que ellos no pudieron disfrutar. A mi hermano y hermana, por acompañarme y ayudarme en los momentos difíciles. A mi pareja, Sonia, por confiar en mí desde el primer momento y ser uno de los principales apoyos en el día a día. Y a mis amigos de la universidad, sin ellos no habría logrado llegar hasta aquí.

Resumen

La medicina, siempre a la vanguardia de la innovación, ha adoptado continuamente avances tecnológicos de otros campos con el objetivo de salvar vidas y mejorar la esperanza de vida de las personas. Un claro ejemplo de esta transferencia tecnológica es el uso de las imágenes hiperespectrales, originalmente desarrolladas para el análisis de la superficie terrestre. Esta técnica no solo ha revolucionado el estudio de nuestro planeta, sino que también ha encontrado aplicaciones prometedoras en el análisis del cuerpo humano. Sus ventajas, como ser una tecnología no invasiva, no ionizante y de alta precisión, la convierten en una herramienta ideal tanto para la detección de enfermedades complejas como para la asistencia en intervenciones quirúrgicas delicadas. No obstante, para que estas aplicaciones sean prácticas y eficientes, es fundamental avanzar en la capacidad computacional.

Este trabajo se centra en la aceleración de un clasificador espacio-espectrales de imágenes hiperespectrales, compuesto por los algoritmos PCA y SVM ejecutados en paralelo, seguidos del algoritmo KNN. Su aplicación está destinada a asistir a los cirujanos en la delimitación precisa del tejido tumoral durante las operaciones de cáncer cerebral, así como en la detección estática de cáncer de piel. La implementación de esta serie de algoritmos se realiza en una FPGA como acelerador de hardware, y su optimización se lleva a cabo utilizando el Intel OneAPI Toolkit, analizando las ventajas que ofrece para el desarrollo de este tipo de aplicaciones.

Palabras Clave

Intel oneAPI Toolkit, FPGA, Imágenes hiperespectrales, PCA, SVM, KNN, Programación en paralelo, Detección de cáncer, Diagnóstico médico, Intel DevCloud

Abstract

Medicine, always at the forefront of innovation, has continuously adopted technological advances from other fields with the aim of saving lives and improving people's life expectancy. A clear example of this technological transfer is the use of hyperspectral imaging, originally developed for the analysis of the Earth's surface. This technique has not only revolutionized the study of our planet, but has also found promising applications in the analysis of the human body. Its advantages, such as being a non-invasive, non-ionizing and high-precision technology, make it an ideal tool both for the detection of complex diseases and for assisting in delicate surgical interventions. However, for these applications to be practical and efficient, it is essential to advance in computational capacity.

This work focuses on the acceleration of a spatial-spectral classifier of hyperspectral images, composed of the PCA and SVM algorithms run in parallel, followed by the KNN algorithm. Its application is designed to assist surgeons in accurately delineating tumor tissue during brain cancer surgeries, as well as in the static detection of skin cancer. This set of algorithms is implemented on an FPGA as a hardware accelerator, and its optimization is carried out using the Intel OneAPI Toolkit, analyzing the advantages it offers for developing these types of applications.

Keywords

Intel oneAPI Toolkit, FPGA, Hyperspectral Imaging, PCA, SVM, KNN, Parallel Programming, Cancer Detection, Medical Diagnostics, Intel DevCloud

Contents

List of Figures

List of Algorithms

Chapter 1: Introduction

There is a deep concern for medical result-oriented analysis diagnosis nowadays. Artificial intelligence and machine learning has developed medical diagnosis accurate tools, and this pushes further the usability and relevance. From classical artificial intelligence to modern AI such as transformers, there are many algorithms available. There are different AI predicting and inference algorithms, with particular system requirements. Thus, depending on the used hardware or available systems, there is a specific need to choose the correct resource.

Different hardware architectures such as Graphic Processing Units (GPUs) really outperform in this field, which has made them grow in popularity. However, the high cost and power consumption still make them a difficult alternative for complex device environments. Here is where Field-Programmable Gate Arrays (FPGAs) emerge.

Despite the popularity of GPUs and the availability of tools that make their setup easy, FPGAs stand out for their low power consumption and, importantly, their reconfigurability to adapt to various devices. In fact, due to their reconfigurable hardware, FPGAs have a significant advantage over fixed hardware; with proper optimizations, their performance can become comparable.

One of the main challenges for these devices is the lack of professionals specialized in developing applications for them, compared to the GPU field. For this reason, Intel has released libraries such as Intel oneAPI Toolkit, which translates high-level programming language to binary FPGA executable code, making algorithm implementation easier on these platforms.

The advantages of these libraries could make a capstone on the current industry, opening the door to improving efficiency in scientific and technological research. Thanks to this platform, researchers and developers can leverage the flexibility and low power consumption of FPGAs without requiring specialized hardware knowledge, thus accelerating innovation

processes. In this work, one of the key perspectives explored is whether Intel oneAPI can be considered a viable alternative to traditional approaches, particularly in terms of implementing complex algorithms on FPGAs.

1.1 Motivation

The early detection of skin cancer and brain tumors is a critical challenge in the medical field, where accuracy and efficiency can significantly impact patient outcomes. Hyperspectral imaging has emerged as a powerful tool in this field, capable of providing detailed information that enhances the ability to diagnose. However, the vast amount of data generated by these images requires substantial computational resources, often leading to prolonged processing times and limiting the feasibility of real-time applications.

FPGAs offer a powerful and transformative solution to current challenges, combining the flexibility of traditional processors with extremely low power consumption and outstanding speed. These programmable devices not only allow for hardware to be tailored to specific needs but can also be seamlessly integrated into almost any instrument or tool, unlocking new possibilities for revolutionary analysis and surgical assistance applications. Their ability to efficiently merge software and hardware is positioning FPGAs as key players in the technological innovations of recent years.

1.2 Objectives

The general objective of this project is the acceleration through Intel oneAPI Toolkit [1] of a spatial-spectral classifier of hyperspectral images [2], composed of the PCA and SVM algorithms run in parallel, followed by the KNN algorithm.

These algorithms, used in an FPGA hardware accelerator, are part of a system for brain cancer detection and assistance during the corresponding surgical operations. Thanks to their acceleration, they can be used in real-time to analyze hyperspectral images and obtain life-saving information.

Achieving this global objective is addressed throughout this report following specific objectives described below:

- Analysis and evaluation of the provided algorithm chain.
- Implementation of the algorithms in C++ with SYCL, adapted for compilation through DPC++, and their operation on FPGA boards.
- Individual optimization of each of the three mentioned algorithms (PCA, SVM and KNN), analyzing their performance and potential improvements.
- Integration of the optimized algorithms back into the complete chain, obtaining combined results from their integration.
- Parallelize the complete pipeline using the available hardware resources.

1.3 Project Plan

In order to achieve the objectives above, a detailed project plan has been drawn up. This planning serves as a reference to measure its progress. Its representation has been made employing a Gantt diagram in which a color legend can be found according to the tasks and phases of the project (see Figure 1.1).

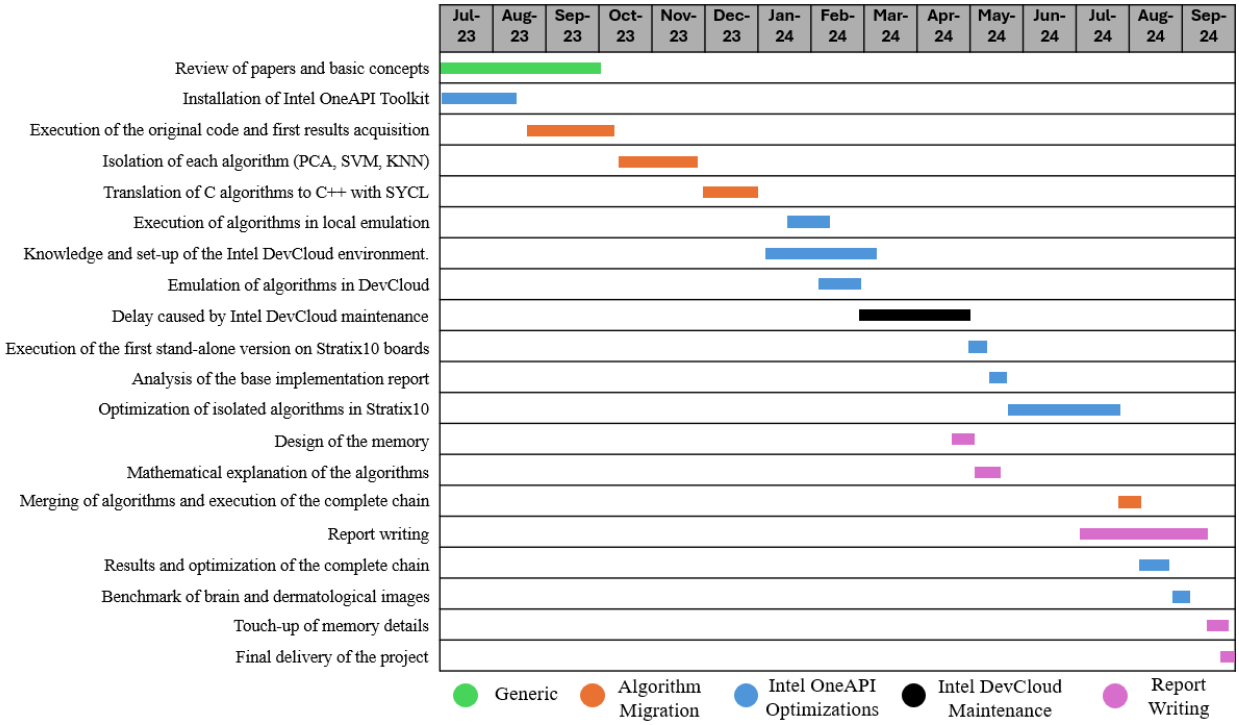


Figure 1.1: Gantt diagram of project planning.

1.4 Document Structure

This document has been organized into chapters that gather all the information about this work, from the research on hyperspectral images and algorithms to the results obtained after optimization. All the chapters are presented below, sorted according to their appearance order, together with a brief description:

- **State of The Art**: This chapter aims to provide an overview of the most recent advances and shows the theoretical aspects of our work. On the one hand, it explains the concept of hyperspectral imaging, its applications, especially in medicine, and its acquisition and processing. On the other hand, the main hardware and software tools for their implementation and analysis are presented.

- **PCA**: This chapter provides an analysis of the Principal Component Analysis(PCA) algorithm for dimensionality reduction of hyperspectral images. It starts with the

analysis of a base implementation, which is later optimized to the final version together with the performance results obtained.

- **SVM**: This chapter addresses the study of the Support Vector Machine(SVM) algorithm for the classification of hyperspectral images. It starts with an analysis of the base implementation, which is optimized to the final version, along with the performance results obtained.

- **KNN**: This chapter includes the study of the K Nearest Neighbors(KNN) algorithm for the filtering and homogenization of hyperspectral images. It starts with an analysis of a base implementation, optimizing it to a final version, and then, shows the obtained performance results.

- **Integration and Optimization of Combined Pipeline**: This chapter explains the full algorithm when PCA, SVM, and KNN are joined. Analyses the integration of the three algorithms, from complexity to resource utilization and performance.

- **Conclusions and Future Work**: These two chapters give a conclusion to the full work, highlighting the results and possible future work to continue improving while investigating.

- **Appendix**: The first appendix lists the contributions of each of the members of the project, collected by name.

Chapter 2: State of The Art

This chapter aims to analyze the current state of the art in hyperspectral image analysis and its use by FPGAs, explaining the hardware and software involved.

It is divided into three main sections. First, in Section 2.1 hyperspectral images are explained, starting from a conceptual point of view, analyzing their applications and their relevance in medicine. Then, their acquisition, processing and finally, the images used in this work are described. Secondly, in Section 2.2 reconfigurable hardware is explained, explaining the functioning of FPGAs and all their components. Thirdly, Section 2.3 explains the existing alternatives to develop programs on FPGAs, with special emphasis on the Intel oneAPI Toolkit, used in our work.

2.1 Hyperspectral Imaging

A hyperspectral image (HSI) is a modality of image spectroscopy where 3D datasets are generated. The third dimension is represented by the spectral bands, which result in a hyperspectral pixel, that is a column vector with the reflectance values of that pixel [3]. The size of the vector corresponds to the number of given bands. A spectral band consists of regions within a spectrum that have a specific range of wavelengths and frequencies. The number of bands used can range from hundreds, allowing the capture of the spectral information from the images. As Figure 2.1 shows, the hypercube represents a three-dimensional matrix where the first two dimensions, samples and lines, reflect the image pixels, while the third dimension includes the spectral bands of the image.

For the study of hyperspectral images, it is necessary to use the spectral reflectance properties [4]. This reflectance is the fraction of reflected radiation over incident energy. It varies for different wavelengths due to the type of material where it is incident, since it can be absorbed or reflected to different degrees, this characteristic is defined as a "spectral

signature", which is unique for each material [5]. For this reason, surfaces can be delimited and materials differentiated thanks to hyperspectral imaging.

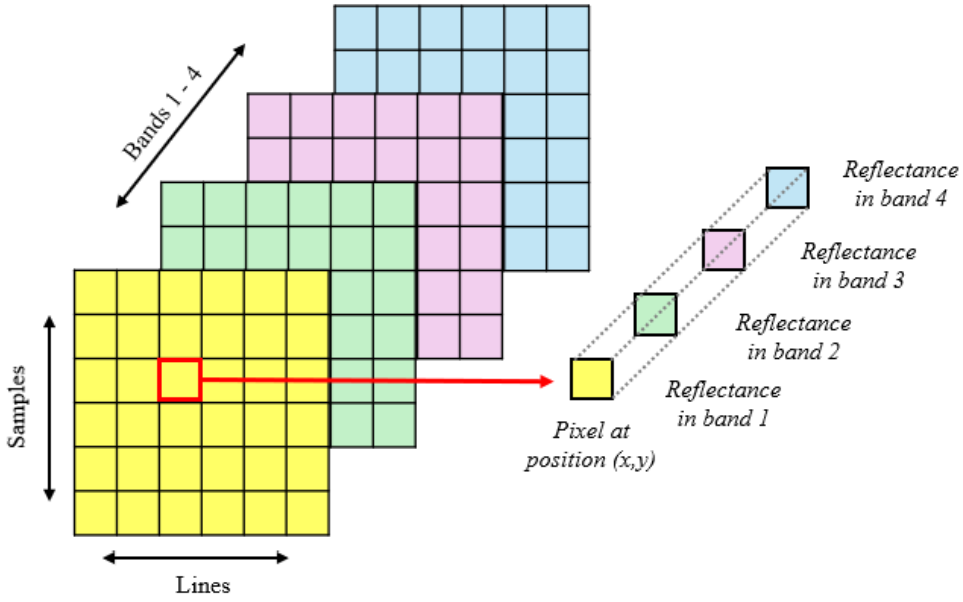


Figure 2.1: Concept of hyperspectral imaging [6].

The main difference between hyperspectral images and other types of images such as RGB ones, widely used on a daily basis, is that the latter loses the third dimension corresponding to the spectral resolution, collecting very few spectral bands (usually 3 for red, green, and blue colors). Figure 2.2 shows the differences between a hyperspectral image and an RGB image. The first shows the spectral signature, while the last only shows the intensity of the three colors mentioned.

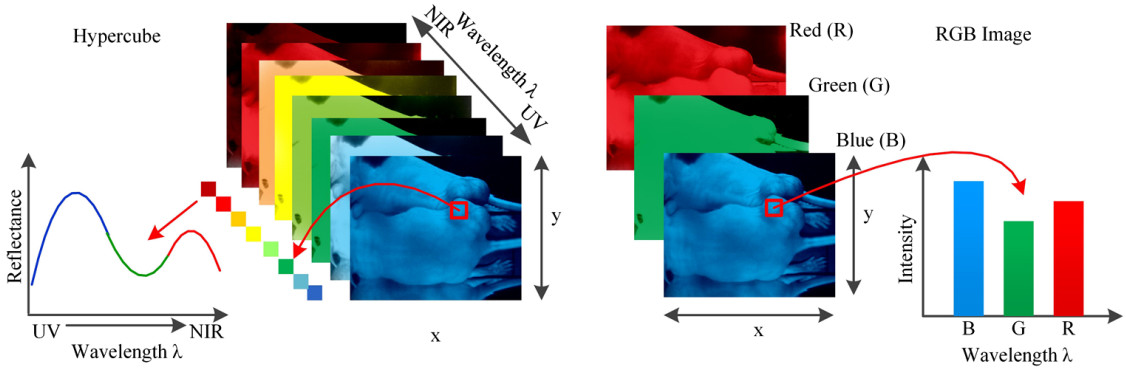


Figure 2.2: Differences between HS Images and RGB Images [7].

Another of the main characteristics of hyperspectral images is the different types of existing pixels [8]. On the one hand, there are endmembers or pure pixels, whose spectral signature is composed of a single material. On the other hand, mixed pixels are pixels composed by different materials at the sub-pixel level. The latter are the ones that make up most of the hyperspectral image since the mixture of materials predominates even at the microscopic level [9]. A visual example of the different types of pixels can be found in Figure 2.3.

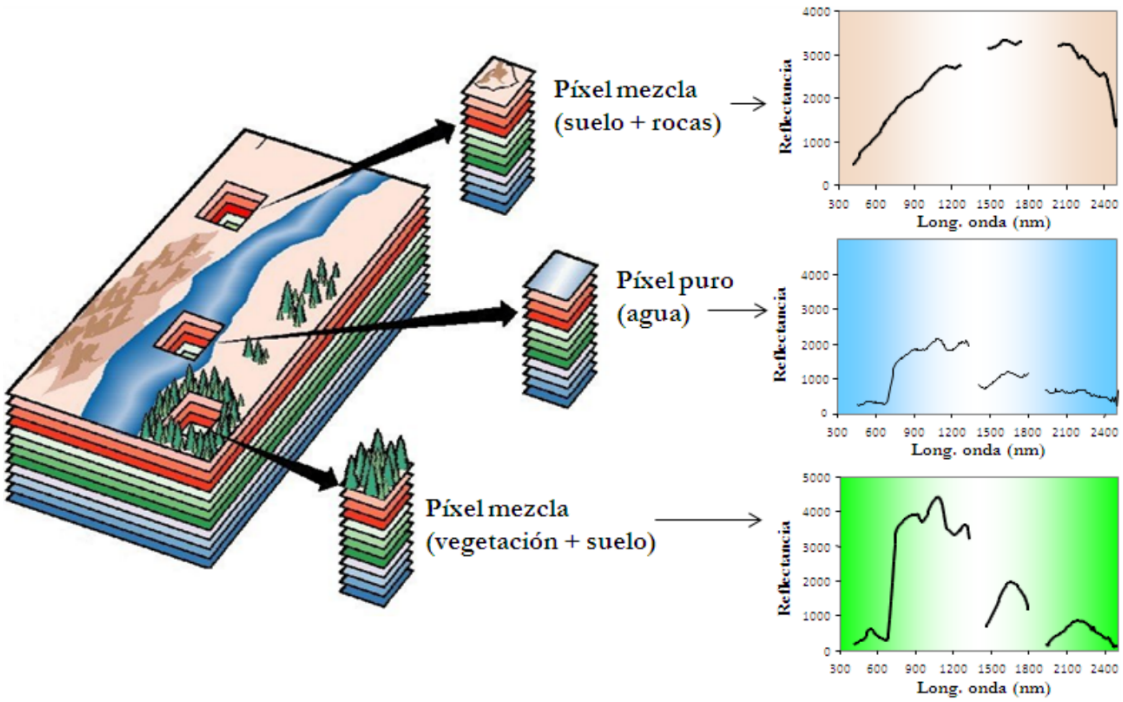


Figure 2.3: Pixel types in hyperspectral imaging [6].

2.1.1 Hyperspectral Imaging Applications

Hyperspectral imaging was first developed for remote sensing, taking advantage of satellite technology in the late 1980s, which made it possible to obtain the first hyperspectral image of the earth for subsequent analysis [10]. However, nowadays, this technology has many more applications. Some of the most important uses are mentioned below:

- **Agriculture**: Precision agriculture is growing with good perspectives to address the

increasing problems facing the agricultural sector. In fact, the number of publications using hyperspectral imaging in agriculture has grown in the decade 2011-2020 (245 publications) compared to the decade 2001-2010 (97 publications) [11]. One of the main uses has been to estimate biochemical properties such as leaf chlorophyll or water content, affecting crop yields [12]. Hyperspectral imaging has also been used to identify crop species, their growth status, or even diseases and plagues [13].

- **Cultural heritage conservation**: Multispectral imaging and later hyperspectral imaging have been used in the field of cultural heritage since the early 1990s. Being a non-invasive technique, it allows investigations on very fragile objects and some of its uses have been to identify different materials, deteriorated areas, and to identify past interventions [14]. Given the diversity of elements to be protected, researches collected the main systems used in the field of conservation, explaining the number of bands, detectors, materials, and surfaces on which they are currently used [15].

- **Geology and Minerals**: Geology has traditionally studied the composition of the terrain. However, it is becoming increasingly multidisciplinary, which has lead to an increment in the usage of remote sensing in geological analysis [16]. Thus, the use of satellites has spread the application of hyperspectral image analysis for the study of the Earth's surface. One of the most common practices is the detection of mineral deposits and strata, allowing to differentiate precious minerals as gold from others such as silicon. Furthermore, they are also used to monitor the impact of mining activities and mineral veins on the mining environment. Some studies have even used hyperspectral imaging to analyze mineral collection by rivers flowing through mining areas, such as the Odiel River flowing through the Iberian Pyrite Belt in Huelva [17].

2.1.2 Hyperspectral Imaging Analysis in Medicine

After exploring the applications of hyperspectral imaging in agriculture, cultural heritage conservation, and geology, it is essential to highlight its impact in the medical field, which

is the focus of this work. The use of hyperspectral imaging has recently been adopted in medicine, where it offers satisfactory results as a non-invasive diagnostic and evaluation technique.

The depth that light penetrates into tissues depends on whether they are weak absorbers. During the absorption process an energy transition occurs where at a specific wavelength can result in a spectral signature for the diagnostic [7]. The light absorbed by tissue components is transformed into heat or radiated in the form of luminescence, subdivided into fluorescence and phosphorescence, where the first has a lifetime of the order of nanoseconds and the second has a lifetime of the microsecond-millisecond order.

Fluorescence makes it possible to investigate tissues for diagnostic purposes. The received light can be reflected into the tissue surface or scattered due to its variations. Alterations in tissue morphology can affect the scattering patterns. Some biomolecules such as NADPH and FAD re-radiate fluorescent light at a longer wavelength, these biomolecules are known to be altered during the progression of diseases, making their autofluorescence a good target for HSI methods [18].

Taking advantage of the information that hyperspectral imaging can provide in medicine, and given its non-contact, non-ionizing, and non-invasive nature, it is being used for various diseases in in-vitro, ex-vivo and in-vivo environments [2]. Many types of cancer have been investigated using HSI [19], such as breast cancer [20], prostate cancer [21], melanoma [22], or tongue cancer [23] among others. Other diseases such as foot ulcers in diabetes [24] or even dental caries [25] have also been investigated using this technique.

In addition to disease detection, of its main use is as a tool for image guidance during surgical operations, which is one of the objects of study. Tools such as intra-operative neuro navigation, intra-operative magnetic resonance imaging (MRI), and fluorescent tumor markers such as 5-aminolevulinic acid (5-ALA) have been used to help surgeons delineate tumor tissues. However, many of these methods have limitations, either at the computational and imaging level or related to the side effects caused to patients [26]. Moreover, in the context of brain tumor operations, this type of cancer diffusely infiltrates the surrounding healthy brain tissue (especially gliomas), making it very difficult for the surgeon to differentiate be-

tween cancerous tissue and healthy tissue with the naked eye. Removing more tissue than necessary can cause neurological problems while leaving some cancerous tissue behind can encourage the progression of the disease [26]. These reasons make hyperspectral imaging a promising alternative to help delineate cancer tissue in in-vivo brain cancer operations.

For its application in dermatology, HSI also represents a major breakthrough. Although it is not used in such delicate situations as the above-mentioned operations, it makes it possible to analyze skin lesions outside the human visible spectrum, which facilitates the differentiation between moles and the detection of malignant areas [27].

2.1.3 Obtaining Hyperspectral Images

The concept of hyperspectral imaging emerged as a NASA mission at the Jet Propulsion Laboratory in 1983, through the design and development of the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) [28]. It measured its first images in 1987 and since then its contributions to HSI research have been constant over the years. It is capable of imaging 224 spectral bands with wavelengths between 400 and 2500 nm. NASA has also promoted other hyperspectral missions that have given rise to sensors such as MODIS, launched in December 1999 on the EOS Terra satellite [29], or EO-1 Hyperion, launched in 2000 as part of the New Millennium mission [30]. There are also other non-NASA initiatives, such as Japan's HISUI launched in 2019 [31]. All of these sensors have made advances in the field of remote sensing and the study of the Earth's surface.

Regarding the cameras used to capture hyperspectral images in laboratory studies, they maintain some differences from those used for remote sensing. There are four approaches for acquiring 3-D hyperspectral image cubes, which are point scanning (whiskbroom method), line scanning (pushbroom method), area scanning, and the single shot method [32]. Figure 2.4 shows these methods in an illustrative way.

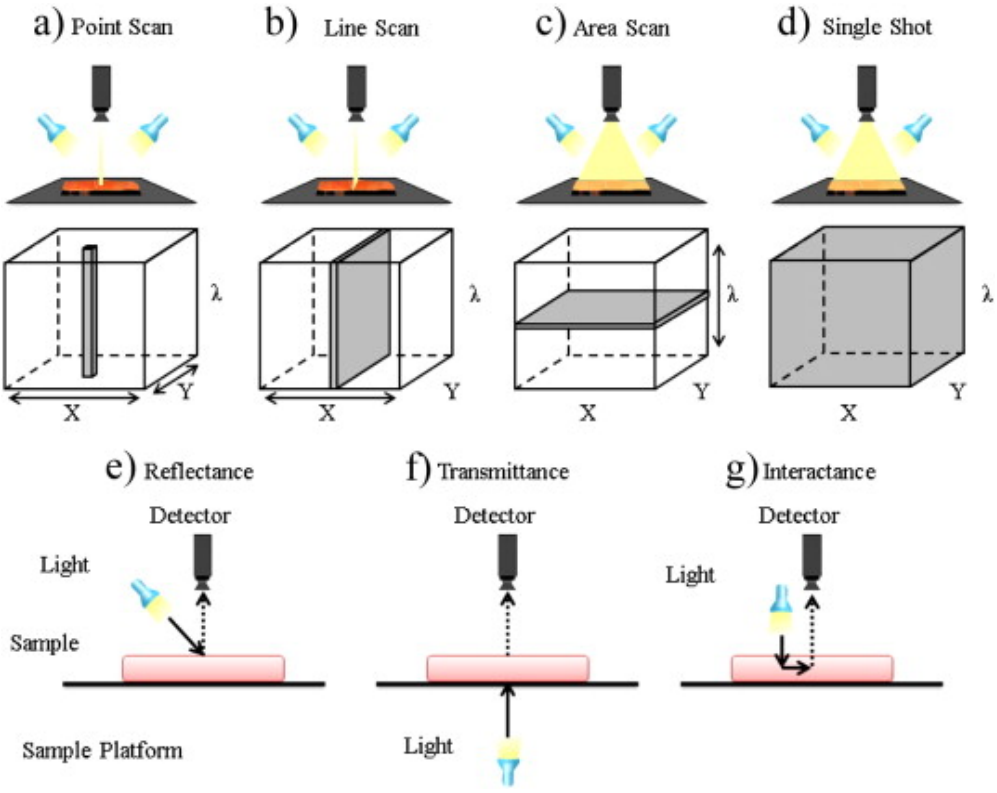


Figure 2.4: Acquisition approaches of hyperspectral images [32].

The camera used to capture the brain images used in our work is a Hyperspec VNIR A-Series, which is based on the line-scanning technique [26]. This camera covers a spectral range from 400 to 1000 nm, being able to capture 826 spectral bands and 1004 spatial bands. This range has been selected to find the most relevant spectral regions where the tumor and normal brain tissues can be distinguished using machine learning algorithms.

2.1.4 Hyperspectral Image Processing

To obtain the final result, the hyperspectral image needs to first be preprocessed (which includes unmixing) and then processed. The final result will depend on the decisions taken throughout the process. This section will explain how the image is processed after obtaining it and the post-process.

The unmixing is a hyperspectral preprocessing part of the algorithm. In this stage, the HSI mixed pixels are analyzed due to their huge amount of pixel resolution. Each pixel in

a hyperspectral image typically contains a mixture of different materials or substances, and spectral unmixing aims to decompose this pixel into its constituent materials (endmembers) [33, 34].

There are two main models to interpret the sensor-received image: the linear model and the nonlinear one. The first assumes that the absorption and refraction phenomena of electromagnetic radiation are structured on a linear pattern. The nonlinear model processes the randomly scattered endmembers through the image [34].

Figure 2.5 shows how linear models (a) reflect the information with a two-dimensional plane and nonlinear models (b) have no linear plane to reflect so it needs to be corrected.

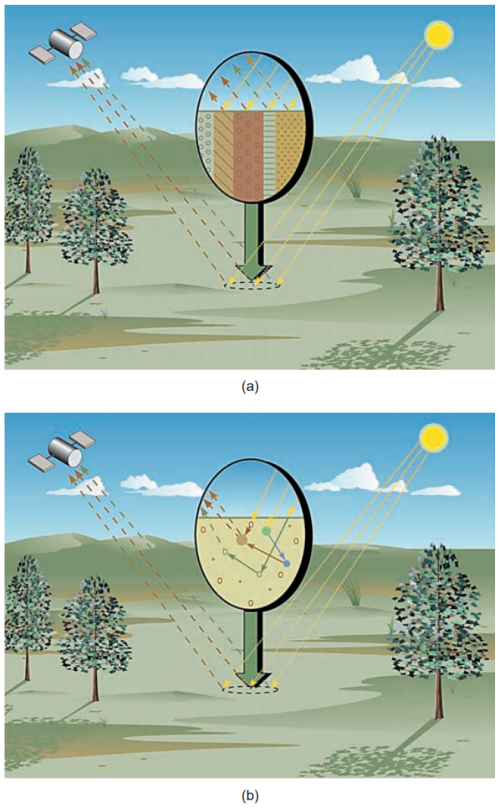


Figure 2.5: Linear and nonlinear mixes [33].

After this process, the raw image needs to be calibrated, extreme noise bands removed, and bands averaged and normalized [26].

All the data analyzed in HSI needs to be processed and classified to obtain the final

result. For this, the image needs to go through a computational and algorithmic process [35]. After the scanner preprocessing, the data will go from the raw stage to the final result stage (see Figure 2.8). Depending on what type of data is analyzed or what the data is for, different processing tools and algorithms will be used.

One of the main differences between the algorithms can be if they are supervised or unsupervised. Supervised algorithms have an attached output, so they are a labeled classification process. This means that the input size of data is the same as the output, so the algorithms only transform the data, not the size. On the other hand, the unsupervised algorithms transform the data size, so they are unlabeled. Figure 2.6 shows some examples of the mentioned algorithm types.

Supervised Algorithms	**Unsupervised Algorithms**
Spectral Angle Mapper (SAM)	Principal Component Analysis (PCA)
Spectral Information Divergence (SID)	Independent Component Analysis (ICA)
Maximum Likelihood Classifier (MLC)	Minimum Noise Fraction (MNF)
Support Vector Machine (SVM)	Non-negative Matrix Factorization (NMF)
Random Forest Classifier	Clustering
Neural Networks (CNN, MLP)	Self-Organizing Maps (SOM)
Endmember Extraction Algorithms	Linear Spectral Unmixing (LSU)
Spectral Matching Algorithms	Feature Extraction
Target Detection Algorithms	Change Detection Algorithms

Figure 2.6: Supervised and unsupervised algorithm examples [35].

There are many other classification types between algorithms. Each one has its benefits, and algorithms can not be only chosen because of the aim, they could also be chosen because of the data size efficiency or performance [36]. Figure 2.7 shows the main algorithm classification categories.

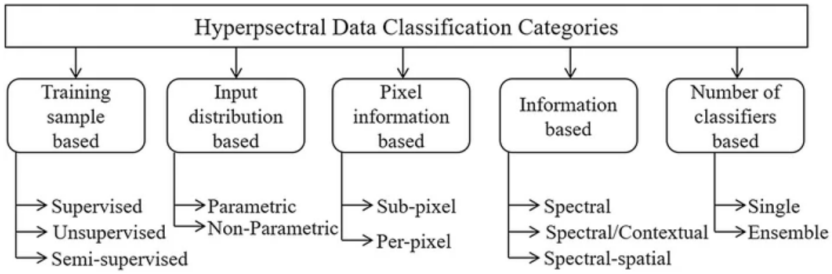


Figure 2.7: Hyperespectral Data Classification Categories [36].

The three stages needed for image processing are, first, dimensionality reduction, then, classification, and finally, segmentation. Each stage can be implemented in different ways, depending on the approach and the data size.

For dimensionality reduction, some of the most used are Principal Component Analysis (PCA), Independent Component Analysis (ICA), and Linear Component Analysis (LCA). PCA is used mainly when there is a need to maximize variance, ICA to maximize statistical independence, and LDA to maximize class separability. Another benefit is that the PCA is unsupervised, while, for example, LCA needs to be labeled [35].

For the classification and segmentation part, there are many popularly used algorithms and methods. Neural networks are becoming incredibly popular for general aim classifying, they are showing really good results and many of them showcase their incredible performance. However, in this case, there are not many images needed to train a neural engine, and the computational resources in case there were enough images would be huge. For this aim, algorithms such as Support Vector Machine (SVM), K-Nearest Neighbors (KNN), Random Forest Classifiers, or Decision Trees are commonly used [37, 36]. In this case, SVM and KNN are going to be used. SVM because of the good management of big sets of data [38] and KNN because of the minimal assumption about the underlying data.

The implemented pipeline includes the preprocessing (calibration, noise canceling, band averaging and normalization) and the processing (PCA, SVM and KNN) as can be seen in Figure 2.8. In previous works, this pipeline has been complemented with a prior preprocessing stage. However, these studies did not use FPGA. For example, [2] used low-power and high-power GPUs (Nvidia GPU) or a low-power manycore platform. For this reason, FPGAs could be a good option to replace such devices, improving results.

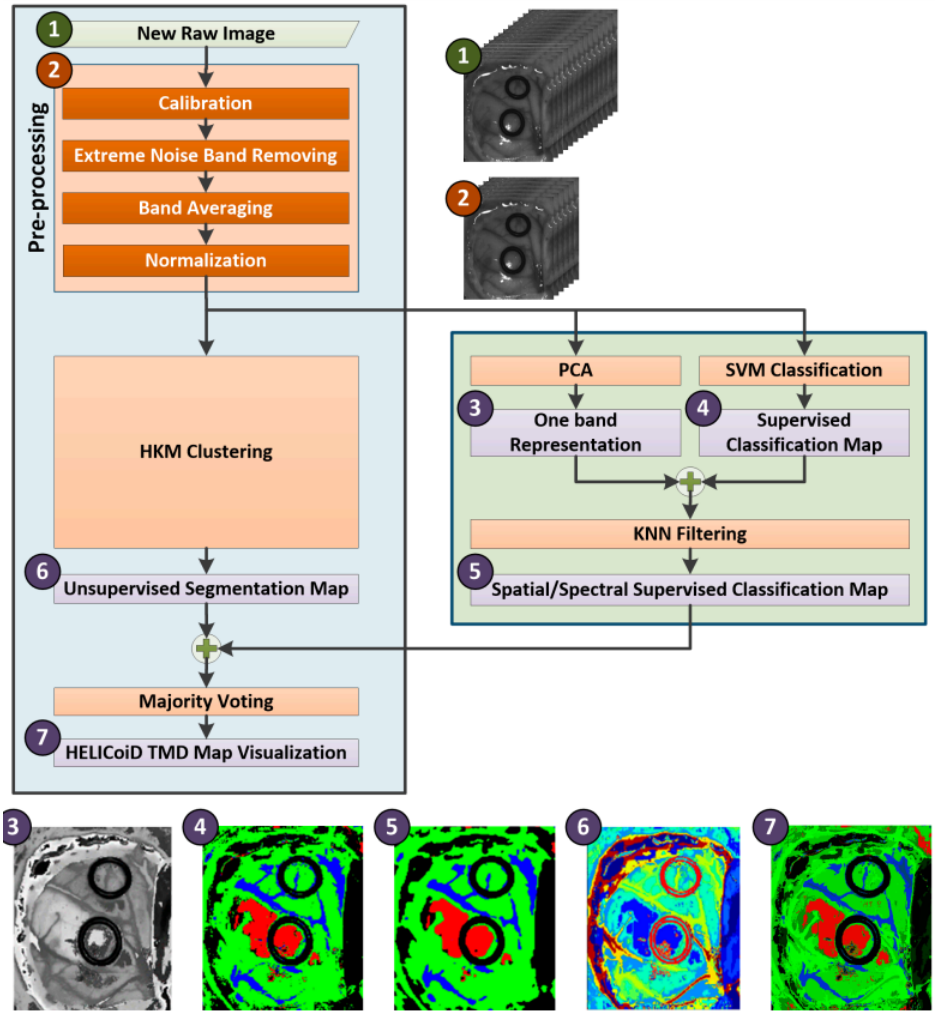


Figure 2.8: Algorithm implementation flow diagram [26].

2.1.5 Images Used

Both the brain cancer images and the skin images used throughout this project have been obtained from the work of Lazcano et al. [2]. On the one hand, brain cancer images have been obtained using the camera mentioned in Section 2.1.3, continuing the visualization system proposed by [26] and using the intra-operation methods for the collection of a database of brain cancer images used by [39]. These are three images from three different patients with a confirmed grade IV glioblastoma tumor by histopathology [2]. On the other hand, in this work, we have used a single skin image obtained using a customized system composed

of an HS sensor and a monochromatic (MC) sensor [2]. Table 2.1 shows the size and name of the images used. As can be seen, all the brain images used have 128 hyperspectral bands, while the skin image has 100 hyperspectral bands.

Dataset	Image ID	Pixels	Size	Size (MB)
Brain	PB1C1	219,232	496 x 442 x 128	107.05
Brain	PB2C1	184,875	493 x 475 x 128	90.20
Brain	PB3C1	124,033	329 x 377 x 128	60.56
Derma	PD1C1	1,000,000	1,000 x 1,000 x 100	381.00

Table 2.1: Specifications of the images used [2].

2.2 Reconfigurable Hardware

In today's rapidly evolving technology, the demand for computational power has grown exponentially. From handling enormous amounts of data to solving complex scientific problems, traditional sequential computing is often not enough. This is where parallel computing comes into play, revolutionizing how computational tasks are solved [40].

Since the nineties, parallel computing has been used to compute large amounts of complex data or real-time processing [41]. There was an inherent need to process the amount of data that linear processing was struggling to process, so the best solution was to split the tasks to compute it faster. As more processing units, as more could the process be split.

All this brings us to the current problem: parallel computing is also needed to compute complex algorithms that manage large amounts of data. In this case, dermatological and brain image processing totally needs parallel computing. The algorithm's complexity would make the linear process too slow for the medical diagnosis aim. This is where specialized hardware such as Field Programmable Gate Arrays (FPGAs) come into play.

2.2.1 FPGAs

An FPGA is a reconfigurable semiconductor integrated circuit (IC) which is different from other devices like Central Processing Units (CPUs) or GPUs. While CPUs and GPUs have

fixed hardware structures that applications must use, FPGAs allow the creation of custom hardware to execute a program. The design of an FPGA involves decisions at several levels, such as high-level architecture. In an architecture design, engineers determine aspects such as the number and type of functional blocks [42].

Although Application-Specific Integrated Circuits (ASICs) outperform FPGAs in a specific task, they require more development time and investment. In contrast, FPGAs offer a faster, more cost-efficient alternative. A crucial aspect of FPGA design is balancing the configurability of hardened blocks with the block footprint in the FPGA [42]. Hardened blocks are pre-optimized units integrated into the FPGA to perform specific tasks more efficiently, such as Digital Signal Processing (DSP) blocks. An ideal goal is to make these hardened functionalities work in as many applications as possible at the same time as minimizing their footprint in the silicon area [42]. Increasing the configurability of the hardened blocks improves the versatility of these blocks, but may reduce the area efficiency [42]. However, a block that cannot be used implies that the area it uses has been wasted and could be of general use [42]. There are different types of hardware design features or blocks in FPGAs, such as programmable logic, programmable routing, input/output (I/O) blocks, memory blocks, and Digital Signal Processing (DSP) blocks.

Logic

Programmable logic is a fundamental component in FPGA architecture. Traditionally Basic Logic Elements (BLEs) of an FPGA consists of two components: Lookup Table (LUTs) and Flips-Flops (FFs) [42]. The logic blocks (LBs) contain multiple BLEs along with local interconnects. These interconnects use multiplexers to route signals between the inputs and outputs of the BLEs and LBs [42]. As FPGA technology has advanced, LUT and LB have increased in size, larger LUTs allow more complex functions to fit into a single LUT, which reduces the number of logic levels in the critical path, hence improving performance [42].

Modern FPGA architectures often include dedicated arithmetic circuits within their logic blocks to optimize performance. For example, for deep learning (DL), FPGA designs

normally incorporate DSP blocks designed for these tasks, increasing the density of hardened blocks within FPGA fabric to improve performance in applications that rely on intensive arithmetic [42].

Routing

Programmable routing in FPGAs plays a crucial role, as it typically uses more than half of the fabric area and constitutes a large part of the critical path, so its efficiency is a key factor [42]. The main purpose of routing is to allow any output of a function block to connect to any input of the FPGA. FPGA routing architectures are usually divided into two types: hierarchical and island-style.

Hierarchical routing reflects the structure of designs in which higher-level modules create instances of lower-level, with frequent communication between nearby modules. This allows hierarchical FPGAs to use short wires to connect small regions of a chip, resulting in optimized routing for adjacent modules [42]. This type of routing is commonly used in smaller FPGAs. Instead, island routing, used in larger FPGAs, involves segments of routing wires, connection blocks, and switching blocks. The connection blocks link the function block inputs to wires together to make longer routes through the chip [42]. When it comes to handling different connection lengths, modern FPGAs incorporate multiple lengths of wiring segments, from short to long, but most segments have a moderate length [42].

A significant challenge in FPGA routing is that the delay of long wires is not improving as the scaling of the process advances, resulting in stagnant or increasing delays. To mitigate this, some vendors have integrated registers into the routing network. For example, Intel's Stratix 10 FPGAs, using its HyperFlex architecture, introduce registers into the routing paths, enabling pipelining and enhancing performance by allowing data to be stored closer to logic, reducing the critical path without requiring additional logic resources [43]. Although this approach slightly increases area and delay, it significantly improves performance over long connections [42].

Input/Output Blocks

FPGAs have highly adaptable, programmable Input/Output (I/O) blocks that enable communication with a wide range of external devices. These I/O structures must be flexible enough to support different requirements from different voltage levels to various communication protocols, among many other features, which presents significant design challenges [42]. These components are crucial because they enable FPGA integration with different systems, therefore the adaptability of these components is key to FPGA usability.

Memory Blocks

In the early stages, FPGAs used FF as the primary form of on-chip memory. As FPGAs became responsible for handling larger and more complex systems, the need for increased memory capacity [42]. To handle this increasing demand, modern FPGAs integrate various types of on-chip memory, with block RAM (BRAM) being the most commonly used [42].

A key challenge in FPGA design is determining the appropriate on-chip memory configuration. Since there is no single RAM configuration, the design must provide flexibility. For this reason, FPGAs include BRAMs with configurable parameters, allowing the memory to be customized to the specific needs of each application [42]. This customizability is crucial to meet the diverse memory requirements of the vast variety of systems that can be implemented in an FPGA [42]. FPGA memory architectures have evolved significantly over time. To match the growing demand for on-chip memory, modern BRAMs offer larger capacities than previous versions, going from 2kb to 20kb in the most current version, such as in the Stratix 10 [42, 44]. This increase in memory reflects the growing need to handle complex tasks, such as artificial intelligence (AI) and data processing. In this work, HSI processing and analysis is an example of such demanding tasks.

Digital Signal Processing Blocks

Previously, in FPGA architectures the only dedicated arithmetic circuits were the carry chains. Multipliers, by contrast, had to be built using logic with LUTs and carry chains,

which carried significant penalties. When applications with high multiplier density increased in demand, designers began to explore innovative methods to mitigate these inefficiencies [42].

The main objective of a DSP block was to minimize the use of soft programming to implement common DSP algorithms by embedding more hardened functionality directly into the DSP block. In the last years, the use of FPGAs has expanded significantly. The evolution of complex task computing needs, such as Deep Learning (DL), has pushed DSP block architectures.

FPGAs have now become a powerful solution for DL inference, offering higher power efficiency than GPUs and lower development costs compared to ASICs [42].

2.2.2 Hardware Used: Stratix 10 FPGA

For this bachelor thesis, the **Stratix10 FPGA** from Intel's FPGA System-on-Chip (SoC) family, specifically the SX 2800, is used as development hardware [44]. The Stratix10 is a leading-edge model in Intel's SoC FPGA family, which combines FPGA technology with high-performance computing (HPC). Built on Intel's 14nm FinFet processor technology, the Stratix 10 brings innovations in FPGA hardware architecture, such as HyperFlex architecture, an ARM Cortex-A53 processor, and broad (Digital Signal Processing) capabilities [44]. These features position the Stratix10 as an ideal solution for demanding software and HPC. The ease of use thanks to the intel dev cloud and the integration with the Intel oneAPI and SYCL, explained in the following sections, are also taken into account.

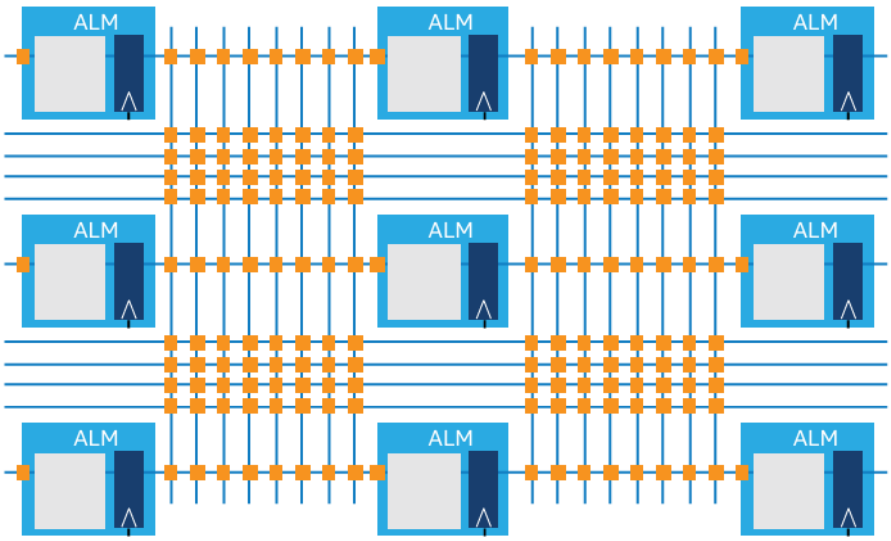
According to intel Stratix 10 SX 2800 the following available resources are seen in the Table 2.2 [44].

Resources	Units
KLE	2,753
Adaptive Logic Modules (ALMs)	933,120
ALM registers	3,732,480
M20K memory blocks	11,721
M20K memory size (Mb)	229
MLAB memory size (Mb)	15
DSP Blocks	5,760
18×19 multipliers	11,520
Peak fixed-point performance TMACS	23
Peak floating-point performance TFLOPS	9.2

Table 2.2: Stratix 10 SX overview of resources and performance [44].

One of the main advantages of using the Stratix 10 over previous FPGAs is the addition of the HyperFlex Architecture.

The main core of this architecture is implementing Hyper-Registers, mentioned in the previous section, which are additional pipeline registers strategically placed all over the FPGA fabric (Figure 2.9) [43]. This approach breaks the link between ALMs and the Hyper Registers used to improve the critical paths, thereby improving the design efficiency. These hyper registers are integrated at the intersections of the horizontal and vertical routing segments within the FPGA. Unlike traditional registers, Hyper-Registers are bridgeable and lack an input routing multiplexer [42], making them more efficient without significantly increasing the silicon area.



- Registers are available in every routing segment
- Registers are available on all block inputs

Figure 2.9: Intel FPGA HyperFlex FPGA Architecture [43].

The Stratix 10 SX 2800 was selected over alternative devices such as GPUs, due to high power consumption and not being the best hardware for the research objective [45], and lower performance versions of the Stratix 10 family [44]. The HyperFlex architecture, combined with high resources such as ALM, DSP blocks, and M20K memory blocks make it ideal for HPC and DL inference tasks [42]. In addition, the integration of Intel Dev Cloud, oneAPI, and SYCL simplifies development workflow, an advantage that is not possible using hardware description languages and local boards. While the Stratix 10 NX offers specialized blocks for AI workloads, it has no availability on Intel Dev Cloud making it impractical for this project, but it is an alternative for future work. Therefore, the Stratix 10 SX 2800 emerged as the optimal choice for this investigation.

2.3 Tools for Development

Developing parallel and hardware designs requires specialized tools that enable efficient implementation. There are many development languages and programs for such aims. In

this section, some alternatives and contexts will be analyzed.

2.3.1 Hardware Description Languages

Hardware Description Languages (HDLs) are specialized computer languages used to describe the structure, design, and behavior of electronic circuits, particularly digital circuits such as processors. Developers not only need to worry about efficiency when programming but also want to have a correct hardware configuration for it. These languages allow designing specific hardware for a specific aim [46].

These languages have many differential features over conventional programming languages. They allow concurrent execution, which shows the paralleling nature of the HDLs, they also can simulate how a circuit will behave and see how the data is transferred in the lowest level.

The two most important HDLs are VHDL and Verilog. VHDL was first developed in the 1980s by the United States Department of Defense, but rapidly taken as a commercial hardware design language [47]. This language can describe all the hardware components that a system will have, without needing to have them, it can set how data is transferred through registers, and it also can configure the logic of the gates and interconnections [48]. Verilog was also created in the 1980s and became a standard for digital system design. Its syntax is similar to C language, making it easier for software developers to learn it [49]. The aim and possibilities are similar to VHDL.

2.3.2 Software Used: Intel oneAPI

Intel oneAPI Base Toolkit is a development platform that allows developers to create, test, and optimize high-performance applications across plenty of hardware architectures, including CPUs, GPUs, and FPGAs. One of its core features is the support for DPC++ (Data Parallel C++), an open-source compiler that supports the SYCL 2020 standard, thus allowing the compilation of parallel applications on several systems. SYCL is an industry-driven Khronos Group standard adding advanced support for data parallelism with C++ to sup-

port accelerated heterogeneous systems [50]. Therefore, thanks to the use of oneAPI and its DPC++ compiler, it is possible to create, compile and run applications on FPGAs using the common C++ language and its SYCL standard. As shown in Figure 2.10, Intel oneAPI fills the gap between high-level applications and hardware accelerators by providing a "level zero" interface to the hardware [51].

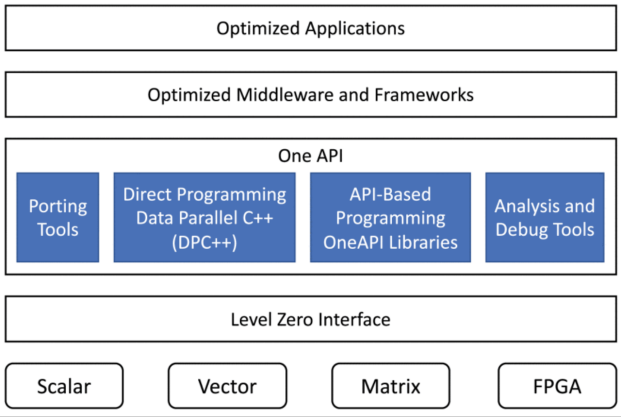


Figure 2.10: Intel oneApi enables integration between software and hardware [51].

In addition, Intel offers a free cloud-based ecosystem called Intel DevCloud. This is a development sandbox that allows you to use the Intel oneAPI Toolkit among other resources. One of the most important features is that it offers support with different intel FPGA boards. Therefore, by using Intel oneAPI Toolkit and DevCloud, developers can take advantage of Intel's powerful hardware accelerators without needing direct access to physical devices. The platform provides a rich environment for debugging and optimizing applications for performance and efficiency, making it a really useful tool for parallel programming or accelerating heterogeneous systems. In addition, it includes many tools to improve the programmer's quality of life especially for development on FPGAs, including:

- **FPGA Emulation**: It allows to quickly compile the program in the Intel Emulator Device, offering an alternative to test different approaches to the final version, saving compilation time.
- **Optimization Report**: The report generated after partial hardware compilation of

DPC++ for FPGA provides critical information for development, including views of structures, memory layout, performance, bottlenecks, and resource estimates. This step helps in developing and refining designs for better performance and more efficient use of resources (see Figure 2.11). Some key elements of the report are:

Visualizations: Illustrates how code structures are mapped onto the FPGA's resources, which helps identify inefficiencies and optimize the design. The report has three main visualizations: system viewer, memory viewer and schedule viewer.

Bottlenecks: The report highlights areas where performance is limited. The bottlenecks may arise from various factors, such as inefficient data paths, suboptimal memory access patterns, etc.

Resource Utilization Estimates: It provides an early assessment of how much of the FPGA's resources will be consumed by the design. This information is crucial to ensure that the design meets the requirements of the target FPGA prior to hardware compilation.

- **FPGA Hardware Imaging**: The final step produces the FPGA bitstream or RTL files, where the final design is compiled for deployment on actual hardware. During this stage, compilation also provides accurate information about the design's resource utilization, such as the use of logic elements, memory and DSP blocks and more. In addition, the compilation generates $fMax$ numbers, which determines the maximum clock frequency at which the FPGA can operate. This information is crucial for measuring the performance potential of the design.

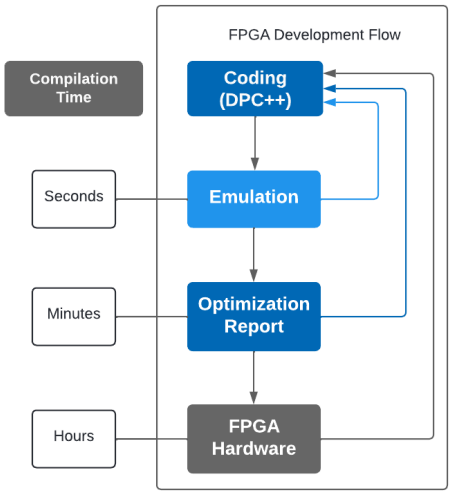


Figure 2.11: Intel oneAPI FPGA development flow (Own work).

Intel oneAPI Toolkit has been chosen to do this work for many reasons and advantages. It uses the C++ syntax, so it is easily learnable for developers, without having to learn new syntax and adapt to a new environment. For example, compared to VHDL, there are much more developers that know the C++ syntax. Furthermore, it also has free FPGA to use online and an advanced cloud system to control all the workflow. There is no need to invest in boards, everything is available in the cloud and for free.

This language offers a multitude of mechanisms that make it possible to control program execution devices and data input and output flows. The most important ones, which are necessary to understand the algorithm flow, will be explained below:

- **Kernels**: C++ with SYCL programs are single-source, meaning that the same source file contains both the code that defines the *kernel* to be executed on SYCL devices and also the host code that orchestrates the execution of those *kernels* [50]. The former will be commonly called **device code** while the latter will be called **host code**. Therefore, a *kernel* will be the code fragment that executes on the FPGA through a connection element called *queue*. The execution of these kernels is done asynchronously since the host sends the work to the queue and continues with its usual execution without waiting for results.

- **Data management**: In addition to controlling where our code runs through kernels, it is also important to control the memory of each of our devices. For proper load and store operations performance, the data should ideally be stored locally in the FPGA. However, even if the above-mentioned operations worsen, it is also necessary to transmit data between host and device (see Figure 2.12), such as the hyperspectral image or the result of the algorithms. For this data movement, there are *buffers*, which are an abstraction of data representing one or more objects of a C++ type [50]. To use them, it is necessary to create *accessors*, which provide information about how the data in the buffer is intended to be used, either read, write, or both.

- **Optimization flags**: Intel offers additional extensions to C++ with SYCL, which provide flags and attributes to optimize program development for specific devices, including FPGAs. It provides customization capabilities with attributes such as *scheduler_target_fmax_mhz* to set the maximum clock frequency of the program, and optimization with attributes such as *loop_fusion* or *max_interleaving*.

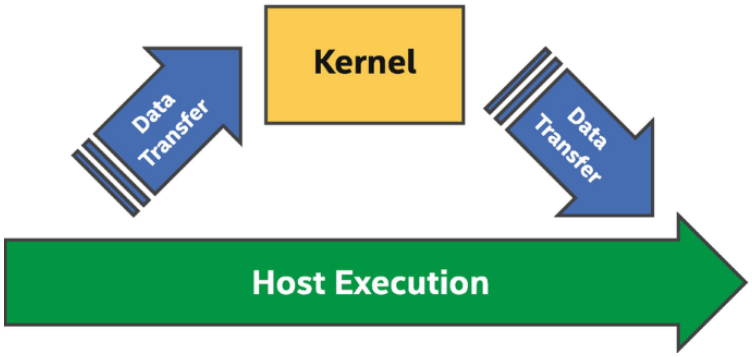


Figure 2.12: Data transfer between kernel and host [50].

Chapter 3: PCA

3.1 PCA Algorithm

The Principal Component Analysis (PCA) algorithm formally introduced by Harold Hotelling [52], is a multivariate statistical method used to reduce the dimensionality of large datasets while retaining as much relevant information as possible. It focuses on transforming a large number of intercorrelated quantitative variables into a smaller set of uncorrelated variables called principal components. These components are linear combinations of the original variables, and their calculation is based on the decomposition into eigenvectors and eigenvalues from the correlation matrix of the dataset. On the one hand, the eigenvectors indicate the directions of greatest variance. On the other hand, the eigenvalues measure the weight of the variances picked up by the eigenvectors. Thus, the principal components are ordered by their eigenvalues in a way where the first component explains most of the variance of the data, the second component explains most of the remaining variability by being orthogonal to the first component, and so on. Figure 3.1 shows an example of the principal component representation.

Despite being a method that allows numerous principal components to be obtained, some studies have shown that the first five bands contain practically all the information contained in the hyperspectral image [53]. In addition, this work only needs one band or principal component for the execution of the KNN algorithm [2]. This allows variations of the algorithm that optimize the calculation of the first principal component.

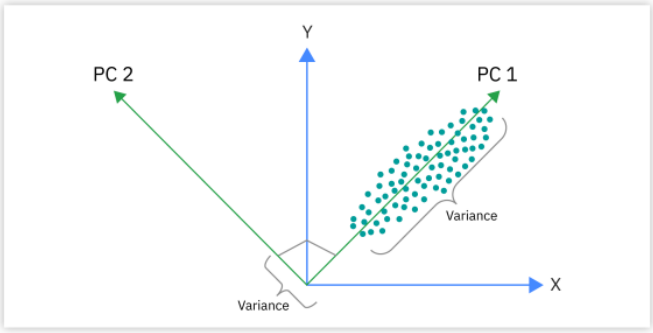


Figure 3.1: Example of orthogonal principal components [54].

The mathematical expression to calculate the PCA [55] of a hyperspectral image $\mathbf{X}$ is: $\mathbf{C} = \mathbf{V}\Lambda\mathbf{V}^{-1}$, where $\mathbf{C}$ is the covariance matrix of the image $\mathbf{X}$, $\mathbf{V} = [\mathbf{v}_1,\mathbf{v}_2,\ldots,\mathbf{v}_n]$ is an orthogonal matrix whose columns are the eigenvectors of $\mathbf{C}$ where n is the total number of spectral bands and Λ is a diagonal matrix containing the eigenvalues of $\mathbf{C}$.

The PCA can be divided into four main steps in the process of obtaining the main components. The steps are presented below and explained mathematically, detailing the implementation possibilities of each step and their suitability for a hardware implementation.

1. **Arrange the data in the original matrix**, calculating the mean of each of the spectral bands and subtracting it from the corresponding pixels, thus centering the matrix at the origin to increase the accuracy of the covariance matrix, eliminating biases caused by the heterogeneous distribution of the data.

2. **Calculation of the covariance matrix** of the original dataset, expressing the correlations between bands.

3. **Obtaining the eigenvalues and eigenvectors** of the covariance matrix, sorting them decreasingly according to their eigenvalue.

4. **Projection of the principal image** on the subspace of the chosen eigenvectors to obtain the desired principal components.

3.2 Base Implementation

This section describes in detail the PCA algorithm implementation used as a basis for this work, following the steps described above.

1. The data is organized in the form of a **two-dimensional** matrix of n data vectors $\mathbf{x}_1$,...,$\mathbf{x}_n$ that collects the pixels of each of the bands, where n is the number of spectral bands collected in the image. Each vector $\mathbf{x}_i$ will have p columns or variables, which collect the pixels. Therefore, a $p \times n$ matrix $\mathbf{X}$ is obtained. For computational reasons, the base algorithm transposes the $\mathbf{X}$ matrix into a $bands \times pixels$ matrix $\mathbf{Y}$ of size $n \times p$.

2. A one-dimensional vector $\mathbf{m}$ n -sized is calculated so that each element m_j contains the **mean** of the corresponding spectral band. Thus, the vector $\mathbf{m}$ will be $m[j] = \frac{1}{p}\sum_{i=1}^{n} Y[i,j]$. This vector is then subtracted from each corresponding pixel, centering the data at the origin and standardizing it. The centered data will be stored in a new $n \times p$ matrix $\mathbf{B}$.

3. Calculation of the **covariance matrix C** of size $n \times n$ such that $C = \frac{1}{p-1} B^T \cdot B$, where B^T is the transpose matrix of $\mathbf{B}$. The result will be a square matrix of size $n \times n$.

4. Computing the eigenvalues and eigenvectors of the covariance matrix using **Jacobi's method**. This is a method that consists of approximating the original matrix into a diagonal matrix by successive iterations of rotations [56]. The off-diagonal elements are converted to zero until all the off-diagonal elements are sufficiently small. For its implementation, two methods exist depending on how the element to be reduced is selected. The classical method, although it guarantees fewer iterations, is more computationally expensive because it looks for the largest element to minimize. That is why the cyclic method is implemented, which reduces the elements in order as they are encountered. Finally, the obtained eigenvalues can be expressed as $C_k = Q_1^T \cdot \cdots \cdot Q_k^T \cdot C \cdot Q_k \cdot \cdots \cdot Q_1$ where Q_i are the successive Jacobi rotation matrices and

C is the original covariance matrix. In turn, the eigenvectors are stored in a matrix **P** of the same size such that $P_k = Q_1 \cdot Q_2 \cdot Q_3 \cdot \cdots \cdot Q_k$.

5. The eigenvalues are stored in a one-dimensional auxiliary vector **r** of size n for reducing accesses to the matrix when reordering the eigenvalues such that $r[i] = C[i, i]$. Using the vector **r** the **Bubble Sort** method is applied to sort the eigenvectors from **P**, being stored in a same-sized matrix **O**, but with the elements ordered according to their eigenvalue.

6. Finally, the original matrix is projected onto the subspace determined by the principal components. For each original pixel centered at the origin, the scalar product is realized with the corresponding eigenvector, stored in a matrix **T** of size $p \times b$ where b is the number of chosen principal components. Thus, $T_{ij} = \sum_{k=1}^{b} B_{ki} \cdot O_{kj}$.

Taking into consideration all the steps described and the methodology to carry them out, the pseudocode corresponding to the base implementation is presented below:

Algorithm 1 Principal Component Analysis - PCA, Base Implementation

Input: Image
Output: PCA

```
# Step 0 - Initialization
VM ← 0
μ ← 0
for i = 0 to BANDS do
    for j = 0 to N do
        Y[ij] = X[ji]
    end for
end for
# Step 1 - Preprocessing
for i = 0 to BANDS do
    for j = 0 to N do
        μ[i]+ = Y[ij]
    end for
    μ[i] = μ[i]/N
end for
for i = 0 to BANDS do
    for j = 0 to N do
        B[ij] = Y[ij] − μ[i]
```

```
    end for
end for
# Step 2 - Covariance matrix
for i = 0 to BANDS do
    for j = i to BANDS do
        for k = 0 to N do
            VM[ij] += μ[ik] · B[jk]
        end for
        VM[ij] = VM[ij]/(N − 1)
        VM[ji] = Vm[ij]
    end for
end for
# Step 3 - Jacobi Method
P ← IdentityMatrix(BANDS)
r[BANDS] ← 0                                   ▷ Eigenvals Vector
O[BANDS × BANDS] ← 0                           ▷ Eigenvecs Ordered
ϵ < −0.00001
end ← 0
while end == 0 do
    end ← 1
    sum ← sum_diagonal(VM)
    for i = 0 to BANDS do
        for j = i + 1 to BANDS do
            if VM[ij] > ϵ · sum then
                end = 1
                α = VM[ij]/(VM[jj] − VM[ii])
                cosA = 1/(sqrt(α · α + 1))
                sinA = cosA · α
                for k = 0 to BANDS do
                    VM[ik] = cosA · VM[ik] − sinA · VM[jk]
                    VM[jk] = sinA · VM[ik] + cosA · VM[jk]
                end for
                for k = 0 to BANDS do
                    VM[ki] = VM[ki] · cosA − VM[kj] · sinA
                    VM[kj] = VM[ki] · sinA + VM[kj] · cosA
                    P[ki] = P[ki] · cosA − P[kj] · sinA
                    P[kj] = P[ki] · sinA + P[kj] · cosA
                end for
            end if
        end for
    end for
end while
r ← diagonal(VM)
```

```
62: s ← (0, 1, ..., BANDS − 1)                                  ▷ Sorting Order
63: for i = 0 to BANDS-1 do
64:     for j = i + 1 to BANDS do
65:         if r[j] > r[j − 1] then
66:             Swap(r[i], r[j])
67:             Swap(s[i], s[j])
68:         end if
69:     end for
70: end for
71: for i = 0 to PCA_BANDS do
72:     for j = 0 to BANDS do
73:         O[ji] = P[js[i]]
74:     end for
75: end for
76: # Step 4 - Projection
77: for i = 0 to N do
78:     for j = 0 to PCA_BANDS do
79:         for k = 0 to BANDS do
80:             PCA[ij]+ = B[ki] · O[kj]
81:         end for
82:     end for
83: end for
```

3.2.1 Algorithm Complexity

As can be seen in the pseudocode (see Algorithm 1), the base implementation is divided into 4 main steps, including a main initialization step where the input image X of structure $pixels \times bands$ is transposed into a matrix Y of structure $bands \times pixels$, with complexity:

$$\mathcal{O}(b \times p)$$

The preprocessing step has the same complexity since all the pixels are traversed for each of the bands, calculating the mean of each band and then traversing the matrix again and centering it by subtracting the mean. The complexity will therefore be:

$$\mathcal{O}(b \times p)$$

The calculation of the covariance matrix (step 2) is the second most complex part of the algorithm. Its calculation consists of the covariance of each pair of variables. For each band, the covariance of all pixels is calculated with the rest of the bands. Which means, going through all the bands and pixels for each band, resulting in a complexity of:

$$\mathcal{O}(b \times b \times p) = \mathcal{O}(b^2 \times p)$$

The Jacobi method step is the **most critical point in the algorithm**. Its complexity is given by an external while loop, which performs the iterations of the method and whose complexity depends on the convergence of the method. For the analysis of its complexity, $\mathcal{O}(bands^2)$ will be taken, since it is usually proportional to the square of the matrix size. As for the inner loops, they are three nested loops with a total complexity of $\mathcal{O}(bands^3)$. At the end of the method, the elements are sorted under a complexity $\mathcal{O}(bands^2)$. The total complexity of the Jacobi step is therefore :

$$\mathcal{O}(b^2 \times b^3 + b^2) = \mathcal{O}(b^5)$$

Finally, the matrix projection step runs through the transposed image (bands x pixels) by the number of principal components analyzed. Since only the first component is taken into account, the final complexity of this step is:

$$\mathcal{O}(b \times p)$$

3.2.2 Algorithm Resource Utilization

When analyzing the resource consumption of this base implementation, Table 3.1 shows the system's resources. These resources are related to the memory initialized throughout the program, as well as all hardware components such as ALUTs (Adaptive Look-Up Tables) or FF (Flip-Flops). Static Partition describes the resource consumption used by the entire system, i.e. all kernels, pipelines, and board logic. On the other hand, Kernel System and

more specifically **PCA - Q** explain the resource consumption made by the kernel in charge of the PCA, being **Q** the queue used for its implementation.

	ALUTs	FFs	RAMs	MLABs	DSPs
Static Partition	466792 (25%)	928428 (25%)	3039 (26%)	0 (0%)	1291 (22%)
Kernel System	76198 (4%)	107254 (3%)	665 (6%)	555 (<1%)	112 (2%)
Global Interconnect	3781	5400	121	0	0
PCA - Q	72415 (4%)	101783 (3%)	542 (5%)	555 (<1%)	112 (2%)

Table 3.1: FPGA's area estimates of PCA base implementation.

```
Compiler Error: The estimation of resources consumed by this design exceeds the size of the target device.
Compiler Error: If you wish to continue compilation anyways, use the flag "-Xsdont-error-if-large-area-est".
Compiler Error: The device 1SX280HN2F43E2VG contains 11721 RAM blocks, while the design is estimated to use 69090 (589%).
```

Figure 3.2: Hyperspectral data exceeding FPGA's RAM limits.

One of the most remarkable aspects is that the base implementation of the PCA hardly uses any RAM resources. This is mainly because the hyperspectral image information, the largest data, is stored in the host and transmitted to the FPGA through buffer accessors. If this data were stored in the FPGA RAM, they would need to use up to 589% of the FPGA RAM (see Figure 3.2). That is why even intermediate structures, such as the image centered at the origin, need a buffer accessor. Table 3.2 below shows the structures required for the PCA implementation and their size based on PB1C1 image.

Buffer	Dimensions	Type	Size of data Type	Total Size
Image Data (X)	$n \times p = 28,061,696$	Float	4 Bytes	107MB
Image Data BP (Y)	$p \times n = 28,061,696$	Float	4 Bytes	107MB
Centered Image (B)	$n \times p = 28,061,696$	Float	4 Bytes	107MB
Mean Vector (m)	$n = 128$	Float	4 Bytes	0.5KB
Covariance Matrix (C)	$n \times n = 16,384$	Double	8 Bytes	128KB
Eigenvectors Matrix (P)	$n \times n = 16,384$	Double	8 Bytes	128KB
Eigenvalues Vector (r)	$n = 128$	Double	8 Bytes	1KB
Eigenvectors Ordered (O)	$n \times n = 16,384$	Double	8 Bytes	128KB
PCA Output (T)	$p \times b = 219,232$	Double	8 Bytes	1.67MB

Table 3.2: Size of PCA's base implementation data according to PB1C1 image.

As can be inferred from Table 3.2, the structures corresponding to the hyperspectral image data as input, and the PCA result as output, are stored on the host and accessed

by read-only and write-only buffer accessors respectively. Limiting them exclusively to input and output avoids load and store delays in these structures throughout the algorithm. However, due to memory issues, both the transposed image and the centered image are also stored on the host. These last structures are used throughout the algorithm, causing delays due to memory accesses, as shown in Section 3.2.3. The rest of the structures presented in the table are properly stored locally in the FPGA, occupying a small amount of memory space.

3.2.3 Performance Analysis

As observed from the results on the maximum clock frequency achieved (see Figure 3.3), the target of 375 MHz is significantly higher than the actual result of 249 MHz. This discrepancy is primarily due to bottlenecks in the algorithm caused by memory dependencies.

Clock Frequency Summary

	Quartus Achieved Clock Frequency (MHz)	Compile Target Frequency (MHz)	Compile Estimated Frequency (MHz)
Kernel clock	249.00	375.00	375.00
Clock 2x	498.00		

Figure 3.3: PCA base implementation clock frequency summary.

These issues are directly related to the loops' Initiation Interval (II), which represents the number of clock cycles between the start of successive loop iterations. Since the target II of each loop is not specified, the compiler makes its best effort to maximize the maximum frequency, even if that means worsening the II of some of the loops [57]. Ideally, these loops should have an II of 1, which would mean that there are no data dependencies. However, many loops have a higher II than usual, some of them, such as the calculation of the centered matrix or the matrix projection are greater than 600. All these loops have in common read and write accesses to memory through accessors, with a **load operation of 634 cycles**.

This means that, if the memory dependency is not resolved by code restructuring, iterations of the loop cannot be chained.

Another problem encountered in these loops is that the memory accesses mentioned have the **stallable** category. This means that an attempt is being made to access the same memory port in the same clock cycle, so one of the instructions must be delayed. Finally, the kernel of the base implementation lasts **4230 clock cycles**, mainly determined by the input and output of data via the accessors. In addition, some operations, such as division operations, cause considerable latency.

In consideration of the execution time, the duration from the start of the kernel to the end of the PCA result has been taken into account. For the base implementation, the execution time is **93800 milliseconds**. Note that this time does not take into account the loading of the image and all the preparation of the algorithm.

3.2.4 Base Implementation Limitations

After analyzing the performance of the base implementation of the PCA from different points of view, several opportunities for improvement arise, both from the conceptual point of view and the mathematical methods used, as well as from the computational point of view and the applicable programming techniques.

- **At a conceptual level**, from the beginning of the explanation it has been shown that for the calculation of eigenvalues and eigenvectors it is not necessary to have a method such as Jacobi's method implemented. Although its accuracy and usefulness are very high for generalized implementations of the PCA, this case only needs the first band, so that much computational load is left over (in the matrix rotations, complexity of approximately $\mathcal{O}(N^5)$ is reached). In addition, the process of restructuring and reordering eigenvectors would be avoided. Another important aspect to restructure is the calculation of the image centered at the origin. Alternatives should be explored to avoid or reduce centering the entire image, since due to its size it cannot be stored in the FPGA, which means slower memory accesses.

- Regarding the **applicable programming techniques**, the alternatives offered by Intel oneAPI must be applied to optimize code performance. With the use of the report information, we should try to reduce accesses to shared memory outside the FPGA due to its high latency. Operations that read from and write to the same memory location, such as cumulative sums, should also be avoided, using instead auxiliary variables that reduce latency.

3.3 Final Implementation

This section explores the final implementation of the PCA for this work, reviewing the changes made and the reasons for them. The limitations explained in Section 3.2.4 leave a latent need to improve the algorithm's shortcomings, and the available optimizations will allow achieving very optimistic results for implementing this algorithm in a real environment.

3.3.1 Improvements

One of the first items addressed to make changes was the process of obtaining the eigenvectors and eigenvalues. In the original implementation of the PCA algorithm, the Jacobi method was used to calculate the eigenvalues and eigenvectors. While the Jacobi method is robust and provides all the eigenvalues and eigenvectors, it is computationally intensive and time-consuming. However, since the primary goal of this project is to extract only the first principal component, a more efficient approach was adopted: **the Power method**. The Power method is particularly well-suited for finding the largest eigenvalue and its corresponding eigenvector, making it ideal for this use case. Compared to other algorithms, such as the QR algorithm, which also provides all eigenvalues, the Power Method is significantly faster and less resource-intensive when only the dominant eigenvector is required.

The Power method, like the Jacobi method, is an iterative method in which the result converges to the eigenvector. Starting from a matrix A of size $n \times n$, it begins with a

non-zero initial vector x_0, forming a sequence such that:

$$
\begin{aligned}
x_1 &= Ax_0 \\
x_2 &= Ax_1 = A(Ax_0) = A^2x_0 \\
&\vdots \\
x_k &= Ax_{k-1} = A(A^{k-1}x_0) = A^kx_0
\end{aligned}
$$

Thus, with the correct number of iterations, a good approximation of the dominant eigenvector is achieved. For this implementation, at most 100 iterations will be performed until the difference between two consecutive eigenvalues is below the value of epsilon = 1e-6. In addition to the eigenvector, it is also necessary to calculate the eigenvalue, which is unknown. For this purpose, the **Rayleigh quotient** is used which, given an eigenvector x and a matrix A, its corresponding eigenvalue λ is calculated as $\lambda = \frac{Ax \cdot x}{x \cdot x}$. Thus, the dominant eigenvector and eigenvalue of the matrix are obtained, reducing the computational complexity of Jacobi's method.

Another point where improvements have been made is in the calculation of the covariance matrix. Storing the entire matrix was very costly in terms of FPGA memory. However, the new implementation solves these problems. Instead of centering the image and calculating the covariance matrix in steps, the operations are grouped in such a way that the mean pixel vector is calculated at the same time as a correlation matrix. The last step uses the latter by fitting it with the pixel mean to compute the covariance matrix. In this way, accesses to the original matrix are considerably reduced, enhancing local memory and grouping operations in the loops. Just as normalization is performed on the fly in the calculation of the covariance matrix, it is also performed in the projection of the matrix at the end of the algorithm.

The pseudocode of the final implementation of the PCA can be found below (see Algorithm 2). The improvements show an enhancement in complexity at various steps of the algorithm. Although adding the computation of the correlation matrix in step 1 increases the complexity to $\mathcal{O}(b^2 \times p)$, the complexity of the covariance matrix computation at step

2 becomes $\mathcal{O}(b^2)$, since it starts from the computed correlation matrix.

The aspect that improves the complexity the most has been the power method. The calculation of the eigenvectors and eigenvalues using Jacobi's method had a complexity of $\mathcal{O}(b^5)$, while the power method achieves a complexity of approximately $\mathcal{O}(b^3)$. The mentioned complexity is the worst case, since it would run 100 iterations to converge (approximates the number of bands given their similarity in the images used).

Algorithm 2 Principal Component Analysis - PCA, Final Implementation

Input: Image
Output: PCA

```
# Step 0 - Initialization
VM ← 0
CM ← 0
μ ← 0
# Step 1 - Correlation matrix and mean pixel
for i = 0 to N do
    for k = 0 to BANDS do
        μ[k]+ = X[ik]
    end for
    for j = 0 to BANDS do
        for k = 0 to BANDS do
            CM[jk] = X[ij] · X[ik]
        end for
    end for
end for
for i = 0 to BANDS do
    μ[i]/N
    for j = 0 to BANDS do
        CM[ij]/N
    end for
end for
# Step 2 - Covariance matrix
for i = 0 to BANDS do
    for j = 0 to BANDS do
        VM[ij] = CM[ij] − μ[i] · μ[j]
    end for
end for
# Step 3 - Power Iteration Method
x ← 0.1
v ← 0
```

```
31: ε < −0.000001
32: for i = 0 to 100 do
33:     for j = 0 to BANDS do
34:         for k = 0 to BANDS do
35:             v[i]+ = VM[ij] · x[j]
36:         end for
37:     end for
38:     eival ← 0
39:     for i = 0 to BANDS do
40:         eival+ = x[i] ∗ v[i]
41:     end for
42:     if (eival_i − eival_{i−1}) < epsilon then
43:         break
44:     end if
45: end for
46: # Step 4 - Projection
47: for i = 0 to N do
48:     for j = 0 to PCA_BANDS do
49:         for k = 0 to BANDS do
50:             PCA[ij]+ = (X[ik] − μ[k]) · x[k]
51:         end for
52:     end for
53: end for
```

3.3.2 Results

Algorithm Resource Utilization

The resource consumption of the final implementation of the PCA is presented below. As can be seen in Table 3.3, one of the most remarkable points is that the RAM memory consumption in the PCA has been reduced by 52.95% mainly due to the elimination of the matrices and vectors that were used to calculate all the eigenvectors and eigenvalues. Concerning the rest of the resources, an increase can be seen in all of them, is remarkable the consumption of ALUTs has grown by 93.23%, or the FFs by 31.49%. However, the overall consumption of the kernel is still quite optimal, since all the consumption percentages are below 10% of the total capacity of the FPGA, thus leaving space for the SVM and KNN algorithms.

	ALUTs	FFs	RAMs	MLABs	DSPs
Static Partition	466792 (25%)	928428 (25%)	3039 (26%)	0 (0%)	1291 (22%)
Kernel System	131944 (7%)	136162 (4%)	378 (3%)	989 (1%)	139 (2%)
Global Interconnect	2012	2253	121	0	0
PCA - Q	129930 (7%)	133838 (4%)	255 (2%)	989 (1%)	139 (2%)

Table 3.3: FPGA's area estimates of PCA final implementation.

Looking at the data structures used (see Table 3.4), the absence of the centered image reduces the input and output of data through buffer accessors to the host shared memory. This reduces latency considerably, limiting I/O operations to the input image and the PCA result respectively. It can also be seen that the original and sorted matrices where the eigenvectors were stored disappear since there is now only one eigenvector. The same happens with the eigenvalue, which has changed from being a vector of eigenvalues to a single variable.

Buffer	Dimensions	Type	Size of data Type	Total Size
Image Data (X)	$n \times p = 28,061,696$	Float	4 Bytes	107 MB
Mean Vector (m)	$n = 128$	Double	8 Bytes	1 MB
Correlation Matrix (CM)	$n \times n = 16,384$	Double	8 Bytes	128 KB
Covariance Matrix (A)	$n \times n = 16,384$	Double	8 Bytes	128 KB
Dominant Eigenvector (x)	$n = 128$	Double	8 Bytes	1 MB
Dominant Eigenvalue (λ)	1	Double	8 Bytes	8 B
PCA Output (T)	$p \times b = 219,232$	Double	8 Bytes	1.67 MB

Table 3.4: Size of PCA's final implementation data according to PB1C1 image.

Performance Analysis

The final implementation of principal component analysis (PCA) has demonstrated a significant improvement in performance and efficiency compared to the base implementation. The clock frequency increases from the previous 249 MHz to almost 300 MHz, while maintaining quite good loop II values. The loops with the highest II such as the image centering loop and the transpose loop have been eliminated thanks to the new algorithm logic, while the projection loop remains as the loop with the highest II with a value of 5. This optimization has been achieved by reducing latency in memory accesses through buffer accessors.

Concerning execution cycles, the kernel shows a total of 2187 cycles of latency, showing an improvement of approximately 48.2% over the 4230 cycles of the base optimization.

Finally, for the execution time of the final implementation, he obtained a result of 2530 milliseconds using the PB1C1 image. The speedup compared to the 93800 milliseconds of the base implementation is therefore 37.07 times.

$$S_{latency} = \frac{T_{old}}{T_{new}} = \frac{93800}{2530} = 37,07$$

Chapter 4: SVM

4.1 The SVM Algorithm

Support Vector Machine (SVM) is a supervised machine learning algorithm primarily used for classification tasks, although it can also be used for regression. The fundamental concept behind SVM is to find a hyperplane that best separates the data points of different classes. The objective is to maximize the margin, the distance between the hyperplane, and the nearest data points from each class, known as support vectors.

The SVM algorithm is used for many reasons, from binary classifying or multiple class classifying to irregularity detection. In this case, it will be used for binary classifying within a matrix and detecting an anomaly within it. This SVM is only a part of a full anomaly-detecting algorithm, so the main target will be to generate only an output of the binary classifiers ordered.

To define the algorithm's process and target, first, we have to decide in which groups are the objects going to be classified. Say that in this case, we have two groups, positive (represented by 1) and negative (represented by -1). The classifying algorithm decides to which group the object belongs. For that, first, we need to draw all the points defined on a plane. The SVM task is to calculate the classification function [38]. If all points are separable, then:

$$
\begin{aligned}
&w \cdot x_i - b \approx 1, \quad \text{if } y_i = 1, \\
&w \cdot x_i - b \approx -1, \quad \text{if } y_i = -1, \quad i = 1, \ldots, l
\end{aligned}
$$

This classification function separates all the objects into two parts with a hyperplane, as we can see in the Figure 4.1.

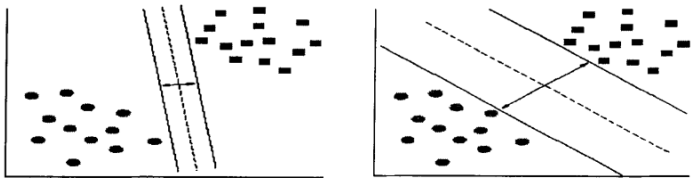


Figure 4.1: Example of a hyperplane [38].

The main aim of this classification in this SVM context is to isolate the anomaly with a hyperplane, maximizing the margin between the two parts of the plane. Figure 4.2 shows how an SVM nonlinear classification separates the anomaly from all other objects. The plane does not find how to separate all the points, it just finds the plane where the distance is maximized. As can be seen in the image, there is a red triangle within the hyperplane, but the only way to maximize the distance is to include it within the circle.

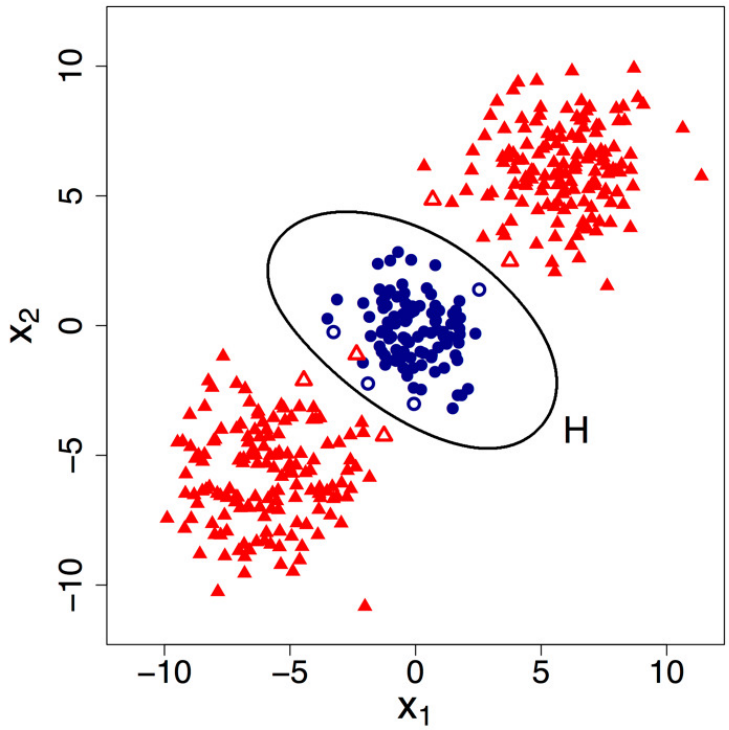


Figure 4.2: Nonlinear SVM classification of survey response [58].

For implementing this SVM algorithm, the process is going to be divided into four parts. First, it handles binary classification by finding an optimal hyperplane to separate two

classes. Then, it estimates class probabilities by converting decision values using a sigmoid function, refining these probabilities across classes. In decision-making, SVM selects the class with the highest probability as the predicted label. Finally, it orders these probabilities according to class labels, allowing for organized and interpretable output, which is crucial for further analysis or decision-making.

Binary Classification

In SVM, binary classification is the foundation. The goal is to find the optimal hyperplane that separates two distinct classes in a feature space. Given a set of training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$, where $\mathbf{x}_i \in \mathbb{R}^d$, where d is the dimensionality (for example, if $\mathbf{d}$ is 2 would be 2D and with 3 would be 3D), represents the feature vector and $y_i \in \{-1, 1\}$ represents the class label, the SVM seeks a hyperplane defined by $\overrightarrow{\mathbf{w}} \cdot \overrightarrow{\mathbf{x}} + \mathbf{b} = \mathbf{0}$ that maximizes the margin between the two classes. $\overrightarrow{\mathbf{w}}$ represents the weight vector, $\overrightarrow{\mathbf{x}}$ represents a feature vector in the input space, and $\mathbf{b}$ is the bias. The margin is the distance between the hyperplane and the nearest data point from either class.

The optimization problem for finding the hyperplane is:

$$\min_{\overrightarrow{\mathbf{w}},b} \frac{1}{2} \|\overrightarrow{\mathbf{w}}\|^2$$

subject to the constraints:

$$y_i(\overrightarrow{w} \cdot \overrightarrow{x}_i + b) \geq \mathbf{1} \quad \text{for all } i$$

This problem can be solved using Lagrange multipliers. Lagrange multipliers are a mathematical technique used in optimization problems to find the maximum or minimum of a function subject to constraints. It leads to the dual formulation:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$$

subject to:

$$0 \leq \alpha_i \leq C \quad \text{and} \quad \sum_{i=1}^{n} \alpha_i y_i = 0$$

Here, α_i are Lagrange multipliers, and C is a regularization parameter. The support vectors are the data points for which $\alpha_i > 0$. The first term is the summation of the Lagrange multipliers and the second involves the inner product of the feature vectors.

1. **Probability Estimation**:

   SVMs traditionally provide a hard classification decision, not probabilities. To estimate probabilities, a common approach is to fit a sigmoid function [59] (a mathematical formula that maps any real value to a range between 0 and 1) to the decision values $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$. This function takes the form:

   $$\sigma(y = 1|x) = \frac{1}{1 + e^{Af(x)+B}}$$

   The parameters A and B are estimated using maximum likelihood estimation on a validation set, where $f(\mathbf{x})$ is the SVM decision value. This probability estimation allows for a probabilistic interpretation of the SVM's output, which is particularly useful in multi-class classification. This provides a probabilistic measure of the classification outcome, which can be useful for understanding the confidence of the prediction and for tasks requiring probability estimates.

2. **Decision Making**:

   In multi-class classification, SVM uses a one-vs-one or one-vs-all approach. In the one-vs-one approach, a binary classifier is trained for each pair of classes, resulting in $\frac{C \times (C-1)}{2}$ classifiers for C classes. The decision-making process involves combining the outputs of these binary classifiers. Typically, a voting scheme is used, where each classifier votes for one class, and the class with the most votes is chosen as the final decision. Alternatively, in a probabilistic setting, the class with the highest probability estimate (from the sigmoid function) is selected.

In the one-vs-all approach, a classifier is trained for each class against all others. The class with the highest decision value is selected as the predicted class.

3. **Result Ordering**:

   After computing the probabilities for each class, the results are ordered according to class labels. This involves storing the probability estimates in a structured manner, often in a matrix form where each row corresponds to an input sample, and each column corresponds to a class. The ordering ensures that the results are easily interpretable and can be used for further decision-making processes, such as ranking, thresholding, or calculating confidence intervals. The structured output is essential for tasks like ensemble learning or hierarchical classification, where multiple classifiers' outputs need to be combined.

4.2 Base Implementation

In this section, it will be explained how the base code (Algorithm 3) separates the SVM algorithm and the complexity of each part of the algorithm. The SVM used in this work will calculate the decision values with probability estimation, will take the decision, and finally sort the result. All the implemented parts are a modification of the SVM explained in the previous sections with computational adaptations. The images will be two-dimensional, so they can be represented as a matrix.

There can be modifications between the previous mathematical explanation and this exact implementation, to let the code use the algorithm for this specific context. The code is a C language adapted version of the SVM algorithm. It will be split into four parts, as explained in the previous section. The base algorithm will gather the first three: binary classification, probability estimation and decision-making. The result will be ordered within the main kernel but after the main part. PB1C1 image will be used to analyze the performance in all the SVM sections below.

Algorithm 3 Support Vector Machine - SVM, Base Implementation

Output: SVM

```
# General for loop
for p = 0 to N do
    p ← 0
    classifier ← 0
    for b = 0 to C do
        for k = b + 1 to C do
            sum1 ← 0.0
            for j = 0 to B do
                sum1 += IMAGE[q × B + j] ∗ W[clasificador × B + j]
            end for
            dec_values[classifier] = sum1 − ρ[p]
            σ_fApB[0] = dec_values[clasificador] × probA[p] + probB[p]
            if σ_fApB[0] ≥ 0.0 then
                σ[0] ← exp(−σ_fApB[0]) / (1.0+exp(−σ_fApB[0]))
            else
                σ[0] ← 1.0 / (1.0+exp(σ_fApB[0]))
            end if
            if σ[0] < min_prob then
                σ[0] ← min_prob
            end if
            if σ[0] > max_prob then
                σ[0] ← max_prob
            end if
            P[b ∗ C + k] = σ[0]
            P[k ∗ C + b] = 1 − σ[0]

            p ← p + 1
            p ← classifier + 1
        end for
    end for
    p ← 0
    for b = 0 to C do
        prob_estimates[b] = 1.0/C
        Q[b × C + b] = 0.0
        for j = 0 to b do
            Q[b × C + b] += P[j × C + b] × P[j × C + b]
            Q[b × C + j] = −P[j × C + b] × P[b × C + j]
        end for
        for j = b + 1 to C do
            Q[b × C + b] += P[j × C + b] × P[j × C + b]
            Q[b × C + b] = −P[j × C + b] × P[j × C + b]
```

```
        end for
    end for
    iterations ← 0
    stop ← 0

    while stop = 0 do
        pQp ← 0
        for b = 0 to C do
            Q_p[b] ← 0
            for j = 0 to C do
                Q_p[b] += Q[b × C + j] × prob_estimates[j]
            end for
            pQp[0] += prob_estimates[b] × Q_p[b]
        end for
        max_error ← 0.0
        for b = 0 to C do
            max_error_aux = Q_p[b] − pQp[0]
            if max_error_aux < 0.0 then
                max_error_aux = −max_error_aux
            end if
            if max_error_aux > max_error then
                max_error[0] = max_error_aux
            end if
        end for
        if max_error[0] < ε then
            stop = 1
        end if
        if stop = 0 then
            for b = 0 to C do
                diff_pQp = (−Q_p[b]+pQp) / (Q[b×C+b])
                prob_estimates[b] = prob_estimates[b] + diff_pQp
                pQp = (pQp+pQp_diff × (pQp_diff × Q[b×C+b]+2×Q_p[b])) / (1+pQp_diff)
                pQp ← pQp / (1+pQp_diff)

                for j = 0 to C do
                    Q_p[j] = (Q_p[j] + pQp_diff × Q[b × C + j])/(1 + pQp_diff)
                    prob_estimates[j] = prob_estimates[j]/(1 + pQp_diff)
                end for
            end for
        end if
        iterations ← iterations + 1
        if iterations = 100 then
```

```
84:         stop = 1
85:       end if
86:     end while
87:     for b = 0 to C do
88:       prob_estimates_result[q * C + b] = prob_estimates[b]
89:     end for
90:     decision ← 0
91:     for b = 1 to C do
92:       if prob_estimates[b] > prob_estimates[decision] then
93:         decision = b
94:       end if
95:     end for
96: end for
```

4.2.1 Algorithm Complexity

This algorithm is mostly within a for loop, which goes through all the pixels. All initial three parts will be in that loop, and the fourth one (sorting) will not be within it.

From line 5 to line 29 in Algorithm 3, the algorithm starts to calculate the decision values and calculates the sigmoid prediction for each point.

The decision value sum1 is calculated by summing the product of the input features (imageIn) and the corresponding weights ($\overrightarrow{w}$) over numberOfBands bands. This sum is a linear combination representing the decision boundary between two classes. The decision value (sum1) is adjusted by subtracting a bias term (ρ), which accounts for the offset in the linear decision function. This adjusted value is stored in decValues[classification]. The adjusted decision value is then used to compute the argument for the sigmoid function. This is done by multiplying the decision value by ProbA[p] and adding ProbB[p]. These coefficients (probA and probB) are parameters that help in adjusting the σ function to better fit the data.

The σ function is then applied to σFApB and σFApA to obtain a probability. The σ function is given by:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

After this, the probability is then clipped to be within the range [minProb, maxProb] to

prevent it from being too close to 0 or 1, which might cause issues in later stages of processing. The total mathematical complexity of this section is $\mathcal{O}(C^2 \times B)$. In the next loop (from line 31 to 38) the Algorithm 3 updates the diagonal elements by summing the squared pairwise probabilities for each class b. The mathematical complexity for this part would be $\mathcal{O}(C^2)$.

The third loop goes from line 41 to 78. This loop calculates an intermediate value of pQp and a temporary matrix multiProbQp based on the currently calculated probability estimates, then, computes the maximum error between multiProbQp and pQp and stops if the maximum error is below the epsilon threshold. After that, updates the probability estimates matrix. To avoid infinite loops, the number of iterations is limited to 100. The complexity of the loop would be $\mathcal{O}(C^2)$. After this loop, an index mapping sort is implemented to sort the answer, with a complexity of $\mathcal{O}(N \times C)$. All these loops are gathered inside the main loop, with a $\mathcal{O}(N)$ mathematical complexity. To conclude, the full algorithm complexity will be the one of the outer loop multiplied by the complexity within that loop.

$$\mathcal{O}((N \times (C^2 \times B + C^2 + C^2)) + (N \times C)) = \mathcal{O}(N \times C^2 \times B)$$

4.2.2 Algorithm Resource Utilization

Table 4.1 shows some of the utilization values. SVM is supposed to use fewer resources than PCA, to enable both to be executed in parallel [2]. However, this version uses more resources than the PCA.

	ALUTs	FFs	RAMs	MLABs	DSPs
Static Partition	466792 (25%)	928428 (25%)	3039 (26%)	0 (0%)	1291 (22%)
Kernel System	71363 (4%)	230085 (6%)	1562 (13%)	982 (1%)	52 (<1%)
Global Interconnect	18860	67276	284	0	0
SVM Kernel	52501 (3%)	162738 (4%)	1276 (11%)	982 (1%)	52 (<1%)

Table 4.1: FPGA's area estimates of SVM.

The logic units' high usage can be because of the poor optimization and the low memory usage (RAM) because most of the algorithm data is stored in the host and accessed through accessors. All the accessors and memory will be analyzed to see how it is distributed.

The SVM algorithm uses plenty of data to calculate the result. All this data is obtained through accessors in this initial code.

The algorithm uses many resources to access the data in Table 4.2, as all of them are stored in the host and not in the kernel itself. The 21 accessors are in read and write mode, so they do not only read the data from the host but also need to store it each time they change it. The size of each variable despite the number of pixels size ones is small enough to be stored within the FPGA, but they are all stored in the host.

Accessor	**Dimension.**
w(read_write)	$BC \times B$
dec_values(read_write)	BC
σ_{fApB}(read_write)	1
σ(read_write)	1
P(read_write)	$C \times C$
$prob\_estimates$(read_write)	C
Q(read_write)	$C \times C$
Q_p(read_write)	C
pQp(read_write)	1
pQp_{diff}(read_write)	1
max_error(read_write)	1
$prob\_estimates\_result$(read_write)	$N \times C$
ρ(read_write)	BC
$probA$(read_write)	BC
$probB$(read_write)	BC
$label$(read_write)	C
$prob\_estimates\_result\_ordered$(read_write)	$N \times C$
$image\_in$(read_write)	$N \times B$
aux(read_write)	1
acc_aux2(read_write)	1
ϵ(read_write)	1

- BC = Binary Classifiers, B = Bands, N = Number of Pixels, C = Number of Classes

Table 4.2: SVM accessors and size.

4.2.3 Performance Analysis

The schedule viewer shows how is the order of the tasks completed by the FPGA [57]. Figure 4.3 shows how is the schedule viewer for the SVM initial algorithm implementation. The base design has **2450** cycles of latency.

Figure 4.3: Base SVM schedule viewer.

This first implementation of the SVM algorithm takes **68643** milliseconds (ms) to execute on average, which seems an excessive time for this type of algorithm. Plenty of implementation and optimization mistakes need to be corrected to bring the FPGA execution to a better version.

4.3 Final Implementation

Optimizing SVM algorithms for FPGA implementation represents a significant advancement in accelerating machine learning tasks. FPGAs offer a unique blend of flexibility and performance, making them ideal for enhancing the computational efficiency of SVMs, particularly in real-time and resource-constrained environments. The inherent parallelism of

FPGA architecture allows for simultaneous processing of multiple data streams, dramatically reducing the time required for training and inference.

This improvement focuses on tailoring the SVM algorithm to leverage FPGA's parallel processing capabilities, optimizing data flow, and minimizing latency. By carefully designing the computational pipelines and memory access patterns, the enhanced SVM implementation on FPGA achieves substantial speed-ups compared to traditional CPU or GPU implementations. This results in faster classification times and lower energy consumption, making the optimized SVM highly suitable for embedded systems, IoT devices, and real-time applications where power efficiency and performance are critical.

Many parts of the code shown in the previous part have been improved and many others remain unchanged. The result-oriented research of improvements has shown that some apparent inefficient parts are more efficient than initially appeared. Every movement and change has been tested to see how the result changes and all changes that slow down the algorithm have been deleted. The pseudocode of the final implementation of the SVM can be found below (see Algorithm 4):

Algorithm 4 Support Vector Machine - SVM, Final Implementation

Output: SVM

```
# General for loop
for p = 0 to N do
    p ← 0
    classifier ← 0
    for b = 0 to C do
        for k = b + 1 to C do
            sum1 ← 0.0
            for j = 0 to B do
                sum1 += IMAGE[q × B + j] ∗ W[clasificador × B + j]
            end for
            dec_values[classifier] = sum1 − ρ[p]
            σ_fApB[0] = dec_values[clasificador] × probA[p] + probB[p]
            if σ_fApB[0] ≥ 0.0 then
                σ[0] ← exp(−σ_fApB[0]) / (1.0+exp(−σ_fApB[0]))
            else
                σ[0] ← 1.0 / (1.0+exp(σ_fApB[0]))
            end if
```

```
        P[b * C + k] = σ[0]
        P[k * C + b] = 1 − σ[0]

        p ← p + 1
        p ← classifier + 1
    end for
end for
p ← 0
for b = 0 to C do
    prob_estimates[b] = 1.0/C
    Q[b × C + b] = 0.0
    for j = 0 to C do
        if j > b then
            sum +← P[j × C + b] × P[j × C + b]
            Q[b × C + j] ← −P[j × C + b] × P[b × C + j]
        else
            sum +← P[j × C + b] × P[j × C + b]
            Q[b × C + b] ← −P[j × C + b] × P[j × C + b]
        end if
    end for
end for
iterations ← 0
stop ← 0

while stop = 0 do
    pQp ← 0
    for b = 0 to C do
        Q_p[b] ← 0
        for j = 0 to C do
            Q_p[b]+ = Q[b × C + j] × prob_estimates[j]
        end for
        pQp[0]+ = prob_estimates[b] × Q_p[b]
    end for
    max_error ← 0.0
    for b = 0 to C do
        max_error_aux = Q_p[b] − pQp[0]
        if max_error_aux < 0.0 then
            max_error_aux = −max_error_aux
        end if
        if max_error_aux > max_error then
            max_error[0] = max_error_aux
        end if
    end for
```

```
        if max_error[0] < ε then
            stop = 1
        end if
        if stop = 0 then
            for b = 0 to C do
                diff_pQp = (−Q_p[b]+pQp) / (Q[b×C+b])
                prob_estimates[b] = prob_estimates[b] + diff_pQp
                pQp = (pQp+pQp_diff×(pQp_diff×Q[b×C+b]+2×Q_p[b])) / (1+pQp_diff)
                pQp ← pQp / (1+pQp_diff)

                for j = 0 to C do
                    Q_p[j] = (Q_p[j] + pQp_diff × Q[b × C + j])/(1 + pQp_diff)
                    prob_estimates[j] = prob_estimates[j]/(1 + pQp_diff)
                end for
            end for
        end if
        iterations ← iterations + 1
        if iterations = 100 then
            stop = 1
        end if
    end while
    for b = 0 to C do
        prob_estimates_result[q ∗ C + b] = prob_estimates[b]
    end for
    decision ← 0
    for b = 1 to C do
        if prob_estimates[b] > prob_estimates[decision] then
            decision = b
        end if
    end for
    for b = 0 to C do
        position ← label[b] − 1
        result[q ∗ C + position] = prob_estimates[b]
    end for
end for
return prob_estimates
```

4.3.1 Improvements

There are six versions of the SVM algorithm. Each one includes changes that improve the execution time or resource utilization. In this case, as SVM does not use many resources, all changes improve the kernel execution time. There are six versions of the algorithm: from v1 (base algorithm) to v6 (last version).

The second version (v2) is the version with the highest speedup. It solves the accessor problem explained in previous versions. The v1 code has many read and write accessors that do not let the FPGA compute smoothly. These are the remaining accessors:

Accessor	**Dimension**
$w(read)$	$BC \times B$
$prob\_estimates\_result(read)$	$N \times C$
$\rho(read)$	BC
$probA(read)$	BC
$probB(read)$	BC
$label(read)$	C
$prob\_estimates\_result\_ordered(read\_write)$	$N \times C$
$image\_in(read)$	$N \times B$
$\epsilon(read)$	1

Table 4.3: SVM accessors and size.

All the remaining ones are only read accessors, despite the result accessor, which needs to be read and written. As it is the result, the FPGA only writes, but it is read too for the correct behavior of DPC++.

With this change, the FPGA does not need to be reading and writing for all the variables but it only reads the input arrays, matrixes, and variables, and stores them if there is enough space within the RAM.

The resulting execution time with this improvement is 6579 milliseconds on average, which is 10.42 times faster.

The v3 implementation tries to use the "pragma omp parallel for reduction" statement for parallelizing the for loops within the while loop, but the result is not as good as it seems to be. The performance is reduced and now the execution time is 6798 milliseconds on

average. All these changes are deleted to return to the v2 version as the performance is quite better.

v4 improves the for loop between lines 50 and 58 in Algorithm 4 and between lines 32 and 43 in Algorithm 3. It replaces the two for loops within the main for in these lines with just one loop and two conditional statements. That leads to a 6469 milliseconds execution, which is a 10.61 times better than v1.

The changes on v5 delete another two accessors from the list. ϵ is declared within the kernel, so there is no need to access the host as it is a constant value, and "prob_estimates_result" is deleted because "prob_estimates_result_ordered" is going to be the only output. This makes the execution time even lower, leading to a 5721 milliseconds execution time, which is 11.99 times better than v1. The new accessor list can be seen in Section 4.3.1.

Accessor	**Dimension**
$w(read)$	$BC \times B$
$\rho(read)$	BC
$probA(read)$	BC
$probB(read)$	BC
$label(read)$	C
$prob\_estimates\_result\_ordered(read\_write)$	$N \times C$
$image\_in(read)$	$N \times B$

Table 4.4: SVM accessors and size.

The last functional version is v6. It restructures the sorting part of the SVM (last step). In this new version, the sorting is included within the main loop and not outside, which reduces the mathematical complexity of the sorting in C, as it uses the main for loops. With this change, v6 is executed in 5662 milliseconds.

From the initial version to the last one, there is a 12.12 speedup. Changes in the code have substantially improved the performance of the code. The following table shows all the explained changes and results:

Version	Execution Time (ms)	Speedup
v1	68643	1.00
v2	6579	10.43
v3	6798	10.10
v4	6469	10.61
v5	5721	11.99
v6	5662	12.12

Table 4.5: Execution time and speedup across versions.

4.3.2 Results

Algorithm Resource Utilization

In the initial implementation of the SVM algorithm code, there is a very low use of the FPGA resources. This algorithm should use fewer resources than the PCA or KNN algorithms, anyway, the usage is still low. Table 4.6 shows some of the utilization values.

	ALUTs	FFs	RAMs	MLABs	DSPs
Static Partition	466792 (25%)	928428 (25%)	3039 (26%)	0 (0%)	1291 (22%)
Kernel System	51277 (3%)	95909 (3%)	462 (4%)	461 (<1%)	79.5 (1%)
Global Interconnect	2030	3524	121	0	0
SVM Kernel	49245 (3%)	92314 (2%)	339 (3%)	461 (<1%)	79.5 (1%)

Table 4.6: FPGA's area estimates of SVM.

Comparing this table with the previous one (Table 4.1), can be seen how the use of resources has decreased. The kernel storing of the variables has mainly improved these statistics. The FPGA had enough logic units and storage, but this can make it easier to later use the SVM in parallel with other algorithms that require more storage.

Performance Analysis

The performance has been improved too. The next graph shows how the cycle delay is much lower than the one in Figure 4.3. The latency in Figure 4.3 was **15362** cycles and in the final version shown in Figure 4.4 is **1450** cycles. Now the improved parts such as the host data accessing processes (Figure 4.4 B5) are much less significant.

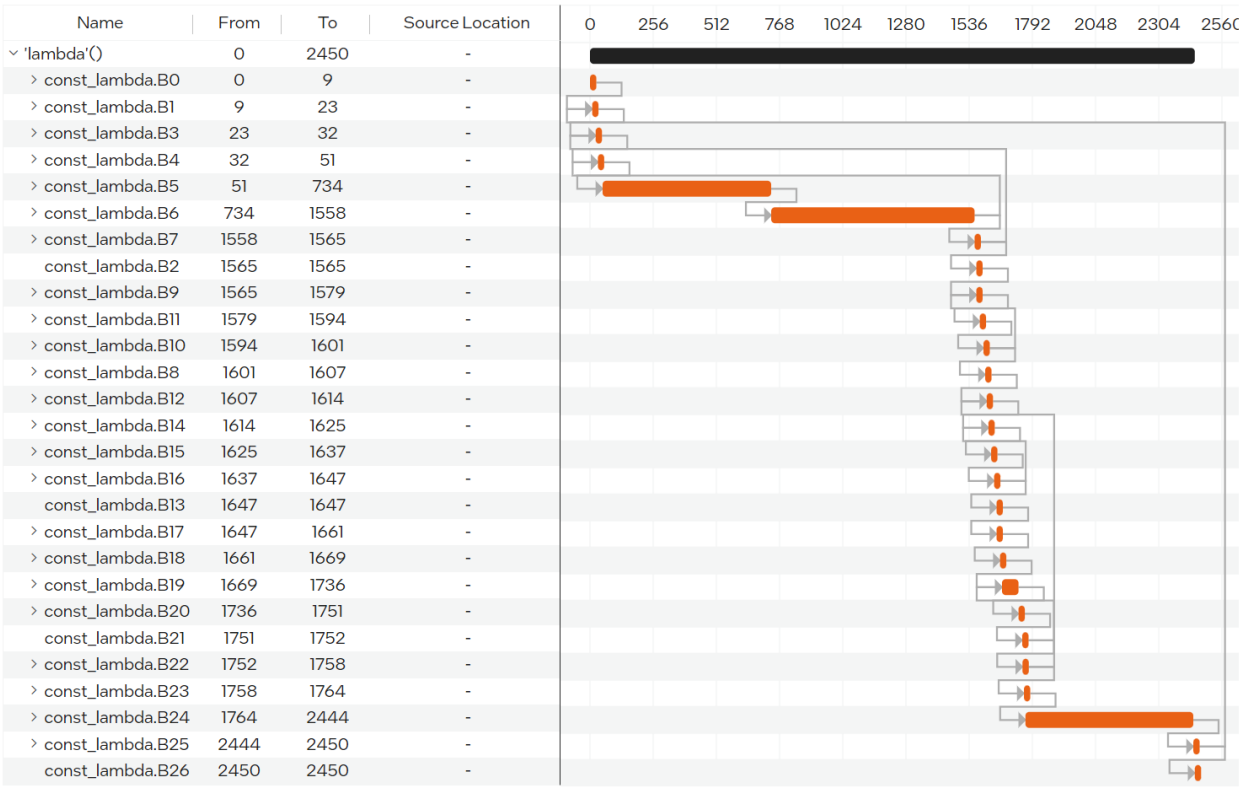


Figure 4.4: Final SVM schedule viewer.

Chapter 5: KNN

5.1 KNN Algorithm

K-Nearest Neighbors (KNN) is a lazy non-parametric supervised learning algorithm used for classification and regression [60]. The algorithm is lazy in the sense that it does not perform any training and delegates all computation to prediction [61]. It is non-parametric, since no assumption is made about the function of the KNN. Finally, it is considered a discriminative model because it models the conditional probability of a sample belonging to a given class [61].

One of the main purposes of the KNN Algorithm is the computer vision and pattern recognition, key aspects in the medical analysis of Hyperspectral Images [61]. Based on the initial thoughts of Evelyn Fix & Joseph Hodges [62], is used to estimate the density function $F(x/C_j)$ of the predictors x for each class C_j. The output is the class based on the probability that an element x belongs to the class C_j [63]. In this hyperspectral image classification, the classes are labeled by the SVM algorithm (Chapter 4). The focus of the algorithm is to find the K pixels with the smallest distance from the reference pixel within a local area, defined by the Window Size (W) and each pixel has a specific value obtained from the PCA step.

Formal definition of the algorithm

Let $\mathcal{D} = (v_1, P_1), (v_2, P_2), ..., (v_n, P_n))$ be a labeled dataset, where each $v_i \in \mathbb{R}^z$ is a data point in a z-dimensional feature space. The feature space has $z = 3$ because each data point v_i is defined as $v_i = (PCA_i, r_i, c_i)$, a vector consisting of:

- PCA_i: The PCA value of the i-th data point.
- r_i: The row index in the original matrix.

- c_i: The column index in the original matrix.

$P_i = (p_{i1}, p_{i2}, ..., p_{it})$ represents the SVM derived classification probabilities for each of the t classes.

Given a query point $v_q = (PCA_q, r_q, c_q)$ the KNN algorithm within a local zone defined by a window size W proceeds as follows:

1. **Define the Local Window**: Identify the local zone around the query point v_q based on the window size W, where left and right parts of W are denoted as SW, being SW the half of the size of W. The local zone $\mathcal{L}_W(v_q)$ is a subset of $\mathcal{D}$ including all data points within the window:

$$\mathcal{L}_W(v_q) = \{v_i \in \mathcal{D} | (rc_q - SW \leq rc_i \leq rc_q + SW)\}$$

   The local zone includes all points within a square region centered at rc_q or (r_q, c_q), in which the bounds of the distance calculations from the query point are defined by W.

2. **Distance Calculation**: Compute the Squared Euclidean distance between the query point v_q and each data point v_i in the local zone $\mathcal{L}_W(v_q)$:

$$d(v_q, v_i) = (v_q - v_i)^2 = (PCA_q - PCA_i)^2 + (r_q - r_i)^2 + (c_q - c_i)^2$$

   The main reason for calculating the Squared Euclidean distance, omitting the square root, is that in the algorithm the focus is on comparing distances to one another. The numerical value itself is not necessary, but the order is, therefore omitting the calculation of the square root gives the same result [64].

3. **Identification of the k Nearest Neighbors**: From the local zone $\mathcal{L}_W(v_q)$, select k nearest neighbors based on the computed Euclidean distances:

$$\mathcal{N}_k(v_q) = \text{the } k \text{ data points in } \mathcal{L}_W(v_q) \text{ with the smallest distances } d(v_q, v_i)$$

4. **Vote Using SVM Probabilities**: For each class $j \in 1, 2, ..., t$, compute the weighted sum of SVM probabilities over the k nearest neighbors and normalize by dividing by the k value to obtain the average probability of each class:

$$\bar{S}_j = \frac{S_j}{k} = \frac{1}{k} \sum_{v_i \in \mathcal{N}_k(v_q)} p_{ij}$$

   Where p_{ij} is the probability that the i-th nearest neighbor belongs to the class j.

5. **Class Assignment**: Assign the class label $\hat{y}_q$ to the query point v_q based on the class with the maximum average probability

$$\hat{y}_q = arg \max_{j \in \{1,2,...,t\}} \bar{S}_j$$

5.2 Base Implementation

The implementation of the original source code consists of three main phases: initialization of the feature matrix, the search of the K neighbors in a local window, and finally the calculation of the average probabilities. The pseudocode where the mentioned steps are implemented can be found below (see Algorithm 5). One of the key aspects is the sliding window, since at the top and bottom of the image it has variable limits that define the region where the distances from the query pixel are calculated. The operation of the sliding window can be seen below:

Example: Window boundaries and distance calculation

The explanation of the mechanism of the window on which the distances are calculated for the query pixel can be seen in the Figures 5.1 to 5.3.

At the beginning of the KNN (see Figure 5.1), the query pixels are at the top of the PCA band, having the problem that the window could not be extended on the *left side* until reaching a particular pixel, so the maximum capacity of the window $W^{'}$, at the beginning, is

SW, in other words, half of W. The *right side* of the window is increased for each iteration that proceeds in the calculation of the K-neighbors of the reference pixel, this window W' is increased until reaching the maximum window size W. Summarizing, only the upper limit W_{sup} is increased, keeping constant the lower limit W_{inf} of the window, or left and right side respectively. This causes that the number of iterations to find the K-neighbors in the distance vector D to vary, as seen in line 28 of the Algorithm 5.

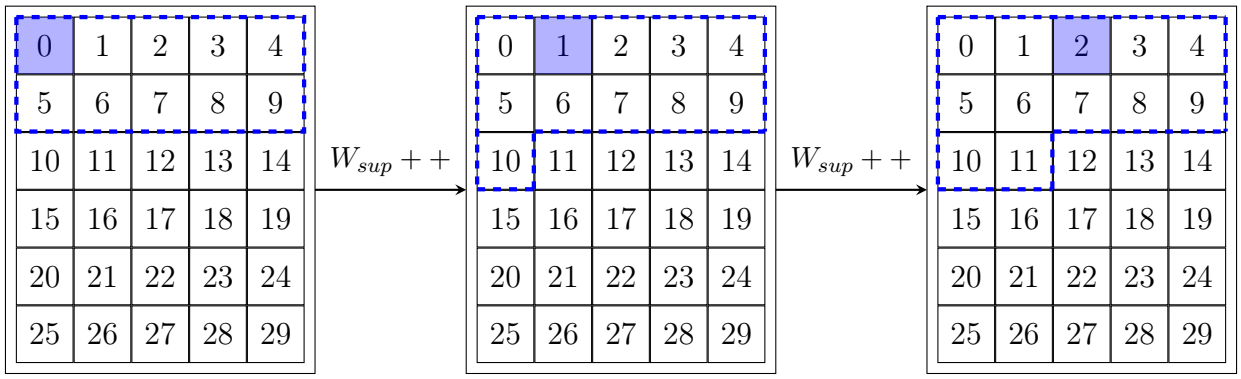


Figure 5.1: Top window.

As soon as the maximum window size is reached, meaning that both the left and right sides summation up to the size W, it remains constant. For each iteration of the assignment of the K-neighbors to the pixel i, both W_{inf} and W_{sup} are increased (see Figure 5.2). This denotes a region, which makes up a large part of the image, with a constant window size, not having the problem of the variable width of the top window, and as discussed below, of the bottom window. In summary, this region of the image is key to achieve further optimization to reduce the execution time of the algorithm.

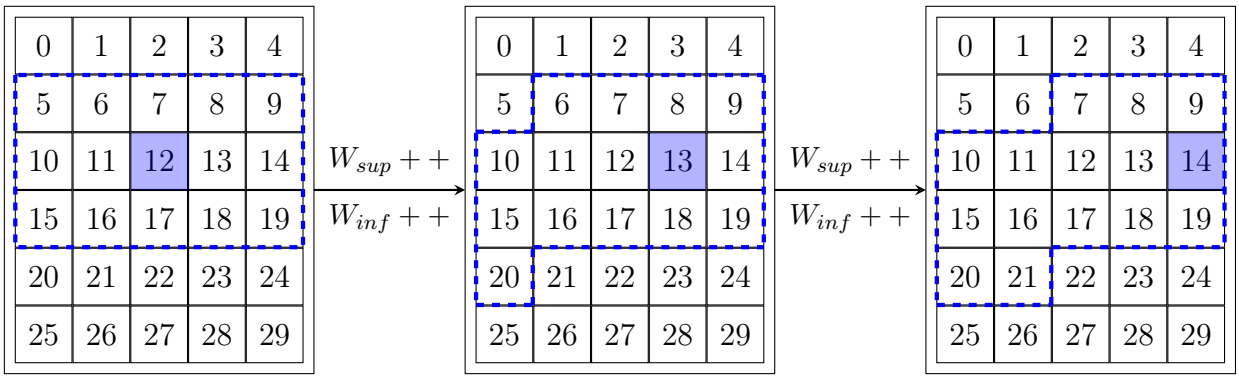


Figure 5.2: Constant size window.

Finally, near the end of the matrix processing (see Figure 5.3), the same situation arises as in the top window, but this time reversed. As the query pixel approaches to the end, the window size W decreases, until it reaches the last value where $W^{'}$ has the value of SW. This is achieved by keeping the W_{sup} fixed and increasing W_{inf} at each iteration, causing $W^{'} = W_{sup} - W_{inf}$ to become smaller and smaller.

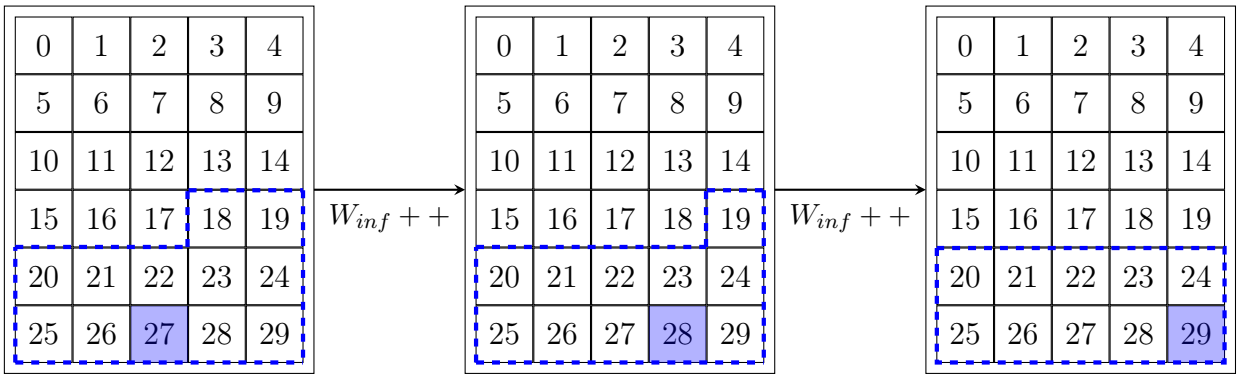


Figure 5.3: Bottom window.

5.2.1 Algorithm Complexity

The Algorithm 5, mainly divided into three main sections that traverse all pixels can break down the complexity around the main loops as follows.

The first part, the initialization of the **Feature matrix initialization** has two nested loops with a total number of iterations of LINES × SAMPLES, equivalent to the number of pixels N. The total mathematical complexity of this part is:

$$\mathcal{O}(\text{LINES} \times \text{SAMPLES}) = \mathcal{O}(N)$$

Following the previous step is the **Searching step unified**, in charge of finding the K neighbors for each pixel. This section iterates for every pixel, where the loop performs the calculation of the distances within the window, then performs the search of the K-neighbors in the calculated distances, and finally stores the K-neighbors in the KNN matrix. All these steps are performed inside the loop that runs through each pixel, resulting in the following complexities: Calculate the distances is $\mathcal{O}(W)$, Search the KNN is $\mathcal{O}(K \times W)$ and Store the KNN is $\mathcal{O}(K)$. As these sections of the code are at the same nesting level, the complexity of this section would be as follows:

$$\mathcal{O}(N \times (K \times W + K)) = \mathcal{O}(N \times K \times W)$$

Finally, in the **Filtering** step, there are two loops at the first level, both traverse the N pixels. The first one also runs through the classes, with a total of $N \times C$ iterations. The other one has a higher complexity since, besides traversing the classes, it goes through the K-neighbors, resulting in $N \times C \times K$ iterations, finally at the same nesting level as the one that traverses the classes, it performs a loop of $C - 1$ iterations for the search of the maximum probability. The complexity of this part of the algorithm results in:

$$\mathcal{O}((N \times C) + (N \times (C \times K + (C - 1)) = \mathcal{O}(N \times C \times K)$$

Joining the three parts of the code, considering that $C < K < W < N$, would result in an algorithm of the following complexity:

$$\mathcal{O}(N + (N \times K \times W) + (N \times C \times K)) = \mathcal{O}(N \times K \times W)$$

Below is the pseudocode (see Algorithm 5), as explained in the previous sections.

Algorithm 5 K Nearest Neighbors - KNN, Base Implementation

Input: PCA, SVM
Output: *label_map*

```
# Initialize Feature Matrix
rc ← 0
knn_Idx ← 0
for row = 0 to LINES do                                   ▷ LINES = Height of Image
    for col = 0 to SAMPLES do                             ▷ SAMPLES = Width of Image
        V[rc] ← {PCA[rc], λrow, λcol, rc}                 ▷ λ = 1
        rc ← rc + 1
    end for
end for
# Searching step unified
SW ← W/2                                                  ▷ SW = Safe Border of window W
W_inf ← 0
W_sup ← SW
for i = 0 to N do                                         ▷ N = LINES × SAMPLES
    for j = W_inf to W_sup do
        D[j − W_inf] ← {d(V_i, V_j), V[j].rc}
    end for
    W ← Wsup − W_inf
    N(k) ← [0, 0, ...0]                                   ▷ N(k) = k-Neighbors
    N_aux(k) ← [0, 0, ...0]                               ▷ N_aux(k) = Auxiliar Vector of k-Neighbors

    min ← 0.0;
    last_min ← 0.0;

    for kk = 0 to K do
        zz ← 0
        min ← ∞
        for ii = 0 to W do
            if (D[ii].dist > last_min) && (D[ii].dist ≤ min) && (D[ii].dist ≠ 0) then
                if (min = D[ii].dist) then
                    zz ← zz + 1
                    N_aux[zz] ← D[ii].rc
                else
                    zz ← 0
                    min ← D[ii].distance
                    N_aux[zz] ← D[ii].rc
                end if
            end if
```

```
        end for
        last_min ← min;

        for x = 0 to zz do
            if (kk + x) ≥ K then
                break
            end if
            N[kk + x] ← N_aux[x]
        end for
        kk ← kk + zz;
    end for
    #Actualizar los límites de la ventana según la posición de i
    if i < SW then
        W_sup ← W_sup + 1
    else if (i ≥ SW) && (i < (N − SW)) then
        W_sup ← W_sup + 1
        W_inf ← W_inf + 1
    else
        W_inf ← W_inf + 1
    end if

    for kk = 0 to K do
        KNN[knn_Idx] ← N[kk] + 1
        knn_Idx ← knn_Idx + 1
    end for

end for
# Filtering
k_rc ← 0
for i = 0 to N do
    for c = 0 to C do
        SVM_aux[c × N + i] = SVM[i × C + c]
    end for
end for

for i = 0 to N do
    for c = 0 to C do
        for z = 0 to K do
            k ← KNN[k_rc]
            k_rc ← k_rc + 1
            if k < N then
                SVM_aux[c × N + i] ← SVM_aux[c × N + i] + P[c × N + kIdx]
            end if
```

```
        end for
        SVM_aux[c × N + i] ← SVM_aux[c × N + i] ÷ KNN
        k_rc ← (k_rc − KNN)
    end for

    max_prob ← 0
    for c = 1 to C do
        if SVM_aux[c × N + i] > SVM_aux[max_prob * N + i] then
            max_prob ← c
        end if
    end for

    label_map[i] ← max_prob + 1
    k_rc ← k_rc + K
end for

return label_map
```

5.2.2 Algorithm Resource Utilization

In the base implementation of the KNN algorithm, executed with the values of the PCA and SVM values obtained from PB1C1 image processing, there is a low use of FPGA resources. As shown in the table 5.1 the system hardly uses the device memory (RAM). This happens because most of the variables are stored in the host and are accessed through accessors.

	ALUTs	FFs	RAMs	MLABs	DSPs
Static Partition	466792 (25%)	928428 (25%)	3039 (26%)	0 (0%)	1291 (22%)
Kernel System	36297 (2%)	85937 (2%)	762.1 (7%)	484 (<1%)	14.5 (<1%)
Global Interconnect	8596	22830	284	0	0
KNN - Q	27699 (1%)	63036 (2%)	476.1 (4%)	484 (<1%)	14.5 (<1%)

Table 5.1: FPGA's area estimates of KNN.

For the feature matrix step, a buffer that stores the PCA result and a buffer that holds the feature matrix (V). For storing the distances (D) a new buffer is created, as well as another buffer to store the KNN of each pixel. For the processing of the SVM the following buffers are created: a buffer for the results obtained in the SVM step, another one to store the reordered probabilities (P) to ease the processing of the K-neighbors and finally a buffer

that stores the calculated probabilities ($\bar{S}$) on which the class with the maximum probability is obtained. Finally, another buffer is declared to store the classes of each query pixel ($\hat{y}_q$), which is the output of the algorithm.

Based on the PB1C1 image each buffer has the following dimensions: For the first step, the dimensions of the Feature Matrix (V) and the PCA are the number of pixels (N) in the image, which is $219,232$. The vector of distances of the query pixel has the size of the window (W), since for each pixel these are the only distances that will be computed, the base implementation declares an SW equals 6×496, so that W is $SW \times 2$. The KNN vector, as mentioned, has the number of pixels times the neighbors (K) found, given that the 40 nearest neighbors are searched would result in $40 \times 219,232$. The vectors of the filtering step have the following dimensions: the resulting vector of the SVM step is of size $C \times N$, which results in $4 \times 219,232$. The vector with the reordered probabilities and the vector of the probability calculation are of the same size. Finally, the output vector of the algorithm, the label assigned to each pixel, is the size of N.

Buffer	Dimensions	Type	Size of Type	Total Size
Feature Matrix (V)	$N = 219,232$	{Float, Short, Short, Int}	12 Bytes	2.6MB
PCA Output	$N = 219,232$	Double	8 Bytes	1.75MB
Feature Distances (D)	$SW \times 2 = 5,952$	{Float, Int}	8 Bytes	47.62KB
KNN	$K \times N = 8,769,280$	Int	4 Bytes	35.07MB
SVM Output	$C \times N = 876,928$	Float	4 Bytes	3.5MB
SVM Reordered (P)	$C \times N = 876,928$	Float	4 Bytes	3.5MB
Filtered Probabilities ($\bar{S}$)	$C \times N = 876,928$	Float	4 Bytes	3.5MB
Label Map ($\hat{Y}$)	$N = 219,232$	Char	1 Bytes	219.23KB

Table 5.2: DPC++ buffers of base KNN.

As shown in Table 5.2, there is excessive use of buffers and host-to-device transactions that impacts the KNN execution time, due to I/O transactions. With an algorithm restructuring that reduces memory size, the buffers could be directly in the device memory. In

this implementation (Algorithm 5) the device memory is used only by $\mathcal{N}$ and $\mathcal{N}_{aux}$ vectors, storing the K-neighbors found that will later be transferred to the KNN buffer. The FPGA implements these vectors as kernel memory, as shown in Figure 5.4. Both memories have a single bank, they lack replicates and private copies.

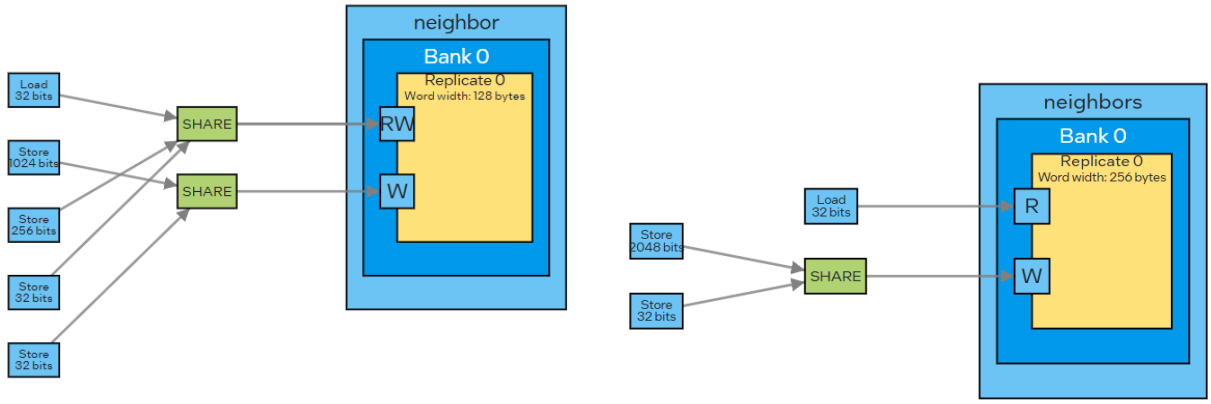


Figure 5.4: Kernel memory representation of $\mathcal{N}_{aux}$ (neighbor) and $\mathcal{N}$ (neighbors).

The Kernel memory system, according to Intel oneAPI [57], consists of the following parts:

- **Port**: serves as an entry point to access the memory system. It can be linked to one or more load-store units (LSUs), and each LSU has the potential to connect to several ports. Access to a memory port can be either stall-free or stallable, depending on whether it encounters contention during memory operations.
  - Stall-free: a memory access is considered stall-free when it can access a memory port without any contention.
  - Stallable: a memory access is stallable when it has access contention to a memory port.
- **Bank**: refers to a segment of the memory system that stores a share of the total data. The entire data for a kernel is distributed over several banks, with each bank containing a distinct subset of the data.

- **Replicate**: is a copy of the data stored within a memory bank. All replicates within a bank contain identical data; the main advantage is that accesses in each replicate can be done independently.

- **Private Copy**: is a duplicate of data from a replicate, created to enable concurrent execution of iterations in nested loops. The private copies are independent and do not necessarily contain the same data across different iterations.

In $\mathcal{N}$ and $\mathcal{N}_{aux}$ all memory accesses are stall-free, in Figure 5.4 these accesses are shown with the label SHARE. This is because load and store accesses are performed at different times during execution. The reason for this is that during the search of the neighbors they are first stored in $\mathcal{N}_{aux}$ and in a different loop, followed by the previous step, the final neighbors are stored in $\mathcal{N}$. Trying to make all memory accesses stall-free is vital to eliminate unnecessary latencies.

5.2.3 Performance Analysis

The schedule viewer display the scheduled cycle and latency of a group of instructions in the design [57]. The base design has **8791** cycles of latency (Figure 5.5). This is because Algorithm 5 has redundant loops, which with a code restructuring could be minimized, as many of them have the same number of iterations, resulting in loop merges, which also could be reduced by modifying the code [57]. Many current bottlenecks (blue lines in Figure 5.5) could be eliminated by reducing data dependencies, reducing loop nesting, and using a less demanding design in Load and Stores transfers between the device and the host.

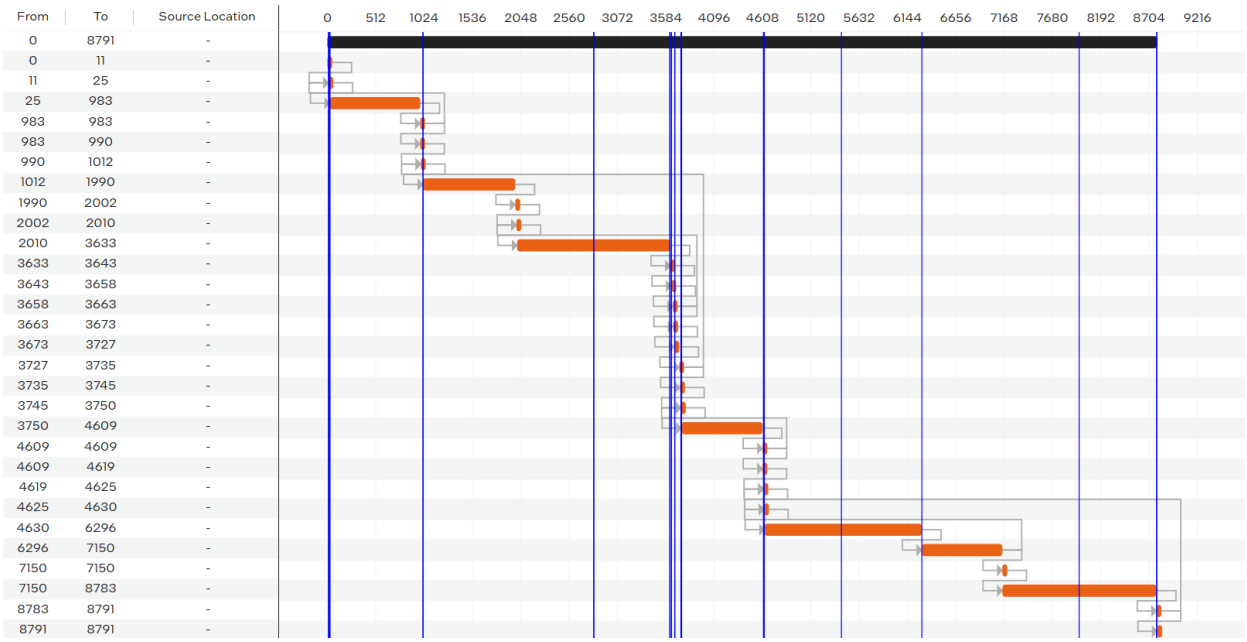


Figure 5.5: KNN base schedule.

The analysis of the loops in this version of the KNN shows some inefficiency caused by the above-mentioned bottlenecks (Figure 5.5). Most of the cycles are due to the large number of accessors performing Loads and Stores (LS) on the host memory. All LS lead to stallable memory transactions, further increasing latencies. Continuing this, there are unnecessary loops and data dependencies:

The *Feature Matrix (V)* is required for distance ordering step to start, when this process could be included in the **Searching step Unified** and thus not have to store the V and the reload it again. The buffer V would be eliminated since the feature matrix would not be needed, since it is only an extra step to calculate the distance, so an improvement would be to calculate the distance directly.

Moreover, the **Filtering** process could be directly in the loop of the search of the K-neighbors search at pixel i, performing only the class assignment of that pixel instead of, as it is in this version, all pixels. This modification would remove multiple unnecessary buffers, such as: the KNN matrix, SVM reordered (P) and filtered probabilities ($\bar{S}$) (Table 5.2).

Loop	Pipelined	BS II	BE fMax	Latency	SI	MII
Initialize Feature Matrix						
for $row = 0$ **to** LINES	Yes	2	480	14	0	1
for $col = 0$ **to** SAMPLES	Yes	1	480	958	0	1
Searching step unified						
for $i = 0$ **to** N	No	n/a	480	22	n/a	1
for $j = W_{inf}$ **to** W_{sup}	Yes	1	480	978	n/a	1
for $kk = 0$ **to** K	No	n/a	480	8	n/a	1
for $ii = 0$ **to** W	Yes	1	432	1623	0	1
for $x = 0$ **to** zz	Yes	1	480	15	3	1
for $kk = 0$ **to** K	Yes	1	480	54	0	1
Filtering						
for $i = 0$ **to** N	Yes	2	480	5	0	1
for $i = 0$ **to** N	Yes	2	480	859	0	1
for $i = 0$ **to** N	Yes	2	480	6	0	1
for $i = 0$ **to** N	Yes	2	480	6	0	1
for $z = 0$ **to** K	Yes	850	480	1666	0	1
for $c = 1$ **to** C	Yes	819	480	1633	0	819

Table 5.3: Loop analysis of KNN base implementation.

Where BS II: Block Scheduled Initiation Interval, BE fMax: Block Estimated max frequency, SI: Speculated Iterations, MII: Max Interleaving Iterations.

According to the report, there is a decrease in the estimated maximum frequency, from 480 to 432, located in the inner loop of the distance search (line 28 of Algorithm 5). The main reason for this decrease in the frequency is the dependence of the variables $last\_min$ and min, both of type *double*.

Another noteworthy information is the Initiation Interval (II) of different loops, the number of clock cycles between loop iterations [57]. For maximum performance, each iteration should start at an II of 1 [57]. In the base implementation there are two remarkable cases of II: the loops of lines 76 and 88 of the Algorithm 5, which have an II of 850 and 819 respectively.

The first implementation of the KNN algorithm takes **279274 milliseconds** (ms) to complete. This implementation has a frequency of 228MHz.

5.3 Final Implementation

Based on the KNN enhancements there are design alternatives for different equipment and different system requirements. To achieve the design of the final version, a code refactoring was performed.

5.3.1 Improvements

To begin with, in the initial version three completely separate processes add complexity and unnecessary latencies, therefore, instead of performing each of the processes for all pixels, the three processes are performed for each pixel of the image.

The algorithm has considerably improved its speed by taking into account different main factors:

- **Complete algorithm per pixel instead of full step path in the algorithm**. Eliminating unnecessary loops using techniques such as loop fusion or reducing loop nesting has increased the speed of the algorithm, since a higher pipeline rate per loop can be achieved, as well as a reduction of data dependencies [57]. The main feature in its favor is the latter, since eliminating bottlenecks caused by data dependencies greatly increases parallelism and greatly reduces IIs, as well as enables the pipelining of loops.

- **Separate sections in use of fixed window and use of variable windows**. Modifying and reducing the use of unnecessary accessors greatly reduces I/O transactions, and consequently improves timing, especially accessors that with the above modification fit perfectly in memory. For example, the **KNN** matrix vector is now unnecessary because only the $\mathcal{N}$ vector would be taken, going from a vector of size $K \times N$ to size K. Therefore, this implementation consists of three buffers: **PCA, SVM and Label Map**, the only ones that have to perform I/O transactions.

- **Separate sections on fixed window usage and variable window usage**. One

of the features of FPGAs is their ability to parallelize, for example using loop unrolls. Unfortunately with a variable code section such as window limits and consequently loop iterations, the loop unrolling adds logic that worsens performance. Therefore, an alternative is to use three kernels, where each one will perform a section of the KNN, top section (Q TOP, Algorithm 6), fixed section (Q CONST W, Algorithm 8) and bottom section (Q BOT, Algorithm 7), behavior explained Section 5.2.

As it is shown in the algorithm, the size over which the calculation of the distances is performed, the window W, is kept constant during all its iterations and can be unrolled, as desired and according to the available space.

- **Pre-loading of external data into a local buffer**. Due to the high dependencies of data in external memory, by the input data of the PCA and the SVM, a preload has been performed before executing the KNN algorithm. This technique may cause problems with the RAM resources when loading the values obtained from dermatological images (PD1C1), due to the dimensions of these images. This measure allows creating a private memory, where unrolls and accesses to the memory values can be performed without entering stalls during the transactions [57]. This measure is especially useful since it greatly reduces the latencies caused by the data load as can be seen in the **Q CONST W** Figure 5.7, compared to the **Q BOT**.

- **Lowering the accuracy, but with great improvement in times**. One of the methods has been to compare the % of accuracy compared to the time achieved, so we have adjusted values in the section of the code where the greatest complexity is: the search of the K neighbors: $\mathcal{O}(K \times W)$ for each pixel and in full would be a complexity of the order $\mathcal{O}(N \times K \times W)$. Considering, for the PB1C1 image, that the standard parameters: W is equal to 5952, K is 40 and there are 219,232 pixels, this results in 52.194 billion iterations in the worst case. Therefore, alternative window sizes have been used, under the logic that in a large window the farthest values lose importance due to the weight of the distances between row and column. For example: going from a window size of 5952 to 2976 reduces by about half the time needed to complete the

algorithm and only about 1.36% of the data are different.

All the implemented improvements are detailed in the pseudocode of Algorithms 6 to 8.

Algorithm 6 K Nearest Neighbors - KNN Top Window, Final Impl.

Input: PCA, SVM
Output: $label\_map$

$SW \leftarrow \frac{W}{2}$ ▷ SW = Safe Border of window W
$W_{inf} \leftarrow 0$
$W_{sup} \leftarrow SW$
$\text{SVM}^{'}(W \times C)$
for $i = 0$ **to** SW **do**
  $r_i \leftarrow i \div \text{SAMPLES}$
  $c_i \leftarrow i \mod \text{SAMPLES}$
  $PCA_i = PCA[i]$
  $\mathcal{N}(k) \leftarrow [0, 0, ...0]$ ▷ $\mathcal{N}(k)$ = k-Neighbors
  $\mathcal{N}_{aux}(k) \leftarrow [0, 0, ...0]$ ▷ $\mathcal{N}_{aux}(k)$ = Auxiliar Vector of k-Neighbors
  $min \leftarrow 0.0f$;
  $last\_min \leftarrow 0.0f$;
  $D(W)$ ▷ $D = \{dist, rc\}$
  $jj \leftarrow W_{inf}$
  $load\_Offset \leftarrow 0$
  for $j = 0$ **to** W_{sup} **do**
    if $load\_Offset < j$ **then**
      $\text{SVM}^{'}[j \times 4] \leftarrow SVM[j \times 4]$
      $\text{SVM}^{'}[j \times 4 + 1] \leftarrow SVM[j \times 4 + 1]$
      $\text{SVM}^{'}[j \times 4 + 2] \leftarrow SVM[j \times 4 + 2]$
      $\text{SVM}^{'}[j \times 4 + 3] \leftarrow SVM[j \times 4 + 3]$
      $load\_Offset \leftarrow load\_Offset + 1$
    end if
    $r_j \leftarrow j \div \text{SAMPLES}$
    $c_j \leftarrow j \mod \text{SAMPLES}$
    $D[j] \leftarrow \{(PCA_i - PCA[j])^2 + (r_i - r_j)^2 + (c_i - c_j)^2, j\}$
  end for
  for $kk = 0$ **to** K **do**
    $zz \leftarrow 0$
    $min \leftarrow \infty$
    for $ii = 0$ **to** W **do**
      if $(D[ii].dist > last\_min)$ && $(D[ii].dist \leq min)$ && $(D[ii].dist \neq 0)$ **then**
        $\mathcal{N}_{aux}[zz] \leftarrow D[ii].rc$
        if $(min = D[ii].dist)$ **then**
          $zz \leftarrow zz + 1$

```
            else
                zz ← 0
                min ← D[ii].distance
            end if
        end if
        ii ← ii + 1
    end for
    last_min ← min;

    for x = 0 to zz do
        if (kk + x) ≥ K then
            break
        end if
        N[kk + x] ← N_aux[x]
    end for
    kk ← kk + zz;
end for

W_sup ← W_sup + 1

P(C) ← 0.0f
P'(c = 0 ... C) ← 0.0f
for z = 0 to K do
    k_z1 ← N[z] + 1
    k_z2 ← N[z + 1] + 1
    P'_{k_z1}(c = 0 ... C) ← 0.0f
    P'_{k_z2}(c = 0 ... C) ← 0.0f

    if k_z1 < N then
        P'_{k_z1}(c = 0) ← SVM'[k_z1 × C + 0]
        P'_{k_z1}(c = 1) ← SVM'[k_z1 × C + 1]
        P'_{k_z1}(c = ...) ← SVM'[k_z1 × C + ...]
        P'_{k_z1}(c = C) ← SVM'[k_z1 × C + C]
    end if
    if k_z2 < N then
        P'_{k_z2}(c = 0) ← SVM'[k_z2 × C + 0]
        P'_{k_z2}(c = 1) ← SVM'[k_z2 × C + 1]
        P'_{k_z2}(c = ...) ← SVM'[k_z2 × C + ...]
        P'_{k_z2}(c = 3) ← SVM'[k_z2 × C + 3]
    end if

    P'(c = 0) ← P_{k_z1}(c = 0) + P_{k_z2}(c = 0)
```

```
        P'(c = 1) ← P_{k_z1}(c = 1) + P_{k_z2}(c = 1)
        P'(c = ...) ← P_{k_z1}(c = ...) + P_{k_z2}(c = ...)
        P'(c = C) ← P_{k_z1}(c = C) + P_{k_z2}(c = C)
        z ← z + 2
    end for
    P(C) ← P'(c = 0, ..., C)
    max_prob ← 0
    for c = 1 to C do
        if P[c] > P[max_prob] then
            max_prob ← c
        end if
        c ← c + 1
    end for
    label_map[i] ← max_prob + 1
end for
return label_map
```

Algorithm 7 K Nearest Neighbors - KNN Bot Window, Final Impl.

Input: PCA, SVM
Output: $label\_map$

```
SW ← W/2                                   ▷ SW = Safe Border of window W
W_inf ← SW
i_BOT ← N − 1
for i = 0 to SW do
    r_i ← (N − 1 − i) ÷ SAMPLES
    c_i ← (N − 1 − i)  mod SAMPLES
    PCA_i = PCA[i]
    N(k) ← [0, 0, ...0]                    ▷ N(k) = k-Neighbors
    N_aux(k) ← [0, 0, ...0]                ▷ N_aux(k) = Auxiliar Vector of k-Neighbors
    min ← 0.0f;
    last_min ← 0.0f;
    D(W)                                   ▷ D = {dist, rc}
    jj ← W_inf
    load_Offset ← 0
    for j = 0 to W_inf do
        r_j ← (N − 1 − j) ÷ SAMPLES
        c_j ← (N − 1 − j)  mod SAMPLES
        D[j] ← {(PCA_i − PCA[j])^2 + (r_i − r_j)^2 + (c_i − c_j)^2, j}
    end for

    for kk = 0 to K do
        zz ← 0
```

```
        min ← ∞
        for ii = 0 to W do
            if (D[ii].dist > last_min) && (D[ii].dist ≤ min) && (D[ii].dist ≠ 0) then
                N_aux[zz] ← D[ii].rc
                if (min = D[ii].dist) then
                    zz ← zz + 1
                else
                    zz ← 0
                    min ← D[ii].distance
                end if
            end if
            ii ← ii + 1
        end for
        last_min ← min;

        for x = 0 to zz do
            if (kk + x) ≥ K then
                break
            end if
            N[kk + x] ← N_aux[x]
        end for
        kk ← kk + zz;
    end for

    W_inf ← W_inf + 1

    P(C) ← 0.0f
    P'(c = 0 ... C) ← 0.0f
    for z = 0 to K do
        k_z1 ← N[z] + 1
        k_z2 ← N[z + 1] + 1
        P'_{k_z1}(c = 0 ... C) ← 0.0f
        P'_{k_z2}(c = 0 ... C) ← 0.0f

        if k_z1 < N then
            P'_{k_z1}(c = 0) ← SVM[k_z1 × C + 0]
            P'_{k_z1}(c = 1) ← SVM[k_z1 × C + 1]
            P'_{k_z1}(c = ...) ← SVM[k_z1 × C + ...]
            P'_{k_z1}(c = C) ← SVM[k_z1 × C + C]
        end if
        if k_z2 < N then
            P'_{k_z2}(c = 0) ← SVM[k_z2 × C + 0]
```

```
        P'_{k_z2}(c = 1) ← SVM[k_z2 × C + 1]
        P'_{k_z2}(c = ...) ← SVM[k_z2 × C + ...]
        P'_{k_z2}(c = 3) ← SVM[k_z2 × C + 3]
    end if

    P'(c = 0) ← P_{k_z1}(c = 0) + P_{k_z2}(c = 0)
    P'(c = 1) ← P_{k_z1}(c = 1) + P_{k_z2}(c = 1)
    P'(c = ...) ← P_{k_z1}(c = ...) + P_{k_z2}(c = ...)
    P'(c = C) ← P_{k_z1}(c = C) + P_{k_z2}(c = C)
    z ← z + 2
  end for
  P(C) ← P'(c = 0, ..., C)
  max_prob ← 0
  for c = 1 to C do
    if P[c] > P[max_prob] then
      max_prob ← c
    end if
    c ← c + 1
  end for
  label_map[N − 1 − i] ← max_prob + 1
  i_BOT ← i_BOT − 1
end for
return label_map
```

Algorithm 8 K Nearest Neighbors - KNN Fixed Window, Final Impl.

Input: PCA, SVM
Output: *label_map*

```
PCA'(N)
SVM'(N × C)
# Pre Load of PCA and SVM
for i = 0 to N do
  PCA'[i] = PCA[i]
  SVM'[i × C] = PCA[i × C]
  SVM'[i × C + 1] = SVM[i × C + 1]
  SVM'[i × C + ...] = SVM[i × C + ...]
  SVM'[i × C + C] = SVM[i × C + C]
  i ← i + 1
end for
SW ← W/2                                  ▷ SW = Safe Border of window W
W_inf ← 0
W_sup ← SW
for i = SW to N − SW do                   ▷ N = LINES × SAMPLES
```

```
    r_i ← i ÷ SAMPLES
    c_i ← i  mod SAMPLES
    PCA_i = PCA[i]
    N(k) ← [0, 0, ...0]                          ▷ N(k) = k-Neighbors
    N_aux(k) ← [0, 0, ...0]                      ▷ N_aux(k) = Auxiliar Vector of k-Neighbors
    min ← 0.0f;
    last_min ← 0.0f;
    D(W)                                         ▷ D = {dist, rc}
    jj ← W_inf
    for ii = 0 to W do
        jj' ← jj + 1
        r_j ← jj ÷ SAMPLES
        c_j ← jj  mod SAMPLES
        D[ii] ← {(PCA_i − PCA[jj])^2 + (r_i − r_j)^2 + (c_i − c_j)^2, jj}
        jj ← jj'
        ii ← ii + 1
    end for

    for kk = 0 to K do
        zz ← 0
        min ← ∞
        for ii = 0 to W do
            if (D[ii].dist > last_min) && (D[ii].dist ≤ min) && (D[ii].dist ≠ 0) then
                N_aux[zz] ← D[ii].rc
                if (min = D[ii].dist) then
                    zz ← zz + 1
                else
                    zz ← 0
                    min ← D[ii].distance
                end if
            end if
            ii ← ii + 1
        end for
        last_min ← min;

        for x = 0 to zz do
            if (kk + x) ≥ K then
                break
            end if
            N[kk + x] ← N_aux[x]
        end for
        kk ← kk + zz;
```

```
    end for

    W_sup ← W_sup + 1
    W_inf ← W_inf + 1
    P(C) ← 0.0f
    P'(c = 0 ... C) ← 0.0f
    for z = 0 to K do
        k_z1 ← N[z] + 1
        k_z2 ← N[z + 1] + 1
        P'_{k_z1}(c = 0 ... C) ← 0.0f
        P'_{k_z2}(c = 0 ... C) ← 0.0f

        if k_z1 < N then
            P'_{k_z1}(c = 0) ← SVM'[k_z1 × C + 0]
            P'_{k_z1}(c = 1) ← SVM'[k_z1 × C + 1]
            P'_{k_z1}(c = ...) ← SVM'[k_z1 × C + ...]
            P'_{k_z1}(c = C) ← SVM'[k_z1 × C + C]
        end if
        if k_z2 < N then
            P'_{k_z2}(c = 0) ← SVM'[k_z2 × C + 0]
            P'_{k_z2}(c = 1) ← SVM'[k_z2 × C + 1]
            P'_{k_z2}(c = ...) ← SVM'[k_z2 × C + ...]
            P'_{k_z2}(c = 3) ← SVM'[k_z2 × C + 3]
        end if

        P'(c = 0) ← P_{k_z1}(c = 0) + P_{k_z2}(c = 0)
        P'(c = 1) ← P_{k_z1}(c = 1) + P_{k_z2}(c = 1)
        P'(c = ...) ← P_{k_z1}(c = ...) + P_{k_z2}(c = ...)
        P'(c = C) ← P_{k_z1}(c = C) + P_{k_z2}(c = C)
        z ← z + 2
    end for
    P(C) ← P'(c = 0, ..., C)
    max_prob ← 0
    for c = 1 to C do
        if P[c] > P[max_prob] then
            max_prob ← c
        end if
        c ← c + 1
    end for
    label_map[i] ← max_prob + 1
    i ← i + 1
end for
```

100: **return** *label_map*

5.3.2 Results

Algorithm Resource Utilization

The first implementation of the KNN, has a low use of FPGA resources. The Algorithm 8, together with Algorithm 6 and Algorithm 7, aims to perform a higher use of FPGA resources, as seen in Table 5.4.

	ALUTs	FFs	RAMs	MLABs	DSPs
Static Partition	466792 (25%)	928428 (25%)	3039 (26%)	0 (0%)	1291 (22%)
Kernel System	43810 (2%)	87720 (2%)	7030 (60%)	952 (1%)	116 (2%)
Global Interconnect	6185	15947	284	0	0
KNN - Q CONST W	20695 (1%)	34497 (<1%)	6527 (56%)	261 (<1%)	88 (2%)
KNN - Q TOP	9576 (<1%)	21500 (<1%)	103 (<1%)	370 (<1%)	14 (<1%)
KNN - Q BOT	7352 (<1%)	15705 (<1%)	114 (<1%)	321 (<1%)	14 (<1%)

Table 5.4: FPGA's area estimates of Final KNN.

The Table 5.4 shows a large increase, compared to the base implementation (Table 5.1), mainly in use of RAMs and MLABs, with an increase of 13.8x in the first case and 1.92x in the second. On the other hand, the use of DSP block has increased by 8.6x. All increases exclude the interconnect system, however, interconnects are reduced in the final version, especially in ALUTs and FFs.

From the memory perspective, the final version uses private memories with the PCA and SVM, to favor unrolls and accesses during algorithm processing (Figure 5.6). The PCA has an unroll factor of 12, with 6 replicates using 3072 RAMs, on the other hand it has an unroll of 2 and has 3328 RAMs. The main aspect is that all accesses have no stall, improving considerably the accesses for Read and/or Write. Finally, in terms of memory, the vector that stores and loads the distances of the pixels inside the window with regard to the query pixel, D. The memory accesses of the vector have stall, in the Figure 5.6 the ARB that describes this stall.

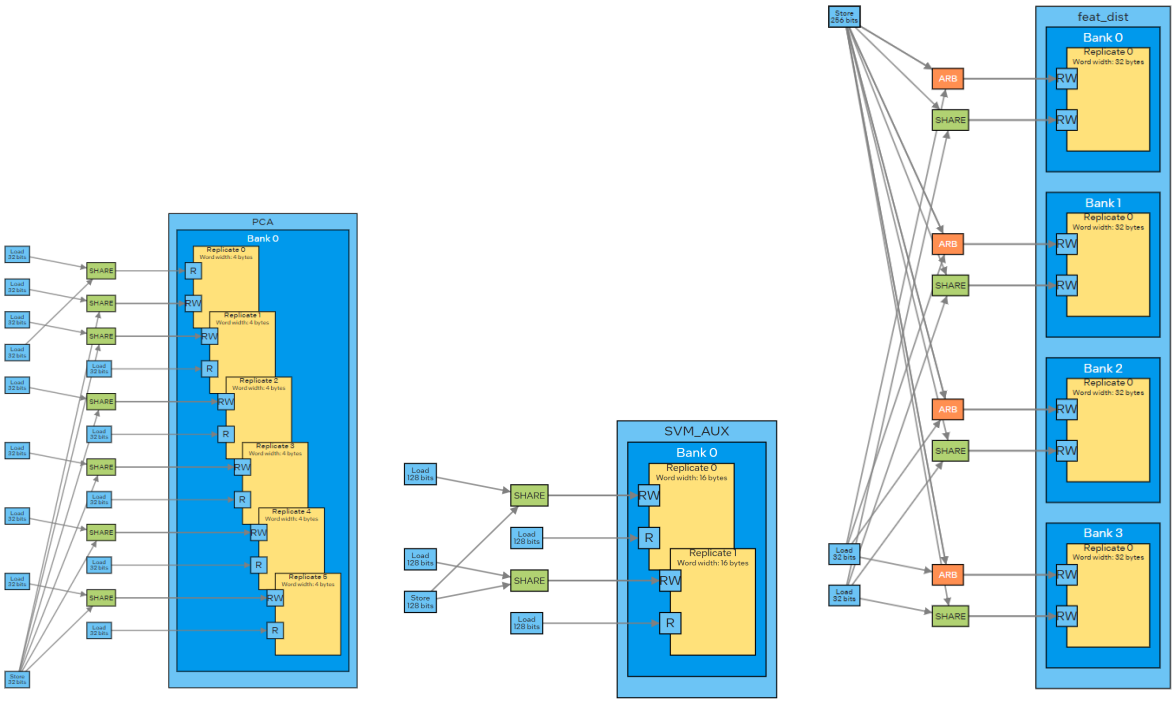


Figure 5.6: Unroll x12 of PCA (left image), Unroll x2 of SVM (center image), feat dist (D) (right image).

The difference between the use of the accessors mentioned in the improvements results in a considerable decrease in their use, going from 8 accessors (Table 5.2) to only 3 (Table 5.5.

Buffer	Dimensions	Type	Size of Type	Total Size
PCA Output	$N = 219,232$	Float	4 Bytes	1.75MB
SVM Output	$C \times N = 876,928$	Float	4 Bytes	3.5MB
Label Map ($\hat{Y}$)	$N = 219,232$	Char	1 Bytes	219.23KB

Table 5.5: DPC++ Buffers of Final KNN.

Performance Analysis

Finally, in terms of performance. The analysis of the schedule cycles shows the following data: for the Fixed Window (Algorithm 8) its latency is 977 (Figure 5.7), the Top Window (Algorithm 6) is 1014 and the Bot Window (Algorithm 7) is 1819. Both, the first and the second, have been improved compared to the base implementation, from 8791 cycles to

around 1000 cycles due to the preloads in memory and other improvements. In the case of the third, the Bot Window, is not the case, but it remains well below the number of the first version, although it is almost x2 of its Top and Fixed counterparts.

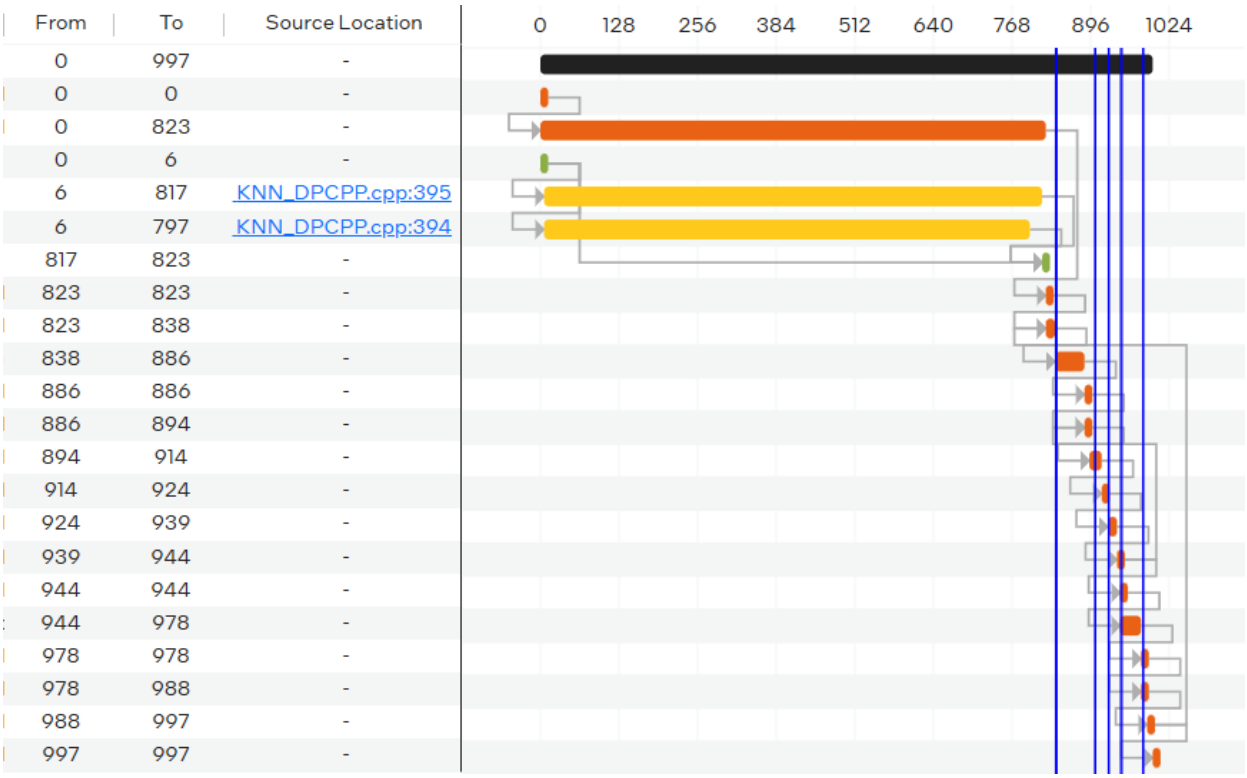


Figure 5.7: KNN schedule fixed window.

On the other hand, talking about the cycle analysis, most of the loops shown in Table 5.3 were modified by the restructuring of the algorithm, and all of them had an II of 1, except for those with n/a, which remained the same. In addition, no frequency was decreased by one loop, decreasing the critical path delay and improving the frequency of the algorithm.

Taking into account all the speed improvements, an isolated algorithm time of 109703 ms has been achieved, giving a speedup of 2.545 [65]. On the other hand, the frequency achieved in this version is 249MHz, a 10% increase over the base implementation.

$$S_{latency} = \frac{T_{old}}{T_{new}} = \frac{279274}{109703} = 2.545$$

Chapter 6: Integration and Optimization of Combined Pipeline

6.1 Integrating the three algorithms

After developing and optimizing the individual components of our machine learning pipeline, Principal Component Analysis (PCA), Support Vector Machines (SVM), and K-Nearest Neighbors (KNN)—we have now undertaken the crucial task of integrating these algorithms into a cohesive system. This step marks a significant step in our project, as it brings together the strengths of each algorithm within a unified framework designed to maximize efficiency and performance.

With the individual components fully developed, the next phase involved carefully integrating these algorithms into a single workflow. This integration process was not merely a matter of combining code; it required a thoughtful approach to ensure that the algorithms work together efficiently. The goal was to create a pipeline where data flows smoothly from one stage to the next, with each algorithm enhancing the performance of the subsequent one.

Moreover, the parallelized components of the algorithms, which were initially developed separately to exploit data parallelism, have been meticulously gathered into this integrated system. This allows the entire pipeline to benefit from parallel computation, significantly reducing execution times and enabling the processing of large datasets that would be impractical with a sequential approach.

6.2 Algorithm and Complexity

The mathematical complexity will play a key role in this, as PCA and SVM are computed parallelly. The mathematical complexity of the full program will have two parts. As PCA and SVM are parallelly executed, the complexity of the first part will be the maximum between the two parts (see next calculation). The second part of the execution will be equal to the KNN complexity. It will be executed individually and after the first two algorithms because it needs both inputs.

$$max(\mathcal{O}(SVM), \mathcal{O}(PCA)) + \mathcal{O}(KNN)$$

$$max(\mathcal{O}(N \times C^2 \times B), PCA) + \mathcal{O}(N \times K \times W)$$

Figure 6.1 shows how KNN uses PCA and SVM output. This makes the computational cost less, taking advantage of parallel computing.

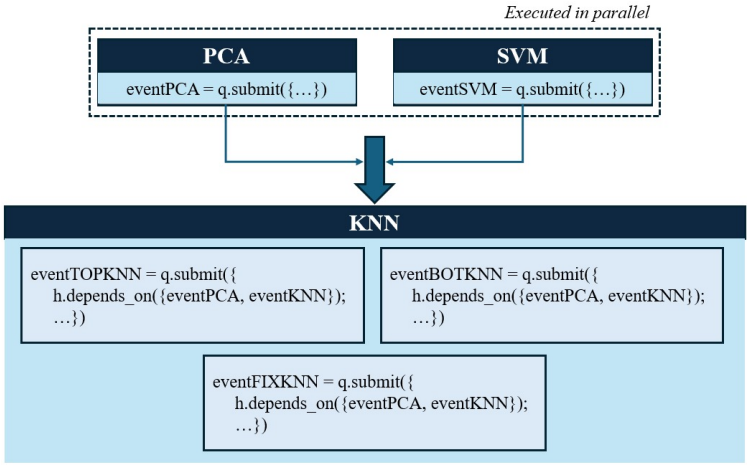


Figure 6.1: PCA, SVM and KNN computation diagram.

6.3 Algorithm Resource Utilization

The resource utilization will now include the three algorithms' hardware requirements. Table 6.1 shows each algorithm's resource utilization.

	ALUTs	FFs	RAMs	MLABs	DSPs
Static Partition	466792 (25%)	928428 (25%)	3039 (26%)	0 (0%)	1291 (22%)
Kernel System	417270 (22%)	483965 (13%)	8295 (71%)	2574 (3%)	94.5 (2%)
Global Interconnect	4515	10406	121	0	0
PCA Kernel	181409 (10%)	127870 (3%)	251 (2%)	1131 (1%)	52 (<1%)
SVM Kernel	70478 (4%)	106836 (3%)	569 (5%)	513 (<1%)	30.5 (<1%)
KNN Kernel	160866 (9%)	238782 (6%)	7352 (63%)	930 (<1%)	12 (<1%)
KNN Kernel Fixed	109195 (6%)	77151 (2%)	6540 (56%)	296 (<1%)	4 (<1%)
KNN Kernel BOT	30285 (2%)	136064 (4%)	671 (6%)	309 (<1%)	4 (<1%)
KNN Kernel TOP	21386 (1%)	25567 (<1%)	141 (1%)	325 (<1%)	4 (<1%)

Table 6.1: FPGA's area estimates of each algorithm in the complete processing pipeline.

The table highlights the huge use of resources represented by KNN compared with PCA or SVM. However, the FPGA has enough resources to process the image and still has space left to include higher-dimension data. The resources available in the FPGA are enough for parallel computing have been enough for computing the image.

These results show that, compared with the resource utilization shown in Table 3.3, Table 4.6 and Table 5.4, the resources used in this case are proportionally higher than in the individual parts. This happens because the parallel computing of the full complete algorithm needs to generate more than the individual algorithms to compute it parallelly and connect the different parts.

Global interconnections have also been reduced. Table 6.2 shows the use of the global interconnections when algorithms were executed individually and when everything is together and parallelized. The use of the interconnections has significantly reduced.

	ALUTs	FFs	RAMs	MLABs	DSPs
Final implementation	4515	10406	121	0	0
Individual PCA+SVM+KNN	10227	21724	526	0	0
Individual PCA	2012	2253	121	0	0
Individual SVM	2030	3524	121	0	0
Individual KNN	6185	15947	284	0	0

Table 6.2: FPGA's interconnections.

The use of parallel computing also has disadvantages. The compiling optimization for the

paralleled program can make some before-optimized components inefficient now. KNN took advantage of DSPs (specialized hardware blocks) before. Now, the use of DSPs is reduced because the specific arithmetic blocks computed need to be changed due to parallelism, so the final execution time is incremented, and FPGA efficiency is decreased.

6.4 Performance

Table 6.3 shows the parameters, time, accuracy, and obtained speedup of the individual algorithms. Comparing the results obtained with the one of the original paper (see Table 6.4) can be seen that the performance is much lower in the FPGA. Due to the type of device, which is focused on efficiency [45], the obtained result is lower than in the ones with GPUs (including high and lower power efficiency GPUs) and other types of technologies. The most approximate comparison between our FPGA and the given results would be taking the Manycore 1:1 [2].

Algorithm	Parameters	Time (ms)	Accuracy[1] (%)	Speedup
PCA	$B = 128; N = 219,232; PC = 1$	2,530	100%	37,03
SVM	$B = 128; N = 219,232;$ $C = 4; BC = 6$	5,662	100%	12.12
KNN	$N = 219,232; K = 40;$ $W = 2965; C = 4$	109,703	98.27%	2.545

Table 6.3: Time and accuracy results of each algorithm isolated.

The optimization by reducing the precision in the KNN algorithm achieves significant reductions in time, with only a small increase in error. Considering that diagnostic speed is also a priority, it would be interesting to determine the optimal trade-off.

The Table 6.4 gathers the results of [2]. Describes the execution time of each algorithm ran on the devices used in the investigation mentioned, and also adds the ones obtained through the FPGA tests, using the same algorithms that the processing chain described at Figure 6.1. All executions are separately computed versions, there is no parallelization between algorithms. All devices have executed the PB1C1 brain image (Table 2.1). The

times when processing on FPGA are higher than in high power GPUs, low power GPUs, or low many core platforms. This is mainly, because the GPUs: Tesla K40, GTX 1060 and RTX 2080, are high-end hardware for parallel processing.

Hardware	PCA (ms)	SVM (ms)	KNN (ms)
High Power GPUs			
Tesla K40	27	8	1,451
GTX 1060	21	4	1,222
RTX 2080	23	1	336
Low Power GPUs			
MaxQ (Low-power)	346	88	2,005
MaxP (Low-power)	220	65	1,655
MaxCLK (Low-power)	122	21	1,390
Low Manycore platforms			
Manycore 1:1	35,612	26,677	129,840
Manycore 1:16	1,683	113	864
FPGA			
Stratix 10 SX 2800	2,530	10,240	109,703

Table 6.4: Execution time (ms) for each algorithm in PB1C1 across different hardware platforms. Other implementations times, except for Stratix 10 SX 2800, are provided by [2]. Time results are obtained with the parameters shown in Table 6.3.

The final execution time of the full algorithm pipeline (Figure 6.1) has resulted in 4,846 ms for the PCA execution and 10,240 ms for the SVM. These time results from the executions in the current pipeline are parallelized. Therefore, it only takes into account for the maximum of both, the SVM. To proceed with the total time calculation, the KNN time is added to the SVM, resulting in a total of 183,305 ms. Nonetheless, notable differences can be seen compared to the isolated version (Tables 6.3 and 6.4). In the work of Lazcano et al., [2], these differences between the time when implementing the complete pipeline do not happen. The main reason is the high configurability that other systems allow rather than the Intel oneAPI Toolkit does. For instance, from the final version of each algorithm DSPs are reduced when there is perfect capacity for them, among others.

Chapter 7: Conclusions and Future Work

7.1 Conclusions

In this work, the tools provided by the Intel oneAPI Toolkit were used to implement the PCA, SVM, and KNN algorithms on a Stratix 10 SX 2800 FPGA, with the aim of optimizing their performance, specifically in terms of time, for the analysis of hyperspectral images in the study of brain tumors and skin cancer. A high-level language such as C++ with SYCL was employed, utilizing the DPC++ compiler, which presents a more accessible alternative to traditional FPGA programming using HDL languages.

At the beginning of this project, several objectives were defined, with the primary goal being the acceleration of the aforementioned algorithms. Based on the work completed, the following conclusions can be drawn:

- **Principal Component Analysis (PCA)**: To optimize the PCA algorithm, the initial implementation had several inefficiencies. On one hand, all eigenvectors and eigenvalues were computed using the Jacobi method, despite only needing the dominant eigenvector. On the other hand, the image was centered at the origin, which led to excessive memory consumption and increased host accesses, lengthening execution times. For the improvement, the power method was implemented to compute the eigenvector, and the matrix centering was optimized to reduce I/O accesses. Along with FPGA adjustments, the optimization goal was successfully achieved, resulting in a speedup of 37.07

- **Support Vector Machine (SVM)**: The base version of the SVM had many issues with accessors. The access time to the host was taking most of the time within the full execution. Additionally, there were redundant loops and variables that could be eliminated and optimized. The main goal was to create local variables to replace

all accessors and restructure the code to eliminate as many redundant variables and loops as possible. To achieve this, host access was optimized, and the main loop was restructured, removing redundant variables and loops, and integrating the sorting algorithm into the main loop. This brought a speedup of 12.12 times.

- **K-Nearest Neighbors (KNN)**: In the optimization of KNN, the base implementation had three main sections, and their separation did not take advantage of the parallelization potential of FPGAs. In response to this observation, the main improvement was to refactor the code and merge compatible loops or sections. Additionally, an important aspect is the search algorithm. Although its complexity did not change, unrolling was used through specific decoupling of the algorithm. Finally understanding the on-chip memory effectively has allowed the available FPGA memory to speed up the algorithm, reducing I/O delays between the device and the host. Another consideration to improve the performance was to readjust the base parameters, such as window size (W), and observe that reducing it significantly improves the times with minimal impact on accuracy. With all these improvements, a 2.545 speedup has been achieved, a significant improvement over the initial one.

- With the proper optimizations, especially those dedicated to hardware implementation, FPGAs achieve acceptable results. These optimizations range from leveraging their data-level parallelization capabilities.

  Analyzing the suitability of FPGAs for this type of application, it is observed that a development focused on optimization and data-level parallelism yields much more efficient results compared to standard programming. One of the issues encountered during the development of these optimizations has been the limited configurability compared to Hardware Description Languages (HDL) like VHDL. Many of the implementations could have been more efficient with specific optimization and design, but unfortunately, the Intel oneAPI Toolkit handles many processes and optimizations differently. The specificity required when developing for these devices is a key characteristic of their nature, and with high-level language libraries, that control is lost, and

consequently, performance is compromised.

- Taking into account the other resources offered by Intel for FPGA application development, Intel DevCloud emerges as an innovative element. It has been observed that it offers many alternatives for developers to use FPGAs with almost no initial investment. However, it is also concluded that due to its lack of resources, documentation, and frequent maintenance downtimes, it lacks reliability, making it a somewhat unstable alternative.

  Despite this, other resources, such as the reports generated by Intel's compiler, provide a very detailed view of FPGA implementations, as well as each cycle and instruction carried out. It offers relevant information to the developer that otherwise would not be accessible. Considering all of Intel's tools together, it is concluded that while they have advantages, they also have drawbacks that should be taken into account when developing on this platform.

7.2 Future Work

In the pursuit of continuing improvements and advances, there is a wide range of future possible work to improve the actual result of the thesis.

First of all, it would be convenient to continue testing code improvements and continue to try even reducing more the execution time. There are many tools still unused in the Intel oneAPI toolkit, they can be tested with the code in a time-optimizing oriented way.

It would also be important to test the current or future code with a wide range of different images. There are infinite different cases in medicine where algorithms can fail or underperform, so it is crucial to test as many cases as possible to improve trustability. It is important to highlight that the real accuracy of the image needs to be analyzed medically, in addition to just the numeric result.

Furthermore, this work has been done with the Stratix 10 SX 2800, but there are many other FPGA and processing units available in the Intel DevCloud and in the market. Testing

different FPGAs or other processing units can showcase improvement evidence.

Finally, it would also be useful to adapt the current work to new market trends. The new Nvidia Blackwell GPU, Nvidia Volta and other processing units' performance has significantly improved computational capabilities at AI and data tasks throughout the last years and seems to continue growing in the early and far future. These computational capacity improvements seem to boost previously ever-seen performance, so it would be probably a good option to experiment with such hardware.

Bibliography

[1] Intel Corporation. Intel oneAPI Base Toolkit. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html, 2021.

[2] Raquel Lazcano, Daniel Madroñal, Giordana Florimbi, Jaime Sancho, Sergio Sanchez, Raquel Leon, Himar Fabelo, Samuel Ortega, Emanuele Torti, Ruben Salvador, Margarita Marrero-Martin, Francesco Leporati, Eduardo Juarez, Gustavo M. Callico, and Cesar Sanz. Parallel Implementations Assessment of a Spatial-Spectral Classifier for Hyperspectral Clinical Applications. *IEEE Access*, 7:152316–152333, 2019.

[3] Chein-I Chang. *Hyperspectral imaging: techniques for spectral detection and classification*, volume 1. Springer Science & Business Media, 2003.

[4] Karel J. Zuzak, Michael D. Schaeberle, E. Neil Lewis, and Ira W. Levin. Visible Reflectance Hyperspectral Imaging: Characterization of a Noninvasive, in Vivo System for Determining Tissue Perfusion. *Analytical Chemistry*, 74(9):2021–2028, 2002. PMID: 12033302.

[5] M. Kamruzzaman and D.-W. Sun. *Chapter 5 - Introduction to Hyperspectral Imaging Technology*. Elsevier, San Diego, second edition edition, 2016.

[6] C Gonzáles. Procesamiento a bordo de imágenes hiperespectrales de la superficie terrestre mediante hardware reconfigurable. *Madrid: Tesis Doctoral, Universidad Complutense de Madrid*, 2012.

[7] Guolan Lu and Baowei Fei. Medical hyperspectral imaging: a review. *Journal of Biomedical Optics*, 19(1), January 2014.

[8] A. Plaza and C.-I. Chang. Impact of Initialization on Design of Endmember Extraction Algorithms. *IEEE Transactions on Geoscience and Remote Sensing*, 44(11):3397–3407, 2006.

[9] A. Plaza, P. Martinez, R. Perez, and J. Plaza. Spatial/spectral endmember extraction by multidimensional morphological operations. *IEEE Transactions on Geoscience and Remote Sensing*, 40(9):2025–2041, 2002.

[10] Alexander F. H. Goetz, Gregg Vane, Jerry E. Solomon, and Barrett N. Rock. Imaging Spectrometry for Earth Remote Sensing. *Science*, 228(4704):1147–1153, 1985.

[11] Bing Lu, Phuong D. Dao, Jiangui Liu, Yuhong He, and Jiali Shang. Recent Advances of Hyperspectral Imaging Technology and Applications in Agriculture. *Remote Sensing*, 12(16), 2020.

[12] Rinaldo R. Izzo, Alan N. Lakso, Evan D. Marcellus, Timothy D. Bauch, Nina G. Raqueño, and Jan van Aardt. An initial analysis of real-time sUAS-based detection of grapevine water status in the Finger Lakes wine country of upstate New York. In J. Alex Thomasson, Mac McKee, and Robert J. Moorhead, editors, *Autonomous Air and Ground Sensing Systems for Agricultural Optimization and Phenotyping IV*, volume 11008, page 1100811. International Society for Optics and Photonics, SPIE, 2019.

[13] Amy Lowe, Nicola Harrison, and Andrew P. French. Hyperspectral image analysis techniques for the detection and classification of the early onset of plant disease and stress. *Plant Methods*, 13(80), 2017.

[14] Haida Liang. Advances in multispectral and hyperspectral imaging for archaeology and art conservation. *Applied Physics A*, 106:309–323, 2012.

[15] 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.

[16] Freek D. van der Meer, Harald M.A. van der Werff, Frank J.A. van Ruitenbeek, Chris A. Hecker, Wim H. Bakker, Marleen F. Noomen, Mark van der Meijde, E. John M. Carranza, J. Boudewijn de Smeth, and Tsehaie Woldai. Multi- and hyperspectral geologic remote sensing: A review. *International Journal of Applied Earth Observation and Geoinformation*, 14(1):112–128, 2012.

[17] Jorge Buzzi Marcos. Imaging spectroscopy to evaluate the contamination from sulphide mine waste in the iberian pyrite belt using hyperspectral sensors (Huelva, Spain), 2012.

[18] Jonghee Yoon. Hyperspectral Imaging for Clinical Applications. *BioChip Journal*, 16(1):1–12, March 2022.

[19] Dan Savastru Mihaela Antonina Calin, Sorin Viorel Parasca and Dragos Manea. Hyperspectral Imaging in the Medical Field: Present and Future. *Applied Spectroscopy Reviews*, 49(6):435–447, 2014.

[20] Yibing Hou, Zhong Ren, Guodong Liu, Lvming Zeng, and Zhen Huang. Design of a Novel LD-Induced Hyper-Spectral Imager for Breast Cancer Diagnosis Based on VHT Grating. In *2011 Symposium on Photonics and Optoelectronics (SOPO)*, pages 1–4, 2011.

[21] Hamed Akbari, Luma Halig, David M. Schuster, Baowei Fei, Adeboye Osunkoya, Viraj Master, Peter Nieh, and Georgia Chen. Hyperspectral imaging and quantitative analysis for prostate cancer detection. *Journal of Biomedical Optics*, 17(7):076005, 2012.

[22] Takashi Nagaoka, Atsushi Nakamura, Haruka Okutani, Yoshio Kiyohara, and Takayuki Sota. A possible melanoma discrimination index based on hyperspectral data: a pilot study. *Skin Research and Technology*, 18(3):301–310, 2012.

[23] Zhi Liu, Hongjun Wang, and Qingli Li. Tongue Tumor Detection in Medical Hyperspectral Images. *Sensors*, 12(1):162–174, 2012.

[24] Dmitry Yudovsky, Aksone Nouvong, and Laurent Pilon. Hyperspectral Imaging in Diabetic Foot Wound Care. *Journal of Diabetes Science and Technology*, 4(5):1099–1113, 2010.

[25] Christian M. Zakian, Iain A. Pretty, and Roger Ellwood. Near-infared hyperspectral imaging of teeth for dental caries detection. *Journal of Biomedical Optics*, 14(6):064047, 2009.

[26] Himar Fabelo, Samuel Ortega, Raquel Lazcano, Daniel Madroñal, Gustavo M. Callicó, Eduardo Juárez, Rubén Salvador, Diederik Bulters, Harry Bulstrode, Adam Szolna, Juan F. Piñeiro, Coralia Sosa, Aruma J. O'Shanahan, Sara Bisshopp, María Hernández, Jesús Morera, Daniele Ravi, B. Ravi Kiran, Aurelio Vega, Abelardo Báez-Quevedo, Guang-Zhong Yang, Bogdan Stanciulescu, and Roberto Sarmiento. An Intraoperative Visualization System Using Hyperspectral Imaging to Aid in Brain Tumor Delineation. *Sensors*, 18(2), 2018.

[27] 1st Lt. Pushkar Aggarwal and Francis A. Papay. Applications of multispectral and hyperspectral imaging in dermatology. *Experimental Dermatology*, 31(8):1128–1135, 2022.

[28] Robert O Green, Michael L Eastwood, Charles M Sarture, Thomas G Chrien, Mikael Aronsson, Bruce J Chippendale, Jessica A Faust, Betina E Pavri, Christopher J Chovit, Manuel Solis, Martin R Olah, and Orlesa Williams. Imaging Spectroscopy and the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS). *Remote Sensing of Environment*, 65(3):227–248, 1998.

[29] C.O Justice, J.R.G Townshend, E.F Vermote, E Masuoka, R.E Wolfe, N Saleous, D.P Roy, and J.T Morisette. An overview of MODIS Land data processing and product status. *Remote Sensing of Environment*, 83(1):3–15, 2002. The Moderate Resolution Imaging Spectroradiometer (MODIS): a new generation of Land Surface Monitoring.

[30] Mark A. Folkman, Jay Pearlman, Lushalan B. Liao, and Peter J. Jarecke. EO-1/Hyperion hyperspectral imager design, development, characterization, and calibration. In William L. Smith and Yoshifumi Yasuoka, editors, *Hyperspectral Remote Sensing of the Land and Atmosphere*, volume 4151, pages 40 – 51. International Society for Optics and Photonics, SPIE, 2001.

[31] Akira Iwasaki, Nagamitsu Ohgi, Jun Tanii, Takahiro Kawashima, and Hitomi Inada. Hyperspectral Imager Suite (HISUI) -Japanese hyper-multi spectral radiometer. In *2011 IEEE International Geoscience and Remote Sensing Symposium*, pages 1025–1028, 2011.

[32] Di Wu and Da-Wen Sun. Advanced applications of hyperspectral imaging technology for food quality and safety analysis and assessment: A review — Part I: Fundamentals. *Innovative Food Science Emerging Technologies*, 19:1–14, 2013.

[33] Nirmal Keshava and John F Mustard. Spectral unmixing. *IEEE signal processing magazine*, 19(1):44–57, 2002.

[34] Carmen Quintano, Alfonso Fernández-Manso, Yosio E Shimabukuro, and Gabriel Pereira. Spectral unmixing. *International Journal of Remote Sensing*, 33(17):5307–5340, 2012.

[35] Manoj Chandak Mayur Akewar. Hyperspectral Imaging Algorithms and Applications: A Review. *TechRxiv*, 2024.

[36] Karbhari V Kale, Mahesh M Solankar, Dhananjay B Nalawade, Rajesh K Dhumal, and Hanumant R Gite. A research review on hyperspectral data processing and analysis algorithms. *Proceedings of the national academy of sciences, India section a: physical sciences*, 87:541–555, 2017.

[37] Bo Wang, Hongwei Ke, Xiaodong Ma, and Bing Yu. Fault diagnosis method for engine control system based on probabilistic neural network and support vector machine. *Applied Sciences*, 9(19):4122, 2019.

[38] Shihong Yue, Ping Li, and Peiyi Hao. Svm classification: Its contents and challenges. *Applied Mathematics-A Journal of Chinese Universities*, 18:332–342, 2003.

[39] Himar Fabelo, Samuel Ortega, Adam Szolna, Diederik Bulters, Juan F. Piñeiro, Silvester Kabwama, Aruma J-O'Shanahan, Harry Bulstrode, Sara Bisshopp, B. Ravi Kiran, Daniele Ravi, Raquel Lazcano, Daniel Madroñal, Coralia Sosa, Carlos Espino, Mariano Marquez, María De La Luz Plaza, Rafael Camacho, David Carrera, María Hernández, Gustavo M. Callicó, Jesús Morera Molina, Bogdan Stanciulescu, Guang-Zhong Yang, Rubén Salvador, Eduardo Juárez, César Sanz, and Roberto Sarmiento. In-Vivo Hyperspectral Human Brain Image Database for Brain Cancer Detection. *IEEE Access*, 7:39098–39116, 2019.

[40] W Daniel Hillis. What is massively parallel computing, and why is it important? *Daedalus*, 121(1):1–15, 1992.

[41] William F McColl. General purpose parallel computing. *Lectures on parallel computation*, 4:337–391, 1993.

[42] Andrew Boutros and Vaughn Betz. FPGA Architecture: Principles and Progression. *IEEE Circuits and Systems Magazine*, 21(2):4–29, 2021.

[43] Intel Corporation. Hyperflex FPGA Architecture. https://www.intel.com/content/www/us/en/docs/programmable/683353/23-4/fpga-architecture-introduction.html, August 2023.

[44] Intel Corporation. Stratix 10 GX/SX Device Overview. https://www.intel.com/content/www/us/en/docs/programmable/683729/current/gx-sx-device-overview.html, June 2024.

[45] Julián Caba, María Díaz, Jesús Barba, Raúl Guerra, and Jose A. de la Torre and Sebastián López. FPGA-Based On-Board Hyperspectral Imaging Compression: Benchmarking Performance and Energy Efficiency against GPU Implementations. *Remote Sensing*, 12(22), 2020.

[46] Lenny Truong and Pat Hanrahan. A golden age of hardware description languages: Applying programming language techniques to improve design productivity. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2019.

[47] David R Coelho. *The VHDL handbook*. Springer Science & Business Media, 2012.

[48] Moe Shahdad. An overview of VHDL language and technology. In *23rd ACM/IEEE Design Automation Conference*, pages 320–326. IEEE, 1986.

[49] Zainalabedin Navabi. *Verilog digital system design*. McGraw-Hill, 1999.

[50] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. *Data Parallel C++: Programming Accelerated Systems Using C++ and SYCL*. Apress, Berkeley, CA, 2023.

[51] Kosiro Obata, Hasitha Muthumala Waidyasooriya, and Masanori Hariyama. Implementation of an FPGA-Oriented Complex Number Computation Library Using Intel OneAPI DPC++. In *2022 IEEE 65th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4, 2022.

[52] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[53] Craig Rodarmel and Jie Shan. Principal Component Analysis for Hyperspectral Image Classification. *Surv Land inf Syst*, 62, january 2002.

[54] IBM. What is principal component analysis (PCA)? https://www.ibm.com/topics/principal-component-analysis, December 2023.

[55] Daniel Fernández, Carlos González, Daniel Mozos, and Sebastián López. FPGA implementation of the principal component analysis algorithm for dimensionality reduction of hyperspectral images. *J Real-Time Image Proc*, 16(5):1395–1406, 2019.

[56] Carlos González, Sebastián López, Daniel Mozos, and Roberto Sarmiento. FPGA Implementation of the HySime Algorithm for the Determination of the Number of Endmembers in Hyperspectral Data. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 8(6):2870–2883, 2015.

[57] Intel Corporation. *Intel oneAPI DPC++/C++ Compiler Handbook for FPGAs*, June 2024.

[58] Antje Kirchner and Curtis S. Signorino. Using support vector machines for survey research. *Medium*, 1, 2014.

[59] Wang An-na, Zhao Yue, Hou Yun-tao, and LI Yun-lu. A novel construction of svm compound kernel function. In *2010 International Conference on Logistics Systems and Intelligent Management (ICLSIM)*, volume 3, pages 1462–1465. IEEE, 2010.

[60] Oliver Kramer. Unsupervised k-nearest neighbor regression, 2011.

[61] Ivo D. Dinov. *Lazy Learning: Classification Using Nearest Neighbors*, pages 267–287. Springer International Publishing, Cham, 2018.

[62] B. W. Silverman and M. C. Jones. E. Fix and J.L. Hodges (1951): An Important Contribution to Nonparametric Discriminant Analysis and Density Estimation: Commentary on Fix and Hodges (1951). *International Statistical Review / Revue Internationale de Statistique*, 57(3):233–238, 1989.

[63] Kashvi Taunk, Sanjukta De, Srishti Verma, and Aleena Swetapadma. A Brief Review of Nearest Neighbor Algorithm for Learning and Classification. In *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, pages 1255–1260, 2019.

[64] VS Prasatha, Haneen Arafat Abu Alfeilate, AB Hassanate, Omar Lasassmehe, Ahmad S Tarawnehf, Mahmoud Bashir Alhasanatg, and Hamzeh S Eyal Salmane. Effects of distance measure choice on knn classifier performance-a review. *arXiv preprint arXiv:1708.04321*, 56, 2017.

[65] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

Appendix A: Contributions

This section details the specific contributions made by each work team member to the project's development. Each one has been in charge of optimizing one of the algorithms, in addition to carrying out joint tasks. The tasks performed together with their author are described below:

A.1 Antonio Álvarez Sánchez

After the algorithm assignments following the initial meetings with the directors, the algorithm I was assigned was Principal Component Analysis (PCA). During the first few months, I dedicated myself to studying its functionality and its various applications in machine learning, noting that it is mainly used for dimensionality reduction.

At the same time, I downloaded and tested the local version of the Intel oneAPI Toolkit and the DPC++ compiler. The initial goal was to isolate each of our algorithms. However, as part of the preliminary work, the PCA algorithm was already isolated in an independent project, allowing me to analyze the initial code and begin translating it from C to C++ with SYCL for FPGAs. The directors recommended that I proceed up to the covariance matrix calculation step and test its functionality. Initially, I implemented the kernel using *parallel_for*, but the directors advised using *single_task* instead. However, this first attempt was valuable in familiarizing myself with the language. I implemented the entire algorithm, and once completed, the next step was to test it on Intel DevCloud. Regarding this platform, one of the first tasks was understanding the various boards available, known as nodes, and how they could be used interactively to submit and execute jobs. This process was somewhat complex, as the documentation was not entirely clear. I even had to post several questions in the Intel DevCloud Forums, where I received helpful guidance. I documented everything I learned to assist my teammates as well.

Once I accessed DevCloud, I emulated the complete functionality of the PCA and then compiled it on an FPGA. I also learned that the behavior of the emulation can differ from that of the FPGA, as they sometimes behave differently. Eventually, I successfully executed the entire algorithm on an FPGA, and it was time to optimize it.

The directors provided significant help in interpreting the report generated from my base implementation, with additional support from my colleague Fabrizio. Carlos recommended investigating the power iteration method for eigenvector calculation, which was highly effective, cutting the initial time in half. Later, he explained another method for implementing the covariance matrix, which is the one used in the final implementation. The final step was unrolling some of the most performance-sensitive loops in the algorithm, leading to my final version.

In the written report, I contributed to the entire chapter on PCA. Additionally, I was responsible for writing the hyperspectral imaging section in the state of the art, specifically Section 2.1.1, Section 2.1.2, Section 2.1.3, and Section 2.1.5. I also wrote the abstract and assisted in drafting the introduction and final conclusions.

A.2 Eneko Retolaza Ardanaz

After assigning the various algorithms among the team members, the algorithm I had was the Support Vector Machine (SVM). I spent the first few months cleaning the provided code and studying the functionality of the algorithm. I analyzed which parts of the algorithm were included in the code and assessed its complexity.

During this time, I acquired a computer with an Intel processor to compile and run the code in DPC++. The Informatics faculty initially helped me, by borrowing the laptop. I installed the necessary programs for the project, including a compatible version of Intel oneAPI Toolkit and the DPC++ compiler. My first task was to separate the KNN and PCA algorithms from the SVM, as all were in the same document. After that, I properly configured the input files for the SVM and created the output file to compare results in later versions.

Once I isolated the code, I added the single task and the accessors to execute the kernel. The project supervisors recommended to implement an accessor for each variable. I sat up Intel's cloud environment, compiled, and ran the code there, adjusting the accessors until I achieved the desired result.

To improve the algorithm, I followed the tutors' advice, starting with kernel access optimizations. I removed unnecessary accesses and moved variables that could fit inside the kernel. This significantly improved execution time. Next, I eliminated unnecessary "for" loops within the main loop, as many were redundant. Additionally, I removed more variables and accessors that were being used redundantly, reducing memory usage, and integrated the final sorting algorithm into the main loop. These final adjustments further reduced execution time, though not as dramatically as optimizing the accessors.

After these improvements, I couldn't reduce the time further, as all changes increased execution time. I consulted with the supervisors, and we determined that the only remaining option was loop unrolling. After testing it with several loops, execution time increased due to the high number of reads in each unrolled line, so I reverted to the previous version. After further discussions, we agreed this would be the final version, as there was no more room to improve the SVM. After this, I executed all the versions of the algorithm to obtain the time, and with the help of Fabrizio, I obtained all the algorithm report information.

In the written report part, I wrote the entire SVM chapter, did the software section (Section 2.1.4, Section 2.3.1 and Section 2.3.2) of the state of the art, and contributed to the introduction, the complete pipeline, conclusions, and future work.

A.3 Fabrizio Nicolás Zeballos

After the algorithm assignment following the first meeting, I was tasked with K-Nearest Neighbors (KNN). The first few months were spent studying the algorithm's functionality and isolating the code from the entire non-parallelized chain given. The main objective during this time was to have a working version of the algorithm in DPC++.

Once the algorithm was isolated in its base language, C, I proceeded to migrate it to

C++ with SYCL using the DPC++ compiler. During this migration, I cleaned up several vectors and reorganized them from 2D to 1D arrays. Throughout this process, I verified that the results matched those obtained in the initial code. I downloaded the Intel oneAPI Toolkit, enabling me to test results using the functionalities provided by this tool.

After translating the code into a correct version of C++ with SYCL and using the DPC++ compiler, I started utilizing specific library instructions to compile for FPGA. This phase presented numerous challenges, and using the cloud required prior training. After managing to use the cloud emulator and obtaining the first results in a version suitable for FPGAs, I attempted hardware compilation. The process of compiling and running on hardware was complex, as Intel DevCloud experienced downtime and its boards were temporarily non-functional. Once this execution was achieved, it became the starting point for implementing improvements.

Following the first execution, I proposed to my supervisors improvements such as unrolls, eliminating division or modulus operations, among other measures, though these were less practical due to the lack of prior report analysis. I introduced, with the help of my directors, architectural changes to the code, which proved to be effective, and also suggested parameter adjustments, as the differences in results were minimal. Finally, I proposed some changes to the search portion of the algorithm, but the implemented alternatives either exceeded resource limits or were even slower for the hardware in use. Ultimately, I achieved the current implementation, which uses three queues (Top, Bottom, and Fixed Window) and includes memory preloading to maximize the FPGA's resources, among other optimizations for FPGA from the Intel oneAPI Toolkit.

For the thesis writing, I have written the entire Chapter 5. Additionally, I was responsible for drafting Section 2.2 of the state-of-the-art review and assisting with the writing of sections Section 2.1 In the common sections, such as the Chapter 1, Chapter 6, and Chapter 7, I worked equally with my colleagues.