Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación

TRABAJO FIN DE MASTER
Máster en Ingeniería de Sistemas y Servicios para la Sociedad de la Información

Manycore real time constraint achievement using OpenMP and OpenCL to detect cancer cells

Jaime Jarrin Valencia

Diciembre 2016
“Your time is limited, so don’t waste it living someone else’s life. Don’t be trapped by dogma—which is living with the results of other people’s thinking. Don’t let the noise of others’ opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition.”

Steve Jobs
Acknowledgements

After a long period researching, today is the day: writing this note of thanks is the finishing touch on my master thesis. It has been a period of intense learning for me, not only on how to perform a research but also on a personal level. Writing this thesis has had an enormous impact on me. I would like to reflect on the people who have supported and helped me so much throughout this period.

First of all, I would like to thank my family, my parents who always supported me in every step of my professional and personal life, I owe them all. To my brother and sister, who always cheered me up when I needed it most, and to my beloved grandmother who supported me when I was in my very early years of College.

Furthermore, I would like to express my sincere gratitude to my advisor Prof. Eduardo Juarez for introducing to me the parallel programming world and for the continuous support throughout my master study and research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of investigation and writing of this thesis.

I have to thank Ecuadorian Government and SENESCYT for allowing me to study abroad and live such an amazing experience.

And last but not least, to all my friends and closest persons who shared this trip with me, even when we were in different countries, working in the same laboratory or living just across the street. I have to say, that thanks to you all being in Madrid while I was doing this Master was an amazing experience which I will never forget.

To all of you... Thank you!
# Table of contents

Table of contents i

List of figures iv

List of tables viii

List of equations x

Resumen xi

Summary xiii

1 Introduction 1

1.1. Motivation 1

1.1. Objectives 2

2 State of the art 4

2.1. Introduction 5

2.2. Hyperspectral images 5

2.2.1. Applications of hyperspectral images 7

2.3. Multicore platforms 10

2.3.1. GPU 13

2.3.2. Manycore 15

2.3.3. GPU and manycore faceoff 19

2.4. Parallel programming 22

2.4.1. Posix threads 22

2.4.2. OpenMP 23

2.4.2.1. Memory model 24

2.4.2.2. Handling parallelism 24

2.4.2.3. API components 25

2.4.2.4. Parallel control structures 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.2.5</td>
<td>Communication and data environment</td>
<td>29</td>
</tr>
<tr>
<td>2.4.2.6</td>
<td>Synchronization</td>
<td>31</td>
</tr>
<tr>
<td>2.4.3</td>
<td>OpenCL</td>
<td>33</td>
</tr>
<tr>
<td>2.4.3.1</td>
<td>Hardware Elements</td>
<td>33</td>
</tr>
<tr>
<td>2.4.3.2</td>
<td>Memory model</td>
<td>34</td>
</tr>
<tr>
<td>2.4.3.3</td>
<td>Program Components</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>Host program</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>OpenCL kernel program</td>
<td>40</td>
</tr>
<tr>
<td>2.4.3.4</td>
<td>Profiling</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>Materials and methods</td>
<td>44</td>
</tr>
<tr>
<td>3.1</td>
<td>Kalray MPPA-256</td>
<td>45</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Technical description</td>
<td>45</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Programming the MPPA-256</td>
<td>48</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Posix and OpenMP on the MPPA</td>
<td>49</td>
</tr>
<tr>
<td>3.1.4</td>
<td>OpenCL on the MPPA</td>
<td>56</td>
</tr>
<tr>
<td>3.2</td>
<td>Sequential algorithms</td>
<td>57</td>
</tr>
<tr>
<td>3.2.1</td>
<td>SVM algorithm</td>
<td>57</td>
</tr>
<tr>
<td>3.2.2</td>
<td>PCA algorithm</td>
<td>59</td>
</tr>
<tr>
<td>4</td>
<td>Application development</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>SVM: OpenMP</td>
<td>66</td>
</tr>
<tr>
<td>4.2</td>
<td>PCA: OpenMP</td>
<td>71</td>
</tr>
<tr>
<td>4.3</td>
<td>SVM: OpenCL</td>
<td>83</td>
</tr>
<tr>
<td>4.4</td>
<td>PCA: OpenCL</td>
<td>86</td>
</tr>
<tr>
<td>5</td>
<td>Analysis of results</td>
<td>93</td>
</tr>
<tr>
<td>5.1</td>
<td>Validation of the algorithms</td>
<td>94</td>
</tr>
<tr>
<td>5.2</td>
<td>Time analysis</td>
<td>94</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Sequential implementation</td>
<td>94</td>
</tr>
</tbody>
</table>
5.2.2. SVM: OpenMP 95
5.2.3. SVM: OpenCL 98
5.2.4. PCA: OpenMP 99
5.2.1. PCA: OpenCL 103
5.2.2. Comparison 105

5.3. Coding methodology 108

6 Conclusions and future lines 110
6.1. Conclusions 111
6.2. Future lines 112

7 References 114

Appendix 119
Annex I 120
Annex II 124
Annex III 127
List of figures

Figure 1. Electromagnetic spectrum 6
Figure 2. Spectral signatures 6
Figure 3. Example of mineral exploration on earth’s surface using Hyperspectral imaging. 7
Figure 4. Example of fish freshness classification (pixels in red are those classified as “non-fresh”) [6] 8
Figure 5. Tumor detection a) oncologist delimitation b) delimitation made by hyperspectral analysis 9
Figure 6. Hyperspectral image acquisition [8] 9
Figure 7. Human’s brain tumor detection using hyperspectral imaging 10
Figure 8. Intel’s processors power density evolution 11
Figure 9. Processor’s performance [10] 12
Figure 10. NVidia Tesla K40 14
Figure 11. GPU Mechanism 14
Figure 12. AMD FirePro GPU a) S-Series b) W-Series 15
Figure 13. Manycore architecture [18] 16
Figure 14. Server Xeon with Xeon Phi as co-processor 17
Figure 15. Intel Parallel Studio XE and Xeon Phi clustering 17
Figure 16. Cavium server with 2 ThunderX processors 18
Figure 17. Kalray MPPA board 19
Figure 18. Fork and join concept 22
Figure 19. Pthread methods to create threads 23
Figure 20. Non-uniform memory access scheme 24
Figure 21. OpenMP API components [22] 25
Figure 22. OpenMP language extensions 26
Figure 23. OpenMP parallel control structure 27
Figure 24. OpenMP parallel for constructor 27
Figure 25 OpenMP Parallel sections 28
Figure 26 OpenMP control constructor comparison 29
Figure 27 OpenMP shared and private variable 29
Figure 28 OpenMP critical example 30
Figure 29 OpenMP atomic example 31
Figure 30 Barrier execution process 32
Figure 31 OpenMP barrier example 32
Figure 32 OpenCL platform model [26] 34
Figure 33 OpenCL Memory model [28] 35
Figure 34 OpenCL program components 36
Figure 35 OpenCL program phases 36
Figure 36 Creating and compiling the OpenCL program 37
Figure 37 Create data and arguments 38
Figure 38 clEnqueueNDRangeKernel prototype 39
Figure 39 Receiving data in OpenCL 40
Figure 40 Releasing the objects in OpenCL 40
Figure 41 Synchronization in OpenCL [27] 41
Figure 42 Kernel example 42
Figure 43 Profiling in OpenCL 42
Figure 44 Global architecture of the Kalray MPPA-256 [31] 46
Figure 45 VLIW implementation on MPPA [31] 47
Figure 46 Outline of the architecture MPPA-256 [34] 48
Figure 47 MPPA Developer 48
Figure 48 Application on Developer workstation 50
Figure 49 MPPA NoC topology 51
Figure 50 Sync connector Rx mode 52
Figure 51 Sync connector TX mode 52
List of tables

Table 1 General specifications of GPU and manycore platforms 21
Table 2 Kalray OpenCL device information [39] 56
Table 3 SVM linear kernel algorithm 59
Table 4 PCA algorithm description 64
Table 5 Average algorithm for clusters 72
Table 6 Covariance matrix algorithm 75
Table 7 PCA algorithm sequential 78
Table 8 Image projection and reduction algorithm 83
Table 9 SVM sequential execution time 95
Table 10 Sequential execution time of the PCA algorithm 95
Table 11 Execution time of the SVM parallelized algorithm 96
Table 11 SVM execution time with OpenCL 99
Table 12 Send, read, process times of SVM kernel 99
Table 13 OpenMP PCA with 14 threads 100
Table 14 OpenMP PCA Execution time 101
Table 15 Execution time of PCA step 3 using OpenMP and POSIX 103
Table 16 PCA OpenCL concurrent elements to be zeroed = 15 104
Table 17 PCA OpenCL concurrent elements to be zeroed = 16 104
Table 18 PCA OpenCL concurrent elements to be zeroed = 112 105
Table 19 Posix Execution time (manual threads) 127
Table 20 PCA sequential steps execution time $\varepsilon = 1 \times 10^{-5}$ 128
Table 21 SVM algorithm with POSIX 129
Table 22 PCA OpenMP $\varepsilon = 1 \times 10^{-4}$ 130
Table 23 PCA OpenMP $\varepsilon = 1 \times 10^{-5}$ 131
Table 24 PCA OpenMP $\varepsilon = 1 \times 10^{-6}$ 131
Table 25 PCA OpenMP $\varepsilon = 1 \times 10^{-7}$ 132
Table 26 PCA OpenMP $\epsilon = 1 \times 10^{-8}$ 132
Table 27 PCA OpenMP $\epsilon = 1 \times 10^{-9}$ 133
Table 28 PCA OpenMP $\epsilon = 1 \times 10^{-10}$ 133
Table 29 PCA OpenMP $\epsilon = 1 \times 10^{-12}$ 134
Table 30 PCA OpenMP $\epsilon = 1 \times 10^{-16}$ 134
Table 31 PCA OpenMP $\epsilon = 1 \times 10^{-20}$ 135
Table 32 SVM OpenMP monocluster 136
Table 33 SVM OpenMP with 13 clusters 136
Table 34 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-5}$ concurrent = 15 137
Table 35 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-6}$ concurrent = 15 137
Table 36 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-7}$ concurrent = 15 138
Table 37 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-8}$ concurrent = 15 138
Table 38 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-9}$ concurrent = 15 139
Table 39 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-10}$ concurrent = 15 139
Table 40 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-12}$ concurrent = 15 140
Table 41 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-16}$ concurrent = 15 140
Table 42 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-20}$ concurrent = 15 141
Table 43 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-5}$ concurrent = 16 141
Table 44 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-6}$ concurrent = 16 142
Table 45 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-7}$ concurrent = 16 142
Table 46 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-8}$ concurrent = 16 143
Table 47 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-9}$ concurrent = 16 143
Table 48 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-10}$ concurrent = 16 144
Table 49 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-12}$ concurrent = 16 144
Table 50 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-16}$ concurrent = 16 145
Table 51 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-20}$ concurrent = 16 145
Table 52 SVM with OpenCL execution times 146
List of equations

(1) Discriminant function 58
(2) Linear kernel 59
(3) Average calculation 61
(4) Covariance Matrix based on eigenvectors 61
(5) Transformation equation 61
(6) Covariance matrix 61
(7) Rotation matrix 62
(8) m value 62
(9) t value 63
(10) cos(α) 63
(11) sin (α) 63
(12) Covariance matrix prima 63
(13) Eigenvector calculation 63
(14) Eigenvalues calculation 63
(15) Size of image for SVM algorithm 66
(16) Vectorization of a matrix 66
(17) Size of block image used in SVM with OpenMP 67
(18) Index calculation to use as offset with multiplex) 70
(19) Size of image for PCA algorithm 71
(20) Block size of the image for PCA with OpenMP 71
(21) Matrix multiplication to find covariance matrix 73
Resumen

En la actualidad, es frecuente que campos como la investigación médica y TI (Tecnologías de la Información) trabajen en conjunto, descubriendo y/o perfeccionando métodos para determinar diferentes tipos de enfermedades que se desarrollan en los seres humanos.

En este contexto, el presente trabajo de investigación describe la aplicación de imágenes hiper espectrales para la detección de tumores en el tejido cerebral mediante la implementación de una cadena de procesado dividida en dos fases: la reducción dimensional –aplicando el Análisis de Componentes Principales (PCA) – y la clasificación –utilizando Máquinas de Vectores Soporte (SVM).

Como ambos procedimientos demandan una gran cantidad de procesamiento, su implementación secuencial requiere mucho tiempo para su ejecución, imposibilitando su utilización en un sistema de tiempo real. Para solucionar este problema, se plantea el uso de programación paralela sobre una arquitectura multinúcleo para poder acelerar el tiempo de ejecución.

En estos momentos no es posible encontrar ningún método probado que transforme automáticamente un código secuencial en paralelo; sin embargo, existen interfaces de programación de aplicaciones (APIs) que permiten crear códigos capaces de explotar el paralelismo del hardware, lo que puede ayudar a convertir el código secuencial en paralelo. En consecuencia, el objetivo principal de esta investigación es estudiar tres métodos diferentes para explotar el paralelismo: (1) POSIX, basado en hilos concurrentes para ejecutar el código en paralelo; (2) OpenMP, que ayuda con la creación de hilos así como también con la sincronización; y, (3) OpenCL, el cual brinda un mayor nivel de abstracción, se encarga principalmente de las comunicaciones y accesos a memoria así como también de definir la forma en la cual las funciones se ejecutan en paralelo.

Estos métodos han sido utilizados en implementaciones secuenciales de SVM y PCA para crear versiones paralelas del código, que han sido probadas en una MPPA-256 de Kalray, una plataforma de alta computación compuesta por 256 núcleos distribuidos en 16 clústeres.
Los resultados muestran que la versión paralela de SVM con OpenCL es 10.79 veces más rápida que la implementación secuencial; mientras que, OpenMP funciona mejor con el algoritmo PCA, alcanzando una aceleración de 2.52.

Estos resultados apuntan a que los procesos descritos permiten explotar adecuadamente el paralelismo del hardware y facilitan a los desarrolladores su trabajo al ejecutar las aplicaciones paralelas.
Summary

Nowadays, the fields of medical research and IT (Information Technology) work together, discovering and refining methods to determine the different types of diseases that affect humans.

In this context, the present research work describes the application of hyperspectral images for the detection of tumors in brain tissue by implementing a processing toolchain divided into two phases: dimensional reduction - applying the Principal Component Analysis (PCA) - and classification - using Vector Support Machines (SVM).

Considering that both procedures require a significant amount of processing, using the sequential implementation takes a long time to process it, making it impossible to use in a real-time system. To solve this problem, this work proposes the use of parallel programming in a multicore architecture to speed up the runtime.

At present, it is not possible to find any proven methodology that automatically transforms a sequential code into parallel one. However, there are Application Programming Interfaces (APIs) that allow creating code capable of exploiting hardware parallelism, which can help to convert the sequential code into parallel one. Therefore, the main objective of this research is to study three different methods to exploit parallelism. (1) POSIX, based on concurrent threads to execute the parallel code. (2) OpenMP, which helps with the creation of threads as well as synchronization. And, (3) OpenCL, which provides a higher abstraction level defining methodologies for communications, memory access and queue instructions to be executed in parallel within a multicore device. These methods have been used to develop parallel implementations of SVM and PCA tested in a Kalray MPPA-256, a high-performance computing platform composed of 256 cores distributed in 16 clusters.

The results show that the parallel version of SVM with OpenCL is 10.79 times faster than the sequential implementation. For the case of PCA algorithm, OpenMP reaches a speed-up of 2.52 times. These results suggest that the described processes allow to exploit the parallelism of the hardware properly and ease developer's work while creating parallel programs.
1 Introduction
1.1. Motivation

This research work is related to the project HELICoiD (HypErspectraL Imaging Cancer Detection), which is an European project funded by the Seventh Framework Program of the European Union and, particularly, by the FET - Open (Future & Emerging Technologies) initiative. The partnership of the project is composed of several institutions of different countries, as Spain, France, United Kingdom and Netherlands.

HELICoiD main objective is to find a generalized methodology to discriminate healthy and carcinogenic cells during surgical interventions by applying hyperspectral images processing techniques to extract information across the electromagnetic spectrum. The hypothesis posed by HELICoiD is the following. As cancer cells implies a change in cellular physiology, therefore it should generate changes in its spectral signature as well. Based on this, it is intended to determine if it is possible to extract the hyperspectral signature of each cancer cell in order to generalize a model for locating them.

Concretely, this work had been developed at the Universidad Politécnica de Madrid (UPM) and specifically at the Electronic and Microelectronic Design Group (GDEM by its Spanish initials) which is the leader of the work package related to the algorithm implementation on a manycore architecture and the subsequent performance analysis considering the ‘surgical real-time’ constrain needed due the sensitivity of its application inside the surgery room. This constrain is not the same as the one established for audio and video standards where 30 to 40 frames per second can be processed. For this implementations we consider a fixed image (somewhat similar to a photograph) taken by the hyperspectral camera in the surgery room while the surgeon is operating the patient. During the surgery, the camera is always focused on the interest area which change only due to surgeon interventions. This interventions are small, slow and accurate so the ‘real-time’ constrain can be reduced as the doctor can wait for a short period of time to analyze the affected area (approximately, 120 s). Thus a real-time constrain for this case is the time to process an image before receiving a new one.

Due to the required amount of computations of the algorithms to process a Hyperspectral image the computational device should be faster enough to compute all the data within the real time constrain. Since powerful general purpose CPUs with sequential software are intended to be used within a controlled environment (e.g., datacenters) it is difficult to build a small system to use inside the surgery room employing
this kind of processors. Note that it is important to find a dependable device to bring in the surgery room, which must be efficient (code-size, run-time, energy consumption) and powerful enough to react with a ‘real-time’ response. Taking the aforementioned in mind, a device with a parallel architecture can be considered, a manycore computing device which can give us a good raw performance, reliability and power management. However, the actual market offers several devices options (architectures, memory models, number of cores, coding methodologies), which makes writing software that can fully benefit from the hardware much harder. Hence, in order to use a sequential code which benefits from the improved hardware to retain the “write once, run faster on new hardware” paradigm, commercial and scientific software must be changed to new software development and support mechanisms. This is where the ‘automatic’ parallel paradigm appears, i.e. a methodology to use already existing sequential code to obtain parallel code which employs hardware resources at full capacity.

The focus of this project is on what has been previously mentioned, to study methods to evaluate automatic mechanisms to adapt sequential codes to parallel architectures, easing the tasks for the developer. This techniques will be applied to two processing stages of the Hyperspectral images used in the HELICoID project: spatial reduction and image classification. The goal is to increase the performance of the program reducing the execution time, while the code is adapted into a parallel architecture. This research will evaluate the manual implementations against the automated implementations on the parallel architecture in terms of speed up. It is expected to reach the same speed-up using both methodologies. At last, this research will try to discover if it is possible to achieve the ‘real-time’ constrain using these methods.

1.2. Objectives

As was explained on the previous section, the main objective of this work is to study methods to automatize the parallelization of a program using developing tools in order to ease programing complexity. In addition, the implementation of these methods on a manycore system has as main goal to check the speed-up achieved against a manual parallelization to find if it is possible to meet the surgical real-time constrain. To reach these global objectives the following specific objectives have been set:

- To study the actual parallelization frameworks schemes, its evolution and application onto a manycore architecture.
- To learn how to use and program a manycore platform, the so-called MPPA-256. In order to be able to develop software using automatic parallelization schemes.
To develop software on the MPPA-256 using automatic parallelization schemes and applying these techniques to the algorithms defined in [1] and [2] in order to evaluate their complexity and performance.

To test the results obtained using automatic parallelization and manual methods. To compare its performance in terms of speed-up to find if it is suitable to apply automatic methods to achieve the ‘surgical real-time’ constrain.
2 State of the art
2.1. Introduction

This chapter will introduce some of the basic concepts and state of the art related to hyperspectral images and manycore platforms which are fundamental to better understand the issues associated to a real-time implementation of the kind of application proposed in this document.

The first part of the current chapter will cover the concepts of hyperspectral images and explain their main characteristics, as well as its usage focused on medical applications. Additionally, information about the usage of hyperspectral images within the HELICoID project will be provided.

Secondly, this chapter will perform a study of the ‘state of the art’ of multicore platforms, their evolution, characteristics and actual market status. The main objective of this study is to show the variety of the market and to provide a general idea of the potential performance that multicore platforms can achieve.

Finally, the methods to develop parallel programs to exploit multicore capacities will be studied. Two perspectives will be introduced. On the one hand, a manual method using POSIX\textsuperscript{1} and, on the other hand, automatic methods such as OpenMP and OpenCL, which use compiler directives instead of pure code to allocate the work over a multicore platform.

2.2. Hyperspectral images

Since human eyes are capable to sense only three bands of the electromagnetic spectrum (blue, green, red) information further away from the visible spectrum cannot be detected. Based on the knowledge that there might be useful information outside the visible spectrum, researchers have tried to create new sensing technologies with the capability to detect this information. Thus, during the last years of the XX century, a new research field known as image spectroscopy, or more commonly known as hyperspectral imaging \cite{1}, has arisen. The main objective of this field is the extraction, study and interpretation of electromagnetic radiation obtained from the interaction between the radiation source, such as light source, and the object under study.

\begin{footnotesize}
\textsuperscript{1} POSIX, Portable Operating System Interface for Unix
\end{footnotesize}
The basis of this research field considers that all materials interact with electromagnetic waves, either absorbing, reflecting or emitting electromagnetic energy. Therefore, this energy presents specific patterns of the molecular composition of the materials under study at different wavelengths. This means that each pattern can be used to extract a unique spectral curve, the so-called spectral signature or endmember, which can identify materials in an unambiguous way. The spectral signature is obtained by sensors that measure parameters such as reflectance (percentage of reflected light by a material or surface) in a specific wavelength of interest.

One key factor of hyperspectral imaging is that it is able to process information outside visible spectrum, allowing researchers to obtain more information from images. Figure 2 is an example of spectral signatures of a hyperspectral imaging. Using a sensor capable to measure reflectance we can detect several types of materials (grass, water-lake, soil, fire, smoke). Every material has its own characteristic curve, allowing researchers to easily classify materials using this technique.

![Electromagnetic spectrum](image1)

**Figure 1. Electromagnetic spectrum**

![Spectral signatures](image2)

**Figure 2. Spectral signatures**

---

2 Source: European Organization for the Exploitation of Meteorological Satellites (EUMETSAT)  
http://www.eumetsat.int
2.2.1. Applications of hyperspectral images

Once the basis of hyperspectral imaging has been covered, this section briefly introduce the applications of this field.

One of the first applications of this field is the analysis and study of Earth’s surface. By using satellites to get images of the Earth, hyperspectral image processing allows researchers identifying agricultural crops, as well as performing resource exploration to find minerals, oil and gas. Additionally, hyperspectral imaging techniques can be applied to spatial research as well. For instance, when a reconnaissance mission found big canyons on Mars and studied the minerals found on the surface to compare its hyperspectral signatures with minerals from earth, Scientists found signs of the existence of liquid water in some moment of the history of the planet [3].

![Figure 3 Example of mineral exploration on earth’s surface using Hyperspectral imaging.](image)

(a) Hyperspectral Spectrometer end members wavelengths. b) Mapping results overlain earth’s Surface. [4]
Hyperspectral imaging analysis can allow detecting the quality of food. There exist several studies to determine the fecal and fungal contamination on fruits, vegetables, meat, chicken and fish. As well as this methods, hyperspectral imaging techniques can be used to determine how good the food is. For instance, hyperspectral imaging techniques can be applied to wheat (e.g. pre-harvest germination) and maize (e.g. moisture and oil content), apples (e.g. bitter pit and bruise detection) and other fruit (e.g. peach, strawberry), and fish fillets (fat and moisture content and detection of nematodes and parasites) [5]. The key point of using hyperspectral imaging on this field is related to the non-intrusive methods which does not alter the components of the meat, fruit or vegetables after the analysis [6].

There are several other applications of hyperspectral imaging, such as forensics analysis, document’s verification, security and defense, disaster management and medical application and others. This document is focused on the last application, specifically on carcinogenic tumors; this field as well the others are deeply explained by Lazcano R. in [1].

There are several studies to find the methodology to detect and locate tumors. Scientists started with animals, they induced a prostate tumor in a laboratory rat [7]. After that, hyperspectral images where used to detect tumors. Initial tests were satisfactory giving to scientists the possibility to remove the affected cells to prevent tumor to spread or even to avoid re-spawn. Considering that tests with animals were successful, scientist decided to apply this methodology to detect ex vivo⁢³ gastronomic tumors and tongue tumors

---

³ Ex vivo, refers to experimentation or measurements done in or on tissue from an organism in an external environment with minimal alteration of natural conditions.
in vivo⁴ without removing the cells from human patient. In Figure 5, the results of the last experiments are shown. In general, both results are quite similar to those generated by an expert (Oncologist) confirming that hyperspectral analysis could provide an accurate classification on cancer cells.

Figure 5 Tumor detection a) oncologist delimitation b) delimitation made by hyperspectral analysis

Since, HELICoiD project aims to detect cancer cell in vivo, project partners have designed a system to capture the hyperspectral images in the surgery room, to process them and to provide information to surgeons in order to assist their decision within the surgery.

Figure 6 Hyperspectral image acquisition [8]

Figure 6 shows the system to acquire and process the images. It is composed by two hyperspectral cameras, a server-class system known as Hyperspectral Data

---

⁴ In vivo, are studies in which the effects of various biological entities are tested on whole, living organisms, such as, animals of humans.
Processing Units (HDPU) which is used to process the images taken by the cameras, a processing system, to classify, detect and locate cancer cells, and a user interface system. This will allow to have results similar to those shown in Figure 7, where the tumor area is delimited giving to doctors an extra ‘point of view’ made by the computational analysis of hyperspectral signatures of cancer cells.

2.3. Multicore platforms

Considering that current applications, such as hyperspectral imaging, require an extremely high computational power, researchers have been developing faster processors with increasing performance in order to process more rapidly a massive amount of data. It is true that a higher processor’s speed will reduce processing time, however, nowadays the questions might be the following: how fast processors must be to achieve real time processing constrains? Is it possible to build such a processor?

According to Gordon Moore, Intel co-founder and author of Moore’s Law in 1965, foretold that the number of transistors in a dense integrated circuit doubles approximately every two years. This increase has been possible due transistor’s size reduction every two years, allowing to have more transistor in the same area of a single chip and to increase processor frequency (processor’s speed) as well. However, there was a problem. Effectively, increasing the number of transistors augments the required energy and dissipating heat capacity of the processor.

Referring to what was mentioned by Marko Bertogna [9], Intel noticed the problem related to increasing the number of transistors in a single processor chip when they were developing a test processor known as Tejas in 2004. As can be seen in Figure 8, the
greater the number of transistors in a single chip, the greater the power generated by squared centimeter (power density), which imply an increase of temperature as well. Tejas was cancelled because it was approaching to the power density of a nuclear reactor making it impossible to manage.

Figure 8 Intel’s processors power density evolution

Hence, answering the aforementioned questions, nowadays it is impossible to build a single chip processor fast enough to compute massively amount of data due of the unmanageable heat and dissipation power that a chip might generate. In consequence, this discovery was known as an inflection point in the market flow. Consequently, manufacturers have had to find solutions to this problem and keep offering processors able to process highly demanded applications. The proposed solution has been to group processors to share workload instead of increasing solely processor’s frequency. Thus, applying the definition “divide and conquer” processors may split the work among them to increase the overall performance of a device.

With that in mind, manufacturers of all kind of computational devices focused their work on developing multicore platforms, allowing to have multiple processors with physically independent cores working together to increase the overall computational power. In Figure 9 it is possible to observe, that in order to keep increasing the processor’s performance, more cores are needed and, as a consequence, parallel programing methods are required as well.
Due to this change of the market flow, many vendors developed general purpose processors with more than one core in one chip, from two up to eight cores, what is currently known as ‘multicore processors’. There are several examples of this processors, such as, Intel series i3, i5, i7 which includes 2 or 4 cores, with core speed up to 3.33 GHz on its best processor. IBM with Power series processors with 4, 6 or 8 cores per chip with speed around 3 up to 4 GHz. AMD with Opteron processor series starting with 4 at 3.5 GHz up to 16 cores with nominal speed of 2.8 GHz per core. This platforms are well suited for general purpose computing for home or office usage. However, for applications with a real time constrains and, specifically, for the applications evaluated in this research work these platforms do not fit for two main reasons. First, even though processing power is extremely high in a single core these platforms cannot exploit all the advantages of parallelism because of its general purpose designed architecture. Secondly, because of its power consumption, increasing the number of cores requires more power to energize the platform. Thus a power efficient platform with the capacity of taking the advantage of the hardware to extract the parallelism is still needed.

To overcome the issues mentioned before it is important to focus on what the application needs. This work looks for raw performance alone (total number of instructions executed within a unit of time). This goal can be alternatively achieved using more but less powerful cores that can be able to outperform chips with few but powerful cores. In this way the power budget and in some way the real time constrains can be fulfilled. Hence, this type of platforms use processors housing hundreds to thousands of
cores; many but simpler cores with lower power requirements targeting a specific application. These platforms are known as a ‘manycore platforms’.

Nowadays there are several options of manycore platforms, each of them with their own strengths and weakness. Thus this section will cover the state of the art of this type of platforms.

Mainly, it is possible to identify two categories. First the general purpose graphical processing units (GP-GPU) category, which in our case is suitable to the project objectives because this research is focused in process hyperspectral images and secondly, the pure manycore platforms category. As this work is intended to be employed in the HELICOiD project, this section will show a brief state of the art of hardware architectures available at this moment in the market. However, to select the most suitable platform in terms of the project scope a deeply study has been needed, which is covered by Eduardo Juárez and Rubén Salvador in [11].

2.3.1. GPU

In this research work the main task is related to image processing. Thus, it is easy to figure out that specialized processors like the GPUs are well suited to the task. A GPU is an electronic circuit designed to process images rapidly and efficiently using a buffer to show the images processed in a display. There are several devices that use this technology, such as, cellphones, workstations, embedded systems, video games and many others [12]. GPUs have been evolving since its creation so they have a wide area of applications from the standard image analysis to hologram generation [13] or even neural networks implementations [14]. The evolution of GPUs started back in 1999 when NVIDIA\(^5\) introduced the language OpenGL and combined with Microsoft’s DirectX specification resulted in more programmability of the graphics rendering process. Eventually, this evolution led to the introduction of a GPU architecture. In the beginning, these manycore devices were capable to handle a single instruction among their cores because the control and instruction cache are shared among all the cores. This feature is known as SIMD (single instruction multiple data) [15]. To handle the limitation in terms of performance of the SIMD architecture, NVIDIA created a Tesla GP-GPU (General Purpose GPU) architecture based on a Single Instruction Multiple Thread architecture (SIMT), which is an hybrid architecture comprising MIMD (multiple instruction multiple data) and SIMD elements that is capable of running many threads across the processor.

\(^5\) http://www.nvidia.com
The organization of the architecture is hierarchical. The processor can host hundreds to thousands of PE (processing elements), which are the SIMD elements. This cores are grouped in clusters conforming the MIMD elements, known also as CUDA (Compute Unified Device Architecture) cores for this proprietary architecture, which are capable to run multiple threads. To get the maximum performance of this architecture programmers should develop an application specifying the number of threads and how those threads are going to be distributed among the hardware. To ease this task NVIDIA developed a proprietary language, called CUDA, which is based in C/C++ to get the maximum performance of this platform. However, other languages are supported as well like C/C++ and OpenCL [16].

This GP-GPU is an accelerator device therefore its usage is intended to speed up a very specific part of the code. Therefore, to run a full application a host with a CPU is needed. As we can see in Figure 10 the NVIDIA Tesla has a PCI Express port to insert it in a host workstation. The process to speed-up an implementation is to deliver to the GPU the parallel section of the code exclusively, the rest of the code must run on the workstation as is shown on the Figure 11.
Nowadays, NVIDIA has only one competitor on the GP-GPU market, AMD\(^6\), with its FirePro devices, the W-series, similar to the Tesla, designed to work in a workstation and the S-Series capable to work independent as a server without needing any other device. One key factor to consider is that AMD bets on OpenCL for programing its devices without creating a proprietary language ensuring the heterogeneous computing (this will be discuss later on the OpenCL section). AMD FirePro supports as well OpenGL, C/C++ and OpenMP [17].

![Figure 12 AMD FirePro GPU a) S-Series b) W-Series](image)

2.3.2. Manycore

As we explained before, a manycore platform or device is characterized by using hundreds to thousands of processors to perform different tasks. In contrast to the GPU, manycore devices usually have far fewer cores than the GP-GPU has PE. However, each core is a complete processor in a manycore platform able to execute independently the same or different programs. As well as GPUs, manycore devices handle parallelism based on threads. In contrast to GPUs, manycore devices have the capacity to execute longer and more heavyweight threads because of the existing total independency among the cores. Typically, most of the manycore platforms have the same architecture as shown in Figure 13. Normally, there is a host device that is the one that handles the overall execution of a program where the external memory is located. External memory data is moved to the manycore device which is composed of cores with local memory. These cores are grouped into clusters with its own memory that could be connected to an internal memory of the manycore device. In turn, internal memory could or could not

---

\(^6\) http://www.amd.com
be shared among the clusters. Typically, core’s memory and internal memory are connected inside the chip through a network on chip (NoC).

Figure 13 Manycore architecture [18]

Briefly speaking, local memory and internal memory are always a constraint in manycore architectures. Currently, manycore platforms differences are related to the (1) memory system architecture, (2) type of networking among the cores, (3) core heterogeneity or homogeneity, (4) programming software paradigms, (5) frequency and (6) power consumption.

Nowadays, there are several examples of manycore devices in the market. For instance, the Xeon Phi processor from Intel\(^7\) is capable of hosting from 61 cores, clocked at a frequency of 1.3 GHz on its 1\(^{st}\) generation, up to 72 cores, clocked at a frequency of 1.5GHz. This architecture is able to reach a high throughput thanks to its high memory bandwidth. This device can work as a co-processor (like an accelerator) in its first version, as it is shown in Figure 14. In addition, this device can work independently using its native execution mode, looking like a Xeon Processor with a higher number of cores. This feature is possible exclusively in the second generation of this processor known as Knights Landing which its main characteristic is the capacity to work without other processor acting as a host [19].

\(^7\) http://www.intel.com
Chapter 2 – State of the art

Figure 14 Server Xeon with Xeon Phi as co-processor

The maximum processing capacity of the Intel Xeon Phi as a coprocessor is 1208 GFLOPS and an energy efficiency of 4~4.5 GFLOP/W. The Xeon Phi is able to run in parallel 240 threads distributed among its 61 cores. The memory that this device can host varies from 6 up to 16 GB. Although Intel has not develop any proprietary programming language for this parallel device, several languages are supported such as C/C++ and Fortran. On top of that, OpenMP, combined with its developing suite called Intel Parallel Studio XE, may help the programmer to exploit the parallelism that the hardware is designed for, even with more than one Xeon Phi co-processor working in cluster mode, as can be seen in Figure 15.

Figure 15 Intel Parallel Studio XE and Xeon Phi clustering

Intel is not the only manufacturer who tries to lead the manycore market. Cavium has launched the Thunder X processor, which can host 48 ARM cores with frequencies up to 2.5GHz. Thunder X is able to handle one thread per core, 48 threads in total. With

---

8 Source: Intel
9 http://www.cavium.com
this powerful device Cavium targets the datacenter and cloud computing market because of the rack server shape of the device. It can host up to 256 GB of memory distributed over the 48 cores. What Cavium introduced in its thunder chip is the ability to perform virtualization tasks in hardware to isolate compute, memory, I/O resources, which means that with this device several accelerators working in parallel and independently can coexist [20]. The C/C++ programming languages is supported in this platform. However to use its full capacity a proprietary hypervisor for virtualization is needed.

![Figure 16 Cavium server with 2 ThunderX processors](image)

The aforementioned manycore examples of Intel and Cavium were designed without the power limitation constrain in mind because those devices were intended to be used in data centers. However, to select the device for the HELICoID project a balance between performance and power consumption is needed. The French manufacturer Kalray has a manycore platform, called MPPA (Multi-Purpose Processor Array) with up to 256 cores where every processor can reach frequencies up to 0.8GHz (using overdrive) with its second generation “Bostan”. The key feature of this device is its reduced power consumption only 5W. The Bostan MPPA is capable to execute 422 GFlops of double point precision. One of the principal objectives of this device is to reach high performance computing reducing the energy consumption. It is possible to create up to 256 threads distributed in all of its 256 processors grouped in 16 clusters with its cluster shared memory (up to 2MB per cluster). Kalray delivers a full developing suite of tools including those of analysis, error detection and compiling. Additionally, it supports
applications developed in C/C++, OpenMP and OpenCL. The full specification of this device will be covered in detail in the next chapter.

![Kalray MPPA board](image)

**Figure 17 Kalray MPPA board**

### 2.3.3. GPU and manycore faceoff

Based on the work made in [11], Table 1 shows a summary of the general specifications of GPU and manycore devices to end up with the strengths and weakness of every platform and the final selected device.

NVIDIA offers a high processing speed with Tesla K40 and its overall performance is high. However, to reach that performance it is mandatory to use a proprietary programming language, CUDA, meaning that the learning curve for a developer could be long. On the other hand, AMD, with its FirePro series GPU, supports open programming languages such as OpenCL and C/C++. FirePro also has an extremely high performance on its server type GPU solution. However its reduced memory could represent a constraint. Thinking about the power consumption of these GPU platforms, both are designed for datacenters and fixed environments. This a strong limitation for the HELICoID project objectives.

Considering the manycore platforms studied, Intel, with its Xeon Phi platform, is really a good contender, capable of perform a 1 TFLOP double precision operations with fewer cores than the GPU. It also includes a developing suite with customized compilers for the architecture and supports many open languages as well as most of the common x86 programming languages. However, it suffers the same problem as the GPU devices,
its power consumption is elevated and the overall performance (4.5 GFLOPS/W) is the lowest of all the devices related in this work.

Cavium, with its ThunderX platforms, promises being a challenger in the future market. With 48 processors at 2.5GHz, ThunderX platforms, with its virtualization capacity, provide developers the option to have multiple accelerators in one board. However, it does not offer the possibility to exploit the hardware at its maximum capacity. As well as the other platforms, this one is intended for datacenter environments. Thus, its power consumption is high and works exclusively in a supported blade server.

Finally, the last option studied is the MPPA from Kalray. This device allows having 256 processors consuming only 5 to 16 Watts, being the most energy efficient platform studied. The computing performance is another key factor. The MPPA can reach up to 634 GFLOPs single-point precision or 315 GFLOPs double-point precision, both using its nominal frequency at 600MHz. Kalray offers a complete suite to develop software for this platforms and supports open programing languages as well. However, the reduced memory size represents a challenge for developers during the software design phases.

The MPPA has been the chosen platform. The main reasons have been its energy efficiency and its dense capacity to handle concurrent threads among the system. This platform is going to be full covered in chapter 3 considering the scope of this project.
<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Platform</th>
<th>Cores/PE</th>
<th>Core Frequency</th>
<th>Power</th>
<th>Single Precision</th>
<th>Double Precision</th>
<th>Memory</th>
<th>Supported Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA</td>
<td>Tesla K40</td>
<td>2880</td>
<td>745 MHz</td>
<td>235 W</td>
<td>4.29 TF</td>
<td>1.43 TF</td>
<td>12 GB @ 288GB/s</td>
<td>CUDA C/C++ OpenCL</td>
</tr>
<tr>
<td>AMD</td>
<td>FirePro S9300</td>
<td>8192</td>
<td>850 MHz</td>
<td>300 W</td>
<td>13.9 TF</td>
<td>0.87 TF</td>
<td>8 GB @ 1024 GB/s</td>
<td>C/C++ OpenGL OpenCL</td>
</tr>
<tr>
<td>AMD</td>
<td>FirePro W9100</td>
<td>2816</td>
<td>930 MHz</td>
<td>275 W</td>
<td>5.24 TF</td>
<td>2.62 TF</td>
<td>GDDR5 16GB – 32Gb @ 320 GB/s</td>
<td>C/C++ OpenGL OpenCL</td>
</tr>
<tr>
<td>INTEL</td>
<td>Xeon Phi</td>
<td>61 (244 logical threads)</td>
<td>1300 MHz</td>
<td>300 W</td>
<td>N/A</td>
<td>1.208 TF</td>
<td>16 GB @ 352GB/s</td>
<td>OpenCL OpenMP most programming languages</td>
</tr>
<tr>
<td>CAVIUM</td>
<td>Thunder X</td>
<td>48</td>
<td>2500 MHz</td>
<td>80 W</td>
<td>N/A</td>
<td>N/A</td>
<td>1TB @ 2400MHZ (using dual sockets)</td>
<td>most programming languages</td>
</tr>
<tr>
<td>KALRAY</td>
<td>MPPA</td>
<td>256 (16 clusters of 16 cores each)</td>
<td>0.6 MHz (0.8 using overdrive)</td>
<td>11 W 16W</td>
<td>0.423 TF 0.634 TF</td>
<td>0.315 TF 0.420 TF</td>
<td>40MB on-chip memory + 5MB caches (available 2MB per cluster @ 102GB/s)</td>
<td>C/C++ OpenMP OpenCL Fortran</td>
</tr>
</tbody>
</table>
2.4. Parallel programing

Once the platform has been selected, it is important to define the programming method to exploit the maximum performance available. In this case the MPPA supports C/C++ using Posix threads, OpenMP and OpenCL. All of them are based in C language. The idea is to apply different abstraction layers to allow developers to program the platform in different ways. This section introduces this three programming methods, which are used in the current research work.

2.4.1. Posix threads

POSIX (Portable Operating System Interface) Threads, commonly referred as Pthread, is an execution model that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time [21]. Each flow of work is referred as a thread. Creation and control over these flows is achieved with calls to the POSIX Threads application programming interface (API). This method is consider as a “manual” programming scheme because developer must create, synchronize and destroy the required threads. In the MPPA, each of those threads are going to be executed in one of the 256 cores independently. Thus, the challenge consist in wisely split the tasks and data, do the processing and finally synchronize the output of all threads to get the desired result.

Essentially, Pthread works using the fork and join concept, shown in Figure 18. This idea means that when there is an available parallel section of the code, the developer shall explicitly create the desired number of threads, perform the operation required over that data, and join the acquired results on every thread.

Figure 18 Fork and join concept

Figure 19 illustrates the available methods to implement threads using the threads API. As can be seen, it is mandatory that a master thread invokes the thread creation, exit and join. Worker threads are the explicit computations a master thread is going to perform.

The process to create and destroy threads is not complex. However, on a platform with 256 cores, skills to identify data independence (made by the developer) and thread synchronization (with the help of the API) are required in order to combine the results. To achieve this goal, the API offers several methods like mutexes and semaphores to prevent data inconsistencies due to operations by multiple threads upon the same memory area performed at the same time or to prevent race conditions\(^\text{12}\), where an explicit operation order upon the memory is expected.

In overall, to work with the MPPA Pthread, a developer must consider, data dependency, fork processing, and synchronization and join processes. Note that all these steps have to be explicitly implemented in that order.

### 2.4.2. OpenMP

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming. It is not a new computer language. Rather, OpenMP works in conjunction with either C, C++ or FORTRAN [22]. OpenMP consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. It is managed by the nonprofit technology consortium OpenMP Architecture Review Board (or OpenMP ARB), jointly defined by a group of major computer hardware and software vendors, including AMD, IBM, Intel, Cray, HP, Fujitsu,

---

\(^{11}\) Source: POSIX Threads Programming, [https://computing.llnl.gov/tutorials/pthreads/](https://computing.llnl.gov/tutorials/pthreads/)

\(^{12}\) Race condition: is an event when two or more different threads try to update the value of a specific variable located in shared memory space.
NVidia, NEC, Red Hat, Texas Instruments, Oracle Corporation, and many other, what makes OpenMP a well-known developing scheme. It is worth noting that OpenMP is widely supported among several platforms.

Among the stronger points of OpenMP, it can be highlighted that OpenMP employs a portable, scalable model that gives programmers a simple and flexible interface to develop parallel applications for platforms ranging from a standard desktop computer to a supercomputer.

The following sections briefly describes the usability and functionality of the OpenMP API to understand how OpenMP constructors and clauses can be applied.

2.4.2.1. **Memory model**

OpenMP was designed considering a shared memory programming model, which means that all threads being executed on the cores have visibility to the same memory space. The MPPA has its own shared memory among 16-cores clusters, resembling to the non-uniform memory access (NUMA) shown in Figure 20. This scheme allows threads within a cluster to have full visibility of shared memory and the capacity to update shared variables.

![Figure 20 Non-uniform memory access scheme](image)

2.4.2.2. **Handling parallelism**

OpenMP parallelism is achieved mainly with concurrent threads. Typically, the number of threads equals the number of available cores in a platform. The procedure to split the work from a serial phase to a parallel one is based on the fork and join model previously described, where a master thread invokes the creation of new threads. Unlike Pthread, OpenMP gives developers a new abstraction layer, where API directives control the creation, join and destruction of threads on the platform cores. The OpenMP API considers the master thread as part of the fork process.
Although OpenMP handles the behavior of threads, it does not extract automatically the inherent parallelism. Developers must explicitly define the parallel zones of the code by means of compiler directives.

2.4.2.3. **API components**

OpenMP is composed by mainly three elements, Compiler directives, runtime library routines and environmental variables (see Figure 21).

![Figure 21 OpenMP API components [22]](image)

However in order to work with OpenMP is not necessary to handle all of those components in every implementation. Developers shall decide how to use them.

**Compiler directives** are statements that appear as comments in source code and are ignored by compilers unless otherwise is told. The common syntax is the following:

\[
\text{Sentinel} \quad \text{directive name} \quad \text{clauses}
\]

Where *sentinel* refers to the type of compiler directive to be translated by the compiling step (**#pragma omp**), *directive name* is the OpenMP statement, and *clauses* are the options and constrains to consider while the OpenMP instruction is going to be executed. The most common format of an OpenMP compiler directive using C/C++ language is shown next:

\[
\text{#pragma omp} \quad \text{directive-name [clause[ ,] clause]...} \text{ new-line}
\]

In addition to directives, OpenMP API is also composed by **runtime library routines** and **environmental variables**. These are typically used to examine and modify execution parameters such as the degree of parallelism in different areas of the program.

The API itself is not dependent to a particular machine or OS. OpenMP compilers are wide spread on major Linux versions as well as Windows. This feature gives a certain degree of portability of the program from one system to other by simply recompiling the
code. C and C++ implementations provide a standard include file, named as `omp.h`, which provides all the required definitions and library functions.

The previous compiler directives, runtime library routines and environmental variables are a set of elements that extend the capacity of languages such as Fortran, C and C++ to express shared memory parallelism. From a certain point of view, within the OpenMP language extensions there are four categories: control structures to express parallelism, data environment constructs for communication between threads, synchronization constructs to coordinate the execution of multiple threads and runtime functions to get and define some native characteristics of the API. A summary of these extensions can be seen in Figure 22.

![OpenMP Language extensions](image)

Figure 22 OpenMP language extensions

The following sections of this research will cover some examples of the languages extensions of OpenMP. A full description of all constructors, clauses and directives is out of the scope of this document and can be found in the full description API of the Architecture Review Board [23]. Besides, some examples and detailed explanations can be found in [22].

2.4.2.4. **Parallel control structures**

Parallel control structures are constructs that alter the control flow in a program. Mainly this constructors define the fork/join sections of the code, like creating new threads or sending specific statements to a set of threads. This research is focused mainly in two control structures the parallel, parallel for and parallel sections.

The parallel construct allow to define the parallel section of a code, creating a team of threads. The code will execute the same instruction or set of instructions among all threads.
#include <omp.h>
#define CORES_COUNT 16
int main()
{
    printf(“The output:\n”);
    /* define multi-thread section */
    #pragma omp parallel num_threads(CORES_COUNT)
    {
        printf(“Hello World\n”);
    }
    /* Resume Serial section*/
    printf(“Done\n”);
}

Figure 23 OpenMP parallel control structure

In Figure 23 an example of the usage of the OpenMP parallel construct is shown. As it was explained before, the C language has a standardized header for OpenMP which must be included in order to let the compiler understand all the language extensions of the API. The parallel section is defined by the #pragma which will create 16 threads that shall execute all the instructions contained between the brackets ({ }). In this example, all threads are going to print “Hello World”. The clause num_threads is used to set the amount of threads that the parallel construct is going to create. Unlike pthreads, the creation and destruction of threads is almost transparent to developers. In fact, it is the API language extension which is in charge of handling this operation.

The parallel for construct divides the loop iterations between the spawned threads. This constructor must always be followed by a “for” loop.

#include <omp.h>
#define N 1000
main(int argc, char *argv[]) {
    int i;
    float a[N], b[N], c[N];

    /*initialize*/
    for (i=0; i < N; i++)
    {
        a[i] = i * 1.0;
        b[i] = i * 1.0;
    }

    /* start of parallel region */
    #pragma omp parallel for
    for (i=0; i < N; i++){ // for loop of the parallel for
        c[i] = a[i] + b[i];
    }
    /* end of parallel region */
}

Figure 24 OpenMP parallel for constructor
In the example exposed in the Figure 24, using the \texttt{#pragma omp parallel for} allows creating a group of threads and share the work among them. In other words, each thread gets a different section of the loop, and they execute their own sections in parallel. With this kind of implementation typically a speedup of the overall execution time is reached. However, there might be some other issues such as race conditions due to synchronization among the team of threads. To solve these kind of problems, the OpenMP API includes clauses and synchronization methods that help developers to avoid them. These methods shall be covered in the following sections.

The last parallel control constructor is \textit{section}. The function of this constructor is to specify areas of the code that must run sequentially within a wider parallel section area. Figure 25 shows an example of sections. The code indicates that any of the sections may run in parallel, however Work2 and Work3 must be run in sequence. This functions run only one time.

```
#pragma omp parallel sections
{
    /* Executes sequential*/
    #pragma omp section
    {
        /* Executes in thread 1 */
        Work1();
    }
    #pragma omp section
    {
        /* Executes in thread 2 */
        Work2();
        Work3();
    }
    #pragma omp section
    {
        /* Executes in thread 3 */
        Work4();
    }
    /* ... */
}
```

Figure 25 OpenMP Parallel sections

Both \texttt{parallel for} and \texttt{parallel sections} must be chosen by developers depending on the application to parallelize. Both of them aim different types of parallelism. Figure 26 shows the behavior of a thread team for a \texttt{parallel for} and a \texttt{parallel section} constructors. Mainly \textit{parallel section constructors} are used to execute the same or different tasks among all threads. On the other hand, \texttt{parallel for constructors} distribute a specific task among all available threads.
2.4.2.5. **Communication and data environment**

OpenMP programs always start with a single thread (master thread) which is associated to the data environment. Typically it is named as a *master thread* and normally contains the *address space* of all variables specified in the program. This includes global variables, static variables, (i.e., variables allocated on the stack), as well as dynamic variables (i.e., allocated on the heap).

A master thread exists for the duration of the whole program. It is responsible of creating threads (when an OpenMP parallel constructor is encountered) as well as threads' data environment. Each thread is created with its own stack. As a result, threads may invoke subroutines and handle threads local variables without interfering with other threads’ stack. However, in order to perform a correct parallel work sharing among threads, developers must define the behavior of the variables among all threads. To achieve this goal, OpenMP defines clauses which must be declared in every control constructor. The options are: *shared*, *private* and *reduction*. Furthermore, the determination of which variables should be shared or private among the threads depends on the data dependence analysis made on the program design phase.

```c
#pragma omp parallel num_threads(16) shared(var1, var2, var3) private(iter,i)
```

*Figure 27 OpenMP shared and private variable*
The *shared* clause specifies that named variables will be shared by all threads in the team for the duration of the parallel construct. In Figure 27, var1, var2 and var3, may be modified by any of the spawned threads, and the updated value will be accessible by all threads. On the other hand, the *private* clause allocates in each thread of the team a private copy of the variables defined in the master thread. The value of the private variable is only visible to its thread. Other threads cannot access to it.

*Private* and *shared* clauses give developers the possibility to handle explicitly data parallelization within a parallel section of a code. However, this flexibility may include variable update problems while updating shared variables (e.g., race conditions) which could make the results of a parallel operation not to be consistent. To solve this issue, two methods can be used, *atomic* variable update or *critical* section [18]. In Figure 28, there is a “for loop” with 16 threads to compute an accumulative add into the variable x. The expected value of ‘x’ at the end of the execution should be 16000. To achieve this, the usage of the clause *critical* is mandatory when the add process is being executed. Otherwise threads shall update arbitrarily the value of the x variable. OpenMP offers this clause *critical* to allow only one thread at a time (emulating a lock on the variable) updating a variable. This is a way to synchronize the data environment shared among threads.

```c
#include <omp.h>
#define THREADS 16
#define TURNS 1000
#pragma omp parallel for private(i) shared(x)
for (i=0; i<THREADS; i++)
{
    int j;
    for (j=0; j<TURNS; j++)
    {
        #pragma omp critical
        x=x+1;
    }
}
```

Figure 28 OpenMP critical example

The critical section stops the execution of other threads to update shared values. As a consequence, the code performance is severely reduced. Due to this constrain OpenMP offers an *atomic update* of variables. In Figure 29 the critical clause is replaced by an atomic one. While a critical section locks a block of code, incurring in a significant overhead every time a thread enters and exits the critical section, an atomic update operates on a specific variable, implying a much lower overhead. Where available,
atomic updates take advantage of a hardware atomic operation without affecting the overall behavior of threads and solving race conditions. Although both clauses provide the same results, the *atomic update* gives developers a higher performance in terms of speed due to its behavior [18].

```c
#include <omp.h>
#define THREADS 16
#define TURNS 1000

#pragma omp parallel for private (i) shared (x)
for(i=0; i<THREADS; i++)
{
    int j;
    for(j=0; j<TURNS; j++)
    {
        #pragma omp atomic update
        x=x+1;
    }
}
```

Figure 29 OpenMP atomic example

2.4.2.6. Synchronization

As it was mentioned in the last section, multiple OpenMP threads communicate with each other to read or write shared variables. Without synchronization it is impossible to achieve coordination between threads. Synchronization is needed to manage the attempt to simultaneously modify or even read a shared variable while preserving the data consistency in the process.

In addition to *atomic update* and *critical sections*, OpenMP provides other constructs to express common synchronization patterns. The most conventional example is the constructor *barrier*, which allows developers to handle event synchronizations. The constructor *barrier* defines a temporal point where all threads in a thread team should synchronize to resume the program execution. As an example, in Figure 30, a barrier precludes section 2 to start if any of the threads spawn in section 1 have not finished its work.
In the example shown in the Figure 31, the constructor \textit{barrier} allows developers to place a synchronization event among the execution of all threads. Thus, after every loop of the \texttt{for} instruction all threads shall execute the function \texttt{worker()} and synchronize \texttt{then} with a \texttt{barrier}.

```c
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>

#define NUM_THREADS 16

/* the function called for each thread */
void worker() {
    /* get our thread id */
    int id = omp_get_thread_num();

    /* start to work */
    printf("Thread %d starting!\n", id);

    /* simulate the threads taking slightly different amounts of time
     * by sleeping for our thread id seconds */
    sleep(id);
    printf("Thread %d is done its work!\n", id);

    /* a barrier */
    #pragma omp barrier

    printf("Thread %d is past the barrier!\n", id);
}

int main () {
    /* have all the threads run worker */
    #pragma omp parallel for num_threads(NUM_THREADS)
    for(int i=0; i<16*5; i++)
        worker();

    return 0;
}
```

Figure 31 OpenMP barrier example
2.4.3. OpenCL

Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. Actually, the portability of the code is achieved because OpenCL is not limited to any specific hardware vendor, operating system or memory architecture.

This framework was created by Apple Inc.\textsuperscript{13} in August 2008 (OpenCL 1.0) and after its creation was submitted to the Khronos Group\textsuperscript{14} which is conformed by several manufactures of CPU and GPU. The first stable standard release (OpenCL 1.2) was published in November 2011. The versions 2.0 and 2.1 were released in November 2013 and November 2015 respectively and it is expected a 2.2 version at the end of 2016. This document will cover the version 1.2 only, mainly because this is the available version on the selected accelerator (MPPA). However, all the explanation of these specifications can be found in [24] and [25] if further information is required.

OpenCL 1.2 defines a programming language based on ISO C99 for programming the devices and application programming interfaces (APIs) to control the platform and execute flows on compute devices. To understand the usage of OpenCL this section is going to describe the hardware elements of the architecture, the memory model, the program components and profiling. These sections introduce a summary of the standard defined by the Khronos Group [26] and the book defined in [27].

2.4.3.1. Hardware Elements

The elements defined by the OpenCL platform model are depicted in Figure 32. It is possible to observe four elements: host, compute device(s), compute unit and processing elements.

\textsuperscript{13} https://developer.apple.com/opencl/
\textsuperscript{14} https://www.khronos.org/
Figure 32 OpenCL platform model [26]

1. **Host**: it is a CPU based device which is responsible of (1) manage the compute devices, (2) arrange and (3) submit kernels to execute onto the compute devices.

2. **Compute Device (CD)**: it is an accelerator (e.g. GPU, many-core platform) dedicated to a parallel computation.

3. **Compute Unit (CU)**: also referred to as work-group. CUs are components of the compute device that group several processing elements. CUs can be associated to clusters in Kalray MPPA.

4. **Processing Element (PE)**: also referred to as work-items. PEs are the systems where raw execution of a kernel is performed. PEs are associated to cores of in Kalray MPPA.

2.4.3.2. **Memory model**

Due to portability requirements, OpenCL defines its own memory model (Figure 32). The OpenCL memory model consists of four types of memory: host memory, global/constant memory, local memory and private memory.
Host memory is used by the host for its own computations.

Global memory is the main memory of the accelerator architecture. All work-items have access to the global memory, but it is the slowest one within the accelerator. Global memory can be dynamically allocated by the host program and can be read and written by the Host and the Compute Device.

Constant memory is a specific case of global memory which uses the same memory space but can only be read by the Compute Device.

Local memory is a shared memory within work-items grouped in a work-group. Work-items employ local memory when synchronous access is required. Local memories are faster than global ones, but are smaller and cannot be accessed by the host.

Private memories can be accessed only by each individual work item. Typically, private memories consists of registers. Thus, they are the fastest memories available on the architecture. At the same time, they are the smallest ones.

Developers should select where to place data in order to achieve a better performance. To do so, developers should have in mind that global memory is the biggest and easier to handle but the slowest one. On the other hand, local and private memory is faster than global memory. However, they are much harder to manage since their usage requires local synchronization.
2.4.3.3. **Program Components**

Every OpenCL program contains two major components: host code and at least one kernel, as shown in Figure 34.

![OpenCL Application diagram](image)

**Figure 34 OpenCL program components**

Each component has its specific function. Host code is responsible to manage the device, generate the inputs and receive the output data. Kernel code function is to perform the parallel computation.

![OpenCL phases diagram](image)

**Figure 35 OpenCL program phases**

**Host program**

The host program is written in C/C++ and executes the following actions (Figure 35):

- a) To initialize the system
- b) To create data and arguments
- c) To send data for execution
- d) To receive data
- e) To finalize and release resources

The *initialization* consists to prepare the device in order to be ready to compute. This is achieved by creating a *context* where all OpenCL applications run. There can be several groups of contexts depending on the number of compute devices available. In this work we are going to use only one context since only one compute device (MPPA) is available.

Figure 36 shows a fragment of the code of the host to initialize the device. First of all, it is important to load the file which contains the kernel (i.e. this code must be previously written in a separated file). Then, using the commands *clGetPlatformIDs* and
clGetDeviceIDs is possible to get the information needed to create a context involving the desired device. Next a command queue is created (clCreateCommandQueue) for the specific context. Finally the program is created using the command clCreateProgramWithSource and compiled with clBuildProgram to be ready in further phases.

```c
/*LOADING THE KERNEL SOURCE*/
char* kernelSource=(char*)malloc(MAX_SOURCE_SIZE);
readKernel("src/kernel/kernel_PCA.cl", MAX_SOURCE_SIZE, kernelSource);
/*LOADING THE KERNEL SOURCE*/

/* creating and compiling the OpenCL program */
/**--PLATFORM INFORMATION--*/
clGetPlatformIDs(1, &mppa_platform, NULL);
clGetDeviceIDs(mppa_platform, CL_DEVICE_TYPE_ACCELERATOR, 1, &device_id, NULL);
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
/**--PLATFORM INFORMATION --*/

/**-QUEUE-*/
queue = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE, &err);
/**-QUEUE-*/

/**BUILD THE PROGRAM*/
program = clCreateProgramWithSource(context, 1, (const char **) kernelSource, NULL, &err);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
/**BUILD THE PROGRAM*/
```

Figure 36 Creating and compiling the OpenCL program

Once the initialization is done, the second phase can start. First, it is required to define which kernel is going to receive and send data from the host to the device and vice versa. To define a kernel, the command clCreateKernel is needed and the name of the desired kernel (kernel_app) is set as a parameter to get the application from the compiled program. Then, it is possible to start moving data to the kernel. Figure 37 depicts the command clCreateBuffer to perform this operation. The key points to consider are (1) a buffer must belong to a context previously created and (2) the value of size must be in bytes and must correspond to the size of the variable/array defined by host_ptr (e.g., float array[10] corresponds a size of 40 since every float requires 4 bytes to allocate them in the buffer). The variable flags is used to determine the operation mode of the buffer, i.e. read, write, read and write and other. This variable can be used to determine different methods to send data like copying the whole variable to the device.
or just reference it to the host memory. All these flags and behaviors can be found in the OpenCL 1.2 standard in [26].

```
Cl_kernel kernel = clCreateKernel(program, "kernel_app", &err);

cl_mem clCreateBuffer(cl_context context, cl_mem_flags flags,
                    size_t size, void *host_ptr, cl_int *errcode_ret)

cl_int clEnqueueWriteBuffer (cl_command_queue command_queue, 
                             cl_mem buffer, cl_bool blocking_write, size_t offset, 
                             size_t cb, const void *ptr, cl_uint num_events_in_wait_list, 
                             const cl_event *event_wait_list, cl_event *event)

cl_int clSetKernelArg(cl_kernel kernel, cl_uint arg_index,
                      size_t arg_size, const void *arg_value)
```

Figure 37 Create data and arguments

The `clCreateBuffer` only allocates the required memory space for the buffer. Thus, to write actual data to the buffer it is necessary the command `clEnqueueWriteBuffer`, in which the following arguments must be considered:

- A command queue shall be created. This queue shall be associated to the data.
- The `blocking_write` parameter, allows to block or not the host when it is writing data to the buffer. It is enable by the flag CL_TRUE and disable by CL_FALSE.
- The value of `cb` defines the amount of bytes required to write, this value must be the same as the one used when creating the buffers.
- The value `*ptr` is the actual variable/array to write into the buffer.
- The `event` variable is not mandatory but it is needed for profiling purposes.

To complete this phase a mapping between kernel arguments and buffers must need be done (next section covers the concept of kernel arguments). The command `clSetKernelArg` allows to assign buffers to kernel arguments. The key points of this command are (1) a kernel should have been created in advance, (2) the indexes of the arguments (`arg_index`) should correspond to positions of the arguments as defined in the kernel and finally, (3) the arguments (`arg_value`) can be defined as buffers or common variables (i.e., arguments can be int, float or any supported data type).

The Send for Execution phase is defined by the command `clEnqueueNDRangeKernel` and its prototype is shown in Figure 38. The purpose of this function is to queue the kernels for the available work-units to perform a computation in the device.
There are a few key points to consider in order to execute the kernel correctly:

- Before the \textit{NDRangeKernel} can be invoked, a \textit{command queue} and a kernel have to be previously loaded into the context.
- \textit{work_dim} refers to work dimension. OpenCL is capable to execute a group of kernels up to dimension three. This means that OpenCL is able to control the execution of threads in different spaces. This feature is very useful when handling matrixes. For instance, a 2 dimensional \textit{NDRangeKernel} allows threads to independently perform computations over row and columns.
- The variable \textit{local\_work\_size} is hardware depended value. It refers to the number of work-units available in a specific work-group.
- The variable \textit{global\_work\_size} defines the number of times that a kernel shall be invoked. This variable must be a multiple of the local work size.
- The \textit{event variable} has a profiling purpose. \textit{Event} grabs specific events of the OpenCL behavior. This argument is not compulsory.
- The remaining values can be set to NULL because they are not mandatory. However, for a deeper coverage of the topic, the OpenCL 1.2 specification can be reviewed \cite{25}.

At this point kernels have two queues to process. The former, a command queue which loads the instructions to perform. The latter, a data buffer queue to compute in parallel instructions using the available work-units.

The last phase of an OpenCL program consists of two stages: finalization and releasing. The finalization stage starts when all computations have been done in the work-units and the host finishes the command queue and reads the results from a buffer.

Figure 39 shows the two commands required for this step, \textit{clFinish}, which indicates finishing the command queue and \textit{clEnqueueReadBuffer} which allows to retrieve data from a buffer to store into a variable.
Chapter 2 – State of the art

/*Finish the command queue*/
c1Finish(queue);
/*Finish the command queue*/
/*Read the results buffer*/
c1_int clEnqueueReadBuffer ( cl_command_queue command_queue,
cl_mem buffer,
c1_bool blocking_read,
s1ze_t offset,
s1ze_t cb,
void *ptr,
c1_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)
/*Finish the command queue*/

Figure 39 Receiving data in OpenCL

The arguments of clEnqueueReadBuffer are similar to those of clEnqueueWriteBuffer. The main difference is that, in this case, the variable defined by ptr is where the buffer data will be stored.

Release is the last step of the final phase. It consists of freeing all used elements in the OpenCL program execution. Figure 40 is an example of this step. All buffers, kernels, program, command queues and contexts should be released in order to free the memory from the device.

clReleaseMemObject (buffer);
c1ReleaseKernel (kernel);
c1ReleaseProgram (program);
c1ReleaseCommandQueue (queue);
c1ReleaseContext (context);

Figure 40 Releasing the objects in OpenCL

OpenCL kernel program

The code executed by the work-units is known as kernel. Kernels are similar to standard C functions and can be thought of as instances of a parallel operation. Figure 42 shows an example of a kernel. Kernels begin with the __kernel indicator and behave like C functions.

The main differences between a standard C function and an OpenCL Kernel are the following:

- Arguments: besides the common argument data types (e.g., int, float, double and others) OpenCL defines the memory area where these variables are stored.
Referring to the memory model explained in section 2.4.3.2, there are three types of memory which can be declared within a kernel: global, constant and local. To define the memory area a double underscore (__) and the memory type should be used before the variable type.

- **Variables**: all local variables declared inside (the “brackets”) of a kernel function shall be stored into private memory.

- **Identifiers**: in order to get control of the execution flow, some methods are provided by the OpenCL API to handle the ids of work-items and work-groups. One of these methods is the command `get_global_id(uint deviceID)` which indicates the global id of the work-item which processes a kernel based on the total number of work-items specified to execute the kernel. The command `get_local_id(uint deviceID)` returns the id of the local processing element (acting as a work-item) within a work-group. The command `get_group_id(uint deviceID)` returns the work-group identifier. These values are really important considering that OpenCL executes its kernel as a loop. The value of N, the number of times the loop is executed, is defined by a `global_size` variable specified in the host. Thus, these methods allows writing data in the desired position of the array to return the computation.

- **Synchronization**: not all kernels require synchronization, however, some applications may require synchronization in order to perform the computations. OpenCL defines a specific method `barrier(CLK_LOCAL_MEM_FENCE)`, which allows to sync work-items within a work-group (see Figure 41). However, OpenCL does not support synchronization among work groups.

Figure 41 Synchronization in OpenCL [27]

Figure 42 shows a kernel which performs a sum of two vectors (arrays a and b) with a scale factor. Each kernel independently computes a position of the array, In this
specific example, kernels ask the global work-item id \((\text{get\_global\_id})\) to perform the computation related to a specific index of arrays \(a\) and \(b\) and writes the result to array \(c\). The scale factor does not change like the operands of the sum. Therefore it is better to use a local variable in order to increase memory access speed.

```
__kernel void vecAdd(__constant double *a,
                    __constant double *b,
                    __global double *c,
                    __local int *d,
                    const unsigned int n)
{
    unsigned int id = get_global_id(0);
    if (id >= n)
        return;
    d = a[0];
    c[id] = d*(a[id] + b[id]);
}
```

Figure 42 Kernel example

Given that kernel functions can be very complex, it is very convenient to design them with their specific function in mind and avoid making them larger than needed. Further examples of OpenCL kernels can be found in [27].

2.4.3.4. Profiling

OpenCL profiling allows developers to obtain timing information from the commands of the API. Since the main objective of this project is to measure execution times, OpenCL profiling is mandatory to assess the performance of program execution. There is the possibility to measure the commands \(\text{clWriteEnqueueBuffer}\), \(\text{clReadEnqueueBuffer}\) and \(\text{clEnqueueNDRangeKernel}\) which represents the time required to send data from the host to a device, the time to receive data from a device to the host and the processing time, respectively. To do so, when any of the previous commands is invoked, an event to check should be included in the event field. This event is recorded by the system itself and developers can process it later. Figure 43 shows the procedure to measure the time of a specific event.

```
double time_start, time_end, time;
clWaitForEvents(1, &event);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(double), &time_start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(double), &time_end, NULL);
```

Figure 43 Profiling in OpenCL
The command `clWaitForEvents` allows detecting the event of the command. Then, the command `clGetEventProfilingInfo` gets the desired event. In this case the flags `CL_PROFILING_COMMAND_START` and `CL_PROFILING_COMMAND_END` allows to receive the time in nanoseconds. Although the profiling action introduce some overhead, it is the most accurate available method to measure time in OpenCL. Therefore, this is the method to be used to measure the total time of applications in this work.
3 Materials and methods
The purpose of this chapter is to describe the materials and methods used to achieve the objectives of this project. As explained on chapter 1, the main goal of this project is to evaluate other methodologies to speed up the implementation of part of the algorithms used in the HELICoID project. The idea is to process hyperspectral images using a many-core platform in such a way that real-time constrains are met.

In order to detail the materials and methods used in this research, this chapter is organized as follows: first, it explains in detail the architecture of the chosen parallel platform, the Kalray MPPA with its latest available version (Bostan). Secondly, it introduces the parallel programming methodologies available on the platform, and finally it analyzes the sequential implementation of the source code to process hyperspectral images described in [1] and [2].

### 3.1. Kalray MPPA-256

#### 3.1.1. Technical description

As it was introduced in Chapter 2, the Multipurpose Processor Array (MPPA) is a manycore platform created by Kalray. There are two versions of platforms available in the actual market. The first one, codenamed as Andey and the second one named as Bostan. A description of Andey, its architecture and components can be found in [29] and [30]. Bostan is the second MPPA generation after Andey. It preserves a similar hardware structure. In addition, Andey adds features such as a 64-bit addressing capability to the cores and an energy efficiency improvement by widening the CPU data path. Whereas Andey PE’s frequency is limited to 400MHz, Bostan can clock as fast as 800MHz (using overclocking). Bostan nominal frequency is 600MHz [31]. This document will discuss about the Bostan MPPA generation.

Briefly speaking, the Bostan MPPA is a 28 nm CMOS chip, which integrates a 4x4 array of 16-core compute clusters (CC), giving the total amount of 256 cores clocked at 600MHz each. Along with the clusters, the chip is integrated with 4 quad-cores I/O subsystems located at the north, south, east and west ends of the chip, all connected with a NoC. Figure 44 shows all the components of the architecture.
The compute clusters are one of the main elements of the MPPA architecture. As it is described in [1] each cluster is composed of the following elements:

- **Resource manager (RM).** It is an extra core reserved for system use in each core. RM responsibility is to manage the resources on behalf of the entire cluster. Effectively, the RM maps and schedules threads on different compute cores and organize the communication among its cluster and other clusters. Besides, RM is in charge of the communication of the cluster with the main off-chip memory (I/O DDR external memory). In order to fulfill all the tasks, RM runs under NodeOS, a specific operating system developed by Kalray.

- **Shared memory.** Each cluster has a shared memory of 2 MB that is structured in 16 parallel banks of 128 KB, providing a total bandwidth of 38.4 GB/s. This memory allocates both instruction cache (IC) and data cache (DC). Nevertheless, if the shared memory is not enough for a certain application, each of the four I/O subsystems embeds a SMP quad-core with 4 I-cache each and shared D-cache on-chip memory. Furthermore, the quad-core includes a DDR controller for accessing up to 64 GB of external DDR3-1600 memory.

- **Direct Memory Access (DMA).** It is the responsible for transferring data from cluster shared memory and I/O shared memory to the NoC.

- **Processing Elements (PE).** There are a total of 16 identical cores which are independent and are intended to perform general-purpose
computations. Each PE implements a 5-way VLIW architecture with two arithmetic-logic units (ALU₀ and ALU₁), a Multiply-Accumulate Unit (MAU), a load/store unit (LSU) and a Branch and control unit (BCU). The MAU can also work as a Floating-point unit (FPU). Both the MAU and the LSU support a subset of the ALU instruction set, the so-called tiny ALU. The microarchitecture of a PE is depicted in Figure 45.

![VLIW implementation on MPPA](image)

**Figure 45 VLIW implementation on MPPA [30]**

Besides the clusters, the other major elements of the MPPA are the I/O subsystems (referred to as IOs as well). IOs major responsibility is to control the execution of programs in the compute clusters and to communicate them with the external world (e.g. external memory or a host device). Each IO is formed by a quad-core processor (each core clocked at 600MHz) and every core runs under a version of RTEMS\(^{15}\), which is a full featured Real Time OS written for embedded platforms. It is worth noting that RTEMS porting to MPPA is a specific customization made by Kalray to better suit to its architecture. The implementation supports all the standardized instructions and most of the APIs such as POSIX [32].

\(^{15}\)Real-Time Executive for Multiprocessor Systems, [https://www.rtems.org/](https://www.rtems.org/).
As a summary, Figure 46 shows the architecture of the MPPA chip as well as the components of one of 16 compute clusters. The key elements of the MPPA are the I/O subsystems and the compute clusters, which runs RTEMS and NodeOS OS, respectively. Both elements implement POSIX-compatible APIs. MPPA-256 applications are then designed as POSIX-like processes deployed on the I/O subsystems and the compute clusters. These processes communicate together through the NoCs using network operations similar to reads and writes on UNIX sockets (i.e. using description files). Finally, a pthread-like interface allows running up to 16 threads in parallel on each compute cluster thanks to their many-core architecture.

3.1.2. Programming the MPPA-256

As we explained in chapter 2, Kalray provides a complete software suite to develop applications for the MPPA-256. This work has been made using the MPPA Developer (see Figure 47) powered by a Bostan board with the SDK 2.2.2-RC1.
The available methods to program the MPPA are the following: (1) a cyclostatic dataflow (CSDF) language, known as ∑C, based in the C-language syntax, (2) a POSIX-level programming API using processes/threads explicitly spawned by the developer in the compute clusters, I/O processors and host, (3) an compiler directives of a version 2.5 OpenMP compiler [34] and, last but not least, (4) a subset of the OpenCL 1.2 specification [35].

Although ∑C is able to adapt the code to the target architecture (i.e. number of cores, NoC, memory), the comprehensive understanding of dataflow programming models that is required to build specifications is out of the scope of this project. Further information about ∑C can be found in [36] and [37]. Hence, this document basically covers POSIX, OpenMP and OpenCL, which are more related to thread mechanisms to exploit parallelism.

### 3.1.3. Posix and OpenMP on the MPPA

POSIX and OpenMP are both extensions of the C language. In this case, the I/O subsystem must spawn the sub-processes on the compute clusters. The way to do so is to pass the corresponding arguments using environmental variables. Once the instructions and data to process are available in the compute clusters, the parallel instructions can be used. As far as the POSIX and OpenMP implementations concern, there are three main elements a program consists of, all of them developed in the C language:

- **Host application**: It is a code that runs on a host system, a CentOS computer in the case of the developer workstation powered by an Intel Core i7-3820 CPU. The host system communicates to the MPPA system through a PCI express connection. Basically, the host application initializes the I/O subsystems and sends/receives the data to/from them.

- **I/O subsystem application**: the main objective of the I/O subsystem application is to provide the communication means to the clusters and host. This application is responsible of spawning the code to the compute clusters and setting the environmental variables to perform the desired computation.

---

16 _Dataflow programming:_ models a program as a directed graph of data flowing between operations, thus implementing dataflow principles and architecture.
• **Cluster executable:** This application implements the thread computations inside a compute cluster.

The details of a program execution are shown in Figure 48, where an example of a Kalray Host + MPPA project is described. First, (1) the host starts one of the available I/O subsystems on the MPPA (host spawns the code to the I/O core), then (2) the I/O initializes the compute clusters needed (up to 16 available in the MPPA), passing the program arguments and spawning the code in the Resource Manager of each cluster. Finally, (3) each cluster creates the threads needed using either Pthread (POSIX threads) or OpenMP directives (#pragmas).

![Figure 48 Application on Developer workstation](image)

To communicate the host and the I/O to the NoC, the management of special files within the classic POSIX IPX operations framework are needed. These communication primitives are known as connectors and can be opened in write only (WRONLY) or read only (RONLY) modes. The available connectors in the MPPA are: `sync`, `signal`, `portal`, `stream`, `rqueue` and `channel`. In this work, the focus is on the sync and portal primitives, because those are the connectors applied in the Posix/OpenMP applications developed.

---

17 Source *Kalray Inc.*
in this research work. If further information is needed about connectors a full description of them can be found in [1] and [29].

The **sync connector** consists in a 64-bit word in the Rx process that can be OR-ed by N TX processes. When the result of the OR equals -1, that is, an all ones 64-bit signed integer variable, the Rx process is notified so the function `mppa_read()` returns non-zero. The common syntax for its definition is `/mppa/sync/rx-tx_nodes:cnoc tag`, where `rx-tx_nodes` is a number which refers to the transmitter or receiver of the communication. This number is related to the I/O subsystem ID number as follows: 128 is allocated to the North I/O, 192 to the South I/O, 160 to the West I/O and 224 the east I/O (see Figure 49). Although the Bostan MPPA is designed to work with four I/O subsystems only two of them are currently available - the North and South. In this work, the north I/O will be used exclusively for developing purposes. The value `cnoc_tag` is an identifier. It can be set in a range from 1 to 255. What follows correspond to the syntax of a sync communication primitive which employs the North I/O.

```c
const char *root_sync = "/mppa/sync/128:1";
```

Figure 49 MPPA NoC topology

---

18 Source Kalray Inc.
To use sync, there is a list of structured steps to apply to have a correct communication flow. In Figure 50, the receive mode is detailed while in Figure 51, the transmission mode is outlined. For the sake of simplicity, only one sender is considered. As can be seen, the IO north is the data sender and the receiver is a cluster defined by a rank variable. It is important to highlight that the operation `mppa_read` is of the blocking type. In other words, `mppa_read` shall block the program flow until the `read` operation receives a match.

```c
const char *pathname = "/mppa/sync/128:2";
//Open the Sync connector for the Rx process
sync_fd = mppa_open (pathname, O_RDONLY);
//Initialize the Rx process match value.
mppa_ioctl (sync_fd, MPPA_RX_SET_MATCH, match)
//wait until the match value is reach. Then the value is written into the value parameter.
mppa_read (sync_fd, &value, sizeof(value))
```

Figure 50 Sync connector Rx mode

```c
//define number of available clusters
int CLUSTER_COUNT = 16;
//define the file descriptor location
const char *pathname = "/mppa/sync/128:2";
//auxiliary variable
float vector[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
int rank = 0;  //for multiplex
int slice = 1;

int ranks[2] = {0, 1};  // for multicast
//Open the sync for a Tx process.
sync_fd = mppa_open (pathname, O_WRONLY)
//multicast
mppa_ioctl (sync_fd, MPPA_TX_SET_RX_RANKS, ranks)
mppa_write (sync_fd, vector, sizeof(vector), 0)
//multiplex
for (rank = 0; rank < CLUSTER_COUNT; rank++) {
    mppa_ioctl (sync_fd, MPPA_TX_SET_RX_RANK, rank)
    mppa_write (sync_fd, vector + rank * slice, sizeof(float) * slice, 0)
}
```

Figure 51 Sync connector TX mode

As can be seen in Figure 51, there are two methods to write the values into the sync connector in the transmission mode, i.e., multicast and multiplex. When the multicast mode is selected, the MPPA_TX_SET_RX_RANKS flag indicates that the command `mppa_write` writes the same information to all the compute clusters defined into the `ranks` variable. In this case, this command will write the whole array, `vector`, to the compute clusters 0 and 1. On the other hand, the multiplex mode allows a developer, for instance, to send different slices of a variable to different clusters of the MPPA. In the
example of the previous figure, the MPPA_TX_SET_RX_RANK flag indicates that the
mppa_write command writes a specific part of a variable to only one compute cluster.
The particular compute cluster is defined by the rank value. Then, within the loop the
rank variable is increased to distribute the array, vector, to the 16 available clusters of
the MPPA. In this example, the slice variable equals ‘1’, meaning that each of the
compute clusters will receive an element of the array vector.

The portal connector targets a memory area on an Rx process. The statement
declaration is as follows /mppa/portal/rx_nodes:dnoc_tag, whose NoC node is specified
by the identifier rx_nodes in the pathname, and a number of TX processes not specified
in the pathname. The dnoc_tag parameter identifies the Rx buffer descriptor that
manages the reader memory area. Its range is from 0 to 255. One of the main
characteristic of portal connectors can be outlined as follows. Rx processes are only
aware of the data received when a notification count that unlocks the Rx process is
received. This notification arrives when the event provided to mppa_aio_read() is
triggered. Rx processes employ asynchronous I/O operations.

Like sync connectors, portal connectors supports multicast and multiplex
communication methods and need specific steps to perform the communication. Figure
52 shows the code to perform a transmission using portals with multicast mode. Similar
to the sync connector case, the mppa_ioctl command is introduced. This command
allows to set the nodes of the given ranks, where ranks are the positions of the nodes
selected as destinations in the ranks array. In this specific case, the transmission will
arrive to all compute clusters). Then, the write command to the connector is performed.

```c
//defining the file descriptor
const char *pathname = "/mppa/portal/[0..15]:2"; // 0 to 15 clusters

//auxiliary variable
int ranks[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
float vector[16]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};

//Open the Sync connector for the Rx process
int portal_fd = mppa_open(pathname, O_WRONLY);
status |= mppa_ioctl(portal_fd, MPPA_TX_SET_RX_RANKS, CLUSTER_COUNT, ranks);
status|=mppa_pwrite(portal_fd,vector,sizeof(vector), 0);
```

Figure 52 Portal TX mode

Figure 53 shows the procedure to set a multicast portal connection within a
compute cluster. As can be seen, after the creation of the file descriptor (pathname
variable) a trigger needs to be set up to know when the variable is able to be read. This is achieved using the `mppa_aiocb_t` data type and the corresponding `mppa_aiocb_set_trigger` function call. Then, the `mppa_aio_read` command is executed. It will only update the value of the variable `vector` when the `mppa_aio_wait` command receives a notification from the trigger. Like the sync connector, the `mppa_read` function is a blocking type function.

```c
vector = (float*)malloc(16*sizeof(float));
//open NoC special files
cnst char *pathname = "/mppa/portal/128:3";
int portal_fd = mppa_open(pathname, O_RDONLY);
// Complete initialization of portals for rx variable/array
mppa_aiocb_t portal_aiocb[1] = { MPPA_AIOCB_INITIALIZER(portal_fd, vector, 16*sizeof(float)});

mppa_aiocb_set_trigger(portal_aiocb, 1);
status |= mppa_aio_read(portal_aiocb);
// Wait for notification of remote writes to local
status |= mppa_aio_wait(portal_aiocb);
```

Figure 53 Portal Rx mode

Once all the elements that involve the execution of a program on the MPPA have been reviewed, the following lines explains how the different steps interact. (These steps are taken using the template Host + MPPA Example available on the developing suite of Kalray MPPA developer).

- First, the host reads the data stored in the HDD using a FILE type operation, `FILE *fopen(const char *filename, const char *mode)`. Then, the connection is established.
- Then, the MPPA is initialized with the multi-binary executable (program for the MPPA) using the functions:
  ```c
  load_ret=mppa_load(0, 0, 0, multibin_name);
  mppa_pid = mppa_spawn(load_ret, NULL, exec_name, NULL, 0);
  ```
- After that, the I/O subsystem has the control of the execution flow of the parallel region. Each time that IO wants to turn on a compute cluster (CC), it executes the function `mppa_open`. This function allows to create a connection between the CC and the IO using any of the connectors described previously.
- Once the connection is established, the IO sends the executable to the CC using the `mppa_spawn` function. Once this is step is completed the execution of the program of the cluster starts. In order to establish a
communication between the IO and the CC, the first task the CC performs is to open a communication using \textit{mppa\_open} function.

- After the IO have initialized and established a communication with all required CC, the whole program execution can continue and both can interchange data. The communication interface uses mainly two functions: \textit{mppa\_write} to send data and \textit{mppa\_read} to receive data. It is worth noting that the read function is a blocking one. Thus, its usage in the IO or CC side blocks the execution of the program until some data is received. The IO waits for all the child processes executed in the CC using the \textit{mppa\_waitpid()}.

- In the following step, CC can perform the execution of tasks using the PE available. Once all processing is finished, CC writes back the result obtained to the IO using a \textit{write} function.

- Finally, when the IO receives all the results from the CC, a notification to finish the \textit{wait} method will be received as well. Finally, the execution of the program resumes until its end.

The exemplification of the aforementioned flow is depicted in Figure 54.

![Diagram](image)

\textbf{Figure 54 Use of POSIX level operation to communicate IO with CC}^{19}

\textsuperscript{19}Source: Kalray Inc.
At last, to compile the source code that has been developed with OpenMP, the reader is referred to Annex I.

### 3.1.4. OpenCL on the MPPA

In the case of OpenCL, Kalray implements a limited version of the 1.2 OpenCL API [38]. Given that OpenCL objective is to give a developer the capability to program any device using the same programming methodology, Kalray has not included any hardware customization of the API. Thus, all commands defined in section 2.4.3 are the programming methods available in the MPPA. Although OpenCL is a general heterogeneous programming methodology, the configuration employed is hardware dependent. Kalray has defined the suggested configurations and established the limits that are described in Table 2.

<table>
<thead>
<tr>
<th>Device Information</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_DEVICE_GLOBAL_MEM_CACHE_SIZE</td>
<td>1,5MB</td>
</tr>
<tr>
<td>CL_DEVICE_GLOBAL_MEM_CACHE_TYPE</td>
<td>CL_READ_WRITE_CACHE</td>
</tr>
<tr>
<td>CL_DEVICE_GLOBAL_MEM_SIZE</td>
<td>1GB</td>
</tr>
<tr>
<td>CL_DEVICE_LOCAL_MEM_SIZE</td>
<td>128KB</td>
</tr>
<tr>
<td>CL_DEVICE_LOCAL_PVT_SIZE</td>
<td>no</td>
</tr>
<tr>
<td>CL_DEVICE_MAX_PARAMETER_SIZE</td>
<td>256</td>
</tr>
<tr>
<td>CL_DEVICE_MAX_WORK_GROUP_SIZE</td>
<td>16</td>
</tr>
<tr>
<td>CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS</td>
<td>3</td>
</tr>
<tr>
<td>CL_DEVICE_MAX_WORK_ITEM_SIZES</td>
<td>{16, 16, 16}</td>
</tr>
<tr>
<td>CL_DEVICE_TYPE</td>
<td>ACCELERATOR</td>
</tr>
</tbody>
</table>

Thus, to use OpenCL within the MPPA, the following considerations must be taken into account:

- Local size: it shall be always 16 because it is related to the number of hardware cores of a cluster.
- Global size: it shall be always a multiple of the local size.
- Local memory size: it is restricted to 128 KB per cluster.
- Global memory size: it is possible to use up to 1 GB, with 1.5 MB for cache memory with read and write capabilities.
- Private memory size: the Kalray implementation does not support private memory. However, it is mentioned that every variable declared within a kernel will partially use the stack and its size cannot be greater than 4KB.
- Kernel `printf` function calls are not supported.
Although there are many other limitations described in the Kalray OpenCL support File, the previous ones are those which have been considered during the development of this work. Despite some of the limitations, OpenCL provides some advantages as well. The OpenCL API handles two main functions, memory accesses through buffers and host-device communications. Compared to the Posix/OpenMP implementation, OpenCL avoids portal creation, executable spawning and memory limit handling within clusters. To compile source code developed with the OpenCL framework, the reader is referred to Annex II.

### 3.2. Sequential algorithms

This section describes the serial/sequential versions of the two algorithms used in the HELICoiD project for the hyperspectral image classification stage. The first one is the Support Vector Machine algorithm [39] and the second one is the Principal Component Analysis introduced in [1] and improved in [40].

#### 3.2.1. SVM algorithm

The *Support Vector Machine (SVM)* is a supervised machine learning algorithm which can be used for classification or regression analysis. This method relies on a pre-processing of the training data to represent patterns in a higher dimensional space. Doing so, data is mapped in such a way that enough room is left to have a hyperplane separating two categories [41]. As can be seen in Figure 55, SVM allows defining an optimal hyperplane to separate the class ‘circles’ from the class ‘crosses’. The optimal margin is defined as the distance of the closest class element to the hyperplane. These data points that are the closest to the hyperplane are called support vectors.

![Example of hyperplane separating two classes](image)

Figure 55 Example of hyperplane separating two classes [42]
Although the previous example in Figure 55 refers to a two-class bi-dimensional classification problem, SVM is capable to identify any number of classes in n-dimensional classification problems using \( n \)-dimensional hyperplanes (\( n \) is related to the number of features\(^{20}\) of the input dataset). The procedure to find the optimal hyperplanes is known as \textit{training}. The training procedure is out of scope of this work because the implementation on the MPPA is only oriented to classification. In case further information is needed, references [41], [42], and [39] depicts the complete mathematical approach to the SVM algorithm.

Several samples of the classes available in the input data set are required for an accurate characterization during the training stage. In general, the input space needs to be transformed into a new space to separate the classes (see Figure 56). Note that the separating hyperplane is found in this transformed space.

![Figure 56 Projection onto a higher dimensional space\(^{21}\)](http://www.cs.toronto.edu/~urtasun/courses/CSC411/tutorial9.pdf)

Equation (1) is used to express the discriminant function (to classify an input based in the training set). The value \( \alpha_i \) and \( \rho_m \) are parameters obtained during the training process. The value is the kernel which defines the hyperplane.

\[
g_m = \sum_i^N \alpha_i K(x_i, x_m) + \rho_m
\]  

(1)

Although there are several types of kernels (linear, polynomial, radial basis function, fisher and so on) this work covers exclusively the linear kernel. The main reason is that the linear kernel is the simplest in terms of computational load (required to fulfill

---

\(^{20}\) Features: characteristics to define an specific class (e.g., width, height, spectral signature)

the real-time constrain). Hence, to complete the discriminant function Equation (2) shows the linear kernel.

\[ K(x_i, x_m) = (x_i x_m^T) \]  

(2)

Basically, the discriminant function using a linear kernel is a dot product of the data times the class weights. Since this procedure is a binary classification, the discriminant function must be applied as many times as available combination of classes. In this case, to classify a pixel three classes are considered: healthy tissue (class 1), cancer tissue (class 2), and others (class 3). Thus, three binary classifiers are identified. For instance, classifier 1/2, classifier 1/3, classifier 2/3 and consequently there will be three independent decisions. Finally, using a majority voting of the results from the binary classifiers the definitive class is assigned to the pixel.

The sequential implementation of this algorithm is depicted in Table 3, the procedure consists on the implementation of equations (1) and (2) per each pixel of the hyperspectral image.

### Table 3 SVM linear kernel algorithm

<table>
<thead>
<tr>
<th>Algorithm 1: SVM linear kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Hyperspectral image X (matrix pixels x bands), weights (class x bands), rho_vector (classes x 1)</td>
</tr>
<tr>
<td>For i=1:3 (classes)</td>
</tr>
<tr>
<td><strong>Step 1:</strong> dot product of X by weights</td>
</tr>
<tr>
<td><strong>Step 2:</strong> Subtract the value ( \rho ) associated to each weight</td>
</tr>
<tr>
<td><strong>Step 3:</strong> perform binary classification</td>
</tr>
<tr>
<td>endFor</td>
</tr>
<tr>
<td><strong>Step 4:</strong> Assign the final corresponding class to the pixel using majority voting</td>
</tr>
<tr>
<td><strong>Output:</strong> Classification output (vector Pixels)</td>
</tr>
</tbody>
</table>

#### 3.2.2. PCA algorithm

The Principal Components Analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.

Considering that hyperspectral images are conformed by a high number of pixels obtained in different bands of the spectrum, hence, there could be the possibility of
redundant information. Thus, PCA algorithm can be employed to perform a
dimensionality reduction and noise removal of the original image by transforming the
original values into a new subspace. This transformation is achieved by finding the
directions (components) that maximize the variance of the original dataset. Those are
called eigenvectors, which are used to project the original dataset into the new subset.
Ideally, this kind of transformation is orthogonal. Therefore the first eigenvectors (ordered
from higher to lower accordingly to their corresponding eigenvalues) contains most of
the valuable information of the image. This ordering allows to perform the dimensionality
reduction and discard the lower bands. To conclude, the benefits of the PCA algorithm
are (1) dimensionality reduction, (2) simpler data representation, and (3) memory
reduction [43].

An application of the PCA algorithm to a hyperspectral image of Washington D.C
(see Figure 57) is shown in Figure 58. It is clearly depicted that the first bands (principal
components) of the PCA transformation contains enough data to describe the image.
However, from bands 10 and up we can consider that are completely composed by
noise. Thus we can discard these bands because its information is irrelevant.

Figure 57 Hyperspectral image of Washington D.C. mall area [43]
Figure 58 Sample of PCA bands of Washington D.C. mall area [43]

Once the basic concepts of the PCA algorithm have been described, next step consist of describing its mathematical basis. First of all, an image of size \( m \times n \) is needed, where \( m \) is the number of pixels and \( n \) is the number of bands. In order to standardize the matrix values, the mean value is subtracted of each band

\[
Y_1 = (Y_{m,n} - \bar{Y}_n)
\]

Where \( Y \) is the original image, and \( A \) is the result of the standardized image. Then, the common PCA formula can be expressed as follows

\[
\Sigma = E \Lambda E^{-1}
\]

\[
AY = X
\]

\[
\Sigma = Y_1 Y_1^T
\]

Where \( \Sigma \) corresponds to the covariance matrix of matrix \( Y_1 \) (Eq. (3)), \( E \) is the orthogonal matrix containing the eigenvectors of \( \Sigma \) and \( \Lambda \) is the diagonal matrix whose diagonal elements are the eigenvalues (\( \lambda \)) [44]. Matrix \( A \) can be seen as the matrix that transforms \( Y \) to \( X \). Hence, the spatial reduction is only achieved if the first eigenvectors
(accordingly to the eigenvalues weights) are used in the transformation process (i.e., first elements of matrix A).

The aforementioned method is the traditional form to calculate the PCA from a mathematical approach. However, there are many other solutions to find the eigenvectors. Thus, in order to follow a more practical approach, this implementation uses the Jacobi method, which is an iterative algorithm that approximates the actual value of the eigenvectors. The accuracy of the mathematical approach depends on a convergence criterion ($\varepsilon$), which implies, in turn, the number of iterations the method employs.

The Jacobi method - only valid for symmetric real matrices - approximates the original matrix to a diagonal one using planar successive rotations. A full description of this method, including the mathematical approach and implementation, can be found in [45] and [40]. There are two variants of this method: classical and cyclic. The classical variant makes zero the largest off-diagonal elements of the matrix in each rotation/iteration. On the other hand, the cyclic one makes zeros the off-diagonal elements in a specific order (e.g., row by row). Unlike the classical variant, the cyclic one does not perform the search of the highest value in the image. The procedure is the following:

- First, select the off-diagonal element to be zeroed.
- Next, the Jacobi rotation matrix ($P$) is calculated as shown in Eq. (7).

However, Note that the size of this matrix is that of the covariance matrix, which typically is symmetric. This matrix $P$ must be stored for further usage to compute the eigenvectors.

$$P = \begin{pmatrix}
1 & 0 & \cdots & 0 \\
0 & \cos \alpha & \sin \alpha & \vdots \\
\vdots & -\sin \alpha & \cos \alpha & \vdots \\
0 & \cdots & \cdots & 1
\end{pmatrix} \quad (7)$$

- Then, the value of the rotation is calculated for every element to be zeroed. Thus, the rotation shall be calculated at the beginning of each iteration. Equations (8) to (11) are used in each iteration to compute the Jacobi rotation matrix.

$$m = \frac{2 \cdot a_{ij}}{a_{ij} - a_{ii}} \quad (8)$$
\begin{align*}
t &= -1 + \sqrt{1 + m^2} \\
\cos \alpha &= \frac{1}{\sqrt{1 + t^2}} \\
\sin \alpha &= t \cdot \cos \alpha
\end{align*}
\begin{equation}
(9)
(10)
(11)
\end{equation}

- Next step is zero the corresponding off-diagonal element by applying Eq. (12). As the result matrix shall be symmetric, the corresponding symmetric off-diagonal element has also the same value.

\[ C_1 = P_1^T \cdot C \cdot P_1 \] 
\begin{equation}
(12)
\end{equation}

- Then, the value obtained in the last step is updated and next value to be zeroed is selected. Considering that in every iteration any zeroed value could be undone due to processing, this procedure requires a convergence criterion or stop condition defined by \( \varepsilon \).

- Finally, when the convergence criterion has been reached and all the ‘\( k \)’ iterations have been calculated, the eigenvectors can be computed as shown in Eq. (13). The eigenvectors are placed in the columns of matrix \( E \).

\[ E = P_1 \cdot P_2 \cdot P_3 \cdots P_k \] 
\begin{equation}
(13)
\end{equation}

Another advantage of the Jacobi method is the computation of the eigenvalues and the eigenvectors associated to them, which represents a saving in the process workload.

In case the eigenvalues are required, they can be obtained using the Jacobi rotation matrix and the covariance matrix as Eq. (14) shows.

\[ C_k = P_k^T \cdot \cdots \cdot P_1^T \cdot \Sigma \cdot P_1 \cdot \cdots \cdot P_k \] 
\begin{equation}
(14)
\end{equation}

Considering that each rotation of the Jacobi method only affects a couple of rows and columns, many rotations can be computed in parallel. For instance, if the elements \( \Sigma_{23} \) and \( \Sigma_{45} \) are considered, they could be processed in parallel; on the contrary, the elements \( \Sigma_{01} \) and \( \Sigma_{31} \) cannot be zeroed simultaneously because of its data dependency [46]. To clarify this statement an example is shown in Figure 59. The algorithm looks for any of the off-diagonal elements above the diagonal (note that the matrix is symmetrical). For instance, as can be seen in the example of Figure 59 a), if the element \( \Sigma_{01} \) (row 0, column 1) is selected, neither the elements located in columns 0 and 1 nor row 0 can be
selected to be zeroed in parallel. Likewise, as can be seen in the example Figure 59 b), the element selected is $\Sigma_{05}$ (row 0, column 5). Therefore, neither the elements located in rows 0 and 5 nor those in columns 0 and 5 can be selected.

![Figure 59 Elements restriction to parallelize Jacobi algorithm](image)

Once the eigenvectors are found, the last step to perform the dimensionality reduction is to project the original image into this new subspace. To do so, a matrix multiplication is required as it is shown in equation (5).

To conclude this section, Table 4 depicts the steps to perform the PCA algorithm which is going to be used to program the application.

Table 4 PCA algorithm description

<table>
<thead>
<tr>
<th>Algorithm 2: Principal Component Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Hyperspectral image $Y$ (matrix $m \times n$)</td>
</tr>
<tr>
<td>Step 1: $X = $ Remove the average of each band of $Y$</td>
</tr>
<tr>
<td>Step 2: Covariance matrix $C = X^T \cdot X$</td>
</tr>
<tr>
<td>Step 3: $E = $ Eigenvector decomposition of $C$</td>
</tr>
<tr>
<td>Step 4: Projection $P = Y \cdot E$</td>
</tr>
<tr>
<td>Step 5: Reduce $P$ to the number of principal components</td>
</tr>
<tr>
<td><strong>Output:</strong> Reduced hyperspectral image $P$ (matrix $N \times P$)</td>
</tr>
</tbody>
</table>
4 Application development
This chapter will describe the procedure used to handle the ‘automatic’ parallelization of the algorithms described in section 3.2 using OpenMP and OpenCL.

4.1. SVM: OpenMP

Before applying parallelism to the SVM algorithm it is important to analyze if there is any data dependency. As it was explained in section 0, all the steps of the algorithm are computed per-pixel because there is no dependency among them. Hence, all the pixels of the image can be processed in parallel.

The parallelization of the algorithm is based in the work made by Madroñal, D. Et al. in [39]. In this work, the authors implement the SVM algorithm onto the MPPA using POSIX threads. The following paragraphs describe the adaptation of the previous implementation using OpenMP. The steps which describe the structure of the OpenMP implemented code are the following:

1. **Input dataset:** The image to test the current algorithm implementation is crucial since hyperspectral sensors generates high volumes of information. The selected image has 1040 bands and 18580 pixels on each one. Every pixel will be mapped in the MPPA as an array of floats (4 bytes), providing a total size of 73.71 MB (see Equation (15))

\[
\text{size} = \frac{18580 \text{ pixels} \times 1040 \text{ bands} \times 4 \text{ bytes}}{1024^2} = 73.71 \text{ MB}
\]  

(15)

In this implementation the image has vector (array) form because the communication API (syncs and portal) between the host and the MPPA does not support matrixes (bidimensional arrays).

\[
\begin{align*}
a_{00} & \quad a_{01} & \quad a_{02} \\
\rightarrow a_{10} & \quad a_{11} & \quad a_{12} & \rightarrow a_{00} & \quad a_{01} & \quad a_{02} & \quad a_{10} & \quad a_{11} & \ldots & \quad a_{21} & \quad a_{22} \\
& \quad a_{20} & \quad a_{21} & \quad a_{22}
\end{align*}
\]  

(16)

2. **Image Splitting:** Since it is impossible to fit the whole image within a cluster due its memory limitation (1 - 1.5 MB), it is necessary to split the image. The splitting process of the image is related to the number of clusters implied in the process. Considering that the algorithm requires to fit a complete pixel (1040 bands) in each compute cluster and also taking into account that 16 cores exists within a cluster, it is preferably to consider image blocks which are in multiple of 16. Thus, it has been
decided to assign 240 pixels in each cluster to fulfill the two previously mentioned conditions. The required memory to allocate an image block is illustrated in Equation (17)

$$\text{size} = \frac{240 \text{ pixels} \times 1040 \text{ bands} \times 4 \text{ bytes float}}{1024^2} = 0.95 \text{ MB} \approx 1\text{ MB}$$

(17)

However, the total number of image blocks, i.e. the total number of pixels of the hyperspectral image divided by the number of pixels per block ($18580/240 = 77.41$) is not an integer number. Consequently the last pixels need to be padded with 0. Specifically, 140 pixels are added to work with an image of 18720 pixels. With this padding, it is possible to compute the algorithm sending iteratively either an image block 78 times to one cluster or 13 image blocks 6 times to 13 clusters of the MPPA.

3. **Classification:** this step involves the SVM algorithm described in Table 3 of Chapter 3. The implementation described below is based on the work developed by D. Madroñal as part of the Helicoid project using POSIX threads. As explained in [39], the values of the SVM model weights required to describe the hyperplane are considered as inputs. This implementation has been developed using only one cluster and 16 cores (1 of the cores is exclusively employed to control the work inside the cluster). The execution flow is as follows: first, a block of 240 pixels of the image is received from the I/O subsystem; then, a master thread creates a team of 15 threads to process 1 pixel per thread (15 pixels in parallel) applying Equations (1) and (2). Figure 60 shows the source code of the master thread. The commands `mppa_aio_wait` and `mppa_aio_read` are used to read a block of the image in every iteration of the while loop. Then the fork and join process is executed. The value `NUM_PTHREADS` defines the number of threads that the forking process uses. In this case the highest value is 15. The highlighted function `task` executes on the remaining cores of the cluster to perform the classification procedure using the algorithm of the SVM. These steps are repeated 78 times within the while loop instanced at the beginning of the code.
iteration = 0;

while (iteration < ITERS) {
    status |= mppa_aio_wait(pixels_in_portal_aiocb);
    status |= mppa_aio_read(pixels_in_portal_aiocb);

    for (j = 0; j < NUM_PTHREADS; j++) {
        param[j] = j;
        if (pthread_create(&tid[j], NULL, task, &param[j])) {
            printf("@* thread creation failed,index=%d\n", j);
            exit(-1);
        }
    }

    for (k = 0; k < NUM_PTHREADS; k++) {
        pthread_join(tid[k], NULL);
    }

    status |= mppa_pwrite(result_portal_fd, result,
                          N_PIXELS_ITERATION_CLUSTER*sizeof(float),
                          slice_offset_result);
    iteration++;
}

Figure 60 POSIX threads creation and joining with SVM

Clearly, each pixel is classified independently in each core of the compute cluster. Based on this idea, the OpenMP directives are introduced. This proposal shows two approaches to exploit the parallelism of the device. The first one follows the same strategy as the POSIX implementation (one cluster) while the second one employs 13 clusters in parallel.

The first approach is similar to that of the POSIX implementation. The difference is that OpenMP eases the thread creation and joining. First, the I/O subsystem performs a unicast transmission to a cluster. Using a portal communication scheme, a unicast transmission differs from a multicast procedure (see Figure 61) just in the selection scheme of the cluster which receives data. Then, in every iteration, the I/O processor sends blocks of image_received size (the actual image size augmented with zero padding) of a VECTOR_LENGTH_DATA_SENT length (240 pixels * 1040 bands).
Once the image has been sent to the cluster, the classification process starts. Figure 62 depicts the OpenMP implementation within a cluster. The first section corresponds to the image block reception. Then, the statement `omp parallel for` defines `NB_THREADS`, i.e. the number of threads involved in the fork and join procedure. In this case, given the capacity of OpenMP to involve the master thread in the forking process up to 16 threads are available.

```c
while (iteration < ITERS){
    status |= mppa_ioctl(image_read_portal_fd,
            MPPA_TX_SET_RX_RANKS,
            CLUSTER_COUNT,
            ranks);
    status |= mppa_pwrite(image_read_portal_fd,
                image_received + (iteration * VECTOR_LENGTH_DATASENT),
                sizeof(float)*VECTOR_LENGTH_DATASENT, 0);

    //OMP THREADS SECTION BEGIN
    #pragma omp parallel for num_threads(NB_THREADS)
    for (i = 0; i < N_PIXELS_ITERATION_CLUSTER; i++){
        cluster_pixel_classify(i);
    }
    //OMP SECTION END

    status |= mppa_pwrite(result_portal_fd, result,
            N_PIXELS_ITERATION_CLUSTER*sizeof(float),
            slice_offset_result);
    iteration++;
}
```

Figure 62 OpenMP thread creation for SVM

Finally, the results are sent to the IO subsystem. As in the POSIX implementation, the previous steps are repeated 78 times to process all the pixels of an image.

The second approach, with 13 clusters, uses the same method to process the pixels within a cluster. In this case, each iteration is going to process 240 pixels per cluster, 3120 pixels in total. Thus, to complete the processing of an image, just 6 iterations are needed. To implement this improvement instead of using a unicast transmission portal a multiplex one is required. Figure 63 shows the source code used to transmit a portion of the image to 13 clusters. As can be seen, the transmission is mainly controlled by the `jumps` indicated in equation (21).
jumps = \text{PIXELS}_{\text{block}} \times (\text{coreID} + \text{iteration} \times 13) \times \text{BANDS} \tag{18}

\begin{verbatim}
for (rank = 0; rank < 13; rank++) {
    status |= mppa_ioctl(image_read_portal_fd,
        MPPA_TX_SET_RX_RANK,
        rank);
    status |= mppa_pwrite(image_read_portal_fd,
        image_received + 240*(rank+iteration*13)*1040,
        sizeof(float)*VECTOR_LENGTH_DATA_SENT, 0);
}
\end{verbatim}

Figure 63 SVM multiplex portal to send an image block

In this manner, every iteration of the I/O subsystem will send the corresponding data to 13 clusters. To process an image block received in a cluster, the implementation employs the same source code as the one depicted in Figure 62. The only change is the method to write back the information to the IO subsystem because since 3120 pixels are managed per iteration instead of 240 pixels. Figure 64 shows the source code. The key point here is to handle correctly the indexes of the values to return to the IO subsystem. In this case, this is achieved with the \text{slice.offset.result} variable that involves, in turn, the variable \text{rank} which refers to the cluster identification number. Thus, in this way every cluster sends a portion of 240 classified pixels of the 3120 pixels required in every iteration of the loop.

\begin{verbatim}
slice_offset_result = sizeof(float)*240*\text{rank};
status |= mppa_pwrite(result_portal_fd,
    result,
    240*\text{sizeof(float),}
    slice_offset_result);
\end{verbatim}

Figure 64 writing back the pixels classified using 13 clusters

In both approaches, after an image is classified the I/O sends data back to the host and removes the additional pixels used in the zero padding process. In chapter 5, the performance results of all three implementations (POSIX, OpenMP one cluster, OpenMP 13 clusters) will be shown.
4.2. PCA: OpenMP

As far as the PCA algorithm concerns, the parallelism study onto the MPPA is based on the algorithm described in Table 4 and the source code with POSIX threads implemented by Lazcano, R.et al. The stages that can be differentiated are the following:

1. **Input dataset**: As has been explained before, it is important to consider the size of the image captured by the hyperspectral sensor. Specifically, the hyperspectral images employed has 224 spectral bands and 10000 pixels (spatial dimensions are 100 x 100 pixels per band). The hyperspectral images are artificially built as the combination of 5 *endmembers*. Every pixel is defined as a float. Thus, the total size of an image is computed in Equation (19).

\[
\text{size} = \frac{10000 \text{ pixels} \times 224 \text{ bands} \times 4 \text{ bytes}}{1024^2} = 8.54 \text{ MB}
\]  

(19)

Because of the shared memory limitation of MPPA clusters, it is worth noting to consider this value. Although the technical documentation mentions that the MPPA has 2MB of shared memory, from a practical point of view, there is only 1.5 MB available to allocate variables. Effectively, it is true that every cluster has 2MB, but this memory is shared between program and data. Hence, it is impossible to fit the whole image into the cluster’s memory. Consequently, the image has to be split according to the algorithm requirements. Moreover, the image is assumed to be a set of vectors as it was considered for the SVM algorithm -see Equation (16).

2. **PCA step 1, Average calculation**: given that the processing time of this step is the lowest one compared to the other steps, it was not considering OpenMP optimization. However, its parallel implementation has been done using 16 clusters (with only one core). This parallel implementation is described in the following paragraphs.

As it was explained before the image is composed of 224 bands and 10000 pixels each. Thus, to parallelize the step, 14 bands can be processed in each cluster (224 bands / 16 clusters =14). Note that the restriction is the available memory. However, as can be observed in equation (20), the amount of memory required to allocate each part of the image on the cluster is less than 1.5 MB

\[
\frac{(14 \times 10000 \times 4)}{1024^2} = 0.5340 \text{ MB}
\]  

(20)
The source code of the I/O subsystem includes a multiplex portal to send parts of the image to the different clusters (see Figure 65).

```c
// Distribute the image to the clusters
for (rank = 0; rank < CLUSTER_COUNT; rank++) {
    status |= mppa_ioctl(image_in_portal_fd, MPPA_TX_SET_RX_RANK, rank);
    status |= mppa_pwrite(image_in_portal_fd, image_in + rank*slice,
                           sizeof(float)*slice, 0);
}
```

Figure 65 PCA transmission of image blocks to clusters (IO subsystem)

The value `rank` corresponds to the clusters which perform the computation. `Slice` is the portion of the image to send to the clusters. Since the hyperspectral image is defined as a vector, `slice` value is 140000, i.e. the `mppa_write` command sends blocks of 140000 floats to each cluster. Table 5 explains the algorithm implemented in each cluster. Briefly speaking, each cluster computes the mean value of 14 bands and subtract the result to every value of the corresponding band. Then, the cluster sends back the sub-cube to the IO subsystem with the corresponding offset to ease the storage in the correct position. After sending the portion of the average matrix, the cluster keeps the result to process next step. Figure 66 shows the command to send back the image block to the I/O subsystem. The offset (`slice_offset_c` variable) determines the correct position to write back the data. This variable depends on the `rank` and `slice` values specified by the I/O subsystem (see Figure 65).

Table 5 Average algorithm for clusters

<table>
<thead>
<tr>
<th>Average algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Hyperspectral image portion Y (vector 140000 → 10000 pixels and 14 bands)</td>
</tr>
<tr>
<td>PIXELS = 10000</td>
</tr>
<tr>
<td>for j=0 to 13</td>
</tr>
<tr>
<td>for k=0 to PIXELS -1</td>
</tr>
<tr>
<td>average[j] += Y[PIXELS]*j+k</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>average[j] = average[j]/10000</td>
</tr>
<tr>
<td>for l = 0 to PIXELS -1</td>
</tr>
<tr>
<td>image_prima[PIXELS]*j+l = Y[PIXELS]*j+l - average[j]</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td><strong>Output:</strong> block of image subtracted the mean value (image_prima)</td>
</tr>
</tbody>
</table>
It is important to highlight that to split correctly the image a complete band or bands must fit into the shared cluster memory. Otherwise, it will be not possible to compute the average. Another restriction to split the bands is that the relationship between the number of bands and the number of cores must be an integer number. If this is not the case, it is possible to round the value in such a way that last cluster computes fewer bands. Specifically, this restriction does not apply in this implementation since the division is an integer number.

Once all clusters have returned their subcubes to the I/O subsystem, they are joined into a single array to continue the process. This new image array is called image prima.

3. **PCA step 2, covariance matrix:** as can be seen in Table 4, the objective of this step is to find the covariance matrix, which involves the multiplication of the transposed image prima matrix and the original one (see Equation (21)).

\[
\begin{bmatrix}
    x'_{0,0} & \ldots & x'_{0,PIXELS} \\
    x'_{1,0} & \ldots & x'_{1,PIXELS} \\
    \vdots & \ddots & \vdots \\
    x'_{BANDS,0} & \ldots & x'_{BANDS,PIXELS}
\end{bmatrix}
\begin{bmatrix}
    \begin{bmatrix}
        x'_{0,0} & \ldots & x'_{0,BANDS} \\
        \vdots & \ddots & \vdots \\
        \vdots & \ddots & \vdots \\
        x'_{PIXELS,0} & \ldots & x'_{PIXELS,BANDS}
    \end{bmatrix}
\end{bmatrix} =
\begin{bmatrix}
    G_{0,0} & \ldots & G_{0,BANDS} \\
    \vdots & \ddots & \vdots \\
    G_{BANDS,0} & \ldots & G_{BANDS,BANDS}
\end{bmatrix}
\]

The data parallelism extraction in this case is related to the rows and columns involved in the process. Figure 67 shows how to obtain the value \( C_{12} \). Note that only the row and column highlighted in black are involved in the process. Other values of the matrix \( C \) can be computed in parallel without affecting the overall multiplication process.
Since matrixes are expressed as vectors, instead of allocating memory to store the transposed matrix, positions of *image prima* are read according to the transposed version. Thus, the I/O subsystem is in charge of sending *image prima* to the clusters. However, because of the memory limitation, images are split in as many slices as available clusters. Hence, every cluster will receive 140000 pixels corresponding to 14 bands (sub matrix of 10000 x 14) to contribute in each iteration with 14x14 elements of the covariance matrix. It is worth noting that each cluster stores the previously computed 14 averaged bands. Therefore, each cluster computes iteratively a strip of the covariance matrix and contributes back its result.

The IO subsystem sends the corresponding portion of *image prima* to all clusters using a multicast portal connector and waits for the results. Since clusters have stored its portion of *image prima* from the previous step, the I/O subsystem sends only the portion required to complete the multiplication. This is called *image prima block*. Then, clusters compute the matrix multiplication (see Figure 68) for each image prima block received using all cores of the compute clusters.
Since all the multiply and add operations are independent it is possible to compute them with up to 16 OpenMP threads working in parallel. In previous works, the implementation for this stage on the MPPA was done using only one core per cluster [1], achieving a certain degree of parallelism but without using all the available cluster resources.

Table 6 depicts the pseudo code of a sequential implementation used in a cluster. Since the procedure to find the covariance matrix is a matrix multiplication, \textit{image prima} and \textit{image prima block} are multiplied. To do so, three loops are needed. The first inner loop iterates the pixels of a band to compute an element of the covariance matrix. Second and third loops iterate the matrixes rows and columns, correspondingly. Then, to perform the matrix multiplication a dot product (i.e. multiply and add) is performed between \textit{image prima} and \textit{image prima block}.

**Table 6 Covariance matrix algorithm**

<table>
<thead>
<tr>
<th>Covariance Matrix algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> image prima block (vector 140000 → 10000 pixels and 14 bands) received by the I/O and image prima (stored in cluster memory)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PIXELS = 10000</td>
</tr>
<tr>
<td>slice = 140000 → total number of pixels to process</td>
</tr>
<tr>
<td>portion=slice/PIXELS</td>
</tr>
<tr>
<td>for cols = 0 to portion – 1</td>
</tr>
<tr>
<td>for iter = 0 to portion - 1</td>
</tr>
<tr>
<td>mat_block [cols<em>portion</em>iter] =0</td>
</tr>
<tr>
<td>for i = 0 to PIXELS -1</td>
</tr>
<tr>
<td>mat_block [cols*portion+iter] +=</td>
</tr>
<tr>
<td>image prima[cols*PIXELS+i] *</td>
</tr>
<tr>
<td>image prima block[i+iter*PIXELS]</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td><strong>Output:</strong> block of multiplication matrix (mat_block size 14 x 14)</td>
</tr>
</tbody>
</table>

---

22 Source: http://polymathprogrammer.com

Figure 68 Matrix multiplication procedure
To exploit the parallelism of the covariance matrix computation, previous works [1], have exclusively employed the separation of the tasks among all the clusters and performed the matrix multiplication using only one core (see Figure 69). This proposal suggests the use of OpenMP to take advantage of all available resources in the cluster.

```c
int i;
const int portion = slice/PIXELS;

int iter = 0;
int cols = 0;

matrix_mult_block = (double*)malloc(portion*portion*sizeof(double));

while(cols < portion){
    while(iter < portion ){
        matrix_mult_block[cols*portion+iter] = 0;
        for (i = 0; i < PIXELS; i++){
            matrix_mult_block[cols*portion+iter] +=
            image_prima[cols*PIXELS+i] * image_prima_block[i+iter*PIXELS];
        }
        iter++;
    }
    cols++;
    iter = cols;
}
```

Figure 69 Cluster implementation using one core

Figure 70 shows the changes from the previous implementation to a methodology based in OpenMP. Since OpenMP does not support while statements in C language all those statements have been changed to for loops. The first outer loop is not parallelized. Note that only the operations of the inner loops are considered. The #pragma constructor configures the following parameters:

- **NB_THREADS**: number of threads available to use, its value can be from 0 to 16. It represents how many cores from the cluster are going to be involved in the computation.
- **Shared variables**: `matrix_mult_block`, `image_prima`, `image_prima_block`. These variables are visible to all threads. They need either to read or write from/to them.
- **Private variables**: `iter` and `i`. This variables are related to the inner loops. Thus, each thread handles them independently.
```c
int i;
const int portion = slice/PIXELS;

int iter=0;
int cols=0;

matrix_mult_block = (double*)malloc(portion*portion*sizeof(double));

#pragma omp parallel num_threads(NB_THREADS)
shared(matrix_mult_block,image_prima,image_prima_block)
private(iter,i)
{
    #pragma omp for schedule(static)
    for(iter=cols; iter<portion; iter++){
        matrix_mult_block[cols*portion+iter] = 0;
        for (i = 0; i < PIXELS; i++){
            matrix_mult_block[cols*portion+iter] +=
                image_prima[cols*PIXELS+i] *
                image_prima_block[i+iter*PIXELS];
        }
    }
} // end of pragma section
} // end of first for
```

Figure 70 Matrix Multiplication with OpenMP

The second `#pragma` initializes the `for` constructor with a schedule static clause. This means that every thread will be executed in order. The decision to use this clause was made based on experimental testing to ensure the correct order of the results obtained from the `for` loop. The following for loop computes a covariance matrix element. Then, the OpenMP API joins all results and stores the values on the corresponding position of `matrix_mult_block`, which is actually a portion of the covariance matrix. Finally, the cluster writes back the data to the IO subsystem and removes the variables to free memory on the clusters.

4. **PCA step 3, eigenvectors decomposition**: this step implements the Jacobi cyclic method described in section 3.2.2. The cyclic method was chosen instead of the classical one because of its parallelism capabilities. Besides, unlike the classical method, the cyclic one does not look for the higher off-diagonal value. Thus, the cyclic method is capable to zeroed any element in any order. To explain the implementation of this step, Table 7 describes a sequential implementation of the Jacobi algorithm.
### Table 7 PCA algorithm sequential

<table>
<thead>
<tr>
<th>Inputs:</th>
<th>covariance matrix (size 224 x 224), convergence criterion, BANDS_PCA iter=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>while</td>
<td>(stop&gt; )</td>
</tr>
<tr>
<td>compute</td>
<td>stop condition (sum of diagonal elements of *)</td>
</tr>
<tr>
<td>for i=0</td>
<td>to bands -1</td>
</tr>
<tr>
<td>for j=i</td>
<td>to bands</td>
</tr>
<tr>
<td>select</td>
<td>elements to be zeroed</td>
</tr>
<tr>
<td>if</td>
<td>is available to be zeroed (no row/column dependency)</td>
</tr>
<tr>
<td>calculate</td>
<td>elements m, t, cos and sin using equations (8) to (11)</td>
</tr>
<tr>
<td>create</td>
<td>Jacobi rotation matrix (P_iter) as a diagonal matrix full of ones</td>
</tr>
<tr>
<td>update</td>
<td>the values P_iter</td>
</tr>
<tr>
<td>P_ij =</td>
<td>cos α</td>
</tr>
<tr>
<td>P_ij =</td>
<td>−sin α</td>
</tr>
<tr>
<td>P_ij =</td>
<td>cos α</td>
</tr>
<tr>
<td>P_ij =</td>
<td>cos α</td>
</tr>
<tr>
<td>Obtain by evaluating equation (12)</td>
<td></td>
</tr>
<tr>
<td>Store P_iter matrix</td>
<td></td>
</tr>
<tr>
<td>Update the zeroed value i,j to</td>
<td></td>
</tr>
<tr>
<td>Endif</td>
<td></td>
</tr>
<tr>
<td>Endfor</td>
<td></td>
</tr>
<tr>
<td>Endfor</td>
<td></td>
</tr>
<tr>
<td>End while</td>
<td></td>
</tr>
<tr>
<td>calculate eigenvalues using equation (14) and P_iter</td>
<td></td>
</tr>
<tr>
<td>order eigenvalues in descending order</td>
<td></td>
</tr>
<tr>
<td>calculate E eigenvectors matrix using equation (13) and P_iter</td>
<td></td>
</tr>
<tr>
<td>order eigenvectors correspondingly to eigenvalues (E_ordered)</td>
<td></td>
</tr>
<tr>
<td>for i = 0 to BANDS</td>
<td></td>
</tr>
<tr>
<td>for j = 0 to BANDS_PCA</td>
<td></td>
</tr>
<tr>
<td>jac_out_{i,j} = E_ordered_{i,j}</td>
<td></td>
</tr>
<tr>
<td>endfor</td>
<td></td>
</tr>
<tr>
<td>endfor</td>
<td></td>
</tr>
</tbody>
</table>

**Output:** jac_out (matrix size 224 x BANDS_PCA)

Considering that a POSIX implementation of this algorithm was made in [40], what follows explain the algorithm parallelization and the usage of OpenMP directives. Later on, the parallelism achieved with those two methods is compared.

First, the input variables are considered, i.e., covariance matrix (Σ), convergence criterion ε and number of principal components kept of the PCA algorithm. First, since is a vectorized matrix of 224x224 of float elements, it employs around 0.2 MB. Thus the whole covariance matrix can be placed inside a cluster. Secondly, the convergence criterion is a user defined parameter to set the accuracy of the method [47]. Finally, the number of principal components is required to
perform the dimensional reduction in such a way that the fewest amount of information possible is lost is related to the spectral signatures of the artificially created input image. The required number of principal components is \( \Omega - 1 \), where \( \Omega \) is the number of spectral signatures of the artificially created input image. This statement has been verified by several tests made in [1]. Hence, for this implementation the image built has 5 spectral signatures and thus \textit{BANDS\_PCA} equals 4.

Then, considering that the covariance matrix fits into one cluster, the parallelism strategy employs only one cluster with 16 cores. Bearing in mind that the task needs to update a common variable (\( \Sigma \)), the usage of multiple clusters is not the wisest solution. If all the elements of the MPPA were used (16 clusters and 16 cores), this shared variable would need to be updated, meaning that a lot of time would be spend sending data rather than processing it. Taking this statement into account, in order to implement the OpenMP code, the strategy addressed in [40], using one cluster and 16 cores, has been followed.

The strategy using a POSIX implementation is the following. The cluster executes its main function, also known as master thread. The master thread creates up to 15 threads and locks them using a mutex, until the data is available to process. Then, the master thread looks sequentially for the available off-diagonal elements to be zeroed (up to 15), considering the parallel restriction shown in Figure 59. Once the data is available to be processed, the master thread awakes the 15 available threads unlocking the mutex (see Figure 71 mutex unlock).

```
for (i=0; i<NUM_THREADS_JACOBI; i++){
    pthread_mutex_lock(&mutex[i]);
    pthread_cond_signal(&condition[i]);
    pthread_mutex_unlock(&mutex[i]);
}
```

Figure 71 mutex unlock

Then, each thread implements Equation (12) in two steps. First, the left multiplication \( P_{\text{iter}}^T \cdot \Sigma \) and secondly \( \Sigma \cdot P_{\text{iter}} \). To do so, among all running threads some synchronization is required. This is achieved using a mutex to lock the current thread and a semaphore to notify the master thread its completion. The master thread waits to receive a semaphore post from all running threads before starting the second operation. As can be seen in Figure 72, sem1 corresponds to the semaphore of the first part of the operation and sem2 corresponds to the second
one. Between each of them, there is a *unlock mutex* procedure to continue the thread processing.

```
while(end_op < values_processed_iter){
    sem_wait(&sem1);
    end_op++;
}
for (i=0; i<values_processed_iter; i++){
    pthread_mutex_lock(&mutex[i]);
    pthread_cond_signal(&condition[i]);
    pthread_mutex_unlock(&mutex[i]);
}
end_op = 0;
while(end_op < values_processed_iter){
    sem_wait(&sem2);
    end_op++;
}
```

**Figure 72 Master thread semaphore synchronization**

The aforementioned steps are repeated until the stop condition is fulfilled. After this, all threads results are joined to get the resultant covariance matrix with off-diagonal elements zeroed. Then, once all threads are done, the *master thread* computes sequentially the eigenvalues and orders them in descending order. Then, the corresponding eigenvectors are computed. Finally, only the first \textit{BANDS\_PCA} eigenvectors (i.e. first four eigenvectors for test image selected) are returned to the I/O subsystem to continue the process.

The proposal presented in this work inherits the strategy of the POSIX implementation but using the available OpenMP constructors. The OpenMP strategy is depicted in Figure 73. Given that the complexity of the POSIX implementation is related, basically, to synchronize the different stages, synchronization in OpenMP can be easily achieved using the *barrier* constructor. The *barrier* constructor waits for all threads to finish their task before continue the next task.
Figure 73 OpenMP implementation of eigenvector decomposition step

The execution of the code is the following. The sequential part executed in the *master thread* to look for the possible elements to be zeroed and to compute equations (8) to (11) remains the same. However, as can be seen in Figure 74, threads are created using OpenMP.

```c
#pragma omp parallel num_threads(NUM_THREADS_JACOBI)
{
    #pragma omp for
    for (i=0; i<NUM_THREADS_JACOBI; i++){
        method1(i);
    }
    #pragma omp barrier
    #pragma omp for
    for (i=0; i<NUM_THREADS_JACOBI; i++){
        method2(i);
    }
}
```

Figure 74 OpenMP thread creation and synchronization
Note that the number of threads (up to 16 threads) that will be created needs to be defined. OpenMP has one advantage with regard to the POSIX implementation. The master thread can be used in the computation, so the code zeros 16 elements instead of 15. Method 1 and method 2 computes the $P_{\text{iter}}^T \cdot \Sigma$ and secondly $\Sigma \cdot P_{\text{iter}}$ respectively. The barrier constructor between both methods, allows the synchronization of all threads. This constructor replaces the usage of mutexes and semaphores. These steps are repeated as many times as needed until the stop condition is fulfilled.

Finally, all threads finish their computations and the OpenMP API join them to deliver the new covariance matrix with all the off-diagonal elements zeroed and the corresponding P matrix computed in the process. Then, the computation of the eigenvalues and eigenvectors is implemented. At last, only the first $BANDS_{PCA}$ eigenvectors are delivered to the I/O subsystem to continue the process.

Briefly speaking, the usage of the OpenMP API allows a developer to ease the synchronization mechanism between threads and avoids the explicit creation, joining and destruction of threads. In addition, the master thread is included in the splitting process. The overall results achieved with both implementations in terms of speed up will be covered in the next chapter.

5. **PCA step 4, projection and image reduction**: the projection of the original hyperspectral image into the new subset involves a matrix multiplication between the image and the eigenvectors. Since the matrix of eigenvectors, obtained in the previous step, has only the required principal components, both projection and dimensionality reduction are achieved. Since the matrix multiplication procedure was covered in step 2 of this algorithm, the same principle is employed in this step. However, this time fewer computations are needed. Thus, the I/O subsystem broadcasts a block of 14 bands of the original image to every cluster to perform the algorithm described in Table 8. OpenMP is employed to distribute the task among the 16 available cores of the cluster.
Table 8 Image projection and reduction algorithm

<table>
<thead>
<tr>
<th>Image projection and reduction algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> original_img_block (vector 140000 → 10000 pixels and 14 bands) and eigenvectors matrix (vector 896 → 224 rows x 4 principal components)</td>
</tr>
<tr>
<td>PIXELELS = 10000</td>
</tr>
<tr>
<td>slice = 140000 → total number of pixels to process</td>
</tr>
<tr>
<td>portion=slice/PIXELELS</td>
</tr>
<tr>
<td>for cols = 0 to PIXELELS – 1</td>
</tr>
<tr>
<td>for iter = 0 to BANDS_PCA</td>
</tr>
<tr>
<td>pca_out [cols*BANDS_PCA *iter] =0</td>
</tr>
<tr>
<td>for i = 0 to BANDS -1</td>
</tr>
<tr>
<td>pca_out[cols*BANDS_PCA+iter] +=</td>
</tr>
<tr>
<td>Image_in[cols*BANDS+i] *</td>
</tr>
<tr>
<td>eigenvectors[iter+BANDS_PCA*i];</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td><strong>Output:</strong> block of pca_out matrix (pca_out size 10000 x 4)</td>
</tr>
</tbody>
</table>

After all clusters sends back portions of the pca_out matrix, which is a dimensionally reduced version of the original image, the I/O subsystem joins the results into one variable and send it back to the host.

### 4.3. SVM: OpenCL

The OpenCL SVM implementation employs the same algorithm and inputs of the OpenMP SVM implementation. Considering the aforementioned, the OpenCL implementation consist of two steps: (1) a projection stage, which corresponds to the steps 1 and 2 of the sequential algorithm and (2) a decision stage, which corresponds to steps 3 and 4 (see Table 3). The execution flow is depicted in Figure 75.

![Figure 75 SVM OpenCL implementation](image)
The host program sends data to the MPPA to compute the projection stage, receives the results and prepares the arguments for the decision stage. The output of this stage is the classified image.

**Projection stage:** The kernel of this stage performs computations on each pixel. This stage calculates a dot product between each pixel and all three hyperspectral plane characteristic vectors. The host prepares the inputs to the kernel, i.e. the image and the hyperspectral plane vectors, with `clCreateBuffer` and `clEnqueueWriteBuffer`. Besides, the host creates the results buffer (see Figure 76).

```c
cl_buf_in_w_vector = clCreateBuffer(context, CL_MEM_READ_ONLY, VECTOR_LENGTH_W*sizeof(float), NULL, NULL); err = clEnqueueWriteBuffer(queue, cl_buf_in_w_vector, CL_TRUE, 0, VECTOR_LENGTH_W*sizeof(float), w_vector, 0, NULL, &omegaSend);

cl_buf_in_image_in = clCreateBuffer(context, CL_MEM_READ_ONLY, DATA*sizeof(float), NULL, NULL); err = clEnqueueWriteBuffer(queue, cl_buf_in_image_in, CL_TRUE, 0, DATA*sizeof(float), image_read, 0, NULL, &imagePart1);

cl_buf_out_results = clCreateBuffer(context, CL_MEM_READ_WRITE, 3*(PIXELS+108)*sizeof(float), NULL, NULL);
```

Figure 76 OpenCL SVM input/output of the projection stage

These buffers are assigned as kernel arguments. Figure 77 shows the projection kernel prototype, which has as arguments `pixels_in` (the hyperspectral image array), `w_vector` (the hyperspectral plane vector), `result` (the output array) and `bands` (the number of bands of the hyperspectral image).

```c
__kernel void projection(__global float *pixels_in,
                        __global float *w_vector,
                        __global float *result,
                        const int bands)
```

Figure 77 Projection kernel prototype

Then, using the `clEnqueueNDRangeKernel` command, the kernel is sent to the device for computation. Note that a key point is the value of the local and global work size arguments of the `clEnqueueNDRangeKernel` command. The local work size value is restricted to 16 due to Kalray specification. In other words, 16 pixel-like work-items are employed. The global work size value determines the number of times kernels are executed within the compute clusters. The input image has 18580 pixels. To make the image size divisible by 16, 108 padding pixels are added. Considering that every compute cluster executes 16 kernels at a time, the global size value is (18580+108)/16.
To avoid the computation of padding pixels inside the OpenCL projection kernel, there is a statement to restrict the computation (see Figure 78). The kernel checks the condition and performs no operation and return.

```c
unsigned int id = get_global_id(0);
if (id>=18580) return;
```

Figure 78 Kernel projection limitation

When all computations are finished, the host reads the buffer `cl_buf_out_results` using the `clEnqueueReadBuffer`. This last step is performed by the host, which prepares the inputs to the next stage. The kernel results vector array contains the information of a pixel projected onto the characteristic vectors of three hyperplanes.

**Decision stage:** Steps 3 and 4, shown in Table 3, are implemented in this stage by means of the kernel named `vote16` (see Figure 79). The displacement of the hyperspectral plane is subtracted to the dot product result computed in the previous state and a majority voting decision is made to classify a pixel. The input arguments are: (1) three arrays containing the projection of the pixel over the characteristic vectors of 3 hyperspectral planes, (2) the rho vector ($\rho$) and (3) the number of pixels to compute. The output is the array of classified pixels. In order to increase the algorithm performance using OpenCL, each kernel processes 16 pixels in each kernel call. In this way the number of calls of the kernel is reduced.

```c
__kernel void vote16(
   __global float *input,
   __global float *input2,
   __global float *input3,
   __constant float *rho_vector,
   __global int *result,
   const int pixels)
```

Figure 79 Kernel vote16 prototype

The local work size and global work size have the same values as those used in the projection kernel. Likewise, a restriction declared in the `vote16` kernel avoids to perform unnecessary computations (see Figure 80)

```c
if (id>pixels/16) return;
```

Figure 80 kernel vote16 limitation

The host program creates the required input and output buffers (see Figure 81) and assign them as kernel arguments (see Figure 82)
Chapter 4 – Application development

4.4. PCA: OpenCL

This section explains the procedure applied to implement the algorithm described in section 3.2.2. Unlike the implementation made with OpenMP/POSIX, the image does not need to be split. Effectively, the shared memory restriction is no longer a problem in OpenCL, since the API automatically handles all memory access (the global memory allocates 1GB) and data transmissions between the host and the device.

The execution flow of the PCA algorithm describe in Table 4 can be seen in Figure 85. First, the host reads the hyperspectral image. Then, the host compiles the OpenCL code to execute on the MPPA. Afterwards, the host is ready to invoke kernels to execute
the algorithm. It is important to highlight that the host is the responsible to create the buffers that will be used as inputs to the kernels. All of them allocates data in global memory. What follows describes more in details the different steps that can be observed in Figure 85.

**PCA step 1, Average calculation:** The first step invokes the **average** kernel (see Figure 83) with the buffers shown in the Figure 84.

```c
_kernel void average(__global float *image_in,
                   __global float *imagePrima)
```

Figure 83 Kernel average prototype

```c
buf_in_image_in = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                  VECTOR_LENGTH*sizeof(float), NULL, NULL);
buf_out_imagePrima = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                                    VECTOR_LENGTH*sizeof(float), NULL, NULL);
err = clEnqueueWriteBuffer(queue, buf_in_image_in, CL_TRUE,
                           0, VECTOR_LENGTH*sizeof(float), image_in, 0, NULL, &sendImage1);
```

Figure 84 Buffers used with average kernel

Each kernel computes the average of each band of the image. Thus, to execute the **clEnqueueNDRangeKernel**, the **global_size** variable will be set to 224, i.e. the number of bands of the image. After the **average** kernel execution, the averaged band results are obtained from the buffer **image_prima**. In turn, this buffer becomes the input to the next step.

During the implementation of this kernel, one performance key factor has been discovered. In order to improve the performance of the kernel, the main part of the computation should be implemented with private variables (declared inside the kernel function). Then, when the computation has been completed, the data should be copied to the buffers. As can be seen in Figure 86, which corresponds to the covariance kernel code, the multiplication process only employs the input **B** (which corresponds to **image_prima**) and the results are stored into local variables. Then, the row processed in this kernel and stored into the local array **products** is copied to the buffer **C**. Using this procedure the buffer is managed more efficiently. In order to do so, private variables should be small enough to allocate them within a 4KB page. This is a restriction of Kalray OpenCL implementation on the MPPA. After the computation finishes, the resulting covariance matrix can be retrieved from the output buffer using the command **clEnqueueReadBuffer** and store in a vectorized form into the variable **covariance**, whose size is 224 x 224.
Figure 85 PCA execution flow with OpenCL
Chapter 4 – Application development

```cpp
__kernel void covariance(__global double* C,
                         __global float* B) {
    int k, p;
    double products[224];
    double aux;
    unsigned int id = get_global_id(0);

    if (id >= BANDS)
        return;

    /* INITIALIZE THE ARRAYS */
    for (k = 0; k < BANDS; k++) {
        products[k] = 0.0;
    }

    /* PERFORMING THE COMPUTATION */
    for (k = 0; k < BANDS; k++) {
        aux = 0;
        for (p = 0; p < PIXELS; p++) {
            aux += (double) B[PIXELS * id + p] * (double) B[(k * PIXELS) + p];
        }
        products[k] = aux;
    }

    /* COPYING THE RESULTS TO BUFFER */
    for (k = 0; k < BANDS; k++) {
        C[id * BANDS + k] = products[k];
    }
}
```

Figure 86 Covariance kernel

**PCA step 3. Eigenvalue and eigenvector decomposition:** The goal of the third step is to obtain the eigenvalues and eigenvectors of the covariance matrix. Since the selected method to perform this computation is the Jacobi cyclic algorithm, it is imperative to use an iterative procedure until the convergence stop criteria is achieved. To do so, two different kernels are utilized to implement equation (12), which allows to approximate the off-diagonal values to zero. The first kernel computes the left part of the matrix multiplication and the second one computes the right part. The complete procedure for this step is the following. First, the host sets up the environment, creates the kernels and declares the required buffers (see Figure 87). It is worth noting that initially no data is written to the buffers.

```cpp
/* CREATE THE BUFFERS */
cl_mem buf_in_out_image_in_jacobi;
buf_in_out_image_in_jacobi = clCreateBuffer(context, CL_MEM_READ_WRITE, BANDS * BANDS * sizeof(double), NULL, NULL);
cl_mem buf_in_in_jacobi;
buf_in_in_jacobi = clCreateBuffer(context, CL_MEM_READ_WRITE, BANDS * BANDS * sizeof(double), NULL, NULL);
cl_mem buf_in_jacobi;
buf_in_jacobi = clCreateBuffer(context, CL_MEM_READ_WRITE, BANDS * BANDS * sizeof(int), NULL, NULL);
cl_mem buf_in_out_matrixP;
buf_in_out_matrixP = clCreateBuffer(context, CL_MEM_READ_WRITE, BANDS * BANDS * sizeof(double), NULL, NULL);
```

Figure 87 Buffer creation in step 3 of the PCA algorithm
Given that the same variables need to be read and written to execute the algorithm (the covariance and rotation matrixes), the corresponding buffers, i.e. `buf_in_out_image_in_jacobi` and `buffer_in_out_matrixP`, are read and written by the kernels using the flag `CL_MEM_READ_WRITE`. Once the environment is set, the host enters a loop which ends when the convergence criterion is reached. Within the loop, the first task the host performs is to look for possible off-diagonal values to be zeroed, considering the parallelism conditions shown in Figure 59. If there are no more elements to be zeroed, the code jumps to find the eigenvalues and eigenvectors. Otherwise, the host invokes the first Jacobi kernel. The host writes all required data to the buffers and determines the number of kernels that shall be executed in parallel based on the number of elements to be zeroed. In other words, the global size for the command `clEnqueueNDRangeKernel` is determined by the number of elements to be zeroed. The `global_size` has a maximum value defined by the constant `JACOBI_CONCURRENT_ITEMS`. This maximum value is bounded by `BANDS/2`, since this is the maximum number of elements that can be zeroed in parallel. Additionally, `global_size` must be a multiple of 16. The code checks at every loop this restriction. If the number of zeroed elements is not a multiple of 16, the value is rounded up to the following multiple value.

Once all the required arguments have been set, the host triggers the first kernel, `Jacobi_method1` (see Figure 88). The arguments of the `Jacobi_method1` kernel are: (1) `matrix_in_jacobi`, which is the covariance matrix, (2) `sinA_array` and `cosA_array`, which are elements required to implement equation (7), (3) the `i_jacobi` and `j_jacobi`, which are the indexes of the element to be zeroed and finally, (4) `limit`, which is the number of elements to be zeroed in this iteration.

```
__kernel void jacobi_method1(
  __global double* matrix_in_jacobi,
  __global double* sinA_array,
  __global double* cosA_array,
  __global int* i_jacobi,
  __global int* j_jacobi,
  int limit)
```

Figure 88 Kernel Jacobi_method1 prototype

The output of the first kernel execution updates the input argument `matrix_in_jacobi`. It is read from the buffer by the host using the `clEnqueueReadBuffer` command. Then, to execute the second kernel of this step, the host writes the updated data to the buffers `buf_in_out_image_in_jacobi` and `buf_in_out_matrixP`. Then, the host assigns the arguments to the kernel `jacobi_method2` and, using the same global size
value as that of the first kernel, invokes the second one using the command `clEnqueueNDRangeKernel`.

```c
__kernel void jacobi_method2(
    __global double* matrix_in_jacobi,
    __global double* sinA_array,
    __global double* cosA_array,
    __global int* i_jacobi,
    __global int* j_jacobi,
    __global double* matrix_P,
    int limit)
```

Figure 89 Kernel Jacobi_method2 prototype

The output of the second kernel is an updated version of covariance and rotation matrixes which are used in the next iteration as inputs for the algorithm. This procedure is repeated until there is no off-diagonal elements higher than the stop condition. When the stop condition has been reached, the modified covariance matrix is read to get the eigenvalues from the diagonal elements. Then, the eigenvalues are ordered and the corresponding eigenvectors are obtained from the accumulated matrix P, which have been computed at the last iteration of the algorithm.

When OpenMP and OpenCL implementations are compared, it is worth noting to highlight that in OpenCL there is no available methods for synchronization between working-groups. However, to use the equivalence of the barrier applied in OpenMP, the invocation of the second kernel implies some kind of synchronization point which is required for the correct execution of the code.

**PCA steps 4 and 5, projection and reduction:** Finally, the last step is made with one last kernel named projection. Using the same principle mentioned in the OpenMP/POSIX implementations, the algorithm preserves only 4 bands of the original hyperspectral image. Thus, the last step consists on the multiplication of the original image by the first four eigenvectors computed in the last step.

The projectionVer2 kernel (see Figure 90) has been implemented applying the same principles employed with covariance kernel. Briefly speaking, this kernel multiplies the vectorized matrix A (original hyperspectral image) and B (eigenvectors) to compute C, the reduced version of the original hyperspectral image of size PIXELS x BANDS_PCA.
Each kernel processes a row of the resulting matrix (4 elements in parallel). Thus, the global size value in this case is set as the number of pixels. The execution is invoked by the `clEnqueueNDRangeKernel` command and the results are collected from the buffer assigned to the variable C of the kernel.

Once the last step is completed, the reduction of the image is finished, this implementation produces similar results to the ones obtained with OpenMP and POSIX. However, if we compare all of these methodologies is clear to see that the OpenCL API handles the communication overhead. In addition, with its memory model, the OpenCL API provides a path to overcome the shared memory limitation found in the clusters when OpenMP/POSIX have been used.

Although the OpenCL standard offers instructions like the bi-dimensional `clEnqueueNDRange`, those are not currently implemented in the available SDK on the MPPA. Besides, the OpenCL implementation on Kalray MPPA is not on its final release yet, and thus their development team continuously publish newer releases of the SDK. With that in mind, next chapter covers the implementation performance achieved in terms of speed up.
5 Analysis of results
This chapter presents the results obtained after implementing the algorithms explained in chapter 4. The results focus on the execution time of the parallelized algorithms and consequently the speed-up versus sequential implementations. This analysis will be covered using a three-way comparison on the execution time: (1) sequential implementation, (2) POSIX version and (3) OpenMP/OpenCL implementations.

### 5.1. Validation of the algorithms

The algorithms presented in this document have been introduced in previous research works conducted by R. Lazcano in [1] and [40] for the PCA algorithm and by D. Madroñal in [39] for the SVM algorithm. This work sets them as ground-truth for further analysis and comparison.

Thus, in order to validate the parallel implementation using OpenMP and OpenCL, the output of the programs have been compared to the reference ones using a file comparison tool (e.g. Meld\(^23\)). The OpenMP and OpenCL programs have been only accepted when both results were exactly the same under the same scenario configuration. Since the main objective of this work is to measure the execution time using OpenMP and OpenCL to figure out if any speed-up can be achieved, the reference programs have been tested with different configurations to define the speed-up achieved. This analysis is introduced in the following section.

### 5.2. Time analysis

This section presents a comparison of the execution time of the applications in terms of speed-up. All experiments are repeated 10 times and the results provide the average time. The execution time of all experiments conducted in this research work can be found in Annex III.

#### 5.2.1. Sequential implementation

The sequential execution time is acquired through the implementation of the SVM and PCA algorithms using a thread of one cluster of the MPPA Bostan at nominal speed (600MHz). This scenario allows getting a baseline to compare it with further parallel implementations.

---

\(^{23}\) [http://meldmerge.org](http://meldmerge.org)
For the case of SVM, Table 9 shows the sequential execution time of it. Only the total time is depicted because all steps are related among them and cannot be measured independently.

Table 9 SVM sequential execution time

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Time (ms)</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read image and writing results</td>
<td>3222.53</td>
<td>66.86 %</td>
</tr>
<tr>
<td>Classification</td>
<td>1228.63</td>
<td>25.49 %</td>
</tr>
<tr>
<td>Total execution time</td>
<td>4819.63</td>
<td>-</td>
</tr>
</tbody>
</table>

On the other hand, Table 10 depicts the sequential implementation of PCA algorithm. It is important to highlight that steps 2 and 3 of the implementation are the ones with the highest time to complete. Moreover, steps 4 and 5 have been measured together because the projection of the image into the new subspace was made using only the required number of bands to achieve the dimensionality reduction.

Table 10 Sequential execution time of the PCA algorithm

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Time (ms)</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>58.12</td>
<td>1.87 %</td>
</tr>
<tr>
<td>Step 2</td>
<td>557.95</td>
<td>17.99 %</td>
</tr>
<tr>
<td>Step 3</td>
<td>2412.11</td>
<td>77.76 %</td>
</tr>
<tr>
<td>Step 4 and Step 5</td>
<td>65.91</td>
<td>2.12 %</td>
</tr>
<tr>
<td>Total processing time</td>
<td>3102.00</td>
<td>-</td>
</tr>
</tbody>
</table>

5.2.2. **SVM: OpenMP**

As it has been previously explained, OpenMP exploits the parallelism using several concurrent threads. In this test scenario, the number of OpenMP threads handled by the *fork-join* process is defined. The maximum number of threads available is 16. Note that each thread is assigned to a core of the compute clusters.

To compare the results of this proposal the OpenMP execution time is compared with that of the sequential implementation (see Table 9). The test scenario consists in classifying a hyperspectral image using three methodologies (1) POSIX with 1 cluster, 24

---

24 Total processing time includes the time used within the communication which was not measured separately in this experiment.
named POSIX-1 (2) OpenMP with 1 cluster, named OpenMP-1 and (3) OpenMP with 13 clusters, named OpenMP-13. The resulting execution times are shown in Table 11.

Table 11 Execution time of the SVM parallelized algorithm

<table>
<thead>
<tr>
<th>Threads</th>
<th>POSIX-1</th>
<th>OpenMP-1</th>
<th>OpenMP-13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1227.42</td>
<td>1237.28</td>
<td>493.98</td>
</tr>
<tr>
<td>2</td>
<td>832.61</td>
<td>835.24</td>
<td>463.53</td>
</tr>
<tr>
<td>3</td>
<td>701.71</td>
<td>702.31</td>
<td>453.57</td>
</tr>
<tr>
<td>4</td>
<td>636.66</td>
<td>635.79</td>
<td>448.24</td>
</tr>
<tr>
<td>5</td>
<td>597.79</td>
<td>596.17</td>
<td>445.07</td>
</tr>
<tr>
<td>6</td>
<td>572.9</td>
<td>569.50</td>
<td>442.90</td>
</tr>
<tr>
<td>7</td>
<td>556.18</td>
<td>553.17</td>
<td>442.01</td>
</tr>
<tr>
<td>8</td>
<td>540.92</td>
<td>536.61</td>
<td>440.65</td>
</tr>
<tr>
<td>9</td>
<td>531.44</td>
<td>527.05</td>
<td>440.13</td>
</tr>
<tr>
<td>10</td>
<td>522.44</td>
<td>517.39</td>
<td>438.84</td>
</tr>
<tr>
<td>11</td>
<td>517.14</td>
<td>510.76</td>
<td>438.73</td>
</tr>
<tr>
<td>12</td>
<td>511.21</td>
<td>503.92</td>
<td>437.93</td>
</tr>
<tr>
<td>13</td>
<td>508.75</td>
<td>501.36</td>
<td>437.82</td>
</tr>
<tr>
<td>14</td>
<td>506.02</td>
<td>498.13</td>
<td>437.81</td>
</tr>
<tr>
<td>15</td>
<td>506.74</td>
<td>491.43</td>
<td>437.31</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>488.25</td>
<td>436.77</td>
</tr>
</tbody>
</table>

Figure 91 plots the previous results. When the execution time of the POSIX-1 implementation is compared with that of the OpenMP-1, it can be seen that there is almost no difference for any number of threads. On the other hand, when POSIX-1 is compared with OpenMP-13, the OpenMP implementation runs significantly faster. On top of that, when the number of threads is greater than eight, it can be observed that the execution time of the POSIX-1 implementation decreases substantially and approaches that of the OpenMP-13.
Figure 91 Execution times of SVM algorithm

To show other perspective of the aforementioned behavior, Figure 92 compares the speed-up of the parallelized SVM algorithm with regard to the sequential implementation. As can be seen, from 2 to 15 threads, POSIX-1 and OpenMP-1 achieve almost the same speed-up. Note that OpenMP-1 is able to employ an additional thread (i.e., 16). Consequently, its speed-up is higher than that of POSIX-1. Moreover, using 13 clusters the OpenMP implementation can achieve its maximum speed-up with 8 threads. It is worth noting that the difference between this speed-up and OpenMP implementation using 16 threads with one cluster is 0.3.
The previous results shows that the OpenMP implementations are capable to exploit the parallelism of the MPPA. Moreover, from the point of view of the speed-up, they are better than the POSIX implementation. The major improvement has been achieved when the hyperspectral image is divided to be processed onto 13 clusters. The speed-up achieved in this case is 2.81. However, given that this implementation employs 13 out of 16 clusters of the MPPA and considering that the OpenMP-1 implementation achieves a speed-up of 2.51, the difference (approximately, 40 milliseconds) is negligible.

The results show that increasing the number of clusters does not proportionally increase the speed-up of the implementation. Effectively, sending data to the clusters from the I/O subsystem implies a time penalty. For instance, the classification of 240 pixels only takes 0.7 ms (measurement confirmed with the three methods) while data communication to/from the cluster takes the time left.

After this analysis, it is possible to conclude that for complex tasks the parallelization effort to achieve a certain speed-up might be better focused on a reduced number of clusters, leaving the place to compute other tasks in the remaining clusters.

5.2.3. SVM: OpenCL

As it has been previously explained, OpenMP exploits the parallelism using several concurrent threads. In this test scenario, the number of OpenMP threads handled by the fork-join process is defined. The maximum number of threads available is 16. Note that each thread is assigned to a core of the compute clusters.

Unlike OpenMP and POSIX, OpenCL sets neither the number of threads/cores to execute nor the communication method, i.e. unicast, multicast, multiplex to one or multiple cores. Thus, only the execution time of one kind of setup, which is decided by the OpenCL compiler, can be measured. The execution time of the algorithm consists of (1) the time of writing to a buffer (one or multiple ones), (2) the processing time, i.e. the execution time of clEnqueueNDRangeKernel, and (2) the time of reading from a buffer. To complement the total execution time of an OpenCL implementation, configuration and compiling times have been also measured.

Table 12 collects the average time of ten instances of the program. The execution time of the decision and projection stages and the total execution time has been measured using profiling. On the other hand, configuration and profiling times have been
measured using time measuring techniques in C (the same procedure used in OpenMP/POSIX implementations) because of the impossibility to profile this steps.

Table 12 SVM execution time with OpenCL

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Time (ms)</th>
<th>Total time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration and Compiling</td>
<td>1056.84</td>
<td>1056.84</td>
</tr>
<tr>
<td>Stage Projection</td>
<td>110.03</td>
<td>113.85</td>
</tr>
<tr>
<td>Stage Decision</td>
<td>3.82</td>
<td></td>
</tr>
</tbody>
</table>

As can be observed in Table 12, the configuration and compiling time is the greatest of all measured times (1056.84 ms). However, the execution times of the projection and decision stages amount a total value of 113.85 ms, i.e. a speed-up of 10.79 compared to the sequential implementation.

Table 13 Send, read, process times of SVM kernel

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Time (ms)</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (clEnqueueWriteBuffer)</td>
<td>47.11</td>
<td>41.38%</td>
</tr>
<tr>
<td>Read (clEnqueueReadBuffer)</td>
<td>0.40</td>
<td>0.34%</td>
</tr>
<tr>
<td>Process (clEnqueueNDRangeKernel)</td>
<td>66.37</td>
<td>58.28%</td>
</tr>
<tr>
<td>Total Execution</td>
<td>113.86</td>
<td></td>
</tr>
</tbody>
</table>

Table 13 collects the writing, reading and kernel invocation execution times. As can be seen, almost 41% of the time is employed to send and receive data. The reading procedure is much faster than the writing one since only one output buffer is utilized in each of the kernels which conform the SVM OpenCL implementation. Note that the execution time of the writing procedure is greater than that of the reading one because every kernel used several buffers to complete its computation. As a conclusion, OpenCL extracts efficiently the parallelism of the algorithm and manage correctly the communications to obtain a certain speed-up when compared to a POSIX or OpenMP implementation.

5.2.4. **PCA: OpenMP**

To compare the total execution times of the sequential and parallel solutions, Table 10 and Table 14 are contrasted. Under the same scenario configuration, on the one hand Table 10 shows the sequential times of each step of the PCA algorithm. On the other hand, Table 14 shows the execution time of the different stages using OpenMP with the best scenario configuration (step 1, using one thread in 16 clusters; steps 2, 4
and 5, 14 threads in 16 clusters; and step 3, up to 16 threads in one cluster). The OpenMP implementation achieves a speed-up of 2.52 with regard to the sequential one. Note that step 1, band average removal, has not been parallelized with OpenMP due to its negligible processing time. However, the speed-up achieve in steps 2, 3 and 4/5 is 3.67, 2.49 and 1.32, respectively.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Time (ms)</th>
<th>Percentage</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>58.04</td>
<td>4.71 %</td>
<td>-</td>
</tr>
<tr>
<td>Step 2</td>
<td>152.01</td>
<td>12.33 %</td>
<td>3.67</td>
</tr>
<tr>
<td>Step 3</td>
<td>968.05</td>
<td>78.50 %</td>
<td>2.49</td>
</tr>
<tr>
<td>Step 4 and Step 5</td>
<td>49.79</td>
<td>4.04 %</td>
<td>1.32</td>
</tr>
<tr>
<td>Total processing time</td>
<td>1233.12</td>
<td>-</td>
<td>2.52</td>
</tr>
</tbody>
</table>

The previous results shows that OpenMP is capable to exploit the available hardware parallelism. To do so, OpenMP allocates threads among cores to share the workload. The result is a faster implementation than the sequential one.

Moreover, to probe the capacity of OpenMP to exploit the inherent parallelism, an experiment with different values of the Jacobi convergence condition has been conducted. As can be seen in Figure 93, different number of threads and values of $\epsilon$ have been tested. The average speed-up for ten instances of each experiment is shown in Figure 93.

25 Total processing time is measured in the Host. Thus, it includes the time used to transmit/receive data to/from the MPPA.
Figure 93 OpenMP PCA speed-up

As can be seen in Figure 93, the smaller the convergence condition, the greatest the speed-up obtained. The reason for this is the behavior of the cyclic Jacobi algorithm in step 3. When it is smaller, more iterations are needed to reach the convergence condition. Besides, note that increasing the number of threads does not always reduce the total execution time. As can be seen in Table 15, the minimum total execution time for certain values of the convergence condition is achieved with less than 14 threads.

Table 15 OpenMP PCA Execution time

<table>
<thead>
<tr>
<th>Number of OpenMP threads</th>
<th>$\varepsilon=1\times10^{-4}$</th>
<th>$\varepsilon=1\times10^{-5}$</th>
<th>$\varepsilon=1\times10^{-6}$</th>
<th>$\varepsilon=1\times10^{-8}$</th>
<th>$\varepsilon=1\times10^{-16}$</th>
<th>$\varepsilon=1\times10^{-20}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.19</td>
<td>3.33</td>
<td>8.08</td>
<td>201.67</td>
<td>235.27</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.90</td>
<td>2.41</td>
<td>5.54</td>
<td>135.56</td>
<td>154.24</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1.44</td>
<td>2.13</td>
<td>4.19</td>
<td>96.76</td>
<td>154.24</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.70</td>
<td>1.94</td>
<td>3.40</td>
<td>64.99</td>
<td>76.05</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.46</td>
<td>2.14</td>
<td>3.23</td>
<td>54.62</td>
<td>62.43</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.47</td>
<td>1.59</td>
<td>3.49</td>
<td>45.12</td>
<td>55.70</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1.55</td>
<td>1.49</td>
<td>3.10</td>
<td>38.37</td>
<td>44.64</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.24</td>
<td>1.87</td>
<td>2.78</td>
<td>35.96</td>
<td>42.70</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1.30</td>
<td>1.75</td>
<td>2.65</td>
<td>37.98</td>
<td>41.22</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.29</td>
<td>1.81</td>
<td>2.41</td>
<td>32.03</td>
<td>36.25</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1.24</td>
<td>1.55</td>
<td>2.21</td>
<td>27.24</td>
<td>32.74</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1.32</td>
<td>1.70</td>
<td>2.67</td>
<td>23.88</td>
<td>30.53</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1.29</td>
<td>1.54</td>
<td>2.12</td>
<td>23.26</td>
<td>27.78</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1.15</td>
<td><strong>1.23</strong></td>
<td>2.48</td>
<td><strong>20.32</strong></td>
<td>25.56</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1.28</td>
<td>1.65</td>
<td>2.14</td>
<td>22.90</td>
<td>26.30</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.11</td>
<td>1.42</td>
<td>2.19</td>
<td>21.21</td>
<td>24.97</td>
<td></td>
</tr>
</tbody>
</table>
The following paragraphs compare an OpenMP implementation against a POSIX one. For the POSIX implementation the results obtained in [40] using a convergence criterion of $\epsilon = 1 \times 10^{-5}$ for step 3 of the implementation are considered as reference. As can be seen in Figure 94 the OpenMP implementation is always faster than the POSIX one. The reasons for this are:

- Unlike the POSIX implementation, all matrix multiplication procedures implemented with OpenMP in this work use all the cores available within a cluster on the MPPA. Since the algorithm requires two matrix multiplications, this reduces the total execution time with regard to the POSIX implementation where only one core per cluster is used.

- OpenMP allows using the master thread to share the work. In this manner, more resources are available to compute tasks. The POSIX implementation of the eigenvectors decomposition utilizes up to 15 threads whereas the OpenMP implementation supports 16.

![Figure 94 POSIX and OpenMP comparison](image)

Table 16 compares the execution times of step 3 using OpenMP and POSIX. As can be seen there is almost no difference in both implementations. Thus, it can be concluded that the better performance of OpenMP is due to the optimized matrix multiplications in steps 2 and step 5 of PCA algorithm.
### Table 16 Execution time of PCA step 3 using OpenMP and POSIX

<table>
<thead>
<tr>
<th>Execution</th>
<th>OpenMP</th>
<th>POSIX</th>
<th>Difference</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1269.77</td>
<td>1279.77</td>
<td>10.00</td>
<td>1.01</td>
</tr>
<tr>
<td>2</td>
<td>1279.77</td>
<td>1280.57</td>
<td>0.79</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>1269.76</td>
<td>1269.77</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>1269.77</td>
<td>1270.57</td>
<td>0.80</td>
<td>1.00</td>
</tr>
<tr>
<td>5</td>
<td>1269.77</td>
<td>1270.57</td>
<td>0.80</td>
<td>1.00</td>
</tr>
<tr>
<td>Min</td>
<td>1269.76</td>
<td>1269.77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>1271.77</td>
<td>1274.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>1279.77</td>
<td>1280.57</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 5.2.5. **PCA: OpenCL**

As it has been mentioned in section 5.2.3, the execution times measured with OpenCL are obtained using the profiling mode. Specifically, all steps of the PCA implementation with OpenCL have been measured independently, i.e. on-line compiling time, average, covariance, eigenvalues decomposition, projection and reduction kernels.

Unlike the implementation made with OpenMP/POSIX, the OpenCL implementation does not allow setting the number of threads utilized in the application. This decision is made by the OpenCL compiler. The only configurable value for this purpose is the `JACOBI_CONCURRENT_ITEMS` parameter, which allows configuring the number of elements the eigenvalue decomposition implementation (step 3) zeroes in parallel.

Table 17 shows the best results obtained with OpenCL and compares them with the sequential and OpenMP implementations. As can be seen, the step 1 obtains a 3.35 speed-up with regard to the sequential implementation. There is no available comparison with OpenMP since this step has not been parallelized using this technique. The step 2 achieves a speed-up of 4.95 and 1.35 compared with the sequential and OpenMP implementations, respectively. The steps 4 and 5 have a speed-up of 1.70 when compared with the sequential implementation and 1.29 times faster than the OpenMP implementation. However, for the step 3, the OpenMP implementation is 2.73 faster than the one made with OpenCL, i.e. the speed-up of OpenCL with regard to OpenMP is 0.37. This experiment has been conducted setting the `JACOBI_CONCURRENT_ITEMS` parameter to 15.
Chapter 5 – Analysis of results

Table 17 PCA OpenCL concurrent elements to be zeroed = 15

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Time (ms)</th>
<th>Percentage</th>
<th>Speed-up sequential</th>
<th>Speed-up OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiling</td>
<td>2154.25</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Step 1</td>
<td>17.33</td>
<td>0.62 %</td>
<td>3.35</td>
<td>3.35</td>
</tr>
<tr>
<td>Step 2</td>
<td>112.33</td>
<td>4.00 %</td>
<td>4.97</td>
<td>1.35</td>
</tr>
<tr>
<td>Step 3</td>
<td>2640.35</td>
<td>94.01 %</td>
<td>0.91</td>
<td>0.37</td>
</tr>
<tr>
<td>Step 4 and Step 5</td>
<td>38.66</td>
<td>1.98 %</td>
<td>1.70</td>
<td>1.29</td>
</tr>
<tr>
<td>Total processing time</td>
<td>2808.68</td>
<td>-</td>
<td>1.10</td>
<td>0.44</td>
</tr>
</tbody>
</table>

This results show that the OpenCL implementation for the PCA is 1.10 times faster than the sequential one but 2.28 times slower than that of OpenMP. This behavior is constant for different values of epsilon and JACOBI_CONCURRENT_ITEMS.

Table 18 and Table 19 collects the execution time of different configurations of the OpenCL PCA algorithm. As expected, changing the number of concurrent elements mainly affects the step 3 of the algorithm. As can be seen, there is a drastic reduction in the processing time when the number of concurrent elements to be zeroed varies from 16 to 112. The explanation of this behavior is the following, the goal of the algorithm is to find as many elements to set to zero as possible. However, the maximum number of concurrent elements to be zeroed in the Jacobi procedure items is found in every iteration. Consequently, the algorithm may execute a higher number of kernels than possible elements to be zeroed, therefore, it may use inefficiently the computing resources.

Table 18 PCA OpenCL concurrent elements to be zeroed = 16

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Time (ms)</th>
<th>Percentage</th>
<th>Speed-up sequential</th>
<th>Speed-up OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiling</td>
<td>2311.16</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Step 1</td>
<td>17.26</td>
<td>0.6 %</td>
<td>3.36</td>
<td>3.36</td>
</tr>
<tr>
<td>Step 2</td>
<td>112.43</td>
<td>3.91 %</td>
<td>4.96</td>
<td>1.35</td>
</tr>
<tr>
<td>Step 3</td>
<td>2705.01</td>
<td>94.13 %</td>
<td>0.89</td>
<td>0.36</td>
</tr>
<tr>
<td>Step 4 and Step 5</td>
<td>38.95</td>
<td>1.36 %</td>
<td>1.69</td>
<td>1.28</td>
</tr>
<tr>
<td>Total processing time</td>
<td>2873.69</td>
<td>-</td>
<td>1.08</td>
<td>0.43</td>
</tr>
</tbody>
</table>
Table 19 PCA OpenCL concurrent elements to be zeroed = 112

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Time (ms)</th>
<th>Percentage</th>
<th>Speed-up sequential</th>
<th>Speed-up OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiling</td>
<td>2146.48</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Step 1</td>
<td>17.13</td>
<td>0.17 %</td>
<td>3.39</td>
<td>3.39</td>
</tr>
<tr>
<td>Step 2</td>
<td>112.31</td>
<td>1.11 %</td>
<td>4.97</td>
<td>1.35</td>
</tr>
<tr>
<td>Step 3</td>
<td>9959.36</td>
<td>98.34%</td>
<td>0.24</td>
<td>0.10</td>
</tr>
<tr>
<td>Step 4 and Step 5</td>
<td>39.11</td>
<td>0.39 %</td>
<td>1.69</td>
<td>1.27</td>
</tr>
<tr>
<td>Total processing time</td>
<td>10127.92</td>
<td>-</td>
<td>0.31</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Observing the results, it is clear to see that the steps of the algorithm which implies only one kernel invocation (steps 1, 2, 4 & 5) executes faster than in the OpenMP implementation. However, step 3, executes iteratively the required number of times up to a convergence criterion is reached. To do so the algorithm is developed with two kernels that performs two different operations. To execute the first kernel a group of buffers (inputs of the kernel) must be written from the global memory of the host. When the operation is done, the algorithm reads the results from the device and store them into global memory. Afterwards, the buffers for the second kernel are written and the arguments for its execution set. Once the computation of the second kernel is finished and the results to store in global memory are read, the variables are set as inputs of the next iteration. As can be seen, there is a read/write procedure to pass the results from one kernel to the other, what implies to access the global memory. This fact is the reason of the greater execution time of this stage of the PCA algorithm. Note that due to the specific implementation of the OpenCL API on the MPPA, there is no possibility to update data within buffers and reuse the same buffers into different kernels.

As a conclusion of this fact, it can be stated that OpenCL works fine on the MPPA just in case there is only one kernel invocation from the host to the device.

5.2.6. **Comparison**

In this section the implementations made with POSIX, OpenMP and OpenCL are compared among them.

First, the SVM algorithm is studied. Figure 95 shows the execution time comparison for the different implementations whereas Figure 96 shows their speed-up. As can be seen both OpenMP and OpenCL execute faster than the sequential implementation. However, it is worth noting to highlight that OpenCL performs better than OpenMP monocluster and OpenMP multicluster (13 clusters). The speed-up achieved with OpenCL is 10.79. This value of speed-up are obtained because the classification
process is completely independent, i.e. each pixel of the image can be processes without affecting any other.

![Figure 95 SVM execution time comparison](image1)

Figure 95 SVM execution time comparison

![Figure 96 SVM speed-up time comparison](image2)

Figure 96 SVM speed-up time comparison

On the other hand, OpenCL and OpenMP behaves differently when they are utilized with the PCA algorithm. For this specific algorithm, OpenMP performs better than OpenCL. Figure 97 and Figure 98 shows the execution time and speed-up of the PCA
implementation, respectively. As has been indicated in previous sections, the step which requires more computational effort (critical step) is the eigenvalue decomposition (step 3). The performance of the whole application is determined by that of step 3. Although OpenCL is faster in every other steps of the algorithm, the total performance becomes lower than that of OpenMP because the critical step is the slowest one of all implementations. The main reason for this is the global memory accesses during consecutive kernel invocations. In contrast, OpenMP, which allows the developer a better control on how to handle threads and data in step 3, achieves a better performance. Figure 98 shows that OpenMP is up to 2.52 times faster than a sequential implementation using a convergence criterion of $\varepsilon = 1 \times 10^{-5}$.

![Figure 97 PCA execution time comparison](image)

*The lower the better
As a conclusion of this analysis, to reach real time constrains, OpenCL is strongly suggested to be used to process the SVM algorithm, whereas OpenMP should be used with the PCA algorithm.

### 5.3. Coding methodology

In this section, the coding strategies and similarities about POSIX, OpenMP and OpenCL are analyzed.

In a POSIX implementation the developer is responsible of defining explicitly the parallelism, splitting the work among threads and joining them to recover the result of a computation. OpenMP eases the work with its API constructors. OpenMP introduces pragmas to manage code sections where parallelism can be exploited. The API itself generates the required number of threads for sections and for loops. The OpenMP API is responsible of the creation, joining and destruction of threads. Thus, these steps are not handled by developer. Moreover, OpenMP includes synchronization among threads using sections or barriers which can replace semaphores or mutexes in a POSIX implementation. However, not all configurations can be automated by OpenMP. For instance, in the MPPA Developer workstation the communications methods between the host and the MPPA remain the same in both coding schemes. Actually, the developer is still responsible of delivering accurately the data to the processing elements (cores). As a conclusion, although OpenMP and POSIX have many commonalities, the OpenMP constructors, substantially ease the coding of parallel tasks.

![Figure 98 PCA speed-up time comparison](image-url)

*The higher the better*
On the other hand, OpenCL introduces a complete different method of exploiting the parallelism. Since its philosophy is based on the concept of heterogeneous computing, threads are not explicitly managed by developers. This scheme is based in the execution of kernels, i.e. pieces of codes that execute on the processing elements but without developer’s control. The API is responsible of queueing in parallel the execution of the kernel. Kernels are executed as many times as required to complete the computation. The only requirement that developers must define is the number of local resources within a workgroup (e.g., Kalray local size is 16) and the global size, which defines the number of times a kernel is executed. In this way, developers do not control the loop but the API itself does. Kernels are not a new concept to developers used to C language. The idea can be easily compare to that of function prototype. Therefore, the coding method is similar in many ways to standard C. Another point which is completely different in OpenCL is the memory model. The definitions of global, local and private memory are hardware related. Thus, the memory used to store data affects drastically the performance (global slower $\rightarrow$ private faster). While using POSIX and OpenMP the challenge is (1) sending data from the host to the processing elements by means of portals and (2) allocating data within the limited memory size of clusters, OpenCL eases significantly this task. The API introduces buffers to send data from global memory to the device (MPPA) which allows storing and sending data when needed. For the specific case of this work, OpenCL avoids splitting the image since it can be directly read from the buffer. OpenCL takes data from the host and delivers data to processing elements using communications which are handled by the API without user intervention. Finally, it is worth noting to recall that synchronization in OpenCL is not an easy task since only syncing inside a work-group (cluster) is allowed. As an OpenCL drawback, syncing among threads is not possible.

At last, moving from a POSIX or even a sequential implementation to OpenMP does not required big changes if the sections to parallelize are identified. OpenMP API focus on handling parallel threads automatically and synchronizing the work in order to ease the developer’s job. In contrast, OpenCL requires to create a completely different program to exploit the parallelism. Therefore, developers must learn this methodology. In OpenCL, developers utilize C-like statements to employ API’s methods to handle communications, buffers and kernel invocations. Synchronization is not intended to be achieved among all the kernels in parallel. Thus, OpenCL is focused to process total independent tasks. Briefly speaking, OpenCL provides a higher abstraction level but leaves less control to developers.
6 Conclusions and future lines
6.1. Conclusions

After the OpenMP/OpenCL implementation of the SVM and PCA algorithms onto the MPPA, the following statements are considered relevant to highlight:

- Based in the tests developed in this research, it has been proven that OpenMP and OpenCL API helps developers to “automate” the parallelism. Each of them has its own behavior and methodologies. Briefly, OpenMP creates, syncs and destroys threads easily, while OpenCL handles communications and execution between the host and the device with its own commands, adding a complete abstraction level to the developer (and consequently, losing control of the program as well). Both, OpenMP and OpenCL work on the MPPA. However, it is worth noting that OpenCL is limited because of the actual implementation of the manufacturer SDK.

- In terms of coding, an OpenMP implementation is similar to a POSIX one, since OpenMP replaces a set of function calls to the Pthread library with a for loop. In this way, OpenMP avoids the creation, joining and destruction of threads. In terms of execution speed, since OpenMP allows using all the resources of the platform, it is possible to achieve greater speed-ups, making the application to run faster on the same architecture.

- Based in the tests made in this research work, it goes without saying that OpenMP and OpenCL works to exploit “automatically” the parallelism of a manycore architecture device. On one hand, OpenMP is typically related to a manual methodology and, in brief, it is easier to implement an OpenMP application from a sequential code. On the other hand, OpenCL might be faster. However, its own coding methodology and memory model need to be learnt. Thus, there is a tradeoff. For efficient implementations, developers must learn OpenCL and realize how to tune kernels in order to exploit the parallelism of the device. The fundamental idea behind OpenCL is its independence from the hardware platform and operating systems. This universality and portability has a cost. The cost is greater programming effort and longer learning curve.

- A speed-up of 2.51 (488.24ms) and 2.81 (436.77 ms) have been obtained using OpenCL to implement the SVM algorithm with 1 and 13 clusters, respectively. However, a speed-up of 10.79 (113.84 ms) has been achieved using OpenCL. Hence, there is a big difference between the speed-ups of both implementations. The reason for this is that the OpenCL API handles all the communication (using all the available hardware devices) in the device whereas
in OpenMP communications are done manually (using only one core of the IO subsystem and portals).

- In the case of the implementation of the PCA algorithm, the parallelism has been better extracted by the OpenMP implementation since it provides more control to the developer. This control is completely needed in an iterative process such as the eigenvalue decomposition. The best speed-up obtained with a balanced configuration is 2.52 (1233.11 ms) using 14 threads on one cluster of the MPPA.

- While implementing the PCA algorithm with OpenCL, it has been found that the execution times of all steps except the eigenvalue decomposition were smaller than those of the OpenMP implementation. However, step 3 (Jacobi Method) requires a greater control than that provided by OpenCL. Since OpenCL handles all communications and the procedure to implement is iterative, several read and write actions in global memory are needed between the invocations of two kernels. These global memory accesses affect drastically the performance of the application.

- The execution times obtained in the implementations of SVM and PCA, are smaller than those of the sequential implementations. As a conclusion, the implementation of the algorithm can be considered faster enough to meet the surgical real-time constrain, i.e. 120 s to process a hyperspectral image.

- With these results, it is possible to conclude that OpenMP and OpenCL open a promising path to reach the goal of the HELICoID project i.e. to detect brain cancer tumors at the operating room employing hyperspectral imaging in manycore devices.

At last, considering that the main objective of this project has been to probe if OpenMP and OpenCL work efficiently, it is possible to conclude that both API help to increase the performance of the application using a parallel approach on the selected MPPA platform. However, programmers still need to know how to parallelize a task. Therefore, OpenMP and OpenCL API only distribute a parallel version of the code on a parallel architecture. In the future, maybe, they will convert a sequential code into a parallel one.

### 6.2. Future lines

- To develop a program which combines the PCA and SVM algorithm to perform a spatial reduction of the image and then the classification. To obtain an OpenCL and OpenMP implementation to evaluate if they are fast enough to fulfill the surgical real time constrain. To investigate new releases of the Kalray
SKD to implement kernels with all features available in the OpenCL standard. To assess the performance of these applications.

- To research and implement new non-linear SVM kernels. To implement them with OpenMP and OpenCL to check their performance and efficiency.
- To measure the power consumption of OpenMP and OpenCL implementations. To port them to an MPPA-based embedded system giving priority to raw performance and battery operational time.

To test the heterogeneity of the OpenCL API to use the code developed in this research with different platforms that supports the API (e.g. Altera FPGAs).
7 References


Appendix
Annex I

OpenMP Makefile modifications

This annex shows the modifications made to a Makefile in order to compile a program on the MPPA using OpenMP. It is based on the makefile of the MPPA + host example found in Eclipse on the MPPA Developer Station.

The following changes were made in order to be used in the applications made in this research work:

- Line 16, to set the developer station.
- Line 18, to set the architecture of the board, in this case Bostan (k1b).
- Line 59, the –fopenmp and –DUSE_OMP flags are added to the compiler flags.
- Line 60, the –fopenmp flag is added to the linker flag.

The compiler and linker flags can be added to the host and I/O program if the application requires to parallelize those codes as well.

```c
# Needed for Ubuntu/Debian since sh points to dash instead of bash...
SHELL := /bin/bash

K1_TOOLCHAIN_DIR := $(shell echo $${K1_TOOLCHAIN_DIR:=-/usr/local/k1tools/})
SRC_DIR := src
HOST_SRC_DIR := ${SRC_DIR}/host
K1_IO_SRC_DIR := ${SRC_DIR}/k1_io
K1_CLUSTER_SRC_DIR := ${SRC_DIR}/k1_cluster
INCLUDE_SRC_DIR := ${SRC_DIR}/include

K1_IO_APPLI_NAME := vector_add_io
K1_CLUSTER_APPLI_NAME := vector_add_cluster

#board can be tc2, emb01 or developer
board := developer

#arch can be k1a (Andey) or k1b (Bostan)
arch := k1b

# emb01 is always a remote target
ifeq ($(board), emb01)
remote := true
```

120
else
remote := false
endif

# Define whether we are using OSes or not. can be os/rtems/nodeos/bare
# If not present, default to os
system-name := os

# Common cflags and lflags (for all targets)
# These flags will be propagated through the hierarchy
-Wuninitialized
LFLAGS := -lm

# The mppa-bin target is used to generate multibinary for specified target
mppa-bin := multibin
multibin-name := vector_add_mppa.mpk

# Define which binaries will composed multibin
# Note that the first binary in the list defines the boot binary
multibin-objs := ${K1_IO_APPLI_NAME} ${K1_CLUSTER_APPLI_NAME}

# You can pass additional flags to "k1-create-multibinary" using following command.
# for example, '-w ".*"' will active all tracepoints
multibin-flags := -w ".*"

# Common flags for k1 binaries
k1-cflags :=
k1-lflags :=

# Cluster binaries
cluster-bin := ${K1_CLUSTER_APPLI_NAME}

# Each *-srcs variable describe sources for the binaries
${K1_CLUSTER_APPLI_NAME}-srcs :=
${K1_CLUSTER_SRC_DIR}/cluster_main.c

${K1_CLUSTER_APPLI_NAME}-cflags := --std=gnu99 -mos=nodeos -fopenmp -DUSE_OMP
${K1_CLUSTER_APPLI_NAME}-lflags := -lmppaipc -fopenmp

# The io-bin var is used to build groups of var
io-bin := ${K1_IO_APPLI_NAME}
# Common cflags for io sources
io-cflags :=
Annex I

68 \( ${K1_IO_APPLI_NAME} \)-srcs := \( ${K1_IO_SRC_DIR}/io_main.c \)
69 # Flags can be specified per binary
70 \( ${K1_IO_APPLI_NAME} \)-cflags :=
71 \( ${K1_IO_APPLI_NAME} \)-lflags :=
72
73 # Same rule as the other but this time for host
74 # It define the host programs we will compile
75 host-bin := vector_add_host
76 # Common flags for host programs
77 host-cflags :=
78 host-lflags := -ldl
79
80 \( ${host-bin} \)-srcs := \( ${HOST_SRC_DIR}/host_main.c \)
81 \( ${host-bin} \)-cflags :=
82 \( ${host-bin} \)-lflags :=
83
84 # Special hooks for user makefile
85
86 # Hooks specified in help-hooks are rules which will be called after displaying
87 # the default help
88 help-hooks := my_help_rule
89 # Pre build hooks will be called before build
90 # pre-build-hooks :=
91 # Post build hooks will be called after build
92 post-build-hooks := post_build_rule
93
94 # Finally, include the Kalray Makefile which will instantiate the rules
95 # and do the work
96 include \( ${K1_TOOLCHAIN_DIR}/share/make/Makefile.mppaipc \)
97
98 # Define your rules after the include, otherwise, the first rule will be treated
99 # as the default rule
100 my_help_rule:
101     @echo Additional Targets:
102     @echo " clean-traces - clean generated traces"
103     @echo " run-sim - run application on simulator"
104     @echo " run-hw - run application on developer"
105
106 post_build_rule:
107     @echo "Build done"
108
109 .PHONY: run-sim
110 run-sim: all
111 \( ${K1_TOOLCHAIN_DIR}/bin/k1-pciesim-runner \)\( ${BIN_DIR}/${host-bin} \)
112 \( ${BIN_DIR}/${multibin-name} \) \( ${K1_IO_APPLI_NAME} \)
.PHONY: run-hw

run-hw: all
    ${BIN_DIR}/${host-bin} ${BIN_DIR}/${multibin-name}
    ${K1_IO_APPLI_NAME}

.clean-traces:
    rm -rf MPPA.*.DSU.* MPPA.*.trace.* MPPA.*.trace0.* MPPA.*.trace1
Annex II

OpenCL Makefile modifications

In order to compile programs with the OpenCL API the code must include the correct compiling instructions. The proposed makefile is based in an example from Kalray, an OpenCL makefile, which can be found on the MPPA Developer Station at this path:

$(K1_TOOLCHAIN_DIR)/share/make/Makefile.opencl

The following changes were made in order to be used in the applications made in this research work:

- Lines 2 to 5, to indicate the locations of the .c files
- Line 48, to add the flag OpenCL and the mathematical library (-lm) as linker flags.
- Line 49, to add the flag OpenCL as compiler flag
- Line 53, to define the kernel architecture, in this case Bostan (k1b).

```bash
1 SHELL := /bin/bash
2 SRC_DIR := src
3 HOST_SRC_DIR := ${SRC_DIR}/host
4 INCLUDE_SRC_DIR := ${SRC_DIR}/include
5 KERNEL_SRC_DIR := ${SRC_DIR}/kernel
6
7 host_ndrange-srce := ${HOST_SRC_DIR}/host_main_final.c
8
9 host-bin := host_ndrange
10
11 #include $(K1_TOOLCHAIN_DIR)/share/make/Makefile.opencl
12
13 ifeq ($(strip $(K1_TOOLCHAIN_DIR)),)
14   $(error K1_TOOLCHAIN_DIR is not set)
15 endif
16
17 cflags += -I.
18
19 K1_PKG_CONFIG := $(K1_TOOLCHAIN_DIR)/bin/k1-pkg-config
20
21 # When no kernel sources are specified, we assume it's a task
22 kernel application
23 ifeq ($(kernels-srce),)
24```
25 # Check for non C files inside the kernel files
26 dummy := $(if $(filter-out %.c,$(kernels-srcs)),$(error Wrong filetype given for kernels-srcs),)
27
28 # k1-opencl-gen only supports C files. filter them from the original list
29 kernels-files := $(filter %.c,$(kernels-srcs))
30
31 kernels-output-dir ?= .
32 # Native kernels requires those file to be compiled in pre-build-hooks += $(kernels-output-dir)/opencl_gen.h
33 host-cflags += -I$(kernels-output-dir)/
34
35 clean-extra-files := $(kernels-output-dir)/opencl_gen.h
36
37 endif
38
39 # LEGACY: Add the host_ocl executable to the list of host binaries
40 host-bin += host_ocl
41
42 # Add opencl flags to host
43
44 #JAIME SAYS:
45 # I added the library for math in the compiling using the -lm instruction... eye on this if it is not needed it must be removed
46 host-lflags += $(shell $(K1_PKG_CONFIG) --libs opencl) -lm
47 host-cflags += $(shell $(K1_PKG_CONFIG) --cflags opencl)
48
49 include $(K1_TOOLCHAIN_DIR)/share/make/Makefile.kalray
50
51 kernels-arch ?= k1b
52
53 # Custom rule for native kernel files
54 ifneq ($(kernels-srcs),)
55
56 quiet_cmd_opencl_gen = OPENCL\tGenerating native kernels for file
57 $(kernels-files) arch $(kernels-arch)
58 cmd_opencl_gen = $(K1_TOOLCHAIN_DIR)/bin/k1-opencl-gen -f
59 $(kernels-files) -a $(kernels-arch) -o $(kernels-output-dir) --
60 $(kernels-cflags) $(kernels-lflags)
61
62 # This rules will generate the needed files for native kernels
63 $(kernels-output-dir)/opencl_gen.h: $(kernels-files) $(kernels-deps)
64 $(call build_cmd,opencl_gen)
65
66 endif

67 # Execution rules

68
run_sim: all
$(K1_TOOLCHAIN_DIR)/bin/k1-pciesim-runner
run_hw: all
$(BIN_DIR)/host_ndrange
Annex III

1. POSIX Times

   a. PCA Algorithm

   Table 20 Posix Execution time (manual threads)

<table>
<thead>
<tr>
<th>Epsilon</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.05</td>
<td>2.13</td>
<td>1.66</td>
<td>1.97</td>
<td>1.80</td>
<td>1.80</td>
<td>1.90</td>
<td>1.60</td>
<td>1.66</td>
<td>1.64</td>
<td>1.58</td>
<td>1.66</td>
<td>1.61</td>
<td>1.51</td>
<td>1.62</td>
</tr>
<tr>
<td></td>
<td>3.10</td>
<td>2.64</td>
<td>2.31</td>
<td>2.17</td>
<td>2.42</td>
<td>1.90</td>
<td>1.82</td>
<td>2.20</td>
<td>2.09</td>
<td>2.14</td>
<td>1.87</td>
<td>2.01</td>
<td>1.84</td>
<td>1.58</td>
<td>1.96</td>
</tr>
<tr>
<td></td>
<td>4.02</td>
<td>3.88</td>
<td>2.89</td>
<td>2.80</td>
<td>2.92</td>
<td>2.51</td>
<td>2.09</td>
<td>1.92</td>
<td>2.34</td>
<td>2.18</td>
<td>1.83</td>
<td>2.13</td>
<td>2.00</td>
<td>2.06</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td>5.62</td>
<td>4.77</td>
<td>3.48</td>
<td>3.55</td>
<td>3.77</td>
<td>2.83</td>
<td>2.98</td>
<td>2.64</td>
<td>2.38</td>
<td>2.39</td>
<td>2.02</td>
<td>2.56</td>
<td>2.35</td>
<td>2.26</td>
<td>2.24</td>
</tr>
<tr>
<td></td>
<td>7.49</td>
<td>5.70</td>
<td>4.20</td>
<td>3.53</td>
<td>3.41</td>
<td>3.69</td>
<td>3.33</td>
<td>3.02</td>
<td>2.91</td>
<td>2.67</td>
<td>2.47</td>
<td>2.89</td>
<td>2.37</td>
<td>2.71</td>
<td>2.39</td>
</tr>
<tr>
<td></td>
<td>9.45</td>
<td>6.48</td>
<td>5.10</td>
<td>4.75</td>
<td>3.55</td>
<td>4.27</td>
<td>3.91</td>
<td>3.14</td>
<td>3.28</td>
<td>3.02</td>
<td>3.06</td>
<td>3.01</td>
<td>2.84</td>
<td>2.79</td>
<td>2.42</td>
</tr>
<tr>
<td></td>
<td>11.41</td>
<td>7.64</td>
<td>5.41</td>
<td>5.37</td>
<td>4.20</td>
<td>5.06</td>
<td>3.73</td>
<td>3.84</td>
<td>3.43</td>
<td>3.81</td>
<td>3.63</td>
<td>3.44</td>
<td>2.83</td>
<td>2.83</td>
<td>3.33</td>
</tr>
<tr>
<td></td>
<td>183.85</td>
<td>131.08</td>
<td>88.27</td>
<td>59.55</td>
<td>49.97</td>
<td>41.40</td>
<td>35.13</td>
<td>32.94</td>
<td>34.81</td>
<td>29.15</td>
<td>24.70</td>
<td>21.73</td>
<td>21.11</td>
<td>18.57</td>
<td>20.64</td>
</tr>
<tr>
<td></td>
<td>214.09</td>
<td>148.60</td>
<td>101.83</td>
<td>69.55</td>
<td>57.06</td>
<td>50.91</td>
<td>40.73</td>
<td>39.09</td>
<td>37.74</td>
<td>32.90</td>
<td>29.60</td>
<td>27.67</td>
<td>25.13</td>
<td>23.25</td>
<td>23.64</td>
</tr>
</tbody>
</table>

*Time in seconds
Table 21 PCA sequential steps execution time $\varepsilon = 1 \times 10^{-5}$

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Step 1</td>
<td>Step 2</td>
<td>Step 3</td>
<td>Step 4 &amp; 5</td>
<td>Total Time</td>
</tr>
<tr>
<td>1</td>
<td>60.23</td>
<td>559.63</td>
<td>2410.19</td>
<td>70.07</td>
<td>3102.01</td>
</tr>
<tr>
<td>2</td>
<td>60.21</td>
<td>559.63</td>
<td>2410.19</td>
<td>70.07</td>
<td>3102.03</td>
</tr>
<tr>
<td>3</td>
<td>60.22</td>
<td>550.42</td>
<td>2420.19</td>
<td>60.06</td>
<td>3102.04</td>
</tr>
<tr>
<td>4</td>
<td>60.20</td>
<td>549.62</td>
<td>2420.19</td>
<td>59.27</td>
<td>3102.05</td>
</tr>
<tr>
<td>5</td>
<td>59.39</td>
<td>560.42</td>
<td>2410.19</td>
<td>70.07</td>
<td>3101.96</td>
</tr>
<tr>
<td>6</td>
<td>60.18</td>
<td>549.63</td>
<td>2420.19</td>
<td>60.07</td>
<td>3102.01</td>
</tr>
<tr>
<td>7</td>
<td>60.20</td>
<td>549.62</td>
<td>2420.19</td>
<td>59.27</td>
<td>3101.96</td>
</tr>
<tr>
<td>8</td>
<td>50.20</td>
<td>559.63</td>
<td>2410.19</td>
<td>70.07</td>
<td>3102.00</td>
</tr>
<tr>
<td>9</td>
<td>59.41</td>
<td>560.42</td>
<td>2409.39</td>
<td>70.07</td>
<td>3101.98</td>
</tr>
<tr>
<td>10</td>
<td>60.21</td>
<td>559.63</td>
<td>2410.19</td>
<td>60.07</td>
<td>3102.01</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>59.04</td>
<td>555.87</td>
<td>2414.11</td>
<td>64.91</td>
<td>3102.00</td>
</tr>
</tbody>
</table>
### b. SVM Algorithm

Table 22 SVM algorithm with POSIX

<table>
<thead>
<tr>
<th>Exe.</th>
<th>POSIX Threads</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>1.23</td>
<td>0.83</td>
<td>0.70</td>
<td>0.63</td>
<td>0.60</td>
<td>0.57</td>
<td>0.56</td>
<td>0.54</td>
<td>0.54</td>
<td>0.52</td>
<td>0.52</td>
<td>0.51</td>
<td>0.51</td>
</tr>
<tr>
<td>2</td>
<td>1.23</td>
<td>0.83</td>
<td>0.70</td>
<td>0.63</td>
<td>0.60</td>
<td>0.57</td>
<td>0.56</td>
<td>0.54</td>
<td>0.54</td>
<td>0.52</td>
<td>0.52</td>
<td>0.51</td>
<td>0.51</td>
</tr>
<tr>
<td>3</td>
<td>1.23</td>
<td>0.83</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.57</td>
<td>0.56</td>
<td>0.54</td>
<td>0.54</td>
<td>0.52</td>
<td>0.52</td>
<td>0.51</td>
<td>0.50</td>
</tr>
<tr>
<td>4</td>
<td>1.23</td>
<td>0.83</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.58</td>
<td>0.56</td>
<td>0.54</td>
<td>0.54</td>
<td>0.52</td>
<td>0.52</td>
<td>0.51</td>
<td>0.50</td>
</tr>
<tr>
<td>5</td>
<td>1.23</td>
<td>0.83</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.58</td>
<td>0.56</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.52</td>
<td>0.51</td>
<td>0.51</td>
</tr>
<tr>
<td>6</td>
<td>1.23</td>
<td>0.83</td>
<td>0.70</td>
<td>0.63</td>
<td>0.60</td>
<td>0.58</td>
<td>0.56</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.52</td>
<td>0.51</td>
<td>0.51</td>
</tr>
<tr>
<td>7</td>
<td>1.23</td>
<td>0.83</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.57</td>
<td>0.56</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.52</td>
<td>0.51</td>
<td>0.51</td>
</tr>
<tr>
<td>8</td>
<td>1.23</td>
<td>0.83</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.58</td>
<td>0.56</td>
<td>0.54</td>
<td>0.53</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.51</td>
</tr>
<tr>
<td>9</td>
<td>1.23</td>
<td>0.83</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.58</td>
<td>0.55</td>
<td>0.54</td>
<td>0.53</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.50</td>
</tr>
<tr>
<td>10</td>
<td>1.23</td>
<td>0.83</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.57</td>
<td>0.56</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.52</td>
<td>0.51</td>
<td>0.51</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td>1.23</td>
<td>0.83</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.57</td>
<td>0.56</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.52</td>
<td>0.51</td>
<td>0.51</td>
</tr>
<tr>
<td><strong>speedup</strong></td>
<td>0.00</td>
<td>1.47</td>
<td>1.75</td>
<td>1.93</td>
<td>2.05</td>
<td>2.14</td>
<td>2.21</td>
<td>2.27</td>
<td>2.31</td>
<td>2.35</td>
<td>2.37</td>
<td>2.40</td>
<td>2.41</td>
</tr>
</tbody>
</table>

*time in seconds
2. OpenMP Execution Times
   a. PCA algorithm

Table 23 PCA OpenMP $\varepsilon = 1 \times 10^{-4}$

<table>
<thead>
<tr>
<th>Exe.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.19</td>
<td>1.90</td>
<td>1.44</td>
<td>1.70</td>
<td>1.46</td>
<td>1.48</td>
<td>1.55</td>
<td>1.24</td>
<td>1.30</td>
<td>1.29</td>
<td>1.24</td>
<td>1.32</td>
<td>1.29</td>
<td>1.15</td>
<td>1.28</td>
<td>1.11</td>
</tr>
<tr>
<td>2</td>
<td>2.19</td>
<td>1.90</td>
<td>1.44</td>
<td>1.70</td>
<td>1.46</td>
<td>1.48</td>
<td>1.55</td>
<td>1.24</td>
<td>1.30</td>
<td>1.29</td>
<td>1.24</td>
<td>1.32</td>
<td>1.29</td>
<td>1.15</td>
<td>1.28</td>
<td>1.11</td>
</tr>
<tr>
<td>3</td>
<td>2.19</td>
<td>1.90</td>
<td>1.44</td>
<td>1.70</td>
<td>1.46</td>
<td>1.47</td>
<td>1.55</td>
<td>1.24</td>
<td>1.30</td>
<td>1.29</td>
<td>1.24</td>
<td>1.32</td>
<td>1.29</td>
<td>1.15</td>
<td>1.28</td>
<td>1.11</td>
</tr>
<tr>
<td>4</td>
<td>2.19</td>
<td>1.90</td>
<td>1.44</td>
<td>1.70</td>
<td>1.46</td>
<td>1.47</td>
<td>1.55</td>
<td>1.24</td>
<td>1.30</td>
<td>1.29</td>
<td>1.24</td>
<td>1.32</td>
<td>1.29</td>
<td>1.15</td>
<td>1.28</td>
<td>1.11</td>
</tr>
<tr>
<td>5</td>
<td>2.19</td>
<td>1.90</td>
<td>1.44</td>
<td>1.70</td>
<td>1.46</td>
<td>1.48</td>
<td>1.55</td>
<td>1.24</td>
<td>1.30</td>
<td>1.29</td>
<td>1.24</td>
<td>1.32</td>
<td>1.29</td>
<td>1.15</td>
<td>1.28</td>
<td>1.11</td>
</tr>
<tr>
<td>6</td>
<td>2.19</td>
<td>1.90</td>
<td>1.44</td>
<td>1.70</td>
<td>1.46</td>
<td>1.48</td>
<td>1.55</td>
<td>1.24</td>
<td>1.30</td>
<td>1.29</td>
<td>1.24</td>
<td>1.32</td>
<td>1.29</td>
<td>1.15</td>
<td>1.28</td>
<td>1.11</td>
</tr>
<tr>
<td>7</td>
<td>2.19</td>
<td>1.90</td>
<td>1.44</td>
<td>1.70</td>
<td>1.46</td>
<td>1.47</td>
<td>1.55</td>
<td>1.24</td>
<td>1.30</td>
<td>1.29</td>
<td>1.24</td>
<td>1.32</td>
<td>1.29</td>
<td>1.15</td>
<td>1.28</td>
<td>1.11</td>
</tr>
<tr>
<td>8</td>
<td>2.19</td>
<td>1.90</td>
<td>1.44</td>
<td>1.70</td>
<td>1.46</td>
<td>1.47</td>
<td>1.55</td>
<td>1.24</td>
<td>1.30</td>
<td>1.29</td>
<td>1.24</td>
<td>1.32</td>
<td>1.29</td>
<td>1.15</td>
<td>1.28</td>
<td>1.11</td>
</tr>
<tr>
<td>9</td>
<td>2.19</td>
<td>1.90</td>
<td>1.44</td>
<td>1.70</td>
<td>1.46</td>
<td>1.47</td>
<td>1.55</td>
<td>1.24</td>
<td>1.30</td>
<td>1.29</td>
<td>1.24</td>
<td>1.32</td>
<td>1.29</td>
<td>1.15</td>
<td>1.28</td>
<td>1.11</td>
</tr>
<tr>
<td>10</td>
<td>2.19</td>
<td>1.90</td>
<td>1.44</td>
<td>1.70</td>
<td>1.46</td>
<td>1.48</td>
<td>1.55</td>
<td>1.24</td>
<td>1.30</td>
<td>1.29</td>
<td>1.24</td>
<td>1.32</td>
<td>1.29</td>
<td>1.15</td>
<td>1.28</td>
<td>1.11</td>
</tr>
<tr>
<td>average</td>
<td>2.19</td>
<td>1.90</td>
<td>1.44</td>
<td>1.70</td>
<td>1.46</td>
<td>1.47</td>
<td>1.55</td>
<td>1.24</td>
<td>1.30</td>
<td>1.29</td>
<td>1.24</td>
<td>1.32</td>
<td>1.29</td>
<td>1.15</td>
<td>1.28</td>
<td>1.11</td>
</tr>
<tr>
<td>speedup</td>
<td>0.93</td>
<td>1.08</td>
<td>1.44</td>
<td>1.20</td>
<td>1.40</td>
<td>1.39</td>
<td>1.32</td>
<td>1.65</td>
<td>1.58</td>
<td>1.59</td>
<td>1.66</td>
<td>1.55</td>
<td>1.59</td>
<td>1.78</td>
<td>1.60</td>
<td>1.84</td>
</tr>
</tbody>
</table>

*Time in seconds
### Table 24 PCA OpenMP $\epsilon = 1 \times 10^{-5}$

<table>
<thead>
<tr>
<th>Exe.</th>
<th>OpenMP Threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.33</td>
<td>2.41</td>
<td>2.13</td>
<td>1.94</td>
<td>2.14</td>
<td>1.59</td>
<td>1.49</td>
<td>1.87</td>
<td>1.75</td>
<td>1.81</td>
<td>1.55</td>
<td>1.70</td>
<td>1.54</td>
<td>1.23</td>
<td>1.65</td>
<td>1.42</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.33</td>
<td>2.41</td>
<td>2.13</td>
<td>1.94</td>
<td>2.14</td>
<td>1.59</td>
<td>1.49</td>
<td>1.87</td>
<td>1.75</td>
<td>1.81</td>
<td>1.55</td>
<td>1.70</td>
<td>1.54</td>
<td>1.23</td>
<td>1.65</td>
<td>1.42</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3.33</td>
<td>2.41</td>
<td>2.13</td>
<td>1.94</td>
<td>2.14</td>
<td>1.59</td>
<td>1.49</td>
<td>1.87</td>
<td>1.75</td>
<td>1.81</td>
<td>1.55</td>
<td>1.70</td>
<td>1.54</td>
<td>1.23</td>
<td>1.65</td>
<td>1.42</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.33</td>
<td>2.41</td>
<td>2.13</td>
<td>1.94</td>
<td>2.14</td>
<td>1.59</td>
<td>1.49</td>
<td>1.87</td>
<td>1.75</td>
<td>1.81</td>
<td>1.55</td>
<td>1.70</td>
<td>1.54</td>
<td>1.23</td>
<td>1.65</td>
<td>1.42</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3.33</td>
<td>2.40</td>
<td>2.13</td>
<td>1.94</td>
<td>2.14</td>
<td>1.59</td>
<td>1.49</td>
<td>1.87</td>
<td>1.75</td>
<td>1.81</td>
<td>1.55</td>
<td>1.70</td>
<td>1.54</td>
<td>1.23</td>
<td>1.65</td>
<td>1.42</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3.33</td>
<td>2.41</td>
<td>2.13</td>
<td>1.94</td>
<td>2.14</td>
<td>1.59</td>
<td>1.49</td>
<td>1.87</td>
<td>1.75</td>
<td>1.81</td>
<td>1.55</td>
<td>1.70</td>
<td>1.54</td>
<td>1.23</td>
<td>1.65</td>
<td>1.42</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3.33</td>
<td>2.41</td>
<td>2.13</td>
<td>1.94</td>
<td>2.14</td>
<td>1.59</td>
<td>1.49</td>
<td>1.87</td>
<td>1.75</td>
<td>1.81</td>
<td>1.55</td>
<td>1.70</td>
<td>1.54</td>
<td>1.23</td>
<td>1.65</td>
<td>1.42</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3.33</td>
<td>2.41</td>
<td>2.13</td>
<td>1.94</td>
<td>2.14</td>
<td>1.59</td>
<td>1.49</td>
<td>1.87</td>
<td>1.75</td>
<td>1.81</td>
<td>1.55</td>
<td>1.70</td>
<td>1.54</td>
<td>1.23</td>
<td>1.65</td>
<td>1.42</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3.33</td>
<td>2.40</td>
<td>2.13</td>
<td>1.94</td>
<td>2.14</td>
<td>1.59</td>
<td>1.49</td>
<td>1.87</td>
<td>1.75</td>
<td>1.81</td>
<td>1.55</td>
<td>1.70</td>
<td>1.54</td>
<td>1.23</td>
<td>1.65</td>
<td>1.42</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3.33</td>
<td>2.41</td>
<td>2.13</td>
<td>1.94</td>
<td>2.14</td>
<td>1.59</td>
<td>1.49</td>
<td>1.87</td>
<td>1.75</td>
<td>1.81</td>
<td>1.55</td>
<td>1.70</td>
<td>1.54</td>
<td>1.23</td>
<td>1.65</td>
<td>1.42</td>
<td></td>
</tr>
<tr>
<td><strong>average</strong></td>
<td><strong>3.33</strong></td>
<td><strong>2.41</strong></td>
<td><strong>2.13</strong></td>
<td><strong>1.94</strong></td>
<td><strong>2.14</strong></td>
<td><strong>1.59</strong></td>
<td><strong>1.49</strong></td>
<td><strong>1.87</strong></td>
<td><strong>1.75</strong></td>
<td><strong>1.81</strong></td>
<td><strong>1.55</strong></td>
<td><strong>1.70</strong></td>
<td><strong>1.54</strong></td>
<td><strong>1.23</strong></td>
<td><strong>1.65</strong></td>
<td><strong>1.42</strong></td>
<td></td>
</tr>
<tr>
<td><strong>speedup</strong></td>
<td>0.93</td>
<td>1.29</td>
<td>1.46</td>
<td>1.60</td>
<td>1.45</td>
<td>1.95</td>
<td>2.09</td>
<td>1.66</td>
<td>1.77</td>
<td>2.01</td>
<td>1.83</td>
<td>2.02</td>
<td>2.52</td>
<td>1.88</td>
<td>2.18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*time in seconds

### Table 25 PCA OpenMP $\epsilon = 1 \times 10^{-6}$

<table>
<thead>
<tr>
<th>Exe.</th>
<th>OpenMP Threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.32</td>
<td>3.66</td>
<td>2.76</td>
<td>2.61</td>
<td>2.67</td>
<td>2.24</td>
<td>1.77</td>
<td>1.59</td>
<td>2.03</td>
<td>1.86</td>
<td>1.52</td>
<td>1.83</td>
<td>1.72</td>
<td>1.75</td>
<td>1.58</td>
<td>1.69</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4.32</td>
<td>3.66</td>
<td>2.76</td>
<td>2.61</td>
<td>2.67</td>
<td>2.24</td>
<td>1.77</td>
<td>1.59</td>
<td>2.03</td>
<td>1.86</td>
<td>1.52</td>
<td>1.83</td>
<td>1.72</td>
<td>1.75</td>
<td>1.58</td>
<td>1.69</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4.32</td>
<td>3.66</td>
<td>2.76</td>
<td>2.61</td>
<td>2.67</td>
<td>2.24</td>
<td>1.77</td>
<td>1.59</td>
<td>2.03</td>
<td>1.86</td>
<td>1.52</td>
<td>1.83</td>
<td>1.72</td>
<td>1.75</td>
<td>1.58</td>
<td>1.69</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.32</td>
<td>3.66</td>
<td>2.76</td>
<td>2.61</td>
<td>2.67</td>
<td>2.24</td>
<td>1.77</td>
<td>1.59</td>
<td>2.03</td>
<td>1.86</td>
<td>1.52</td>
<td>1.83</td>
<td>1.72</td>
<td>1.75</td>
<td>1.58</td>
<td>1.69</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4.32</td>
<td>3.66</td>
<td>2.76</td>
<td>2.61</td>
<td>2.67</td>
<td>2.24</td>
<td>1.77</td>
<td>1.59</td>
<td>2.03</td>
<td>1.86</td>
<td>1.52</td>
<td>1.83</td>
<td>1.72</td>
<td>1.75</td>
<td>1.58</td>
<td>1.69</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4.32</td>
<td>3.66</td>
<td>2.76</td>
<td>2.61</td>
<td>2.67</td>
<td>2.24</td>
<td>1.77</td>
<td>1.59</td>
<td>2.03</td>
<td>1.86</td>
<td>1.52</td>
<td>1.83</td>
<td>1.72</td>
<td>1.75</td>
<td>1.58</td>
<td>1.69</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4.32</td>
<td>3.66</td>
<td>2.76</td>
<td>2.61</td>
<td>2.67</td>
<td>2.24</td>
<td>1.77</td>
<td>1.59</td>
<td>2.03</td>
<td>1.86</td>
<td>1.52</td>
<td>1.83</td>
<td>1.72</td>
<td>1.75</td>
<td>1.58</td>
<td>1.69</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4.32</td>
<td>3.66</td>
<td>2.76</td>
<td>2.61</td>
<td>2.67</td>
<td>2.24</td>
<td>1.77</td>
<td>1.59</td>
<td>2.03</td>
<td>1.86</td>
<td>1.52</td>
<td>1.83</td>
<td>1.72</td>
<td>1.75</td>
<td>1.58</td>
<td>1.69</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4.32</td>
<td>3.66</td>
<td>2.76</td>
<td>2.61</td>
<td>2.67</td>
<td>2.24</td>
<td>1.77</td>
<td>1.59</td>
<td>2.03</td>
<td>1.86</td>
<td>1.52</td>
<td>1.83</td>
<td>1.72</td>
<td>1.75</td>
<td>1.58</td>
<td>1.69</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4.32</td>
<td>3.66</td>
<td>2.76</td>
<td>2.61</td>
<td>2.67</td>
<td>2.24</td>
<td>1.77</td>
<td>1.59</td>
<td>2.03</td>
<td>1.86</td>
<td>1.52</td>
<td>1.83</td>
<td>1.72</td>
<td>1.75</td>
<td>1.58</td>
<td>1.69</td>
<td></td>
</tr>
<tr>
<td><strong>average</strong></td>
<td><strong>4.32</strong></td>
<td><strong>3.66</strong></td>
<td><strong>2.76</strong></td>
<td><strong>2.61</strong></td>
<td><strong>2.67</strong></td>
<td><strong>2.24</strong></td>
<td><strong>1.77</strong></td>
<td><strong>1.59</strong></td>
<td><strong>2.03</strong></td>
<td><strong>1.86</strong></td>
<td><strong>1.52</strong></td>
<td><strong>1.83</strong></td>
<td><strong>1.72</strong></td>
<td><strong>1.75</strong></td>
<td><strong>1.58</strong></td>
<td><strong>1.69</strong></td>
<td></td>
</tr>
<tr>
<td><strong>speedup</strong></td>
<td>0.93</td>
<td>1.10</td>
<td>1.46</td>
<td>1.54</td>
<td>1.50</td>
<td>1.80</td>
<td>2.27</td>
<td>2.53</td>
<td>1.98</td>
<td>2.16</td>
<td>2.65</td>
<td>2.19</td>
<td>2.34</td>
<td>2.30</td>
<td>2.54</td>
<td>2.37</td>
<td></td>
</tr>
</tbody>
</table>

*time in seconds
### Annex III

Table 26 PCA OpenMP $\epsilon = 1 \times 10^{-7}$

<table>
<thead>
<tr>
<th>Exe.</th>
<th>OpenMP Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>6.04</td>
</tr>
<tr>
<td>2</td>
<td>6.04</td>
</tr>
<tr>
<td>3</td>
<td>6.04</td>
</tr>
<tr>
<td>4</td>
<td>6.04</td>
</tr>
<tr>
<td>5</td>
<td>6.04</td>
</tr>
<tr>
<td>6</td>
<td>6.04</td>
</tr>
<tr>
<td>7</td>
<td>6.04</td>
</tr>
<tr>
<td>8</td>
<td>6.04</td>
</tr>
<tr>
<td>9</td>
<td>6.04</td>
</tr>
<tr>
<td>10</td>
<td>6.04</td>
</tr>
<tr>
<td>average</td>
<td>6.04</td>
</tr>
<tr>
<td>speedup</td>
<td>0.93</td>
</tr>
</tbody>
</table>

*Time in seconds

Table 27 PCA OpenMP $\epsilon = 1 \times 10^{-8}$

<table>
<thead>
<tr>
<th>Exe.</th>
<th>OpenMP Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>8.08</td>
</tr>
<tr>
<td>2</td>
<td>8.08</td>
</tr>
<tr>
<td>3</td>
<td>8.08</td>
</tr>
<tr>
<td>4</td>
<td>8.08</td>
</tr>
<tr>
<td>5</td>
<td>8.08</td>
</tr>
<tr>
<td>6</td>
<td>8.08</td>
</tr>
<tr>
<td>7</td>
<td>8.08</td>
</tr>
<tr>
<td>8</td>
<td>8.08</td>
</tr>
<tr>
<td>9</td>
<td>8.08</td>
</tr>
<tr>
<td>10</td>
<td>8.08</td>
</tr>
<tr>
<td>average</td>
<td>8.08</td>
</tr>
<tr>
<td>speedup</td>
<td>0.93</td>
</tr>
</tbody>
</table>

*Time in seconds
Table 28 PCA OpenMP $\epsilon = 1 \times 10^{-9}$

<table>
<thead>
<tr>
<th>Exe.</th>
<th>OpenMP Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10.22</td>
</tr>
<tr>
<td>2</td>
<td>10.22</td>
</tr>
<tr>
<td>3</td>
<td>10.22</td>
</tr>
<tr>
<td>4</td>
<td>10.22</td>
</tr>
<tr>
<td>5</td>
<td>10.22</td>
</tr>
<tr>
<td>6</td>
<td>10.22</td>
</tr>
<tr>
<td>7</td>
<td>10.22</td>
</tr>
<tr>
<td>8</td>
<td>10.22</td>
</tr>
<tr>
<td>9</td>
<td>10.22</td>
</tr>
<tr>
<td>10</td>
<td>10.22</td>
</tr>
<tr>
<td>average</td>
<td>10.22</td>
</tr>
<tr>
<td>speedup</td>
<td>0.92</td>
</tr>
</tbody>
</table>

*Time in seconds

Table 29 PCA OpenMP $\epsilon = 1 \times 10^{-10}$

<table>
<thead>
<tr>
<th>Exe.</th>
<th>OpenMP Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>12.37</td>
</tr>
<tr>
<td>2</td>
<td>12.37</td>
</tr>
<tr>
<td>3</td>
<td>12.37</td>
</tr>
<tr>
<td>4</td>
<td>12.37</td>
</tr>
<tr>
<td>5</td>
<td>12.37</td>
</tr>
<tr>
<td>6</td>
<td>12.37</td>
</tr>
<tr>
<td>7</td>
<td>12.37</td>
</tr>
<tr>
<td>8</td>
<td>12.37</td>
</tr>
<tr>
<td>9</td>
<td>12.37</td>
</tr>
<tr>
<td>10</td>
<td>12.37</td>
</tr>
<tr>
<td>average</td>
<td>12.37</td>
</tr>
<tr>
<td>speedup</td>
<td>0.92</td>
</tr>
</tbody>
</table>

*Time in seconds
### Table 30 PCA OpenMP $\varepsilon = 1 \times 10^{-12}$

<table>
<thead>
<tr>
<th>Exe.</th>
<th>OpenMP Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>17.88</td>
</tr>
<tr>
<td>2</td>
<td>17.88</td>
</tr>
<tr>
<td>3</td>
<td>17.88</td>
</tr>
<tr>
<td>4</td>
<td>17.88</td>
</tr>
<tr>
<td>5</td>
<td>17.88</td>
</tr>
<tr>
<td>6</td>
<td>17.88</td>
</tr>
<tr>
<td>7</td>
<td>17.88</td>
</tr>
<tr>
<td>8</td>
<td>17.88</td>
</tr>
<tr>
<td>9</td>
<td>17.88</td>
</tr>
<tr>
<td>10</td>
<td>17.88</td>
</tr>
<tr>
<td>average</td>
<td>17.88</td>
</tr>
<tr>
<td>speedup</td>
<td>0.92</td>
</tr>
</tbody>
</table>

*Time in seconds

### Table 31 PCA OpenMP $\varepsilon = 1 \times 10^{-16}$

<table>
<thead>
<tr>
<th>Exe.</th>
<th>OpenMP Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>201.67</td>
</tr>
<tr>
<td>2</td>
<td>201.67</td>
</tr>
<tr>
<td>3</td>
<td>201.67</td>
</tr>
<tr>
<td>4</td>
<td>201.67</td>
</tr>
<tr>
<td>5</td>
<td>201.67</td>
</tr>
<tr>
<td>6</td>
<td>201.67</td>
</tr>
<tr>
<td>7</td>
<td>201.67</td>
</tr>
<tr>
<td>8</td>
<td>201.67</td>
</tr>
<tr>
<td>9</td>
<td>201.67</td>
</tr>
<tr>
<td>10</td>
<td>201.67</td>
</tr>
<tr>
<td>average</td>
<td>201.67</td>
</tr>
<tr>
<td>speedup</td>
<td>0.91</td>
</tr>
</tbody>
</table>

*Time in seconds

134
### Table 32 PCA OpenMP $\varepsilon = 1 \times 10^{-20}$

<table>
<thead>
<tr>
<th>Exe.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>235.27</td>
<td>154.24</td>
<td>154.24</td>
<td>76.05</td>
<td>62.43</td>
<td>55.70</td>
<td>44.64</td>
<td>42.70</td>
<td>41.22</td>
<td>36.25</td>
<td>32.74</td>
<td>30.53</td>
<td>27.78</td>
<td>25.56</td>
<td>26.30</td>
<td>24.97</td>
</tr>
<tr>
<td>2</td>
<td>235.27</td>
<td>154.24</td>
<td>154.24</td>
<td>76.05</td>
<td>62.43</td>
<td>55.70</td>
<td>44.64</td>
<td>42.70</td>
<td>41.22</td>
<td>36.25</td>
<td>32.74</td>
<td>30.53</td>
<td>27.78</td>
<td>25.56</td>
<td>26.30</td>
<td>24.97</td>
</tr>
<tr>
<td>3</td>
<td>235.27</td>
<td>154.24</td>
<td>154.24</td>
<td>76.05</td>
<td>62.43</td>
<td>55.70</td>
<td>44.64</td>
<td>42.70</td>
<td>41.22</td>
<td>36.25</td>
<td>32.74</td>
<td>30.53</td>
<td>27.78</td>
<td>25.56</td>
<td>26.30</td>
<td>24.97</td>
</tr>
<tr>
<td>4</td>
<td>235.27</td>
<td>154.24</td>
<td>154.24</td>
<td>76.05</td>
<td>62.43</td>
<td>55.70</td>
<td>44.64</td>
<td>42.70</td>
<td>41.22</td>
<td>36.25</td>
<td>32.74</td>
<td>30.53</td>
<td>27.78</td>
<td>25.56</td>
<td>26.30</td>
<td>24.97</td>
</tr>
<tr>
<td>5</td>
<td>235.27</td>
<td>154.24</td>
<td>154.24</td>
<td>76.05</td>
<td>62.43</td>
<td>55.70</td>
<td>44.64</td>
<td>42.70</td>
<td>41.22</td>
<td>36.25</td>
<td>32.74</td>
<td>30.53</td>
<td>27.78</td>
<td>25.56</td>
<td>26.30</td>
<td>24.97</td>
</tr>
<tr>
<td>6</td>
<td>235.27</td>
<td>154.24</td>
<td>154.24</td>
<td>76.05</td>
<td>62.43</td>
<td>55.70</td>
<td>44.64</td>
<td>42.70</td>
<td>41.22</td>
<td>36.25</td>
<td>32.74</td>
<td>30.53</td>
<td>27.78</td>
<td>25.56</td>
<td>26.30</td>
<td>24.97</td>
</tr>
<tr>
<td>7</td>
<td>235.27</td>
<td>154.24</td>
<td>154.24</td>
<td>76.05</td>
<td>62.43</td>
<td>55.70</td>
<td>44.64</td>
<td>42.70</td>
<td>41.22</td>
<td>36.25</td>
<td>32.74</td>
<td>30.53</td>
<td>27.78</td>
<td>25.56</td>
<td>26.30</td>
<td>24.97</td>
</tr>
<tr>
<td>8</td>
<td>235.27</td>
<td>154.24</td>
<td>154.24</td>
<td>76.05</td>
<td>62.43</td>
<td>55.70</td>
<td>44.64</td>
<td>42.70</td>
<td>41.22</td>
<td>36.25</td>
<td>32.74</td>
<td>30.53</td>
<td>27.78</td>
<td>25.56</td>
<td>26.30</td>
<td>24.97</td>
</tr>
<tr>
<td>9</td>
<td>235.27</td>
<td>154.24</td>
<td>154.24</td>
<td>76.05</td>
<td>62.43</td>
<td>55.70</td>
<td>44.64</td>
<td>42.70</td>
<td>41.22</td>
<td>36.25</td>
<td>32.74</td>
<td>30.53</td>
<td>27.78</td>
<td>25.56</td>
<td>26.30</td>
<td>24.97</td>
</tr>
<tr>
<td>10</td>
<td>235.27</td>
<td>154.24</td>
<td>154.24</td>
<td>76.05</td>
<td>62.43</td>
<td>55.70</td>
<td>44.64</td>
<td>42.70</td>
<td>41.22</td>
<td>36.25</td>
<td>32.74</td>
<td>30.53</td>
<td>27.78</td>
<td>25.56</td>
<td>26.30</td>
<td>24.97</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td><strong>235.27</strong></td>
<td><strong>154.24</strong></td>
<td><strong>154.24</strong></td>
<td><strong>76.05</strong></td>
<td><strong>62.43</strong></td>
<td><strong>55.70</strong></td>
<td><strong>44.64</strong></td>
<td><strong>42.70</strong></td>
<td><strong>41.22</strong></td>
<td><strong>36.25</strong></td>
<td><strong>32.74</strong></td>
<td><strong>30.53</strong></td>
<td><strong>27.78</strong></td>
<td><strong>25.56</strong></td>
<td><strong>26.30</strong></td>
<td><strong>24.97</strong></td>
</tr>
<tr>
<td><strong>speedup</strong></td>
<td>0.91</td>
<td>1.39</td>
<td>1.39</td>
<td>2.82</td>
<td>3.43</td>
<td>3.84</td>
<td>4.80</td>
<td>5.01</td>
<td>5.19</td>
<td>5.91</td>
<td>6.54</td>
<td>7.01</td>
<td>7.71</td>
<td>8.38</td>
<td>8.14</td>
<td>8.57</td>
</tr>
</tbody>
</table>

*Time in seconds*
b. SVM algorithm

Table 33 SVM OpenMP monocluster

<table>
<thead>
<tr>
<th>Exe.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.24</td>
<td>0.84</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.57</td>
<td>0.55</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>2</td>
<td>1.24</td>
<td>0.84</td>
<td>0.71</td>
<td>0.64</td>
<td>0.60</td>
<td>0.57</td>
<td>0.55</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>3</td>
<td>1.24</td>
<td>0.84</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.57</td>
<td>0.56</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>4</td>
<td>1.24</td>
<td>0.84</td>
<td>0.70</td>
<td>0.64</td>
<td>0.59</td>
<td>0.57</td>
<td>0.55</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.51</td>
<td>0.50</td>
<td>0.50</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>5</td>
<td>1.24</td>
<td>0.84</td>
<td>0.71</td>
<td>0.64</td>
<td>0.60</td>
<td>0.57</td>
<td>0.55</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>6</td>
<td>1.24</td>
<td>0.84</td>
<td>0.70</td>
<td>0.63</td>
<td>0.59</td>
<td>0.57</td>
<td>0.55</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>7</td>
<td>1.24</td>
<td>0.84</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.57</td>
<td>0.55</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>8</td>
<td>1.24</td>
<td>0.83</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.57</td>
<td>0.56</td>
<td>0.53</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.51</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.49</td>
</tr>
<tr>
<td>9</td>
<td>1.24</td>
<td>0.83</td>
<td>0.70</td>
<td>0.63</td>
<td>0.60</td>
<td>0.57</td>
<td>0.56</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.49</td>
</tr>
<tr>
<td>10</td>
<td>1.24</td>
<td>0.84</td>
<td>0.70</td>
<td>0.63</td>
<td>0.59</td>
<td>0.57</td>
<td>0.55</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.49</td>
</tr>
<tr>
<td>average</td>
<td>1.24</td>
<td>0.84</td>
<td>0.70</td>
<td>0.64</td>
<td>0.60</td>
<td>0.57</td>
<td>0.55</td>
<td>0.54</td>
<td>0.53</td>
<td>0.52</td>
<td>0.51</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.49</td>
</tr>
<tr>
<td>speedup</td>
<td>0.99</td>
<td>1.47</td>
<td>1.75</td>
<td>1.93</td>
<td>2.06</td>
<td>2.16</td>
<td>2.22</td>
<td>2.29</td>
<td>2.33</td>
<td>2.37</td>
<td>2.40</td>
<td>2.44</td>
<td>2.45</td>
<td>2.46</td>
<td>2.50</td>
<td>2.51</td>
</tr>
</tbody>
</table>

*time in seconds

Table 34 SVM OpenMP with 13 clusters

<table>
<thead>
<tr>
<th>Exe.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.50</td>
<td>0.46</td>
<td>0.46</td>
<td>0.45</td>
<td>0.45</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>2</td>
<td>0.49</td>
<td>0.46</td>
<td>0.46</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>3</td>
<td>0.49</td>
<td>0.46</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>4</td>
<td>0.50</td>
<td>0.46</td>
<td>0.46</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>5</td>
<td>0.49</td>
<td>0.46</td>
<td>0.46</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>6</td>
<td>0.50</td>
<td>0.46</td>
<td>0.46</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>7</td>
<td>0.49</td>
<td>0.46</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>8</td>
<td>0.49</td>
<td>0.46</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>9</td>
<td>0.49</td>
<td>0.47</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>10</td>
<td>0.50</td>
<td>0.46</td>
<td>0.46</td>
<td>0.45</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>average</td>
<td>0.49</td>
<td>0.46</td>
<td>0.45</td>
<td>0.45</td>
<td>0.45</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>speedup</td>
<td>2.49</td>
<td>2.65</td>
<td>2.71</td>
<td>2.74</td>
<td>2.76</td>
<td>2.77</td>
<td>2.78</td>
<td>2.79</td>
<td>2.79</td>
<td>2.80</td>
<td>2.80</td>
<td>2.80</td>
<td>2.80</td>
<td>2.80</td>
<td>2.81</td>
<td>2.81</td>
</tr>
</tbody>
</table>

*time in seconds
3. OpenCL Execution Times
   a. PCA Algorithm

   Table 35 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-5}$ concurrent = 15

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4 &amp; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>3939.19</td>
<td>2803.06</td>
<td>17.19</td>
<td>112.52</td>
<td>2638.25</td>
<td>35.10</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2117.49</td>
<td>2883.21</td>
<td>17.36</td>
<td>112.00</td>
<td>2714.22</td>
<td>39.63</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2129.04</td>
<td>2873.53</td>
<td>17.06</td>
<td>111.13</td>
<td>2707.16</td>
<td>38.18</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2135.41</td>
<td>2884.94</td>
<td>17.48</td>
<td>112.68</td>
<td>2714.80</td>
<td>39.99</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>2119.60</td>
<td>2884.37</td>
<td>17.61</td>
<td>112.96</td>
<td>2713.75</td>
<td>40.05</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>2149.48</td>
<td>2850.73</td>
<td>17.24</td>
<td>111.95</td>
<td>2682.84</td>
<td>38.70</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>2137.53</td>
<td>2874.28</td>
<td>16.09</td>
<td>112.02</td>
<td>2706.46</td>
<td>39.71</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>2131.78</td>
<td>2903.40</td>
<td>18.27</td>
<td>113.26</td>
<td>2729.74</td>
<td>42.13</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>2131.38</td>
<td>2861.72</td>
<td>17.48</td>
<td>112.75</td>
<td>2692.28</td>
<td>39.20</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>2120.72</td>
<td>2917.63</td>
<td>17.17</td>
<td>113.07</td>
<td>2750.61</td>
<td>36.78</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td>2311.16</td>
<td>2873.69</td>
<td>17.30</td>
<td>112.43</td>
<td>2705.01</td>
<td>38.95</td>
</tr>
</tbody>
</table>

   Table 36 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-6}$ concurrent = 15

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4 &amp; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>2156.40</td>
<td>3871.55</td>
<td>18.04</td>
<td>112.65</td>
<td>3702.43</td>
<td>38.41</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2120.88</td>
<td>3890.21</td>
<td>16.43</td>
<td>111.18</td>
<td>3725.02</td>
<td>37.57</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2131.91</td>
<td>3898.73</td>
<td>17.32</td>
<td>112.47</td>
<td>3728.37</td>
<td>40.58</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2156.57</td>
<td>3907.95</td>
<td>17.46</td>
<td>113.18</td>
<td>3736.69</td>
<td>40.62</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>2137.10</td>
<td>3898.64</td>
<td>17.19</td>
<td>113.13</td>
<td>3729.27</td>
<td>39.05</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>2127.32</td>
<td>3896.45</td>
<td>17.79</td>
<td>112.12</td>
<td>3728.91</td>
<td>37.63</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>2141.08</td>
<td>3862.95</td>
<td>17.65</td>
<td>112.38</td>
<td>3692.21</td>
<td>40.70</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>2119.98</td>
<td>3844.47</td>
<td>16.88</td>
<td>111.84</td>
<td>3676.49</td>
<td>39.26</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>2128.92</td>
<td>3852.90</td>
<td>17.25</td>
<td>113.00</td>
<td>3680.52</td>
<td>42.12</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>2137.70</td>
<td>3883.20</td>
<td>17.33</td>
<td>111.81</td>
<td>3715.75</td>
<td>38.31</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td>2135.79</td>
<td>3880.70</td>
<td>17.33</td>
<td>112.38</td>
<td>3711.57</td>
<td>39.42</td>
</tr>
</tbody>
</table>
Table 37 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-7}$ concurrent = 15

<table>
<thead>
<tr>
<th>Execution</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4 &amp; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2131.43</td>
<td>3867.74</td>
<td>16.26</td>
<td>113.08</td>
<td>3699.65</td>
<td>38.75</td>
</tr>
<tr>
<td>2</td>
<td>2137.68</td>
<td>3861.21</td>
<td>16.56</td>
<td>111.61</td>
<td>3696.20</td>
<td>36.85</td>
</tr>
<tr>
<td>3</td>
<td>2142.14</td>
<td>3903.06</td>
<td>17.13</td>
<td>112.31</td>
<td>3732.02</td>
<td>41.60</td>
</tr>
<tr>
<td>4</td>
<td>2150.57</td>
<td>3871.18</td>
<td>17.39</td>
<td>113.89</td>
<td>3700.52</td>
<td>39.38</td>
</tr>
<tr>
<td>5</td>
<td>2153.56</td>
<td>3881.69</td>
<td>17.42</td>
<td>112.37</td>
<td>3712.97</td>
<td>38.93</td>
</tr>
<tr>
<td>6</td>
<td>2152.21</td>
<td>3885.33</td>
<td>17.88</td>
<td>111.87</td>
<td>3715.45</td>
<td>40.12</td>
</tr>
<tr>
<td>7</td>
<td>2146.49</td>
<td>3888.95</td>
<td>17.52</td>
<td>111.88</td>
<td>3719.73</td>
<td>39.82</td>
</tr>
<tr>
<td>8</td>
<td>2137.96</td>
<td>3879.36</td>
<td>17.15</td>
<td>112.30</td>
<td>3710.09</td>
<td>39.82</td>
</tr>
<tr>
<td>9</td>
<td>2255.27</td>
<td>3829.29</td>
<td>16.13</td>
<td>112.22</td>
<td>3661.88</td>
<td>39.07</td>
</tr>
<tr>
<td>10</td>
<td>2168.44</td>
<td>3754.88</td>
<td>17.27</td>
<td>114.15</td>
<td>3582.00</td>
<td>41.46</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>2157.57</td>
<td>3862.27</td>
<td>17.07</td>
<td>112.57</td>
<td>3693.05</td>
<td>39.58</td>
</tr>
</tbody>
</table>

Table 38 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-8}$ concurrent = 15

<table>
<thead>
<tr>
<th>Execution</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4 &amp; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2155.86</td>
<td>5714.12</td>
<td>17.05</td>
<td>112.39</td>
<td>5546.62</td>
<td>38.06</td>
</tr>
<tr>
<td>2</td>
<td>2116.68</td>
<td>5696.54</td>
<td>17.11</td>
<td>112.12</td>
<td>5530.24</td>
<td>37.08</td>
</tr>
<tr>
<td>3</td>
<td>2143.26</td>
<td>5716.17</td>
<td>17.07</td>
<td>113.01</td>
<td>5544.39</td>
<td>41.71</td>
</tr>
<tr>
<td>4</td>
<td>2119.94</td>
<td>5713.59</td>
<td>17.94</td>
<td>113.69</td>
<td>5541.73</td>
<td>40.23</td>
</tr>
<tr>
<td>5</td>
<td>2133.61</td>
<td>5723.04</td>
<td>17.37</td>
<td>113.75</td>
<td>5552.45</td>
<td>39.46</td>
</tr>
<tr>
<td>6</td>
<td>2144.44</td>
<td>5706.05</td>
<td>16.44</td>
<td>112.67</td>
<td>5535.50</td>
<td>41.45</td>
</tr>
<tr>
<td>7</td>
<td>2138.05</td>
<td>5712.75</td>
<td>17.58</td>
<td>113.26</td>
<td>5541.70</td>
<td>40.21</td>
</tr>
<tr>
<td>8</td>
<td>2137.91</td>
<td>5708.20</td>
<td>17.61</td>
<td>112.45</td>
<td>5537.80</td>
<td>40.34</td>
</tr>
<tr>
<td>9</td>
<td>2120.72</td>
<td>5725.96</td>
<td>18.26</td>
<td>111.33</td>
<td>5555.74</td>
<td>40.63</td>
</tr>
<tr>
<td>10</td>
<td>2136.10</td>
<td>5708.74</td>
<td>17.67</td>
<td>112.16</td>
<td>5541.76</td>
<td>37.15</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>2134.66</td>
<td>5712.52</td>
<td>17.41</td>
<td>112.68</td>
<td>5542.79</td>
<td>39.63</td>
</tr>
</tbody>
</table>
Table 39 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-9}$ concurrent = 15

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4 &amp; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2167.66</td>
<td>6618.78</td>
<td>15.75</td>
<td>112.93</td>
<td>6452.02</td>
<td>38.08</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2143.54</td>
<td>6636.07</td>
<td>17.91</td>
<td>111.34</td>
<td>6467.24</td>
<td>39.58</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2144.08</td>
<td>6631.11</td>
<td>18.33</td>
<td>112.43</td>
<td>6459.11</td>
<td>41.24</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2131.32</td>
<td>6653.43</td>
<td>17.07</td>
<td>111.30</td>
<td>6487.97</td>
<td>37.10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2142.43</td>
<td>6617.68</td>
<td>16.27</td>
<td>112.17</td>
<td>6451.13</td>
<td>38.12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2136.19</td>
<td>6310.62</td>
<td>17.33</td>
<td>113.87</td>
<td>6142.70</td>
<td>36.72</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2122.47</td>
<td>6635.03</td>
<td>16.22</td>
<td>111.29</td>
<td>6467.49</td>
<td>40.04</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2159.78</td>
<td>6637.99</td>
<td>17.39</td>
<td>112.65</td>
<td>6469.26</td>
<td>38.69</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2149.51</td>
<td>6656.67</td>
<td>17.92</td>
<td>113.91</td>
<td>6486.45</td>
<td>38.39</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2148.87</td>
<td>6631.82</td>
<td>17.03</td>
<td>112.15</td>
<td>6462.87</td>
<td>39.77</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>2144.58</strong></td>
<td><strong>6602.92</strong></td>
<td><strong>17.12</strong></td>
<td><strong>112.40</strong></td>
<td><strong>6434.62</strong></td>
<td><strong>38.77</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 40 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-10}$ concurrent = 15

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4 &amp; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2147.24</td>
<td>7276.42</td>
<td>17.02</td>
<td>113.03</td>
<td>7109.50</td>
<td>36.87</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2151.13</td>
<td>7230.17</td>
<td>17.76</td>
<td>113.02</td>
<td>7062.48</td>
<td>36.90</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2130.59</td>
<td>7259.45</td>
<td>17.08</td>
<td>112.13</td>
<td>7090.09</td>
<td>40.16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2119.74</td>
<td>7266.62</td>
<td>18.01</td>
<td>112.13</td>
<td>7099.41</td>
<td>37.07</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2129.76</td>
<td>7251.99</td>
<td>17.74</td>
<td>112.01</td>
<td>7081.67</td>
<td>40.57</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2148.59</td>
<td>7257.47</td>
<td>16.85</td>
<td>113.03</td>
<td>7088.29</td>
<td>39.29</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2163.66</td>
<td>7279.20</td>
<td>17.61</td>
<td>113.21</td>
<td>7105.68</td>
<td>42.69</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2146.96</td>
<td>7210.97</td>
<td>16.49</td>
<td>112.41</td>
<td>7044.31</td>
<td>37.76</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2136.40</td>
<td>7259.90</td>
<td>16.69</td>
<td>111.90</td>
<td>7090.89</td>
<td>40.42</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2154.92</td>
<td>7243.86</td>
<td>18.07</td>
<td>112.64</td>
<td>7073.50</td>
<td>39.65</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>2142.90</strong></td>
<td><strong>7253.60</strong></td>
<td><strong>17.33</strong></td>
<td><strong>112.55</strong></td>
<td><strong>7084.58</strong></td>
<td><strong>39.14</strong></td>
<td></td>
</tr>
</tbody>
</table>
Table 41 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-12}$ concurrent = 15

<table>
<thead>
<tr>
<th>Execution</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Step 1</td>
</tr>
<tr>
<td>1</td>
<td>2144.61</td>
<td>10667.84</td>
<td>16.64</td>
</tr>
<tr>
<td>2</td>
<td>2192.38</td>
<td>10799.45</td>
<td>17.47</td>
</tr>
<tr>
<td>3</td>
<td>2121.18</td>
<td>10804.44</td>
<td>16.82</td>
</tr>
<tr>
<td>4</td>
<td>2152.93</td>
<td>10741.43</td>
<td>17.64</td>
</tr>
<tr>
<td>5</td>
<td>2138.27</td>
<td>10479.59</td>
<td>17.39</td>
</tr>
<tr>
<td>6</td>
<td>2153.08</td>
<td>10729.63</td>
<td>17.86</td>
</tr>
<tr>
<td>7</td>
<td>2160.65</td>
<td>10769.86</td>
<td>17.82</td>
</tr>
<tr>
<td>8</td>
<td>2114.67</td>
<td>10774.73</td>
<td>16.59</td>
</tr>
<tr>
<td>9</td>
<td>2156.20</td>
<td>10769.91</td>
<td>16.66</td>
</tr>
<tr>
<td>10</td>
<td>2172.74</td>
<td>10732.43</td>
<td>17.42</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>2150.67</td>
<td>10726.93</td>
<td>17.23</td>
</tr>
</tbody>
</table>

Table 42 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-16}$ concurrent = 15

<table>
<thead>
<tr>
<th>Execution</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Step 1</td>
</tr>
<tr>
<td>1</td>
<td>2122.56</td>
<td>87585.20</td>
<td>17.34</td>
</tr>
<tr>
<td>2</td>
<td>2138.78</td>
<td>87477.04</td>
<td>16.31</td>
</tr>
<tr>
<td>3</td>
<td>2141.74</td>
<td>87606.60</td>
<td>16.96</td>
</tr>
<tr>
<td>4</td>
<td>2141.25</td>
<td>87701.94</td>
<td>16.33</td>
</tr>
<tr>
<td>5</td>
<td>2130.75</td>
<td>87759.11</td>
<td>17.03</td>
</tr>
<tr>
<td>6</td>
<td>2138.10</td>
<td>87888.87</td>
<td>16.11</td>
</tr>
<tr>
<td>7</td>
<td>2157.65</td>
<td>87634.82</td>
<td>16.95</td>
</tr>
<tr>
<td>8</td>
<td>2147.37</td>
<td>87950.65</td>
<td>16.89</td>
</tr>
<tr>
<td>9</td>
<td>2153.92</td>
<td>87922.32</td>
<td>17.70</td>
</tr>
<tr>
<td>10</td>
<td>2145.54</td>
<td>87948.19</td>
<td>17.59</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>2141.77</td>
<td>87747.47</td>
<td>16.92</td>
</tr>
</tbody>
</table>
Table 43 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-20}$ concurrent = 15

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compile</td>
</tr>
<tr>
<td>1</td>
<td>2144.29</td>
</tr>
<tr>
<td>2</td>
<td>2131.97</td>
</tr>
<tr>
<td>3</td>
<td>2148.27</td>
</tr>
<tr>
<td>4</td>
<td>2139.52</td>
</tr>
<tr>
<td>5</td>
<td>2173.58</td>
</tr>
<tr>
<td>6</td>
<td>2165.02</td>
</tr>
<tr>
<td>7</td>
<td>2126.04</td>
</tr>
<tr>
<td>8</td>
<td>2152.23</td>
</tr>
<tr>
<td>9</td>
<td>2172.28</td>
</tr>
<tr>
<td>10</td>
<td>2137.40</td>
</tr>
<tr>
<td>Average</td>
<td>2149.06</td>
</tr>
</tbody>
</table>

Table 44 PCA implemented with OpenCL $\epsilon = 1 \times 10^{-5}$ concurrent = 16

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compile</td>
</tr>
<tr>
<td>1</td>
<td>3939.19</td>
</tr>
<tr>
<td>2</td>
<td>2117.49</td>
</tr>
<tr>
<td>3</td>
<td>2129.04</td>
</tr>
<tr>
<td>4</td>
<td>2135.41</td>
</tr>
<tr>
<td>5</td>
<td>2119.60</td>
</tr>
<tr>
<td>6</td>
<td>2149.48</td>
</tr>
<tr>
<td>7</td>
<td>2137.53</td>
</tr>
<tr>
<td>8</td>
<td>2131.78</td>
</tr>
<tr>
<td>9</td>
<td>2131.38</td>
</tr>
<tr>
<td>10</td>
<td>2120.72</td>
</tr>
<tr>
<td>Average</td>
<td>2311.16</td>
</tr>
</tbody>
</table>
Table 45 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-6}$ concurrent = 16

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4 &amp; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2156.40</td>
<td>3871.55</td>
<td>18.04</td>
<td>112.65</td>
<td>3702.43</td>
<td>38.41</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2120.88</td>
<td>3890.21</td>
<td>16.43</td>
<td>111.18</td>
<td>3725.02</td>
<td>37.57</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2131.91</td>
<td>3898.73</td>
<td>17.32</td>
<td>112.47</td>
<td>3728.37</td>
<td>40.58</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2156.57</td>
<td>3907.95</td>
<td>17.46</td>
<td>113.18</td>
<td>3736.69</td>
<td>40.62</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2137.10</td>
<td>3898.64</td>
<td>17.19</td>
<td>113.13</td>
<td>3729.27</td>
<td>39.05</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2127.32</td>
<td>3896.45</td>
<td>17.79</td>
<td>112.12</td>
<td>3728.91</td>
<td>37.63</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2141.08</td>
<td>3862.95</td>
<td>17.65</td>
<td>112.38</td>
<td>3692.21</td>
<td>40.70</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2119.98</td>
<td>3844.47</td>
<td>16.88</td>
<td>111.84</td>
<td>3676.49</td>
<td>39.26</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2128.92</td>
<td>3852.90</td>
<td>17.25</td>
<td>113.00</td>
<td>3680.52</td>
<td>42.12</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2137.70</td>
<td>3883.20</td>
<td>17.33</td>
<td>111.81</td>
<td>3715.75</td>
<td>38.31</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>2135.79</td>
<td>3880.70</td>
<td>17.33</td>
<td>112.38</td>
<td>3711.57</td>
<td>39.42</td>
<td></td>
</tr>
</tbody>
</table>

Table 46 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-7}$ concurrent = 16

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4 &amp; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2131.43</td>
<td>3867.74</td>
<td>16.26</td>
<td>113.08</td>
<td>3699.65</td>
<td>38.75</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2137.68</td>
<td>3861.21</td>
<td>16.56</td>
<td>111.61</td>
<td>3696.20</td>
<td>36.85</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2142.14</td>
<td>3903.06</td>
<td>17.13</td>
<td>112.31</td>
<td>3732.02</td>
<td>41.60</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2150.57</td>
<td>3871.18</td>
<td>17.39</td>
<td>113.89</td>
<td>3700.52</td>
<td>39.38</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2153.56</td>
<td>3881.69</td>
<td>17.42</td>
<td>112.37</td>
<td>3712.97</td>
<td>38.93</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2152.21</td>
<td>3885.33</td>
<td>17.88</td>
<td>111.87</td>
<td>3715.45</td>
<td>40.12</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2146.49</td>
<td>3888.95</td>
<td>17.52</td>
<td>111.88</td>
<td>3719.73</td>
<td>39.82</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2137.96</td>
<td>3879.36</td>
<td>17.15</td>
<td>112.30</td>
<td>3710.09</td>
<td>39.82</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2255.27</td>
<td>3829.29</td>
<td>16.13</td>
<td>112.22</td>
<td>3661.88</td>
<td>39.07</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2168.44</td>
<td>3754.88</td>
<td>17.27</td>
<td>114.15</td>
<td>3582.00</td>
<td>41.46</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>2157.57</td>
<td>3862.27</td>
<td>17.07</td>
<td>112.57</td>
<td>3693.05</td>
<td>39.58</td>
<td></td>
</tr>
</tbody>
</table>
Table 47 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-8}$ concurrent = 16

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4 &amp; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2155.86</td>
<td>5714.12</td>
<td>17.05</td>
<td>112.39</td>
<td>5546.62</td>
<td>38.06</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2116.68</td>
<td>5696.54</td>
<td>17.11</td>
<td>112.12</td>
<td>5530.24</td>
<td>37.08</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2143.26</td>
<td>5716.17</td>
<td>17.07</td>
<td>113.01</td>
<td>5544.39</td>
<td>41.71</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2119.94</td>
<td>5713.59</td>
<td>17.94</td>
<td>113.69</td>
<td>5541.73</td>
<td>40.23</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2133.61</td>
<td>5723.04</td>
<td>17.37</td>
<td>113.75</td>
<td>5552.45</td>
<td>39.46</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2144.44</td>
<td>5706.05</td>
<td>16.44</td>
<td>112.67</td>
<td>5535.50</td>
<td>41.45</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2138.05</td>
<td>5712.75</td>
<td>17.58</td>
<td>113.26</td>
<td>5541.70</td>
<td>40.21</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2137.91</td>
<td>5708.20</td>
<td>17.61</td>
<td>112.45</td>
<td>5537.80</td>
<td>40.34</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2120.72</td>
<td>5725.96</td>
<td>18.26</td>
<td>111.33</td>
<td>5555.74</td>
<td>40.63</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2136.10</td>
<td>5708.74</td>
<td>17.67</td>
<td>112.16</td>
<td>5541.76</td>
<td>37.15</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>2134.66</td>
<td>5712.52</td>
<td>17.41</td>
<td>112.68</td>
<td>5542.79</td>
<td>39.63</td>
<td></td>
</tr>
</tbody>
</table>

Table 48 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-9}$ concurrent = 16

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4 &amp; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2167.66</td>
<td>6618.78</td>
<td>15.75</td>
<td>112.93</td>
<td>6452.02</td>
<td>38.08</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2143.54</td>
<td>6636.07</td>
<td>17.91</td>
<td>111.34</td>
<td>6467.24</td>
<td>39.58</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2144.08</td>
<td>6631.11</td>
<td>18.33</td>
<td>112.43</td>
<td>6459.11</td>
<td>41.24</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2131.32</td>
<td>6653.43</td>
<td>17.07</td>
<td>111.30</td>
<td>6487.97</td>
<td>37.10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2142.43</td>
<td>6617.68</td>
<td>16.27</td>
<td>112.17</td>
<td>6451.13</td>
<td>38.12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2136.19</td>
<td>6310.62</td>
<td>17.33</td>
<td>113.87</td>
<td>6142.70</td>
<td>36.72</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2122.47</td>
<td>6635.03</td>
<td>16.22</td>
<td>111.29</td>
<td>6467.49</td>
<td>40.04</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2159.78</td>
<td>6637.99</td>
<td>17.39</td>
<td>112.65</td>
<td>6469.26</td>
<td>38.69</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2149.51</td>
<td>6656.67</td>
<td>17.92</td>
<td>113.91</td>
<td>6486.45</td>
<td>38.39</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2148.87</td>
<td>6631.82</td>
<td>17.03</td>
<td>112.15</td>
<td>6462.87</td>
<td>39.77</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>2144.58</td>
<td>6602.92</td>
<td>17.12</td>
<td>112.40</td>
<td>6434.62</td>
<td>38.77</td>
<td></td>
</tr>
</tbody>
</table>
Table 49 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-10}$ concurrent = 16

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4 &amp; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2147.24</td>
<td>7276.42</td>
<td>17.02</td>
<td>113.03</td>
<td>7109.50</td>
<td>36.87</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2151.13</td>
<td>7230.17</td>
<td>17.76</td>
<td>113.02</td>
<td>7062.48</td>
<td>36.90</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2130.59</td>
<td>7259.45</td>
<td>17.08</td>
<td>112.13</td>
<td>7090.09</td>
<td>40.16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2119.74</td>
<td>7266.62</td>
<td>18.01</td>
<td>112.13</td>
<td>7099.41</td>
<td>37.07</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2129.76</td>
<td>7251.99</td>
<td>17.74</td>
<td>112.01</td>
<td>7081.67</td>
<td>40.57</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2148.59</td>
<td>7257.47</td>
<td>16.85</td>
<td>113.03</td>
<td>7088.29</td>
<td>39.29</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2163.66</td>
<td>7279.20</td>
<td>17.61</td>
<td>113.21</td>
<td>7105.68</td>
<td>42.69</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2146.96</td>
<td>7210.97</td>
<td>16.49</td>
<td>112.41</td>
<td>7044.31</td>
<td>37.76</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2136.40</td>
<td>7259.90</td>
<td>16.69</td>
<td>111.90</td>
<td>7090.89</td>
<td>40.42</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2154.92</td>
<td>7243.86</td>
<td>18.07</td>
<td>112.64</td>
<td>7073.50</td>
<td>39.65</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>2142.90</td>
<td>7253.60</td>
<td>17.33</td>
<td>112.55</td>
<td>7084.58</td>
<td>39.14</td>
<td></td>
</tr>
</tbody>
</table>

Table 50 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-12}$ concurrent = 16

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
<th>Compile</th>
<th>Total Execution</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4 &amp; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2144.61</td>
<td>10667.84</td>
<td>16.64</td>
<td>112.09</td>
<td>10499.08</td>
<td>40.04</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2192.38</td>
<td>10799.45</td>
<td>17.47</td>
<td>113.18</td>
<td>10630.49</td>
<td>38.31</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2121.18</td>
<td>10804.44</td>
<td>16.82</td>
<td>112.81</td>
<td>10634.24</td>
<td>40.58</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2152.93</td>
<td>10741.43</td>
<td>17.64</td>
<td>112.56</td>
<td>10572.32</td>
<td>38.91</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2138.27</td>
<td>10479.59</td>
<td>17.39</td>
<td>113.30</td>
<td>10307.76</td>
<td>41.14</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2153.08</td>
<td>10729.63</td>
<td>17.86</td>
<td>111.96</td>
<td>10561.87</td>
<td>37.94</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2160.65</td>
<td>10769.86</td>
<td>17.82</td>
<td>112.07</td>
<td>10601.68</td>
<td>38.29</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2114.67</td>
<td>10774.73</td>
<td>16.59</td>
<td>111.76</td>
<td>10609.01</td>
<td>37.37</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2156.20</td>
<td>10769.91</td>
<td>16.66</td>
<td>114.07</td>
<td>10600.50</td>
<td>38.68</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2172.74</td>
<td>10732.43</td>
<td>17.42</td>
<td>112.97</td>
<td>10562.80</td>
<td>39.25</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>2150.67</td>
<td>10726.93</td>
<td>17.23</td>
<td>112.68</td>
<td>10557.97</td>
<td>39.05</td>
<td></td>
</tr>
</tbody>
</table>
Table 51 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-16}$ concurrent = 16

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compile</td>
</tr>
<tr>
<td>1</td>
<td>2122.56</td>
</tr>
<tr>
<td>2</td>
<td>2138.78</td>
</tr>
<tr>
<td>3</td>
<td>2141.74</td>
</tr>
<tr>
<td>4</td>
<td>2141.25</td>
</tr>
<tr>
<td>5</td>
<td>2130.75</td>
</tr>
<tr>
<td>6</td>
<td>2138.10</td>
</tr>
<tr>
<td>7</td>
<td>2157.65</td>
</tr>
<tr>
<td>8</td>
<td>2147.37</td>
</tr>
<tr>
<td>9</td>
<td>2153.92</td>
</tr>
<tr>
<td>10</td>
<td>2145.54</td>
</tr>
<tr>
<td>Average</td>
<td>2141.77</td>
</tr>
</tbody>
</table>

Table 52 PCA implemented with OpenCL $\varepsilon = 1 \times 10^{-20}$ concurrent = 16

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compile</td>
</tr>
<tr>
<td>1</td>
<td>2144.29</td>
</tr>
<tr>
<td>2</td>
<td>2131.97</td>
</tr>
<tr>
<td>3</td>
<td>2148.27</td>
</tr>
<tr>
<td>4</td>
<td>2139.52</td>
</tr>
<tr>
<td>5</td>
<td>2173.58</td>
</tr>
<tr>
<td>6</td>
<td>2165.02</td>
</tr>
<tr>
<td>7</td>
<td>2126.04</td>
</tr>
<tr>
<td>8</td>
<td>2152.23</td>
</tr>
<tr>
<td>9</td>
<td>2172.28</td>
</tr>
<tr>
<td>10</td>
<td>2137.40</td>
</tr>
<tr>
<td>Average</td>
<td>2149.06</td>
</tr>
</tbody>
</table>
### b. SVM Algorithm

Table 53 SVM with OpenCL execution times

<table>
<thead>
<tr>
<th>Exe.</th>
<th>Time (ms)</th>
<th>Compile</th>
<th>Projection</th>
<th>Decision</th>
<th>Total Read</th>
<th>Total Send</th>
<th>Raw process</th>
<th>Total Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1042.77</td>
<td>109.64</td>
<td>3.93</td>
<td>0.39</td>
<td>47.03</td>
<td>66.15</td>
<td>113.57</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1042.80</td>
<td>110.80</td>
<td>3.31</td>
<td>0.37</td>
<td>47.12</td>
<td>66.62</td>
<td>114.11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1061.54</td>
<td>108.91</td>
<td>3.81</td>
<td>0.42</td>
<td>47.12</td>
<td>65.19</td>
<td>112.72</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1035.53</td>
<td>110.34</td>
<td>3.80</td>
<td>0.43</td>
<td>47.27</td>
<td>66.44</td>
<td>114.14</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1073.21</td>
<td>108.77</td>
<td>3.50</td>
<td>0.38</td>
<td>47.07</td>
<td>64.81</td>
<td>112.27</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1049.74</td>
<td>109.85</td>
<td>4.71</td>
<td>0.37</td>
<td>47.11</td>
<td>67.08</td>
<td>114.56</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1075.96</td>
<td>111.55</td>
<td>4.34</td>
<td>0.37</td>
<td>47.08</td>
<td>68.44</td>
<td>115.89</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1057.99</td>
<td>109.37</td>
<td>3.40</td>
<td>0.40</td>
<td>47.08</td>
<td>65.29</td>
<td>112.77</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1066.63</td>
<td>111.05</td>
<td>3.67</td>
<td>0.36</td>
<td>47.06</td>
<td>67.30</td>
<td>114.72</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1062.17</td>
<td>109.93</td>
<td>3.80</td>
<td>0.40</td>
<td>47.11</td>
<td>66.23</td>
<td>113.73</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>1056.84</td>
<td>110.02</td>
<td>3.83</td>
<td>0.39</td>
<td>47.10</td>
<td>66.35</td>
<td>113.85</td>
<td></td>
</tr>
</tbody>
</table>