ENERGY-EFFICIENT PARALLEL AND DISTRIBUTED MULTI-OBJECTIVE FEATURE SELECTION ON HETEROGENEOUS ARCHITECTURES

Thesis submitted by

JUAN JOSÉ ESCOBAR PÉREZ

To obtain the International Ph.D. degree as part of the

DOCTORAL PROGRAM IN INFORMATION AND COMMUNICATION TECHNOLOGIES

Supervisors

MIGUEL DAMAS HERMOSO
JESÚS GONZÁLEZ PEÑALVER

February 20, 2020

Editor: Universidad de Granada. Tesis Doctorales
Autor: Juan José Escobar Pérez
ISBN: 978-84-1306-636-3
URI: http://hdl.handle.net/10481/63898

A mi sobrina Laura,
para que algún día recojas el
testigo y continúes el camino

«*Yo no creo en eso de los grandes genios. Pienso en gente que trabaja y ha trabajado mucho, y que tiene talento*»

— Francisco Sánchez Gómez (Paco de Lucía)

ACKNOWLEDGEMENTS

Os agradezco todo lo que habéis hecho por mí, por ser mis guardianes día y noche, por tenderme la mano cuando me sentía perdido y confiar siempre en mí. Entiendo que no ha sido fácil: pensábais que me gastaría la herencia en videojuegos y maquinicas, y al final nos pasamos las horas en casa jugando juntos al dominó. A veces me ofusco y rabio, pero aunque no lo demuestre a menudo, os quiero más que a nada.

Papá y Mamá

Tengo la suerte de contar con mis dos ángeles más preciados. Parecen estar ausentes, como volando lejos, pero me observan, atentos en desplegar sus alas para levantarme. A ellos les agradezco, desde pequeñito, que sean mi modelo a seguir, motivándome día a día a seguir creciendo, con tesón, y aun así no los alcanzo, porque ángeles son...

Eva Belén y Rosa María

Gracias por oír mis gruñidos. Te quiero siempre a mi lado, disfrutando de la comida, de tus viajes... Oye. ¿Crees que ya está abierto el Kōkyo?

Pilar

Gracias chicos por los momentos de risas, de comentarios que a nadie le importa, de estar "trabajando" y basta con deciros «¡Hay una incursión en la iglesia!», para dejarlo todo y salir disparados a cazar Pokémons. ¡Y todo eso dentro del tiempo que íbais por el despacho! Un 5% vamos...

Los antiguos compis del D1-7 que se den por aludidos

He aprendido mucho de tí, procuras que no me falte de nada y aún con las adversidades de la vida, no te rindes, sigues adelante, aprendiendo y enseñando. Gracias por todo, nunca lo olvidaré.

Julio

A mis supervisores agradezco la oportunidad de realizar esta Tesis, de poder avanzar y aportar mi granito de arena al grupo de investigación.

Miguel y Jesús

Thank you for your company and support during my stay in Colchester. I still remember the jokes and the funny moments, especially when we used gestures to explain some things due to my English level was... :D.

Eirini and Rukiye

ACRONYMS

A

ACPI	Advanced Configuration and Power Interface
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
ANOVA	ANalysis Of VAriance
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit

B

B&B	Branch-and-Bound
BCI	Brain-Computer Interface
BCSS	Between-Cluster Sum of Squares

C

CMOS	Complementary Metal–Oxide Semiconductor
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSV	Comma-Separated Values
CU	Compute Unit
CUDA	Compute Unified Device Architecture
CVI	Cluster Validity Index

D

DBN	Deep Belief Network
DNA	Deoxyribonucleic Acid
DPM	Dynamic Power Management
DPS	Decisive Path Scheduling
DSSI	Dynamic Scheduling of Subpopulations or Individuals
DTLZ	Deb, Thiele, Laumanns, and Zitzler
DVFS	Dynamic Voltage and Frequency Scaling
DVS	Dynamic Voltage Scaling

E

EA	Evolutionary Algorithm
ECoG	Electrocorticography
EEG	Electroencephalogram

ERD	Event-Related Synchronization
ERDF	European Regional Development Fund
ERS	Event-Related Desynchronization
ES	Evolution Strategy

F

FPGA	Field-Programmable Gate Array
FS	Feature Selection

G

GA	Genetic Algorithm
GCC	GNU Compiler Collection
GPGPU	General-Purpose computing on Graphics Processing Unit
GPU	Graphic Processing Unit

H

HBM	High-Bandwidth Memory
HBP	Human Brain Project
HPC	High-Performance Computing

I

I/O	Input/Output
ICC	Intel C++ Compiler
ID	Identifier
IoT	Internet of Things

K

KNN	K-Nearest Neighbors

M

MI	Motor Imagery
MICIU	MInistry of sCience, Innovation, and Universities
MIMD	Multiple Instruction Multiple Data
MINECO	MINistry of Economy and COmpetitiveness
MISD	Multiple Instruction Single Data
MOGA	Multi-Objective Genetic Algorithm
MPI	Message Passing Interface
MRA	MultiResolution Analysis

N

NDS	Non-Dominated Sorting

NRMSE Normalized Root-Mean-Squared Error
NSGA Non-dominated Sorting Genetic Algorithm
NUMA Non-Uniform Memory Access

P

PAFS Productivity-Aware Frequency Scaling
PC Personal Computer
PCIe Peripheral Component Interconnect Express
PE Processing Element

R

R&D Research and Development
RAM Random Access Memory
RAW Read-After-Write
ReLU Rectifier Linear Unit

S

SA Simulated Annealing
SBX Simulated Binary Crossover
SIMD Single Instruction Multiple Data
SISD Single Instruction Single Data
SM Streaming Multiprocessor
SPM Static Power Management
SPMD Single Program Multiple Data
SVL Supply Voltage Level

T

TCP Transmission Control Protocol
TDP Thermal Design Power
TLP Thread-Level Parallelism
TSP Travelling Salesman Problem

U

UMA Uniform Memory Access
USB Universal Serial Bus

W

WCSS Within-Cluster Sum of Squares

X

XML eXtensible Markup Language

Z

ZDT Zitzler, Deb, and Thiele

ABSTRACT

The objectives and requirements of computer science are constantly changing. Nowadays, when developing an algorithm, it is not enough to solve the problem itself since the energy-time performance or memory usage should also be taken into account, especially in high-dimensional problems such as FS. For some years, energy-aware computing has gained importance as it allows data centers to save costs by reducing energy consumption, and it is even today a topic of global interest due to environmental reasons. The present trend in the development of computer architectures that offer improvements in both performance and energy efficiency has provided distributed platforms with interconnected nodes including multiple multi-core CPUs and accelerators. In these so-called heterogeneous systems, the applications can take advantage of different parallelism levels according to the characteristics of the devices in the platform. Precisely, these differences between computing devices are what make heterogeneous computing, unlike homogeneous, present other problems to deal with. However, this process is not automatic and requires the intervention of the developer to properly program the applications and thus achieve good results.

With this in mind, the objective of this thesis is the development of parallel and energy-efficient codes for time-demanding problems that frequently appear in bioinformatics and biomedical engineering applications. Specifically, this thesis tackles with unsupervised EEG classification, which is one of the aforementioned high-dimensional problems due to the characteristics of the EEG signals. To cope with the high number of features that each EEG contains, the implemented procedures are based on a multi-objective FS approach. The codes have been designed to take advantage of the heterogeneous architectures by exploiting the computing capabilities of their devices. In addition, they have been developed in a procedural way due to their complexity. This thesis also studies and compares the codes to identify the advantages and drawbacks of each, as well as analyzes the performance behavior in terms of energy consumption, execution times, and quality of the solutions under different situations such as workload, available computing resources, device clock frequency, and others that will be described in the corresponding chapters. The results show the importance of developing efficient methods to meet the energy-time requirements, pointing out the methodology to be followed and demonstrating that energy-aware computing is the way to continue on the right track.

FS: Feature Selection

CPU: Central Processing Unit

EEG: Electroencephalogram

RESUMEN

Los objetivos y requisitos de las ciencias de la computación cambian constantemente. Hoy día, los algoritmos deben desarrollarse pensando tanto en el problema a resolver como en factores relacionados con la energía, el tiempo y el uso de memoria, especialmente en problemas de alta dimensionalidad como la FS. Desde hace años, la computación eficiente ha ganado importancia ya que permite a los centros de datos ahorrar costes al reducir el consumo energético, siendo actualmente un tema de interés mundial por razones medioambientales. La tendencia actual en arquitectura de computadores está proporcionando mejoras de rendimiento a través de plataformas distribuidas y heterogéneas cuyos nodos interconectados incluyen CPUs multi-núcleo y aceleradores. En estos sistemas, las aplicaciones pueden aprovechar diferentes niveles de paralelismo según las características de sus dispositivos. Sin embargo, las diferencias entre dispositivos hacen que la computación heterogénea, a diferencia de la homogénea, presente otros inconvenientes que también deben tratarse. Como este proceso no es automático, la intervención del desarrollador para programar adecuadamente las aplicaciones y lograr buenos resultados es necesaria.

Con esto en mente, el objetivo de esta tesis es desarrollar códigos paralelos y energéticamente eficientes para problemas costosos en tiempo que aparecen con frecuencia en aplicaciones de bioinformática e ingeniería biomédica. Específicamente, esta tesis trata con la clasificación no supervisada de señales EEG ya que es uno de los problemas de alta dimensionalidad mencionados anteriormente. Para hacer frente a la gran cantidad de características que cada EEG contiene, los procedimientos implementados hacen uso de la FS multi-objetivo y aprovechan las capacidades computacionales de los dispositivos presentes en las plataformas heterogéneas utilizadas. Además, han tenido que ser desarrollados de forma procedural debido a su gran complejidad. A lo largo de esta tesis, todos los procedimientos son evaluados para identificar sus ventajas e inconvenientes. El rendimiento de cada uno de ellos es analizado en términos de consumo energético, tiempo de ejecución y calidad de las soluciones bajo diferentes condiciones experimentales tales como la carga de trabajo, recursos de cómputo disponibles, frecuencia de operación del dispositivo y otras que se describirán en el capítulo correspondiente. Los resultados muestran la importancia de desarrollar métodos eficientes para cumplir con los requisitos de tiempo y energía, señalando la metodología a seguir y demostrando que la computación energéticamente eficiente es el camino a seguir.

FS: Del Inglés: Feature Selection

CPU: Del Inglés: Central Processing Unit

EEG: Del Inglés: Electroencephalogram

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF EXPERIMENTS

Part I

PRELIMINARY & BACKGROUND

1

INTRODUCTION

CONTENTS

Present heterogeneous computer architectures allow different strategies to accelerate the applications and optimize their energy consumption according to specific power-performance trade-offs. The needs of the market have caused that the computer systems are no longer only formed by general purpose processors, i.e., CPUs, but also include accelerators that act as coprocessors and whose purpose is to address more specific tasks. *Machine Learning* tasks for classification, clustering, FS, and optimization problems are present in many useful applications in the field of bioengineering, which usually require high-performance platforms whose cost in both economic and environmental terms should be carefully taken into account. Indeed, the energy-aware computing has emerged as one of the main issues of research in computer systems, and efficiency today not only means good speedups but also optimal energy consumption. Moreover, some tasks present in *Machine Learning* also show different processing characteristics and thus diverse profiles in energy consumption when they are parallelized for heterogeneous systems.

Storage technology and distributed platforms, including networks, sensors, and other data capture devices, make it possible to have large datasets, from which applications of high socio-economic interest are emerging. By integrating systems with increasing processing and communication capabilities in a wide variety of everyday devices, applications with different requirements in terms of speed, energy consumption, or portability can be addressed, leading to paradigms such as IoT, *Cloud Computing*, or *Big Data*. These paradigms imply important transformations in the way in which they interact with the environment and access to information and communication technologies.

IoT: Internet of Things

1.1 CONTEXT AND MOTIVATION

The rapid development of digitalization together with the adoption of technology by people, is contributing to increase dramatically the ever-growing amount of data created in the world. According to a study of the *Statista* online portal [1], in the last decade the volume of data generated per year has increased from 2 ZB to 41 ZB, as can be seen in Figure 1.1. In addition, *Statista* also estimates 612 ZB and 2,142 ZB for 2030 and 2035, respectively, which indicates an exponential growth of generated data.

Currently, information and data are synonyms for knowledge in the field of computer science, and it is well-known that knowledge is power. Companies are investing a lot of effort in capturing and analyzing that data to later sell them to third parties due to the economic interests generated by the traffic and use of this information, leaving in the background the methodology of their processing and therefore, the use of hardware resources.

However, it is not possible to create applications capable of analyzing large datasets or high-dimensional data without the efficient use, in terms of performance and energy consumption, of parallel and heterogeneous computer architectures. The grouping of these computers as a whole is called a *cluster*, and can include multiple accelerators, such as GPUs and storage resources managed by distributed file systems that allow the processing of large amount of existing data. To all this the problem of climate change is added, which together with the increase in the world population, is creating the need to optimize the use of resources, especially energy consumption, not only in smaller and domestic devices but also in large supercomputers.

GPU: Graphic Processing Unit

On the other hand, high-dimensional multi-objective optimization opens promising approaches to *Machine Learning* applications once efficient parallel procedures are available. Frequently, these kind of applications require simultaneous optimization of several conflicting objectives, and it is quite common to use EAs to solve the problem [2, 3]. The goals of these applications deal with discovering useful models on large datasets, which include high-dimensional samples. Both unsupervised and supervised EEG classification are good examples of applications that deal with these issues.

EA: Evolutionary Algorithm

EEG classification is motivated by the current importance of the study of the brain. In fact, there are several projects worldwide created specifically for this, such as the *BRAIN Initiative*, proposed by the United

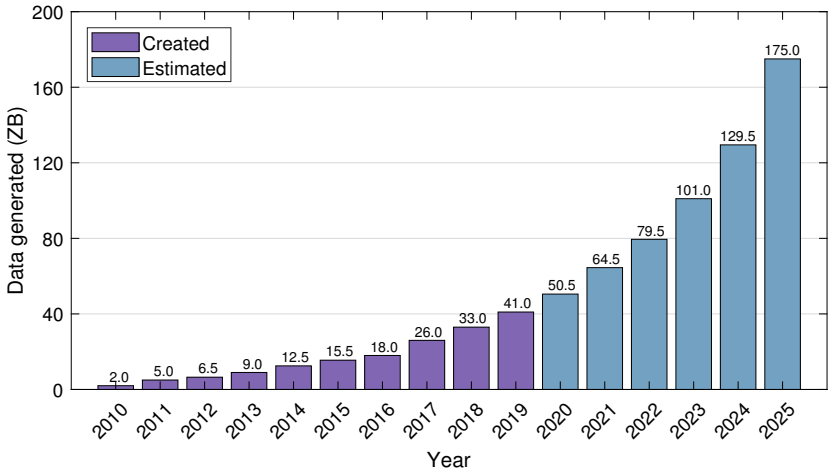


Figure 1.1: Volume of data worldwide since 2010. Data source: *Statista Digital Economy Compass 2019* [1].

States government in 2013, or the HBP, funded under the *Horizon 2020* framework[1]. An EEG is a multivariate signal whose classification has to cope with difficult problems. The solution to these problems usually implies the definition of high-dimensional feature vectors, despite the fact that few EEG signals are available to determine the classifier parameters. This way, FS techniques should be applied to remove noisy, irrelevant features, or to improve the learning accuracy and result comprehensibility of an EEG whenever its number of features is higher than the number of available signals. However, as mentioned before, none of this is possible without the development of efficient algorithms that take advantage of existing heterogeneous computer architectures.

HBP: Human Brain Project

1.2 OBJECTIVES

Taking into account the motivation shown in Section 1.1, a deep literature review will be carried out first to identify the fields of greatest potential within the context of heterogeneous and distributed programming, energy-aware computing, as well as the current progress of EAs to cope with EEG classification. Therefore, in general the objective is to make relevant contributions in parallel and energy-efficient multi-objective optimization for applications related to neuroengineering, since they are also potentially useful in rehabilitation technologies and medical image processing. The specific objectives are detailed below.

1 Program that finances various research and innovation projects in the European context

1.2.1 *First Approach for Multi-objective Optimization*

The first step is to create a first functional version. Although there are already tools to carry it out, the best way to have full control over the implementation for future optimizations is through a design that starts from scratch. With that in mind, the following objectives are addressed:

GA: GENETIC ALGORITHM

NSGA: NON-DOMINATED SORTING GENETIC ALGORITHM

1. Development of the first functional version, not yet parallel, on which the following implementations will be based. This application will be a multi-objective procedure that allows in general to address the analysis of large volumes of data, although it will be focused on the EEG classification problem. In addition, it will be based on a *wrapper* approach where a GA (a kind of EA) performs the FS technique to decrease the dimensionality of the dataset, i.e., an EEG dataset. Specifically, the well-known NSGA-II algorithm [4] is considered to address the multi-objective problem since it uses a GA as an optimization method.

2. Design an efficient learning method. *K-means* algorithm is proposed as a clustering method for unsupervised classification. Initially, a sequential version for CPU will be developed since it is the primary component of a computer. After validating the implemented procedure, a parallel version capable of taking advantage of the multiple cores present in this kind of microprocessors will be created. Moreover, as it is quite common today to find GPU accelerators on high-performance heterogeneous systems, a third version adapted to them will also be implemented. None of the first parallel versions will alter the behavior of the sequential program, so it is guaranteed that the quality of the solutions obtained by the classifier is exactly the same in all of them.

3. Optimize the GPU implementation of the classifier. The characteristics of the GPU architecture differ greatly from those of the CPU, as well as being more complex. However, they allow a greater degree of parallelism, which provides less execution time and energy consumption. In this way, it is necessary to focus on this type of architecture to take advantage of its full potential.

4. Experimental analysis to determine the performance of the developed algorithms. The analysis will be mainly focused on the execution times, energy consumption, and quality of the solutions obtained to identify possible improvements and optimizations. To do that, a set of tests will be designed to evaluate the behavior of the program under different experimental conditions.

1.2.2 *New Paradigms for Distributed Computing*

Once the first approaches have been obtained, the following objectives related to the optimization, expansion, and exploration of new alternatives to the implemented algorithms are proposed:

1. Adapt the algorithms already implemented to the distributed high-performance architectures. The objective is to take advantage of the resources of these platforms by balancing the computing cost and using asynchronous communication between nodes. Again, the goal is to reduce execution times and energy consumption without losing quality in the solutions obtained.

2. Creation of new procedures taking into account the connection topology of the nodes provided by distributed *clusters*. Thus, it is intended to create new alternatives to previously developed GAs with the aim of improving the quality of the solutions obtained. These procedures will be based on existing paradigms such as the distributed multi-population approaches.

3. Design of workload distribution strategies for heterogeneous platforms. This offers the possibility of full cooperation between CPU and GPU to distribute the work. The new implementations will be subjected to several experiments to evaluate their behavior, both in ideal and adverse conditions: tasks with unbalanced and/or low workload, or the disparity in computing capacity between devices or nodes are some of the situations to deal with.

4. Build an energy-time model according to the elements present in the platform and adjust the corresponding parameters by multiple linear regression from the experimental measurements. The purposes are: (i) predict the energy consumption and execution times of a given workload distribution in the computing platform at hand and (ii) understand the behavior of the program to identify the situations to improve.

5. Use supervised classification as an alternative method to GA and *K-means* algorithm. For this study, the use of neural networks is proposed, and more specifically CNNs, since they are currently gaining great importance in data analysis and implicitly perform FS. This does not mean that the previous implementations are in disuse, since the strategies of workload distribution are perfectly applicable in this area.

CNN: Convolutional Neural Network

1.3 THESIS STRUCTURE

This section provides a brief description of each of the chapters that make up this thesis with the objective of providing a global view of its structure. The document is divided into three main parts depending on their type of content. Each of these parts is made up of more than one chapter, including appendices and bibliography. In this way, the structure of the thesis is as follows:

PART I. PRELIMINARY & BACKGROUND

- Chapter 1. Introduction: this chapter introduces the problem considered in this thesis, the objectives that it proposes and allows to know what are the motivations to carry it out. An overview of the structure of the document is also provided in Section 1.3.

- Chapter 2. Computer Architectures: this chapter presents the technological fundamentals that provide the basic hardware knowledge for the study carried out in this thesis: the classification of computer systems, code optimization, and parallel programming languages are exposed as an introduction to computer architectures. Nevertheless, it goes deeper into the part related to parallel and heterogeneous systems and energy-aware computing due to its importance for the development of the thesis.

- Chapter 3. Bioinspired Multi-objective Optimization: this chapter, following the line of Chapter 2, presents the more relevant topics related to software applications and algorithms used: the problem of EEG classification, *K-means* algorithm, neural networks, or the theory of evolutionary computing are some of the topics addressed in this chapter.

PART II. CASE STUDY & DISCUSSION

- Chapter 4. Methodology: this chapter shows the methodology applied, analysis methods, evaluation metrics, and experimental conditions to which the experiments of Chapters 5 and 6 will be subjected. The resources used for the evaluation of the algorithms, such as the origin of the EEG dataset, programming tools, or the energy measurement system are also described. The chapter presents the different algorithms to be evaluated and their particularities, as well as the characteristics of the platforms and devices on which the codes will be executed, i.e., a desktop PC, the heterogeneous *cluster*, and the CPU and GPU devices.

PC: Personal Computer

- Chapter 5. Development on Single-computer Systems: this chapter is focused on the development of procedures for monocomputer systems. A first version of the MOGA is created, which serves as a basis for the following implementations. The developed versions are analyzed and compared in terms of execution times, energy consumption, and quality of the solutions. Moreover, an energy-time model is built to explain and predict the behavior of the algorithms in a heterogeneous system.

MOGA: Multi-Objective Genetic Algorithm

- Chapter 6. Development on Distributed Systems: this chapter deals with the development of new implementations adapted to multi-computer systems, as well as to explore new solutions based on CNNs. The advantages and limitations of each implementation are also discussed, and the energy-time model of Chapter 5 is modified for integration with distributed systems.

- Chapter 7. Conclusions: this chapter briefly presents the final conclusions according to the results obtained and the objectives proposed in this thesis. Possible future works are also exposed.

Part III. APPENDICES & BIBLIOGRAPHY

- Appendix A. Publications: this appendix lists the different publications obtained during the course of the thesis. Publications in international journals with impact factor are included, as well as publications in international and national conferences.

- Appendix B. Wattmeter for Energy Measurements: this appendix describes in detail the wattmeter used to measure the energy consumption of the developed algorithms.

- Appendix C. Getting Started Guide to HPMoon: this appendix provides a user guide for those interested in using the software developed in this thesis. Specifically, the best version of the algorithms implemented. The explanations range from the compilation process to its execution and operation, including some tips to get the most out of the application since the program has certain limitations, which will be explained in Chapter 6.

- Appendix D. Grants and Special Acknowledgements: this appendix lists the grants received for the development of the thesis and thanks other researchers for their help and support.

- Bibliography: the bibliography lists the set of scientific publications and web links that support the content of this thesis.

2

COMPUTER ARCHITECTURES

CONTENTS

The word *computer* is derived from the word *compute*, which means *to calculate*. The computer was originally defined as a super fast calculator with the capacity to solve complex arithmetic and scientific problems at very high speed. Nowadays, in addition to handling complex arithmetic computations, computers perform many other tasks like accepting, sorting, selecting, moving, and comparing information. They also perform arithmetic-logical operations on alphabetic, numeric, and other types of information, which are known as *data* and whose growth is currently skyrocketing.

However, today almost all the people in the world make use of computers in one way or another, being part of our lives and conditioning the way we live. They are present everywhere, and are no longer limited to sophisticated areas such as science: cars, video games, vending machines, washing machines, PCs, and in general those devices we use daily. In addition, its use also is interesting for industry and companies since they have interests for commercial purposes. All this has been possible thanks to the constant evolution of computer architectures from its origins, but like any technology, it has a limit and if there are no new ways to move forward, that evolution slows down. In fact, it

is already happening, since one of the laws that shows the evolution in the number of transistors per microprocessor for almost 60 years, is nearing its end. This law is known as the Moore's law [5], and states the following:

> The number of transistors on a microchip doubles about every two years, though the cost of computers is halved.

The problem is that computing capacity currently evolves thanks to the reduction of lithography in the manufacture of integrated circuits, and the forecast is that this law will be valid for about 10 more years [6]. The miniaturization of transistors is becoming increasingly difficult since silicon, the main element of microprocessors, has physical limitations [7]. Although much research is being done to move forward with new forms of computing, such as quantum computing [8] or graphene [9] to build new superconductors, they are not viable today.

HBM: High-Bandwidth Memory

For now, a more feasible solution could be to design processors with a 3D structure, similar to that already used in HBM memories, allowing the increase of storage capacity and speed without altering their physical size. However, unlike memory, processors are much more complex, so the challenge is to create 3D-stacked layers of transistors that are interconnected with each other. In any case, while these technologies arrive, and since parallel data computing is also booming due to its multiple scientific applications in bioinformatics, medicine, cosmology, among others, the need to change the computing paradigm is created in order to improve the performance of algorithms and applications.

HPC: High-Performance Computing

In this context, HPC systems are presented as one of the alternatives to overcome the hardware limitations. They are highly demanded to perform heavy tasks in simulation and prediction, such as fluid dynamics, aeronautics, thermodynamics, meteorology, or the proteins synthesis to obtain new drugs, which is of vital importance for pharmaceutical companies. For some years, the economic cost and time required to acquire HPC systems has dropped dramatically, and the multiple hardware architectures that arise allow to choose the right platform according to the requirements of the problem to be solved. Also, they represent a fast response to the continuous market demands.

Hereafter, some of the current alternatives to cope the problem described above are shown, as well as a theoretical basis that helps to address the studies in Chapters 5 and 6.

2.1 CLASSIFICATION OF COMPUTER SYSTEMS

Computer systems can be classified into four major categories using the well-known Flynn's taxonomy [10]. The classification is done according to the instruction and data streams that can be executed simultaneously in the PEs, as can be seen in Figure 2.1. The different types of systems are described below:

PE: Processing Element

- **SISD**: system with a uniprocessor machine capable of executing a single instruction, operating on a single data stream. The result of the execution is the same as if the instructions were executed sequentially. However, it is known that sequential processors do not exactly follow this model since it is usual to insert multiple functional units or use the *pipelining* technique, introducing some parallelism in the execution of the instructions.

SISD: Single Instruction Single Data

- **SIMD**: system with a multiprocessor machine capable of executing the same instruction for all PEs, but operating on different data streams. Operations with vectors or matrices are very suitable for SIMD systems.

SIMD: Single Instruction Multiple Data

- **MISD**: system with a multiprocessor machine capable of executing different instructions on different PEs, but all of them operating on the same data stream. These systems are not commercially available due to their practical limitations.

MISD: Multiple Instruction Single Data

- **MIMD**: system with a multiprocessor machine capable of executing different instructions on multiple data streams. Unlike the SIMD and MISD systems, the PEs work asynchronously. Currently, most computers are MIMD systems because they allow to run any type of application. In addition, they are divided into two subtypes: shared memory systems and distributed memory systems, depending on the way in which PEs access memory. In later sections, this type of systems will be discussed.

MIMD: Multiple Instruction Multiple Data

2.2 CODE OPTIMIZATION

Compilers incorporate a code optimization stage, in which from the original source code they generate an optimized intermediate code that takes advantage of the processor architecture. Despite the advances in compilers in recent years, the degree of optimization that a compiler can reach is not comparable to what a programmer with knowledge

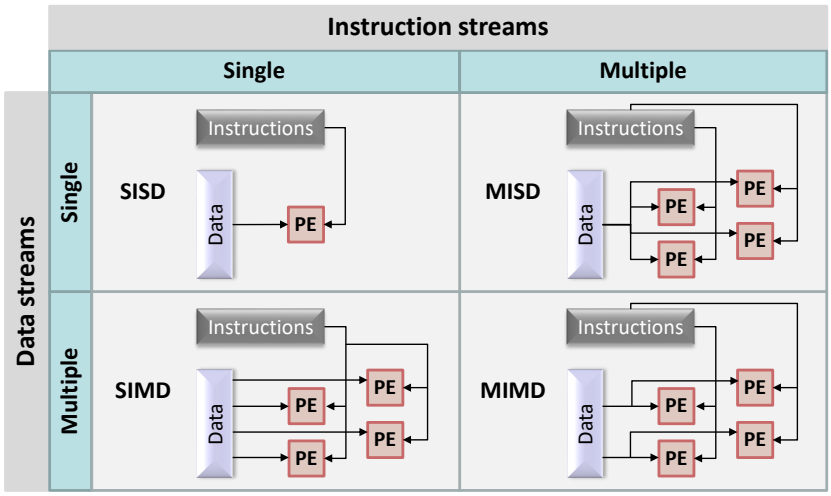


Figure 2.1: Flynn's taxonomy [10]. Classification of computer systems.

of the processor architecture could achieve. There are mainly two kind of optimizations to improve the performance of an application depending on whether the architecture should be taken into account or not. Table 2.1 summarizes the optimization possibilities described in Sections 2.2.1 and 2.2.2.

2.2.1 *Architecture-independent Optimizations*

These types of optimizations are valid for any processor since they do not depend on how the architecture is internally designed. Even some of them are quite intuitive and easy to apply. The optimizations are listed below:

RAW: Read-After-Write

- **Code refactoring**: useful to avoid data dependency, e.g., RAW dependencies. Basically it means that an operation cannot be completed until the previous one has finished, leaving an instruction paused on the pipeline.

- **Remove unused variables**: some variables that are no longer necessary remain in the code, consuming resources. This situation is caused by the constant changes in the source code.

- **Loop-invariant code motion**: this occurs when some operations are repeated throughout the iterations of a loop. The solution is

Table 2.1: Independent and dependent-architecture optimizations.

#	Independent	Dependent
1	Code refactoring	Use of registers
2	Remove unused variables	Pipelining technique
3	Loop-invariant code motion	Use of SIMD instructions
4	Strength reduction	Multi-threaded programming
5	Minimize dynamic memory use	-
6	Set an induction variable	-

to move those operations outside the body of the loop since they will not affect the semantics of the program.

- **Strength reduction**: some operations are replaced with equivalent but less expensive operations. E.g., replacing an exponentiation with an addition, a division with a multiplication plus an addition, or some multiplications with bit level operations.

- **Minimize dynamic memory use**: the abuse of requests to the operating system to reserve memory at runtime is very expensive in terms of performance. Therefore, as far as possible, static memory should be used so that the memory allocation is made at compile time.

- **Set an induction variable**: An induction variable or recursive strength reduction replaces a function of some systematically changing variable with a simpler calculation using previous function's values.

2.2.2 *Architecture-dependent Optimizations*

When the code is modified to be more efficient on a particular machine, an architecture-dependent optimization is being performed. For this kind of optimization, it is required to know the features of the target machine to produce code that runs faster. Currently, compilers are becoming more sophisticated and allows part of these optimizations, but as commented before, the intervention of the programmer usually improves the optimization. Some of those optimizations are as follows:

- **Use of registers**: registers are the most critical processor resources since accessing them is much faster than memory access. In addition, the number of instructions goes down since there is no need for load and store instructions. However, the number of available registers is very limited, so it is convenient to use the registers to store variables that are used very frequently.

- **Pipelining technique**: *pipelining*, or pipeline concurrency, consists on the possibility of organizing the hardware to execute more than one instruction simultaneously, reducing the number of cycles per instruction by a factor equal to the depth of the pipeline or number of stages that a CPU has. As with an affordable cost the performance improvement is remarkable, all current general purpose microprocessors incorporate this technique. However, increasing the number of stages too much can reduce performance, since the amount of control logic needed to manage intermediate buffers and dependencies between pipeline stages also grows. E.g., the *Ice Lake* microarchitecture of the *Intel Corporation*[1] has pipelines with 14-19 stages.

- **Use of SIMD instructions**: instructions that allow operations with multiple scalar or floating-point data simultaneously. The set of available operations varies depending on the architecture used, although for years the number of SIMD instructions and the size of their records has been growing to accelerate certain operations. The typical set of SIMD instructions includes arithmetic-logical and load/store operations.

- **Multi-threaded programming**: multi-threaded programming is becoming one of the most relevant fields since the current trend is to integrate more processing cores in the same CPU chip. The appearance in recent years of multi-core processors allows, through parallel programming techniques, the opportunity to take advantage of all cores to increase performance in applications that require intensive calculation and have some degree of parallelization. This will be discussed in more detail in later sections since it is one of the main research focuses of this thesis.

The optimization process, contrary to what many people think, must be done during the coding process, not at the end. As this thesis is in line with the code optimization to obtain efficient parallel codes, the optimizations mentioned above have been taken into account during the development of the algorithms described in Chapters 5 and 6.

1 Semiconductor chip manufacturer that invented the x86 series of microprocessors

2.3 PARALLEL ARCHITECTURES

The need for parallel computing is caused by the limitations of sequential computers, which already left the race to increase the CPU clock speed. By integrating several processors it is possible to solve problems that require more memory or faster computing speed. There are also economic reasons, since the price of sequential computers is not proportional to their computational capacity. E.g., to acquire a machine four times more powerful it is usually necessary to invest more than four times its price. On the contrary, the acquisition of several computers interconnected through the network allows to obtain performances almost proportional to the number of processors but with a lower additional cost. Even so, CPU devices are not capable of performing some tasks or at least not efficiently even by grouping several computers. In addition, there is no expectation that they will improve significantly, since the rate of increase in computing power has started to fall because the R&D is more focused on energy efficiency to meet environmental requirements, and less on increasing the brute power. Therefore, new computing paradigms are necessary to address this issue. At this point, the use of parallel architectures and accelerators, in combination with heterogeneous platforms open alternatives for efficient computing. In the following sections, concepts related to parallelism and the tools for obtaining it are discussed. It is also emphasized the importance of energy-aware computing and the use of heterogeneous platforms in high-performance applications due to their relevance in the field.

R&D: Research and Development

2.3.1 *Multi-core CPUs and Multiprocessors*

The first response to poor performance of sequential microprocessors was the emergence of multi-core processors. Although its use has become popular in consumer computers in the early 21st century, in the mid-1980s the *Rockwell International* company[2] manufactured versions of the 6502 microprocessor[3] with two cores on a single chip. In general, multi-core microprocessors allow a computing device to provide parallelism through a TLP model without including multiple microprocessors in separate physical packages, or *dies*[4]. The instructions are ordinary instructions, such as add, move data, and branch, but these can be run on separate cores at the same time, increasing overall speed for programs that support multi-threading or other parallel computing

TLP: Thread-Level Parallelism

2 Was a manufacturing conglomerate involved in electronics and aircraft, among others
3 8-bit microprocessor designed by *MOS Technology, Inc.* in 1975
4 Small block of semiconducting material on which a functional circuit is fabricated

SPMD: Single Program Multiple Data

techniques. This allows instructions to be executed following the SPMD model, which should not be confused with the SIMD model seen in Section 2.1. The difference is that in SIMD each instruction processes multiple data elements using vector registers. In contrast, SPMD, which is a subtype of the MIMD model, splits the program across multiple cores that operate on different data subsets.

The parallelization of code can be carried out in different ways depending on the memory model available. That is, depending on whether the system has shared or distributed memory. Shared memory can be found on systems where several processors and cores share a single memory space, such as UMA multiprocessors. Therefore, the distribution of the cores is local since they are physically close to each other within the same motherboard. In this parallel programming model, the main problem comes when the number of processors is high. This seems contradictory to the idea that the more cores there are, the better execution time. The right thing is to say that up to a point.

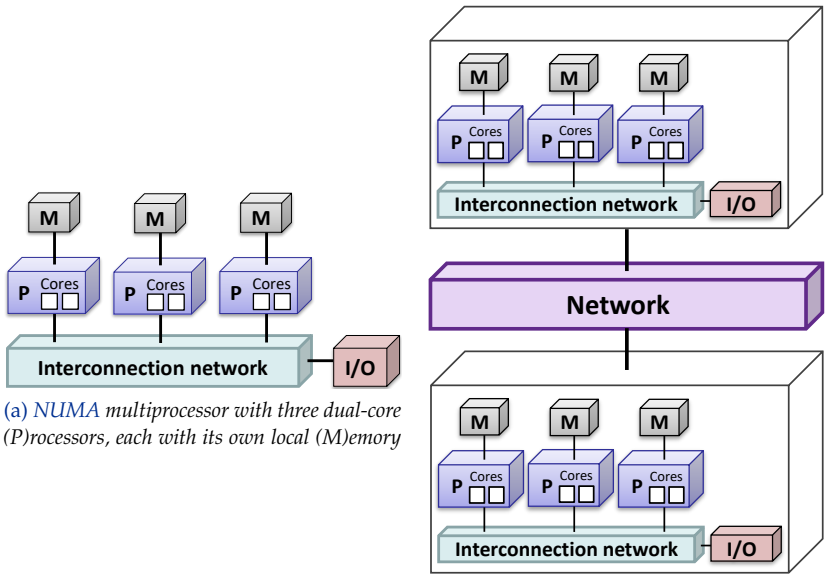
UMA: Uniform Memory Access

To explain this, the following case is considered: an algorithm does something usual like accessing memory to store or read data. However, access to RAM is slower than executing an instruction on a CPU core and there are also limitations on the amount of simultaneous memory accesses. Today, HPC systems have multi-channel memory architectures that allow simultaneous access to RAM. The problem is that currently the maximum number of channels does not usually exceed six, but the number of cores in the processors that these machines usually equip is around several tens. As a consequence, if an application requires many memory accesses, and since not all cores can access at the same time, arbitration policies are required to decide which one has permission to access memory. It may be the case that by considerably increasing the number of processors, the overhead imposed by this arbitration policy penalizes enough to make the performance even worse.

RAM: Random Access Memory

In distributed memory systems, such as multicomputers and NUMA multiprocessors (Figure 2.2), all processors have a local memory. The communication between CPUs in this model takes place through the interconnection network, which can be configured in several topologies depending on the purpose of that machine. The shared memory systems are easier to program but are less tolerant to failures because a simple failure affect the entire system, whereas this is not the case of the distributed model since each CPU can be easily isolated. Moreover, shared memory systems are less scalable because the addition of more CPUs leads to memory contention, as discussed above. For these reasons, multicomputer-based schemes are normally used for HPC since they usually meet the requirements of the users.

NUMA: Non-Uniform Memory Access



(a) *NUMA multiprocessor with three dual-core (P)rocessors, each with its own local (M)emory*

(b) *Multicomputer composed of two NUMAs*

Figure 2.2: Scheme of distributed memory systems.

Although many multiprocessors and multicomputers are available, if programming techniques are not used to extract parallelism, their potential will not be exploited. To facilitate these tasks, there are parallel programming tools that allow the programmer to design source code based on the available architecture. Among others, *OpenMP* [11] and MPI [12] enjoy the greatest popularity for their flexibility and potential. *OpenMP* is based on the paradigm of shared variables and compiler directives. Each of them has its advantages and disadvantages. *OpenMP* is easier to program and debug than MPI, and is integrated into the main compilers, so it is not necessary to install other libraries for its use. In addition, the directives can be added incrementally and the code can be executed as the sequential code would.

MPI: Message Passing Interface

The main disadvantage is that the resulting code can only be executed on a multiprocessor because multicomputers do not share RAM memory between nodes. MPI, meanwhile, works through message-passing. The parallel code obtained can be executed on both multiprocessors and multicomputers. This allows the MPI code to scale better than an *OpenMP* code because a multicomputer is expanded more easily by adding a new computer to the system if more memory or processors are needed. Also, each process has its own variables, so consistency problems cannot occur. However, depending on the application, performance may be limited by network communication between nodes.

2.3.2 *Accelerators*

The race to integrate more cores in the same integrated circuit did not start in the field of CPUs, but is preceded by a similar one in that of GPUs due to competition between manufacturers. These devices, or accelerators, emerged at the end of the 1990s fuelled by the demands of 3D graphic applications. Although initially they were used exclusively to execute shaders[5], thanks to the GPGPU [13], it is now also possible to use them for other purposes. Therefore, they have evolved towards parallel multi-threaded architectures with capabilities that could provide high performances, but with lower costs than CPUs since they devote more resources for data processing.

GPGPU: General-Purpose computing on Graphics Processing Unit

For several years, the GPUs incorporate hundreds or thousands of processing cores and large amounts of memory, offering unthinkable power so far. Although the term *core* has been used by both CPUs and GPUs manufacturers, it should be noted that it does not refer to exactly the same concept. In a CPU, each core is equivalent to a complete microprocessor, with its own registers, functional units, and even memory, although it shares memory, buses, and I/O lines with the other cores. The GPU cores are much simpler, since they are basically ALUs with the capacity to operate with floating-point data and carry out some operations, but not to execute general purpose code. However, recently GPUs incorporate cores for more advanced purposes. E.g., the GPUs of the *NVIDIA Corporation*[6] include cores to accelerate operations related to real-time ray-tracing[7], or the so-called *Tensor* cores, which are dedicated to *Deep Learning* and AI tasks. In order to facilitate programming with GPUs, *NVIDIA* has also developed its own libraries and the CUDA programming language, although they are only compatible with their own devices. However, languages such as *OpenCL* [14], which will be seen in more detail in Section 2.3.3, allow to create parallel code regardless of the manufacturer.

I/O: Input/Output

ALU: Arithmetic Logic Unit

AI: Artificial Intelligence

CUDA: Compute Unified Device Architecture

When parallelizing source code, it is necessary to be clear about the characteristics of the CPU and GPU devices to choose the one that best suits the program requirements. GPUs are ideal for SIMD parallelism. E.g., perform operations with vectors and matrices. To parallelize tasks in which different streams of instructions are applied to datasets that can be independent or not, it is convenient to resort to the CPU cores, since each of them can execute a different program, i.e., following the MIMD model. Table 2.2 shows the differences between the two devices.

5 Small programs that calculate rendering effects with a high degree of flexibility
6 Company specialized in the design of GPUs for workstations and consumer devices
7 Technique to calculate in real-time the reflection and refraction of light

Table 2.2: Differences between CPU and GPU.

#	CPU	GPU
1	General purpose processor	Graphics computing
2	Needs more memory than GPU	Minor memory consumption
3	High core clock rate	Moderate core clock rate
4	Has few powerful cores	Contain more weak cores
5	Low bandwidth	High bandwidth
6	Serial instruction processing	Parallel instruction processing
7	Emphasis on low latency	Emphasis on high throughput
8	Moderate energy consumption	High energy consumption

The GPU plays the role of a coprocessor connected, through a bus, to a CPU host that can share the RAM memory. During the execution of programs, data have to be transferred between the host memory and the GPU memory. The PEs are the basic computing elements of the GPU, and several of them, along with one or more instruction units and a register file comprise a SM. They do not contain instruction units and are only able to execute scalar operations. A GPU can include multiple SMs, which only has one program counter and allow the simultaneous execution of the same program on different data according to the SPMD model. This is done by multiple threads that are organized in blocks in such a way that all threads in a block are assigned to a single SM. Moreover, the blocks are also partitioned into *warps* containing threads with consecutive and increasing identity numbers that start together at the same program address. While the threads in a block are able to cooperate and share the instruction unit and the register file, threads in different blocks can only communicate through the off-chip memory.

SM: Streaming Multiprocessor

Contrary to the paradigm seen so far, other accelerators such as FPGAs can be reprogrammed according to the requirements of the desired application even after its manufacture. These semiconductor devices are based on an array of configurable logic blocks connected through programmable interconnections. This feature distinguishes them from ASICs that are custom-made for specific tasks. In addition, these accelerators usually have lower energy consumption than those of CPUs and GPUs, so they are being used in applications related to image processing, speech recognition, signal processing, or hardware emulation. Although FPGAs can be programmed in hardware description languages such as *Verilog*, the tendency is to use languages with syntaxes already known as *C++*.

FPGA: Field-Programmable Gate Array

ASIC: Application-Specific Integrated Circuit

2.3.3 *OpenCL as a Multi-platform Language*

Creating applications for heterogeneous platforms is difficult since the traditional programming approaches for multi-core CPUs, GPUs, and FPGAs are very different. Parallel programming in CPUs normally assumes the shared memory model and does not cover vector operations. The GPU programming model addresses complex memory hierarchies and vector operations with more specific hardware. FPGAs, in this sense, are more similar to GPUs due to their large number of configurable logic gates, which offer fine-grained parallelism since they have more independent threads than CPUs. In short, the differences in the hardware architecture of all these devices make it difficult for the developer to exploit their full potential. Also, the programming languages used for each device usually differ, which further complicates the work. At this point, it is clear that a common language is needed for the different platforms. The *OpenCL* language [14] is presented as a candidate to meet these requirements, which could be defined as:

> *OpenCL* is an open standard for parallel programming of heterogeneous platforms, which allows to design applications that run on a host and launch kernels on other devices.

This means that this language is multi-platform and therefore valid for both CPUs, GPUs, and even some FPGAs. In addition, *OpenCL* is compatible with a wide range of applications, from embedded software to HPC solutions and supports both data and task parallelism. By creating an efficient programming interface, *OpenCL* forms an abstraction layer on the hardware implementation while offering high performance. As an additional feature, it is also suitable in new interactive graphic applications that combine general parallel computation algorithms with 3D graphics rendering, since for example it offers interoperability with graphic processing APIs such as *OpenGL*[8]. The model consists of a central or host device connected to one or more *OpenCL* devices. An *OpenCL* application runs on the host according to the native model of the host platform, which sends commands to the devices. Based on the capabilities of *OpenCL*, the code developed for the kernels should be the same on all devices. However, this is far from reality since each device has a different hardware architecture and therefore it is necessary to make adjustments to the code to adapt and optimize it. Even depending on the type of code, it may not work properly.

API: Application Programming Interface

8 Cross-platform API for rendering 2D and 3D vector graphics

2.3.3.1 *Execution Model*

The execution of a program in *OpenCL* occurs in two parts: the kernels that run on one or more *OpenCL* devices and a host program that runs on the main machine. The host program defines the context for the kernels and manages their execution. When a kernel is sent for execution, an index space is defined, and an instance of the kernel is assigned to a point in the space. This instance is called *work-item* and is identified by its point in the index space, which provides a global ID for the *work-item*. Each *work-item* executes the same code, but could act on different datasets and execution paths.

ID: IDENTIFIER

Work-items are organized into *work-groups*, which provide a coarse-grained decomposition of the index space. These *work-groups* are identified by a unique *work-group* ID with the same dimensionality as the index space used for *work-items*. A *work-item* is also assigned a unique local ID within a *work-group*, and can be identified by both its global ID and the combination of its local ID and the ID of the *work-group* it belongs to. Finally, an *OpenCL* device is divided into one or more CUs, which can contain one or more *work-groups*.

CU: COMPUTE UNIT

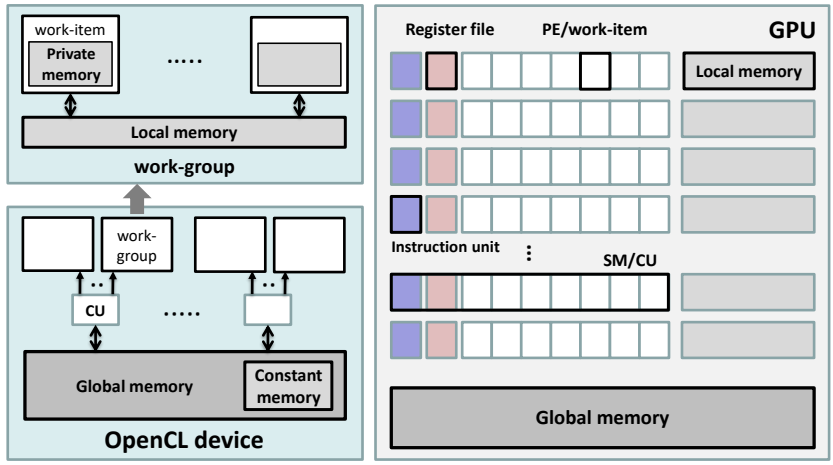
At this point, this way of structuring a computing device should be familiar. Indeed, this has been seen before when the CPU and GPU structures was discussed. Using the GPU case as a basis, the equivalence is as follows: the *OpenCL* device would be the GPU, a CU, an SM, and a *work-item*, a PE. A *work-group*, would be the grouping of several PEs that run concurrently within a SM. Figure 2.3 shows the relationship between the GPU architecture and the *OpenCL* device model.

2.3.3.2 *Memory Model*

OpenCL offers four different regions of memory that can be accessed by work-items when running a kernel:

- **Global**: this memory region allows read and write access to all *work-items* of all *work-groups*. *Work-items* can read and write on any element of a memory object. Readings and writes to global memory can be cached depending on the capabilities of the *OpenCL* device.

- **Constant**: it is a region of global memory that remains constant during the kernel execution. The host reserves and initializes memory objects located in this memory.



(a) *OpenCL device model* (b) *GPU architecture scheme*

Figure 2.3: Relationship between the *OpenCL* device model and GPU.

- **Local**: memory that is local to a *work-group*. This memory region can be used to store variables that can be shared by all *work-items* of that *work-group*. Depending on the *OpenCL* device, local memory can be implemented as a logical abstraction of global memory or as on-chip memory. E.g., GPUs usually offer very fast local memory, but also very limited due to its high cost.

- **Private**: private for each *work-item*, so the variables defined here are not visible by any other *work-item*.

Figure 2.3a represents the different memory regions and their relationship with the *OpenCL* device model. On the other hand, Table 2.3 describes the scope of the kernel and the host over the different memory regions, such as the type of memory allocation (static or dynamic) or the access allowed (read or write).

The application that runs on the host uses *OpenCL* API functions to create the necessary memory objects for the kernel and the command queues to operate with them. The host and the memory model of the *OpenCL* device are, in most cases, independent. This is necessary considering that the host is defined outside the *OpenCL* domains. However, sometimes they need to interact with each other, for which *OpenCL* offers two alternatives: explicitly copying the data or mapping and unmapping regions of a memory object, allowing the host to access the memory object from its address space. When the read or write operation is completed, the host unmaps the region.

Table 2.3: Kernel and host scope over the different *OpenCL* memory regions. R: Read; W: Write.

Code	Scope	Global	Constant	Local	Private
Host	**Allocation**	Dynamic	Dynamic	Dynamic	✗
	Access	R/W	R/W	✗	✗
Kernel	**Allocation**	✗	Static	Static	Static
	Access	R/W	R	R/W	R/W

OpenCL uses a relaxed memory consistency model, i.e., it is not guaranteed that the status of the memory visible to a *work-item* is consistent across all *work-items* at all times. In the private memory of a *work-item* there is consistency in the readings and writings. Local memory is consistent through *work-items* belonging to a particular *work-group* if synchronization barriers are used. The consistency of the global memory is similar to that of the local memory, but there is no guarantee of consistency between the different *work-groups* that run the kernel. Nor is consistency guaranteed for memory objects shared between commands already entered in the command queue.

2.3.3.3 *Programming Model*

The *OpenCL* programming model supports both data and task parallelism, and a hybridization between both models. However, the main model that *OpenCL* drives is data parallelism. The index space associated with the *OpenCL* execution model defines the *work-items* and how they are mapped onto the data. In a strictly data parallelism model, there is a one-to-one correlation between *work-items* and the data to be processed. *OpenCL* implements a relaxed and hierarchical version of the data parallelism model, so a strict one-to-one correlation is not a requirement. There are two ways to specify hierarchical subdivision. In the explicit model, the programmer defines the total number of *work-items* to execute in parallel and also how the *work-items* are divided into *work-groups*. In the implicit model, the programmer only specifies the total number of *work-items* to execute in parallel, and the division into *work-groups* is handled by the *OpenCL* implementation. In the task parallelism model, only one instance of the kernel is executed. This is equivalent to executing a kernel on a CU with only one *work-group* and one *work-item*, so the programmer must express the parallelism in another way. E.g., vectorizing the kernel following the SIMD model or entering the queue multiple kernels.

2.3.4 *Energy-aware Heterogeneous Computing*

The boom in data analysis, along with other paradigms such as IoT, *Deep Learning*, or the creation of advanced AIs, has shown that multicore processors and accelerators are no longer sufficient on their own for highly complex tasks. As discussed earlier in this chapter, HPC systems are presented as one of the alternatives to overcome the hardware limitations and address this problem. If in addition to CPU devices, other accelerators such as GPUs or FPGAs are incorporated to the equation, a heterogeneous platform is obtained. This type of platforms constitute the present mainstream approach to take advantage of technology improvements that these devices can offer [15]. In fact, many of the world's most powerful supercomputers, collected on the *TOP500* list [16], have heterogeneous configurations.

Besides offering opportunities to execute efficient parallel codes, the heterogeneous architectures including CPU and GPU devices could also constitute an efficient approach for energy-saving, and papers such as [17] consider the efficient cooperation between both devices as an important concern to reach exascale performances. This objective is very close to being fulfilled since there is also an incessant competitiveness among organizations to see who has the most powerful supercomputer. Figure 2.4 shows the performance evolution of supercomputers from 1993 to the present. Extrapolating the data, it seems that the coveted exaflop could arrive by 2020-2022.

Although this sounds good, it presents some difficulties. The development of energy-performance efficient codes for heterogeneous platforms needs to address hardware and software issues related with the cooperation among the CPU-GPU nodes, along with the challenges of the heterogeneous computing. Among those, the difference between the size of the CPU and GPU memories, the CPU-GPU memory bandwidth limitations, the workload balancing among both devices, the overlapping in data transfers between CPU and GPU, and the parallelism profile of the application considered.

The problem does not end there. The need to control energy consumption and reduce the emission of greenhouse gases is imposing performance limitations in all computer systems. The motivations are multiple: while mobile devices need to do more calculations with a fixed battery capacity, in embedded systems, such as consumer electronics, customers demand responsiveness as the main feature. Other technologies, such as *Bluetooth* or *Zigbee* [18], even require more aggressive power profiles to maximize the battery life of their devices. But,

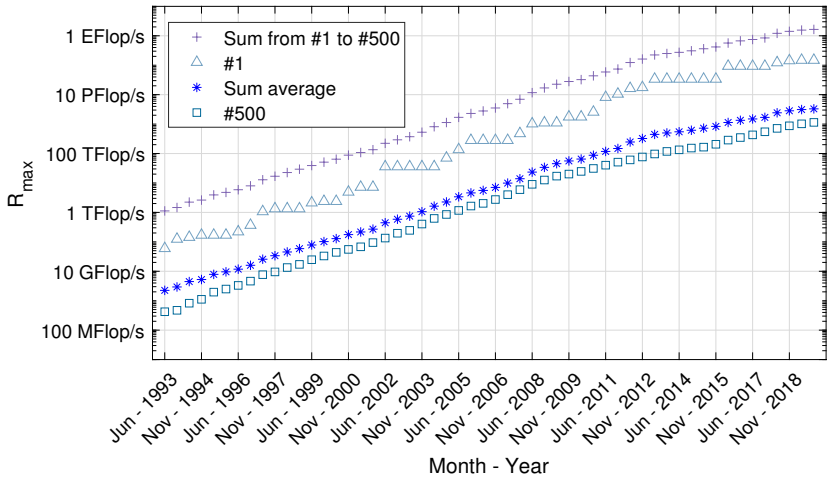


Figure 2.4: Evolution of R_{max} in supercomputers since 1993. Data source: *TOP500* list [16].

when the topic is scaled to the scope of HPC, this changes. Here, what is mainly sought is the highest possible performance. The increase in computation is not difficult, since this could be achieved by grouping a multitude of computers to create huge heterogeneous platforms. The concern, therefore, lies in the energy consumption of these machines. To get an idea, the supercomputer that occupies the top position of the *TOP500* list, called *Summit*, has a consumption of 10,096 kW, which is equivalent to the average consumption of 12,000 homes with three inhabitants. For all this, it is not surprising that the agency in charge of the *TOP500* list also contemplates another list, called *Green500* [19], with the 500 systems that provide the best performance per watt consumed.

> The information and communication technology sector faces two challenges to grow in a sustainable way: increasing energy efficiency while still offering increased performance.

This goal is not a novelty, but it becomes more compelling with the passing of the years. The question then is how to solve this problem. A reasonable response would be to provide new efficient computing models and paradigms, which would allow significant energy-saving. One of the first bottlenecks lies in software design, which tends to focus on abstraction layers, wasting energy and performance potential that could be exploited with the optimization of the entire coordinated

DVS: Dynamic Voltage Scaling

system. Although CPU and GPU devices have specific mechanisms to control energy consumption, such as DVS, most are not used due to the difficulties they present to the programmer. Thus, instead of including these mechanisms in the architecture of the device, where the greatest energy-saving can be achieved, the optimization of energy consumption is relegated to the software that will be executed on the platform, which is often left in background. Although it requires a little effort, to achieve the next level of energy efficiency the interaction of hardware and software must be optimized through an iterative co-design process.

This thesis aims to address just the problem mentioned above, but applied to the EEG classification problem through bioinspired algorithms in order to minimize both execution time and energy consumption. For this, aspects related to the optimization process, workload distribution among heterogeneous devices, and the creation of energy-time models that explain and predict the behavior of the application will be taken into account.

2.4 RELATED WORKS

Many works in the literature have reported high speedups on both CPU and GPU devices with lower energy consumption [20]. Precisely, as GPUs offer the opportunity to take advantage of massive parallelism, their energy efficiency has been previously analyzed in several papers [21–23]. Nevertheless, as it is pointed out in [24], there is a controversy about whether real applications can benefit from GPUs, since the data location and the overhead when moving data between CPU and GPU have to be taken into account to achieve an efficient code. To understand the behavior of this applications, some models relevant to performance optimization on GPUs have been proposed in [25–27]. Indeed, the availability of accurate performance models constitutes an important topic to distribute the workload optimally.

The use of heterogeneous architectures has been discussed in some papers. Paper [28] proposes approaches to cope with the difficulties that appear in the parallelization of frequent algorithms in data mining applications. It also analyzes the effect of factors such as the communication patterns and the data partition on application performance. In [29], some strategies fuelling the trend towards heterogeneous processors are considered. Among others, strategies for memory optimizations, overhead reduction between the host (CPU) and the accelerators, or mechanisms to migrate tasks to the available CPU cores.

With respect to energy efficiency of hybrid CPU-GPU platforms, recent workload balancing approaches have been proposed, demonstrating that suitable workload distributions among CPU and GPU devices allow reductions in energy consumption [30]. E.g., [31] provides analytical models to get an insight into performance gains and energy consumption of different heterogeneous platforms, concluding that greater parallelism allows opportunities for energy-saving. In [32], integrated CPU-GPU platforms and iterative algorithms based on *OpenCL* are analyzed, while paper [33] considers a set of regular and irregular applications with problem sizes high enough to justify the workload distribution among both devices.

In addition, [34] proposes two alternatives for energy efficiency: (i) determine a workload distribution so that both CPU and GPU finish at the same time, and (ii) coordinately throttle the CPU or GPU frequencies according to their utilization. In both cases, the target is to minimize execution time and reduce energy consumption. The paper also points out the need to take into account the architecture of the devices, since although GPUs are more efficient than CPUs, allocating all workload in the GPU is surely not the most efficient alternative. Other approaches, however, investigate the effect in energy consumption of different implementations of a specific application and try to derive energy-aware strategies and power models by applying a multiple linear regression to the experimental data [35].

An important research issue is the development of energy-efficient codes that take into account both execution time and energy consumption [36]. Paper [37] distinguishes between SPM approaches, which are based on the use of low-power components, and techniques that use software approaches and power-scalable components, called DPM. In turn, DPM techniques are also classified into two alternatives:

SPM: Static Power Management

DPM: Dynamic Power Management

- **Based on the dynamic adjustment of energy consumption**: by taking advantage of power-scalable components, this enables the user to control the frequency and voltage of the devices in the platform [38, 39]. These approaches are based on DVFS, a technique similar to DVS but also allowing dynamic scaling of the CPU frequency. This way, the time spent by lighter tasks waiting for heavy tasks to finish can be reduced and the required levels of performance could be still satisfied. The algorithm described in [38] considers DVFS to provide a procedure capable of determining the best CPU core with its corresponding voltage and frequency values to optimize energy consumption. In general, the paper shows relevant improvements in energy consumption. However, it is difficult to provide some fair comparisons to conclude which

DVFS: Dynamic Voltage and Frequency Scaling

strategy is the best since the performance improvements of an approach depend on the characteristics of the application. Even in some cases, the experimental results are obtained from tasks with randomly generated workloads where the computational cost is already known.

For this reason, other papers such as [40] are exclusively focused on performing energy analysis to study the impact of parallel programming models on HPC systems. Also in [41] the energy efficiency of computer systems is analyzed when their frequency is modified by DVFS techniques. In addition, the paper proposes a new DVFS policy, named PAFS, that aims to optimize energy consumption while still satisfying performance requirements of a given application. On the other hand, [39] provides an approach to handle multi-core heterogeneous platforms that includes the makespan[9] and the minimization of energy consumption. It uses the computing capacity and the TDP information provided by vendors and supposes that the tasks to be distributed among the CPU cores have neither deadlines nor precedence constraints.

PAFS: Productivity-Aware Frequency Scaling

TDP: Thermal Design Power

- **Based on energy-aware workload balancing strategies**: current microprocessors usually implement power management policies that usually change between microprocessors and are not visible to the user. In this case, it is necessary to devise a black-box approach [42] that models the main characteristics of the applied power management policy. In this line, paper [37] also comments that an effective energy-aware online scheduler requires an accurate prediction of the effects of different voltage and frequency levels in different phases of the application, whose computational costs are difficult to know in advance. Papers [43, 44] deal with the determination of power and energy consumption models either by running micro-benchmarks [43] or through the energy consumption evaluation of the platform components [44]. Finally, [45] shows an energy consumption model in codes for sparse linear systems and analyzes different CPU power-saving modes to define energy-aware strategies.

Other approaches present hybrid alternatives. Paper [46] describes a first phase performing a priority based task ordering and scheduling, followed by a second phase where integer programming is used to optimize voltage scaling. A procedure that combines DVS and the efficient DPS algorithm [47] is provided in [48] to minimize both execution time and energy consumption. In that procedure, DPS is firstly applied to the corresponding task graph to obtain low execution times. After that,

DPS: Decisive Path Scheduling

9 Total time needed to fully process a set of jobs

DVS is applied during idle times to reduce energy consumption while the computing time achieved by the scheduler algorithm is maintained. The results obtained by simulation show average energy consumption reduction of about 40% over DPS.

Paper [49] proposes two cost functions based on different approximations for energy measurement to tackle scheduling on processors with DVS. It aims to reach a trade-off between execution time and energy consumption in precedence-constrained parallel applications. The simulation results provided show that schedules generated by non energy-aware procedures consume between 16 and 51% more energy than their alternatives. In [50], it is described a two-level method to schedule large workloads of a data center. The results obtained after generating more than 100,000 workflows with the tool described in [51], show that the best schedulers achieve improvements up to 46.8% in makespan, and up to 29.0% in energy consumption with respect to a typical *round-robin* strategy.

Nevertheless, it is not always possible to handle the runtime power management, and some rules or principles should be taken into account to develop efficient procedures for a given platform. To do that, other kind of models are required to estimate the time and energy used by a program. Among others, the inherent parallelism, required synchronizations, or memory and I/O requirements. In this context, paper [52] provides a complete survey of energy models with the corresponding set of references. These models are classified according to their parameters, abstraction level, and instantaneous or average power. The paper also gives information about the accuracy in predicting energy consumption and instantaneous power, portability to different architectures, and complexity of the model.

Other approaches go further. As energy consumption and execution time are competing objectives, a multi-objective (more specifically a bi-objective) approach is required to tackle the development of an energy-aware scheduling problem. To this end, [50] proposes as future work the use of multi-objective optimization to find a trade-off between execution time and energy consumption. Nevertheless, a scheduling algorithm based on that scheme would require a large computing time to choose the best alternative among those evaluated. Depending on the application, multi-objective optimization could be a solution to the problem. But for others, such as real-time applications, this approach would not be possible since they demands speed and rapid response. For these situations, another type of energy-aware scheduling is mandatory, as [53] indicates. However, a previous study would determine the most appropriate option for each case.

3

BIOINSPIRED MULTI-OBJECTIVE OPTIMIZATION

CONTENTS

Hardware is not everything, since finally software applications are responsible for providing usefulness to platforms. The evolution of hardware is motivated by the needs of real-world applications, and as discussed in Section 1.1, huge amounts of data are created worldwide that must be processed. Paradigms such as *Cloud Computing*, AI, or *Deep Learning* are responsible for dealing with these issues, and techniques such as optimization, FS, statistical classification, clustering, or EAs help filter such data. In fact, today approximately 73% of *Machine Learning* problems correspond to classification and clustering tasks [54].

Machine Learning is an area of research that allows the theoretical and practical advancement of solutions in multiple disciplines. Today, various AI algorithms based on statistical or mathematical methods have been successfully applied to solve optimization and classification problems. However, nature has been one of the main sources of inspiration for the development of AI algorithms. Bioinspired algorithms [55] are based on the analogy between the problems to be solved and the natural or social systems. Simulating the behavior of those systems, the

objective is to design non-deterministic *heuristic* methods of search and learning. E.g., the theory of evolution has supported the design of EAs, such as GAs or ESs. These algorithms are mainly defined by biological processes and strategies to decide who is part of the next generation, similar to what happens in real life. On the other hand, imitating the behavior of animals such as birds, bats, bees, ants, and fireflies, has served as a guide to design efficient algorithms. But in addition, inspiration not only comes from biological processes and social behaviors, since there are also algorithms based on physical phenomena such as SA, used in metallurgy with steel and ceramics.

ES: Evolution Strategy

SA: Simulated Annealing

While Chapter 2 presented the technological fundamentals in computer architecture, this chapter discusses the problems of optimization and classification, and how *metaheuristics*, especially bioinspired algorithms based on EAs or neural networks, can address these problems.

3.1 THE OPTIMIZATION PROBLEM

Optimization is a process present everywhere and almost daily. Using the shortest path or arriving as quickly as possible to go somewhere is a way to optimize. In general, people's lives are conditioned by two optimization problems: maximization and minimization. Depending on the context, the objective will be one or the other, and the same happens in the field of computing. After all, computing is often used to solve everyday life problems. Over the years, many algorithms have been developed to address optimization problems. Currently, there are two types of methods to address it: exact and approximate methods.

3.1.1 *Exact and Approximate Methods*

Exact methods achieve optimal solutions and guarantee their optimality whereas approximate methods, also known as *heuristics*, obtain plausible solutions in reasonable times but without guaranteeing their optimality. Exact methods can be divided into four subtypes: based on dynamic programming, B&B, constrained programming, and those based on A^* familiy techniques [56]. Briefly they can be described as:

B&B: Branch-and-Bound

- **Dynamic programming**: it aims to recursively divide the problem into simpler subproblems. This process is based on Bellman's optimality principle [57], that says:

> Any subpolicy of an optimum policy from any given state must itself be an optimum policy from that state to the terminal state.

This method of optimization by stages is the result of a sequence of partial decisions. The process avoids fully enumerating the search space by pruning sequences of decisions that do not lead to optimal solutions.

- **B&B and A^***: implicitly list all possible solutions to the problem. The search space is explored through the dynamic construction of a tree whose root node represents the problem to be solved, the leaf nodes represent potential solutions, and the internal nodes possible subproblems. A branch of the tree is pruned when the boundary functions decide that the branch cannot contain any optimal solution to the problem.

- **Constrained programming**: language built around concepts such as search trees and logical implications. The optimization problem is done through the definition of a set of variables that, in turn, are linked to a set of restrictions. Variables take their values from a finite value domain while restrictions can be represented by mathematical symbols or expressions.

With regard to approximate methods, *heuristics* are able to find good solutions in large instances and allow to obtain good performance with an acceptable cost in many problems. They can be organized into two subtypes: *specific heuristics* and *metaheuristics*. *Specific heuristics* are those tailored to solve a specific problem. On the contrary, *metaheuristics* are general purpose algorithms that can be used to solve almost any optimization problem. In a way, they can be seen as high-level templates used as a guide to design *specific heuristics*. In addition, they have demonstrated in different applications their effectiveness in solving complex problems, which has given them popularity in the last 20 years [58]. When designing *metaheuristics*, two completely opposite criteria must be taken into account: the exploration of the search space and the exploitation of the best solutions found. This is known as *diversification* and *intensification*, respectively. In *diversification*, unexplored regions must be visited to address all possible regions of the search space and ensure diversity in the search for solutions. On the contrary, in *intensification*, promising regions are explored in depth with the hope of finding better solutions. These regions are determined based on the quality of the solutions found.

3.1.2 *Single and Multi-objective Optimization*

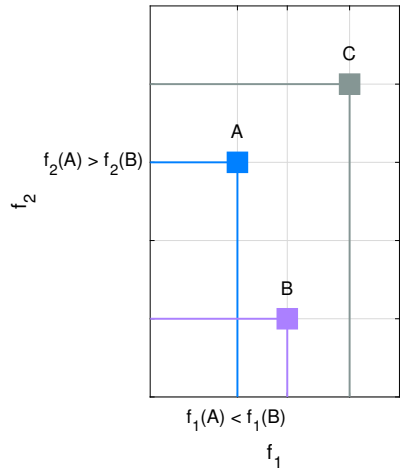
Single-objective optimization problems are focused on improving the quality of a single solution. The *metaheuristics* used for these problems follow a certain path through the search space. This path is generated by iterative processes that move the current solution to another search space, and apply generation and replacement procedures on the current solution. In the generation phase, a set of solutions is generated from the current solution. This set is usually obtained by local transformations of the solution. In the replacement phase, one of them is selected from the set of candidate solutions as the current new solution. The process is repeated until the stop criterion is met.

The generation and replacement phases may not use memory during the process, so they only use the current solution. If memory is used, information related to searches can be stored and then used to generate new lists of candidate solutions or to select new solutions. An example of this type of *metaheuristics* is found in the SA algorithm, which is inspired by the annealing process of steel and ceramics. The annealing technique consists of heating and then slowly cooling a material to vary its physical properties. The heat causes the atoms to increase their energy, allowing them to move from their initial positions (a local minimum of energy). On the other hand, slow cooling gives them a greater chance of recrystallizing in configurations with less energy than the initial one (global minimum).

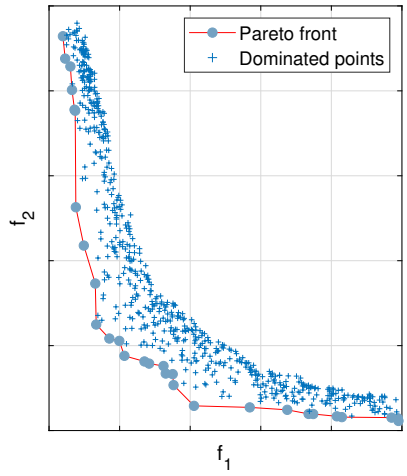
On the other hand, unlike single-objective optimization problems, multi-objective optimization is characterized by a set of objective functions, generally of a conflicting nature, which must also be optimized. This problem can be stated as follows:

$$\begin{aligned} &\underset{f(x)}{\text{Minimize}}\ (f_1(x), f_2(x), \ldots, f_M(x)) \\ &\text{s.t. } x \in X \to \mathbb{R}^k \wedge k \geq 2 \end{aligned} \tag{3.1}$$

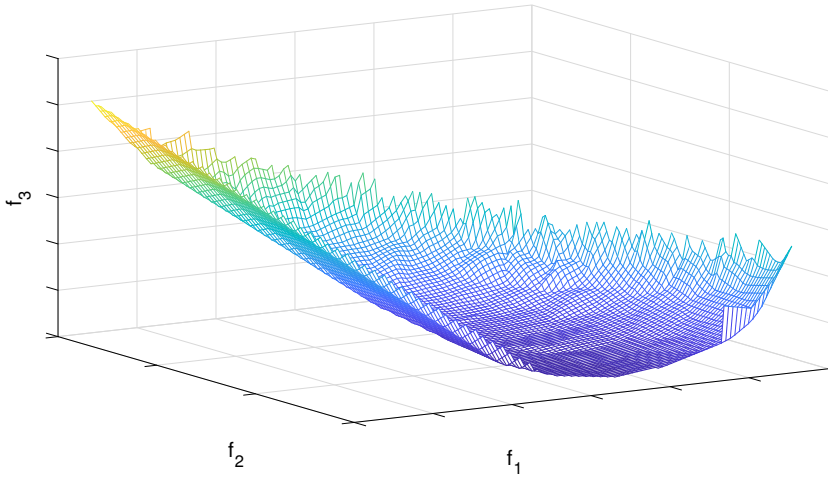
where M is the number of objective functions and X is the set of feasible solutions. X represents the decision space of the multi-objective problem, while the space to which the objective vector f belongs is called the objective space. Vector f can be defined as the cost functions of the decision space in the objective space that evaluate the quality of each solution $(x_1, \ldots, x_n)$. Multi-objective problems do not usually have an optimal solution, but their solutions consist of all those that are feasible. This means that the components of the objective function



(a) *Non-dominance between points A and B*



(b) *Pareto front in 2D space*



(c) *Pareto front in 3D space*

Figure 3.1: Multi-objective problem representation where f_1, f_2, and f_3 are the objective functions.

$f(x)$ cannot be strictly improved by any other feasible solution. E.g., Figure 3.1a shows a non-dominance situation between points A and B, while point C is dominated by both A and B. These solutions are called *Pareto-optimal* solutions, and can be formally described as:

Definition 3.1 ***Pareto optimality***. A solution $x^* \in X$ is *Pareto-optimal* if $\nexists x \in X$ such that $f_i(x) \leq f_i(x^*) \wedge f_j(x) < f_j(x^*)$; $\forall i, j \in \{1, \ldots, M\}$. This is denoted as $f(x) \nprec f(x^*)$.

The definition of *Pareto optimality* comes directly from the concept of dominance. With that in mind, the main purpose of multi-objective optimization is to find the *Pareto front*, which is composed of the set of solutions that are not dominated by anyone:

$$\begin{gathered} PF = \{F(x);\ x \in X\} \\ \text{s.t. } \nexists x^* \in X \text{ such that } f(x) \prec f(x^*) \end{gathered} \tag{3.2}$$

Figures 3.1b and 3.1c show *Pareto fronts* in 2D and 3D spaces, respectively. Depending on the space considered (objective or decision), the number of *Pareto-optimal* solutions may differ. Whatever the case, the ideal is to obtain a solution that optimizes all objectives, but this does not usually occur in real applications. However, some decision-making methods set thresholds for each objective function. In this way, the method can specify levels of aspiration that indicate the degree of acceptance in the objective space. A *Pareto-optimal* solution that satisfy all levels of aspiration is called a *satisfactory solution* [59].

3.2 STATISTICAL CLASSIFICATION

A classification system tries to classify into different classes or categories a series of samples that represent part of the information of a problem. In the field of *Machine Learning*, the objective of these systems is to predict which class the new unlabeled samples belong to. There are two types of classification depending on whether the classes are known in advance or not:

- **Supervised** [60]: the classes to which the instances of the training dataset belong are known in advance. From these instances, the system learns and establishes certain parameters that will be used to classify new instances. Now, depending on the number of classes, the supervised classification can be binary or multi-class. In the first one, only two different classes can be assigned. An example is given in cancer detection systems. One of the classes will be used for cases where the diagnosis is positive and the other for the opposite case. In the multi-class classification, multiple classes can be assigned to observations. A typical example is given in handwritten text recognition applications. In the case of numbers, 10 classes are needed to represent each digit (0 to 9).

- **Unsupervised** [61]: the classes to which the instances of the training dataset belong are unknown. In this case, the system apart from classifying has to establish the classes through clustering algorithms and use some metric to evaluate the quality of the classification. Due to the great diversity of existing clustering algorithms, these are classified according to different aspects such as the way they process the data, how the obtained clusters are organized, the membership of data to the clusters, or the mechanism used to group.

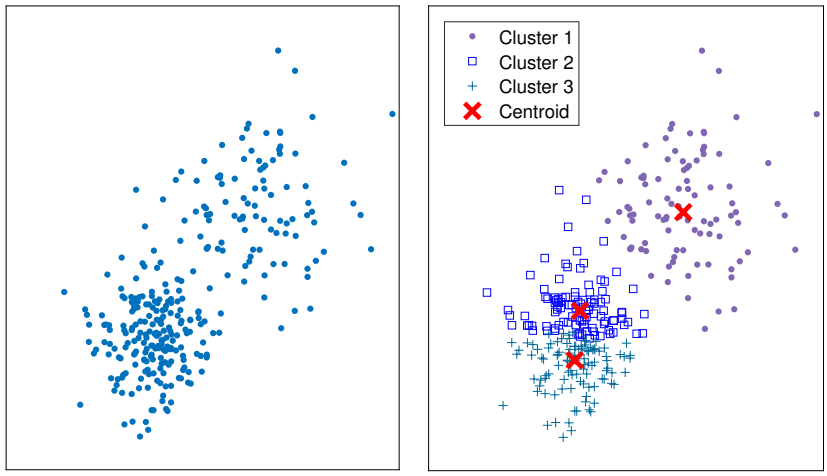
Supervised classification facilitates the task. However, it is not always applicable and therefore leaves the user with the remaining possibility: the unsupervised classification. This is very useful when the instances do not have class labels, when the cost of labeling them by an expert is high, or when instances may vary over time. However, this implies extra computing time to process the data before classification.

3.2.1 *K-means Clustering*

K-means clustering, also referred as *K-means* algorithm, is an *NP-hard* problem that aims to partition a set of points into clusters, in which each observation belongs to the cluster with the nearest mean. Its computational complexity when finding the global optimum in multi-dimensional spaces is $\mathcal{O}(n^{d \cdot K+1} \cdot \log(n))$, where n is the number of d-dimensional points and K is the number of clusters. Therefore, in practice, a stop criterion is applied when the method meets a specific condition, e.g., reaching a certain number of iterations or stopping when minimal changes in the position of the centroids are detected.

The iterative variant is known as Lloyd's algorithm [62], and is focused on converging towards local optimum. This allows to reduce the algorithmic complexity to $\mathcal{O}(i \cdot n \cdot d \cdot K)$, where i is the number of iterations needed until convergence. Even so, this version is not the fastest, because according to the purpose there are other implementations of *K-means* in the literature [63]. An example of how this algorithm works can be seen in Figure 3.2. The main steps are described below:

1. **Initialize centroids**: set the iteration counter to $i = 1$ and generate the K initial centroids, k_j^i; $\forall j = 1, \ldots, K$, by randomly selecting them from the n points, p_t; $\forall t = 1, \ldots, n$. This random selection is known as the Forgy's method [64].



(a) *Points before clustering* (b) *Clustering result after applying K-means*

Figure 3.2: Example of *K-means* algorithm in 2D space with $K = 3$.

2. **Clustering**: assign each point p_t to the cluster corresponding to its nearest centroid. Each cluster C_j^i is built as follows:

$$C_j^i = \left\{ p_t : \left\| p_t - k_j^i \right\|^2 \leq \left\| p_t - k_l^i \right\|^2 ; \forall j, l = 1, \ldots, K \right\} \tag{3.3}$$

3. **Update centroids**: calculate the new centroids, k_j^{i+1}, using all points belonging to cluster C_j^i:

$$k_j^{i+1} = \frac{1}{\left| C_j^i \right|} \cdot \sum_{p_t \in C_j^i} p_t \tag{3.4}$$

4. **Check the stop criterion**: if the condition is not met, increase the iteration counter to $i = i + 1$ and repeat Steps 2 and 3. Among the existing stop criteria, some common ones are reaching a certain error threshold or number of iterations.

Since Lloyd's variant of *K-means* is a *heuristic* algorithm, there is no guarantee that it converges to the global optimum. In addition, the result of the clustering depends not only on the points but also on the initial centroids, so it is common to run the algorithm several times with different starting conditions. However, in the worst case, *K-means* can be very slow to converge. It has been demonstrated that there are certain point sets that, even in 2D spaces, make the algorithm converge in exponential time, that is, $2^{\Omega(n)}$ [65].

3.2.2 *EEG Classification*

Understanding the brain of living beings is the current challenge of society, especially when it comes to the human brain. One of its applications is related to EEG classification in the context of BCI tasks [66] due to its great socio-economic interest. Its study is becoming possible thanks to the advancement of computer technology, and a global career has begun to find out how it works. However, the brain is tremendously complex since it is made up of millions of neurons, where each one has about 1,000 connections. Although today it is not possible to sequence it in a similar way to DNA, recent advances in its study allow other types of applications. E.g., analyzing brain activity allows detecting diseases, as well as relating the EEG of a thought to a limb movement using classification algorithms and *Machine Learning* techniques. The latter is known as MI [67].

BCI: Brain-Computer Interface

DNA: Deoxyribonucleic Acid

MI: Motor Imagery

EEG classification, applied to MI-based BCI tasks, is one of the most interesting research areas due to its potential application in different fields, such as games [68] or in health care, where some patients have suffered amputations or paralysis in a limb [69]. But its applications go further: in the same way that there are already visual, tactile, and voice methods for handling devices, more and more companies and users are demanding to control the systems with the brain. However, BCI tasks usually have to deal with EEG signals defined by a large number of features, and thus require FS techniques to remove redundant, noisy-dominated, and irrelevant inputs from signals. Therefore, high-dimensional EEG classification often has to be addressed with a number of training signals much lower than the number of features, which is known as the *curse of dimensionality* problem [70, 71].

This phenomenon may be caused by the following reasons, among others: (i) the presence of noise or outliers since EEG signals have a low signal-to-noise ratio; (ii) the need to represent time information in the features because brain signal patterns are related to changes in time, and (iii) the non-stationary character of EEG signals, which may change quickly over time or between experiments even for the same subject. The problem is that the MI-based BCI paradigm uses series of attenuations and amplifications of short duration conditioned by limb movement imagination [72]. ERD is the short-lasting attenuation within the alpha[1] and beta[2] bands and is found before and during the visual stimulation. In contrast, ERS represents an amplitude increase of rhythmic activity. ERD and ERS analysis is complex because the

ERD: Event-Related Synchronization

ERS: Event-Related Desynchronization

1 Brainwaves in the frequency range of 8-12 Hz that arise from the electrical activity
2 Brainwaves similar to alpha waves but in the frequency range of 12.5-30 Hz

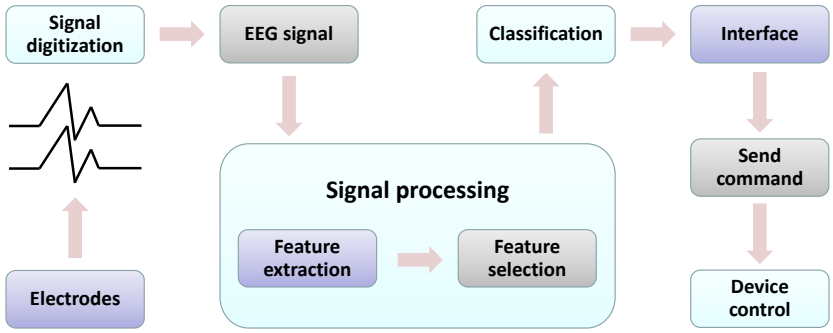


Figure 3.3: EEG classification process in MI-based BCI tasks.

signals are weak and noisy, occur at different locations of the cortex, at different instants within a trial, and in different frequency bands. With this scenario, a good FS method is mandatory to reduce the dimensionality of the input signals. In addition to reducing execution time, FS can provide an improvement in the accuracy of the classifiers when reducing the dimensionality.

Generally, applications related to BCI need a portable, moderate-cost, and real-time system. For these reasons, the methods that obtain EEGs are the most commonly used. These systems are connected between the brain and the environment without using peripheral nerves or muscles, directly transforming brain events into actions. Normally, the EEGs of an individual are obtained in a non-invasive way, placing electrodes on the scalp, outside the skull. Other methods, such as ECoG, use electrodes placed directly on the exposed surface of the brain to record the electrical activity. Whatever the method, once the EEG signal is obtained through the electrodes it is digitized for subsequent treatment, which usually includes FS and classification techniques. Figure 3.3 shows the EEG classification process in BCI tasks.

ECoG: Electrocorticography

3.3 EVOLUTIONARY ALGORITHMS

EAs are *heuristic* search and optimization techniques based on natural evolution and genetics. What mostly distinguishes them from classic optimization techniques is that EAs process in each iteration more than one potential solution to the problem. This feature gives them a great advantage in problems where it is difficult to use classical methods. E.g., solving multi-objective optimization problems, where they have also proven to be very effective in approaching a solution to a large number of problems.

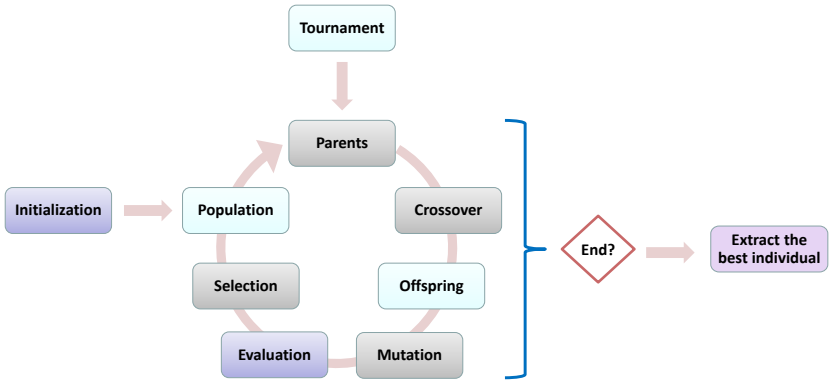


Figure 3.4: Main steps of a GA with parent selection by tournament.

3.3.1 *Genetic Algorithms*

GAs [73] are algorithms that mimic the natural selection process. The objective of these algorithms is to find a solution for optimization and search problems using the individuals of a population as candidates for the solution. Each of them is a chromosome composed by genes that encode the parameters of the problem to be solved. They evolve and in each generation the candidates that best adapt survive, that is, those that are closest to the best solution according to the objective functions. Each individual is evaluated, obtaining a score related to the quality of that solution. The better the score, the greater the probability of reproduction for that individual, and when two individuals recombine, the offspring generated share the genes of both parents. In this way, the idea is that after each generation better solutions to the given problem are obtained. The basic steps of a GA can be seen in Figure 3.4.

3.3.1.1 *Representation and Evaluation Function*

The structure of a GA is usually similar for most of the problems in which it is used. However, when solving a specific problem, other aspects must also be taken into account. The first thing to decide is the number of chromosome genes and their representation. The number of genes is usually defined by the decision space of the problem to be solved, while to represent the genes both real and integer coding are currently accepted. An example of integer representation can be found in TSP: as many genes as possible destinations and an integer that identifies each destination. A subtype of integer representation is the binary representation, or binary-coded, widely used in problems

TSP: Travelling Salesman Problem

related to feature and instance selection. In FS, each gene (0 or 1) represents the decision to select or discard one of the dimensions (feature) of the dataset when classifying. In instance selection, the decision corresponds to selecting a sample (instance) from the dataset.

Another aspect to consider is the evaluation function, or cost function, which is used to calculate the quality of each individual in the population. The value returned by this function is called *fitness* and represents the cost of that solution to the problem. If the problem is maximization, the greater the *fitness* the better the individual, and if it is minimization, the opposite. Taking the case of the TSP problem again, the evaluation function should calculate the sum of the distances between the pairs of destinations visited. As the goal is to visit all destinations in the shortest possible way, this is a minimization problem.

However, as discussed in Section 3.1.2, many problems are related to multi-objective optimization. In such cases, each objective has its own cost function that must be maximized or minimized according to the problem characteristics, which adds complexity and increased execution time to the algorithm. Although it seems unimportant, the greater the complexity, the greater the number of generations needed to find a satisfactory solution to the problem, so the stop criterion of the GA must be chosen carefully to ensure that the algorithm converges towards an acceptable solution. A typical stop criterion is to fix the number of generations, but other conditions such as reaching a threshold for solution quality are also common. On the other hand, genetic operators are responsible for creating the new population of individuals, which will be different from the current one if the GA has not yet converged. These operators are designed to increase the average *fitness* of the population, and although there are others such as *regrouping* or *migration*, the three main ones of a GA are *crossover*, *mutation*, and *selection*:

3.3.1.2 *Crossover Operator*

Also called *recombination*, and is responsible for conducting the search in the solutions space. Although there are several *crossover* operators, they all share the same purpose: generate offspring from the individuals of the current population. For this, some of the individuals are selected, acting as parents. Although children have different chromosomes, their genes are composed of the genetic information of both parents. The *crossover* operators may differ depending on the representation of the chromosome. Some common for binary-coded chromosomes are:

- **Single-point** [74]: a random point is chosen and the chromosomes of the parents are divided by that point. The first child will be composed of the left part of the first parent followed by the right part of the second. The second child will be composed of the left part of the second parent followed by the right part of the first. An illustration can be seen in Figure 3.5a.

- **Two-point**: similar to the previous one but using two cut-off points, dividing the chromosome into three parts. The first child will be composed of the left and right part of the first parent followed by the central part of the second. The second child will consist of the left and right part of the second parent followed by the central part of the first. This is depicted in Figure 3.5b.

- **Uniform**: the chromosome genes of both parents are compared individually. Those that differ are marked to exchange with a fixed probability (usually 0.5). Each child will have the same genes as one of their parents except those that have been marked, which will be obtained from the other parent. This behavior can be seen in Figure 3.5c, where the genes highlighted in black are those that will be exchanged.

Regarding the *crossovers* used for real-coded chromosomes, one of them is the SBX *crossover* [75], which is applied gene to gene. A spread factor, β, is defined as the ratio of the absolute difference in offspring values to that of the parents:

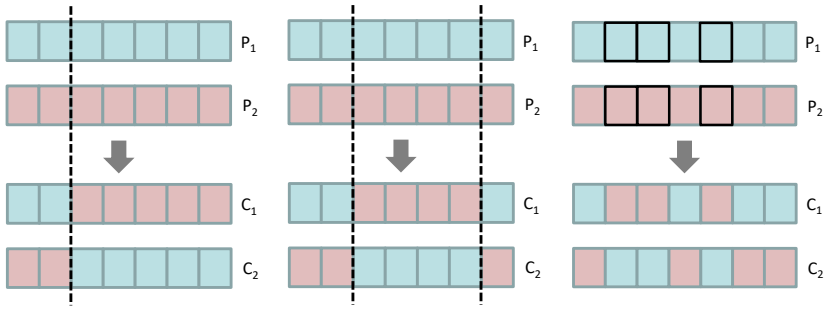
SBX: Simulated Binary Crossover

$$\beta = \left| \frac{C_1 - C_2}{P_1 - P_2} \right| \tag{3.5}$$

where C_1 and C_2 are the children, and P_1 and P_2 the parents. The operator also involves a distribution index, η, which is kept fixed to a non-negative value. If a large value of η is chosen, the resulting offspring solutions are close to the parent solutions. For a small value of η, solutions away from parents are likely to be created. Thus, this parameter has a direct effect in controlling the spread of offspring solutions. Taking that into account, the probability distribution of β can be calculated as:

$$P(\beta) = \begin{cases} 0.5 \cdot (\eta + 1) \cdot \beta^{\eta}, & \text{if } \beta \leq 1 \\ 0.5 \cdot (\eta + 1) \cdot \frac{1}{\beta^{\eta+2}}, & \text{if } \beta > 1 \end{cases} \tag{3.6}$$

Thus, children C_1 and C_2 are defined by the following equations:



(a) *Single-point crossover* (b) *Two-point crossover* (c) *Uniform crossover. Genes highlighted in black are the ones that will be exchanged*

Figure 3.5: Traditional binary *crossover* operators in GAs to generate two (C)hildren from the genetic information of the (P)arents.

$$C_1 = 0.5 \cdot [(1+\beta) \cdot P_1 + (1-\beta) \cdot P_2] \tag{3.7}$$
$$C_2 = 0.5 \cdot [(1-\beta) \cdot P_1 + (1+\beta) \cdot P_2] \tag{3.8}$$

There are several ways to select parents. E.g., directly choose individuals that belong to the elite, or the *roulette-wheel* method. This method consists in giving all individuals a probability of selection proportional to their *fitness*, so that the sum of probabilities p_i of all N individuals is 1. In more formal terms:

$$\sum_{i=1}^{N} p_i = 1 \tag{3.9}$$

This means that the best individuals have a higher value than the worst and therefore are more likely to procreate. However, low quality individuals may also be used to ensure genetic diversity. They can be used within the genetic pool of the parents and in the subsequent generation of children. Another method to select parents is by tournament. In this case, some individuals are chosen randomly and the best of them is selected. The situation in which two individuals are selected is called a binary tournament and is the most common way to select parents. There is a probabilistic variant of this method, where a random number is generated in the interval $[0,1]$ that according to its relation to a prefixed threshold, the best or worst individual is chosen. Given the number of existing methods, choosing the most appropriate one will depend on the characteristics of the problem to be solved.

3.3.1.3 *Mutation and Selection Operators*

The *mutation* is the modification of some of the genes of a chromosome. For each gene the *mutation* is performed with a fixed probability, although it is usually too low to not alter the natural course of evolution in excess. However, its use provides diversity in the search space and serves as a measure to avoid falling into local optimum quickly. For binary-coded chromosomes (0 or 1), the *mutation* consists in assigning the opposite value, which is known as *bit-flip mutation*. In the case of real-coded chromosomes, one option is to alter the value of the gene with a constant. Another option is to use some probability function similar to that of the SBX *crossover* that allows a random variation within previously established ranges.

The *selection* operator is responsible for deciding which individuals will be part of the next generation. This operator is based on providing solutions with a score, which for single-objective problems can be directly the value of the cost function (*fitness*). However, in multi-objective problems, it is very unlikely to obtain a solution that totally dominates the rest. The probability decreases further when the number of individuals in the population increases since each of them represents a solution to the problem. In addition, as discussed in Section 3.1.2, the more objective functions the greater the likelihood of conflict between them. Therefore, since a total order cannot be defined on the set of solutions, other *selection* operators are needed. NSGA-II algorithm, which is described in Section 3.3.2, incorporates a *selection* operator based on rankings and non-dominance levels to address the issue.

3.3.2 *Non-dominated Sorting Genetic Algorithm*

NSGA algorithm was proposed in 1994 [76] as a kind of EA that, in addition to performing the usual steps of a GA, also includes the so-called NDS to rank the individuals into different levels of non-dominance. These individuals are classified according to their rank, their *fitness*, and the result obtained by a distribution parameter. Once this group of individuals is classified, the process is repeated but with the remaining individuals, and so on until the entire population is classified. Since individuals with better rank have higher quality, they will be more likely to create offspring, allowing a deeper search in non-dominated regions. Although the algorithm achieved good results, it was strongly criticized mainly for three reasons: (i) its non-elitist behavior; (ii) the need to specify the distribution parameter, and (iii)

NDS: Non-Dominated Sorting

Algorithm 3.1: Pseudocode of NSGA-II algorithm.

```
1  Function NSGA-II(N, M, G)
       Input  : Number of individuals in the population, N
       Input  : Number of objectives, M
       Input  : Number of genes, G
       Output : The final population of individuals, I
2      I ← initPopulation(N, G)
3      I ← evaluation(I, N, G)
4      repeat
5          for i ← 1 to N do
6              Parents ← parentSelection(I, N)
7              Child ← crossover(Parents, G)
8              Child ← mutation(Child, G)
9              O ← O ∪ Child
10         end
11         O ← evaluation(O, N, G)
12         R ← I ∪ O
13         R ← nonDominatedSorting(R, 2 · N, M)
           // Replacement process
14         for i ← 1 to N do
15             I_i ← R_i
16         end
17     until the stop criterion is not reached;
18     return I
19 End
```

the high computational complexity, $\mathcal{O}(M \cdot N^3)$, where M is the number of objectives and N the size of the population.

Later, the same authors proposed NSGA-II [4], an improved version of its predecessor that solves the previous limitations (Algorithm 3.1). This is achieved through a NDS mechanism of complexity $\mathcal{O}(M \cdot N^2)$, a *selection* operator to combine parents and offspring, and the choice of the best N individuals according to their *fitness* quality and distribution in the *Pareto front*. Thanks to the good results achieved in many multi-objective applications, it has become a standard algorithm for optimization problems. However, there is a third version, NSGA-III [77], which is an extension of NSGA-II that works well with problems where $M \geq 3$. The main difference between both algorithms is that NSGA-III uses a set of reference points to maintain the diversity of the *Pareto's* points during the search.

3.3.3 *Multi-population Approaches*

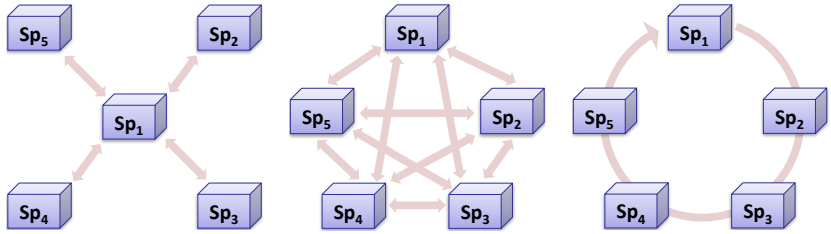
The GAs mentioned in Sections 3.3.1 and 3.3.2 evolve a single population of individuals, but there are also multi-population models that allow the evolution of several of them simultaneously. The basic idea is to divide the total population into several subpopulations, or islands, where each one evolves independently. This scheme allows some GAs the possibility of exchanging information between subpopulations every certain number of generations. The process is known as *migration*, and it is considered as an exclusive genetic operator of multi-population models. The introduction of *migration* gives the algorithm the ability to exploit the differences between subpopulations, thus allowing genetic diversity. The determination of the *migration* rate is a sensitive matter since an inappropriate value can cause premature convergence. Different multi-population models can be devised depending on the type of communication and topology between subpopulations [78]. Some of them are depicted in Figure 3.6 and described as follows:

- **Star**: one subpopulation is selected as master and the rest are considered slaves. The master subpopulation can be chosen either randomly or by identifying the subpopulation with the best value of its objective functions. In any case, all slave subpopulations periodically send their best individuals to the master subpopulation and receive as many others from it.

- **Fully-connected**: there is no hierarchy between subpopulations and each of them has direct communication with the rest of subpopulations. This model allows greater genetic diversity by sharing more information, but entails much higher communication costs.

- **Ring**: each subpopulation is connected to two more. Normally the communication flow is carried out in a single direction, so that a subpopulation can only send individuals to one of its neighboring subpopulations and receive from the other.

In general, EAs are easily parallelizable [79]. The reason is that each individual in the population can be evaluated independently. The multi-population paradigm allows another level of parallelism since following the same philosophy, subpopulations evolve independently of each other. However, at more levels, more complexity. This can cause other problems that must also be addressed. Among them, the cost of communications, since according to the topology used the number of communications and the size of the messages varies considerably.



(a) *Star* (b) *Fully-connected* (c) *Ring*

Figure 3.6: Connection topologies in multi-population EAs.

Another problem is workload balancing. Depending on the parallel approach, workload imbalances and consequent loss of performance can occur. Assuming that the time needed to evaluate an individual is the same in all subpopulations, an example of workload imbalance appears when the size of the subpopulations differs. Another imbalance situation is present in heterogeneous computing because the devices responsible for evaluating individuals probably have different computational capabilities. Also the granularity of parallelism must be taken into account, since depending on its type the cost of communications and workload imbalance can be less or greater, so finding a trade-off between the two is important. In fact, as in this thesis approximately half of the algorithms developed are based on multi-population GAs, these topics and some more will be analyzed in depth.

There are two main parallel models for a multi-population GA: (i) take advantage of a master-worker approach to evaluate in parallel the individuals of the subpopulations, or (ii) parallelize the whole evolution of each subpopulation [80]. While the first alternative has the same behavior than the sequential algorithm, the second one does not. The parallel evaluation of the individuals could be implemented in GPU while the CPU executes the rest of the EA steps, so the GPU would be used as a coprocessor. A drawback of this scheme is the number of data transfers between both devices, since a copy of the individuals in both directions is needed for each generation.

PCIe: Peripheral Component Interconnect Express

The GPU bus (usually a PCIe bus), has worse bandwidth and latency than those provided by the CPU memory bus. In addition, the copy of the dataset from main memory to the GPU memory could also require a significant amount of time. Although this transfer only has to be done once, the size of the dataset to be processed is usually large. On the other hand, since CPUs have incorporated multiple cores for a long time, this method wastes part of their computing capacity because if only one core is used for the master, the rest of them remain idle.

In the second model, the whole algorithm can be parallelized on CPU or GPU. In the CPU case, all evolutionary steps, including the *fitness* evaluation, can be done in the cores. However, in the GPU case, each subpopulation could be assigned to a thread block and each individual to a thread. In this way, the application of *crossover*, *mutation*, and *selection* operators to the individuals requires the use of barriers to synchronize the threads. Although this approach could reduce the data transfers between CPU and GPU, the synchronization requirements have to be taken into account. Moreover, the GPU memory hierarchy should be carefully managed because this memory is not cached and its accesses mean many additional cycles. For this reason, global memory should be used for the communication among subpopulations according to the devised *migration* policy, while shared memory should store the data structures of the individuals. If there is no space to store all the data, *tiling*[3] strategies should be defined. Examples of parallel EAs can be found in [81]. It includes many details regarding the *migration* and *selection* criterion, a topology to exchange individuals between subpopulations, and the synchronization between threads required by a synchronous multi-population model.

3.4 ARTIFICIAL NEURAL NETWORKS

ANNs, or simply neural networks, are mathematical models defined primarily by computational units that emulate neurons, the communication links that define synaptic connections between network neurons, and the type of messages that the networks handle. Although there are different types of ANNs, such as CNNs or DBNs, they all share the same basic principle: try to mimic the functioning of the brain. It is known that the human brain is one of the most complex natural learning mechanisms that exist, and hence the desire to replicate its functioning. These bioinspired models can be used to solve classification problems, image reconstruction, meteorological prediction based on the historical records of a region, autonomous movement of robots and vehicles, or even intelligent voice assistants. In [82], the inventor of one of the first neurocomputers defines an ANN as:

ANN: Artificial Neural Network

DBN: Deep Belief Network

> A computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.

3 GPU technique to increase performance through caching and data reuse

The simple processing elements are the neurons, which basically transmit electrical impulses to other neurons through the network. They provide the ability to remember, think, and learn from previous experiences. However, the ability to learn is the one that interests in the field of computing. The question is how to implement an ANN in computer systems. The first thing is to create the artificial neuron, also called *perceptron*. The *perceptron* takes several inputs, $(x_1, x_2, \ldots, x_n)$, and produces only one output, y. To calculate the output value, each entry is assigned a real number, called weight, $(\omega_1, \omega_2, \ldots, \omega_n)$, which expresses the degree of importance that an entry has in the calculation of the output. Weights are crucial because the training process of an ANN basically consists in adjusting the values of the weights to obtain the desired results. The training process has two steps: (i) perform ϑ as the weighted sum of the input variables taking into account their weights, and then (ii) calculate the output of the neuron, y, by applying an activation function φ on the result of ϑ. This is defined as:

$$\vartheta = \sum_{i=1}^{n} x_i \cdot \omega_i \tag{3.10}$$

$$y = \varphi(\vartheta) \tag{3.11}$$

The activation function is what determines the output value, so choosing carefully an appropriate activation function is essential in *Deep Learning*. There are many activation functions, such as the *threshold* function, whose output is binary because it returns 0 or 1 depending on whether the value obtained in ϑ is greater than or less than a threshold. Other more complex functions like *tanH* or ReLU allow outputs with real values. Currently, most *Deep Learning* models are composed of several layers of neurons, creating a multi-layer *perceptron* where the output of all neurons in layer j can be connected to the input of all neurons in layer $j+1$. Depending on the number of layers, concepts such as input, hidden, and output layers appear. This allows to improve the model although the calculation of the weights is more complicated. However, if the number of layers is increased excessively, the result of the model could get worse. Figure 3.7 shows the training process of a neural network.

ReLU: Rectifier Linear Unit

Neural networks are able to learn and relate the data they process to later know how to identify the similarity between a new sample and the data learned. For this, the networks must be adjusted iteratively, through epochs. At the end of each epoch, the model compares its prediction of y with its real value through a loss function. This function estimates the margin of error in the network, so the objective is to

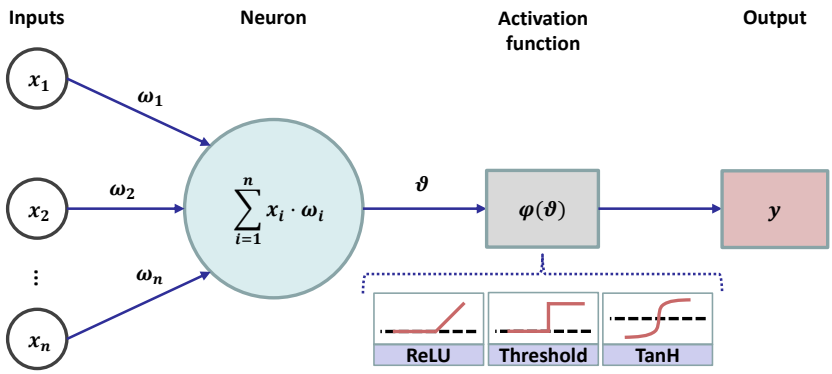


Figure 3.7: Training process of a neural network. φ can use the ReLU, *threshold*, or *tanH* activation functions to calculate the output, y.

minimize it. The lower its value, the more accurate the model is because there is less difference between prediction and error. The adjustment of the weights minimizes the loss function, which propagates the error backwards in the network. This is known as *backpropagation* [83], and thanks to this technique, multi-layer networks can be trained in a supervised manner. When calculating the error obtained and propagating it to the previous layers, small adjustments are made to make the network learn and classify the inputs correctly. Although this sounds good, problems such as *overfitting* appear during training. This occurs when the neural network has a close relationship with training data and subsequently is not able to correctly classify new samples. To solve this, techniques such as *dropout*[4] or *cross-validation*[5] are used.

One type of multi-layer network is CNN [84], which takes its inspiration from the visual cortex of animals and therefore is focused on image processing. Its architecture consists of several layers whose objective is the feature extraction and classification. The first step is to use a convolution layer to divide the input image into receptive fields and thus extract features from the image, such as edges or vertices. The next step is called *pooling*, a technique that can reduce the dimensionality of the features extracted but maintaining the most important information. In general, the convolution and *pooling* processes are repeated until the final output of the network is obtained. The output is formed by a group of units that classify the result (usually as many units as number of classes). Although CNNs are primarily designed for image processing, they can also be applied to EEG classification or natural language processing. Due to their versatility, some experiments will be performed in Chapter 6 to analyze them during EEG classification.

4 Layers that randomly remove certain features by setting them to zero
5 Technique that evaluates whether a model will be generalized to an independent dataset

3.5 RELATED WORKS

The benefits of FS in data mining applications for both supervised and unsupervised classification have been previously surveyed in [85–89], showing that FS in unsupervised problems is inherently a multi-objective problem [90]. Nevertheless, as the number of features involved in that applications is huge, an optimal FS becomes an *NP-hard* problem [91, 92], for which efficient *metaheuristics* are required. Even for a modest number of features, FS procedures based on B&B, SA, EAs, or combinations of them have been previously proposed. For that reason, [93, 94] consider parallel processing as an interesting approach for FS, and papers such as [95] have used *OpenCL* for its parallelization. In addition, [96] describes parallel approaches to address the presence of irregular samples and the dimension reduction problem, illustrating the effect of that approaches in algorithms like KNN[6] or *K-means*.

KNN: K-NEAREST NEIGHBORS

EAs can be applied to many different optimization problems as they do not require thorough knowledge of the problem at hand [97–99]. Indeed, in [100] a parallel multi-objective FS procedure is parallelized through different evolutionary strategies, but they do not parallelize the *fitness* evaluation of the individuals in the population. A similar situation occurs in [101], where the authors focus more on energy consumption analysis in different platforms for a sequential EA. A relatively high number of contributions on parallel implementations of EAs considering CPU-GPU can be found in the literature. Nevertheless, most of them do not completely exploit the energy-saving capabilities in HPC systems because they do not consider CPU and GPU as resources that can be equally used to distribute the workload. Instead, they usually tend to use the GPU and only one or few CPU threads to control the GPU activity. As examples, in [102] a parallel EA using MPI on only one platform is described. Paper [103] proposes a methodology to solve optimization problems in heterogeneous systems, and points out the usefulness of further researching on this approach.

The use of GPUs in evolutionary computation has been described in many previous papers [104]. In [105], an overview on parallelization of EAs is provided, and states that the most direct option to use the GPU is performing the evaluation step taking advantage of data parallelism present in the *fitness* function. An alternative in which the entire implementation falls to GPU could alleviate these problems [106]. Even so, these approaches have to take into account the memory requirements of the application, since the individuals and the dataset necessary to compute the *fitness* should be allocated in the GPU memory. E.g., paper

6 Classification method based on the most repeated class among the nearest K neighbors

[107] describes a CUDA implementation of a multi-population GA. It provides efficient implementations of genetic operators especially designed for GPU, and uses global memory for the *migration* process between subpopulations according to a unidirectional ring topology.

On the other hand, the algorithm proposed in [108] corresponds to an alternative implementation to NSGA-II. It also includes a *selection* strategy and a parallel implementation of the evaluation step. With a population of 10,000 individuals, the results show speedups of about 5,000 with respect to a CPU implementation of NSGA-II when using the ZDT3, ZDT4, and ZDT5 benchmarks[7] with two objectives. In [109], an efficient implementation of the NDS step of NSGA-II is evaluated with the DTLZ test suite[8], while paper [110] implements a MOGA for a data mining application on marketing. The approach executes all steps of NSGA-II in GPU except the non-dominated *selection* since there are some inconveniences such as the cost of the NDS function or the synchronizations between threads.

ZDT: Zitzler, Deb, and Thiele

DTLZ: Deb, Thiele, Laumanns, and Zitzler

The *K-means* parallelization has been considered and compared to CPU versions in [111–115]. E.g., in [113] a speedup of up to 68 is reported using a dataset with one million points and 4,000 centroids. The use of large datasets is considered in [114], where the clustering is applied to 1,000 two-dimensional centroids and one billion samples, achieving a speedup of more than 11 with respect to an optimized octa-core CPU. Moreover, paper [115] uses datasets with 500,000 samples and 2, 20, and 200 dimensions, showing speedups of up to 43 with respect to a sequential version, and speedups of 1.5-14 with respect to an optimized version that uses the SIMD instructions of the CPU.

EEG classification, like other high-dimensional applications, requires the processing of a huge amount of data and therefore it is a time-demanding problem. In this way, parallel processing and FS techniques are commonly applied to decrease the execution time and improve the quality of the solutions [116]. In paper [117], a MOGA is applied to solve an optimization problem in a MI-based BCI application. It implements a *wrapper* procedure for FS together with an unsupervised procedure whose performance is used to evaluate the *fitness* of each individual. The individuals of the population correspond to different chromosomes that define the features of the EEG signals to be classified. Moreover, with the recent boom in *Deep Learning* due to new processing algorithms and the performance growth of GPUs, some neural networks such as CNNs or DBNs are being used for EEG classification since they present good performances [118, 119].

7 Test suite for two-objective problems whose name comes from its authors
8 Test suite similar to ZDT but its tests problems are scalable in any number of objectives

Part II

CASE STUDY & DISCUSSION

4

METHODOLOGY

CONTENTS

This chapter aims to detail the methodology used. This includes a brief description of each algorithm developed, the platforms on which they are executed, as well as the statistical methods, performance metrics, and dataset used for their analysis. Since this thesis has been developed over several years, the methodology applied to analyze the algorithms has been subject to the available resources and needs at the moment. In order not to confuse the reader, for each experiment the setup and the resources for its realization are described. In any case, the same experimental conditions are set in those cases in which the results of different implementations are compared, so that a fair comparison between experiments is guaranteed.

4.1 PROPOSED APPROACH

The thesis proposes different parallel MOGA implementations to cope with the EEG classification problem in a MI-based BCI application, which can take advantage of parallelism at multiple levels. The evolutionary procedures are based on a *wrapper* approach where an NSGA-II algorithm evolves one or multiple populations of individuals that codify different FSs. This application, as discussed in previous chapters, is computationally heavy as it addresses a high-dimensional problem. For that reason, the proposed procedures described in Section 4.1.2 are designed to parallelize the task and/or distribute the work among the devices coupled to the heterogeneous processing systems. The characteristics of these platforms are described in Section 4.2.

4.1.1 *Basic Scheme*

As the implementations are based on the NSGA-II algorithm, their structure is similar to that shown in Algorithm 3.1. In summary, they include the initialization of the individuals with a maximum of 10 features (genes) initially set to 1, their evaluation, binary *crossover*, *mutation*, *selection*, and *migration* in the case of multi-population procedures. The difference lies in the parameters used for these procedures and the method to evaluate individuals. The multi-objective optimization either performs uniform *crossover* with a probability of 0.75 or bit-flip *mutation* with a probability of 0.25. In case of crossover, the parents are selected by binary tournament and the probability of exchanging a matching gene in both parents is 0.5. In case of mutation, each bit is flipped with a probability of 0.1. The algorithms developed use the unsupervised learning method *K-means* to evaluate the individuals. All of them perform one execution of *K-means* in each generation to compute their cost functions, f_1 and f_2, which are defined by two CVIs [120] (see Figure 4.1). The individuals are evaluated through a bi-objective *fitness* where f_1 and f_2 must be minimized and maximized, respectively. Specifically, f_1 correspond to the WCSS minimization, whose value can be calculated as the sum of the distances between each point P_t and the centroid k_j^i of the cluster to which it belongs:

CVI: Cluster Validity Index

WCSS: Within-Cluster Sum of Squares

$$f_1 = \sum_{j=1}^{K} \sum_{P_t \in C_j^i} \left\| P_t - k_j^i \right\|^2 \tag{4.1}$$

where K is the number of clusters, C_j^i the cluster j in the iteration i, and $\left\| P_t - k_j^i \right\|^2$ the Euclidean distance between point P_t and the centroid K_j^i. On the other hand, the cost function f_2 corresponds to the BCSS maximization, whose value can be calculated as the sum of the distances between each centroid:

BCSS: Between-Cluster Sum of Squares

$$f_2 = \sum_{j=1}^{K-1} \sum_{l=j+1}^{K} \left\| k_l^i - k_j^i \right\|^2 \tag{4.2}$$

being $\left\| k_l^i - k_j^i \right\|^2$ the Euclidean distance between the centroids k_l^i and k_j^i. Once the evaluation of the N individuals is finished, the *fitness* of each of them is normalized in the interval $(0, 1)$ according to the *softmax* function, which fulfills that the sum of all the normalized values is 1.

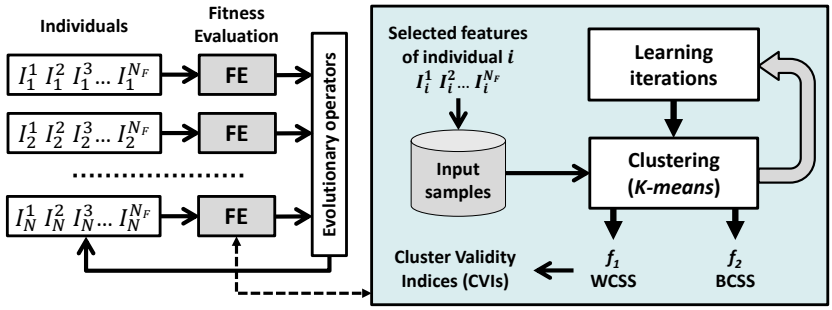


Figure 4.1: The proposed MOGAs implement a *wrapper* approach for FS where the performance of *K-means* clustering is used to evaluate the *fitness* of the individuals.

Therefore, The new value of the cost function f_m for a given individual i can be calculated as:

$$f_m^i = \frac{e^{f_m^i}}{\sum_{j=1}^{N} e^{f_m^j}} \tag{4.3}$$

4.1.2 *Description of the Algorithms*

Throughout this thesis a total of 10 procedures have been developed with *C++*, of which nine are different versions of the MOGA described in Section 4.1.1. Each of them can be seen as an evolution of the previous one that allows adding new functionalities, parallelism levels, or degrees of optimization with the aim of reducing execution time, energy consumption, and sometimes improving the readability of the code. Also, they do not have a graphical interface and can only be executed using the command line interpreter. The tenth procedure corresponds to a *Python*[1]-*C++* hybrid approach of an ANN as an alternative method to MOGA. A brief description of each procedure is provided below, whose main characteristics are summarized in Table 4.1:

- **Sequential Genetic Algorithm** (*SGA*): naive implementation that can only be executed sequentially on a single CPU core. This version is an adaptation of an equivalent *MATLAB*[2] code that, although executed sequentially, offers much better performance.

1 High-level interpreted language designed to create clear and logical code
2 Numerical computing environment and programming language developed by *MathWorks*

- **Parallel Genetic Algorithm** (*PGA*): this version exploits the potential of *OpenCL* to parallelize the evaluation of individuals by distributing each individual among CPU cores or GPU CUs. The rest of the MOGA steps are executed sequentially since, according to previous analysis, most of the execution time corresponds to the calculation of the *fitness* of the individuals.

- **Optimized Parallel Genetic Algorithm** (*OPGA*): similar to the previous one but it adds optimizations in both CPU and GPU kernels mainly related to memory management. This allows shorter execution times due to the reduction in memory consumption and coalescing access to data by transforming the original dataset into another stored in column-major order.

- **Multi-Device Genetic Algorithm** (*MDGA*): the host code is modified to add a scheduler designed with *OpenMP* that dynamically distributes the work between the CPU and GPU devices present on the platform through a master-worker scheme. In this way, the evaluation of individuals is carried out simultaneously on all available devices until each individual in the population is evaluated. The CPU and GPU kernel code do not present changes.

- **Multi-Population Genetic Algorithm** (*MPGA*): the evolutionary procedure is transformed into a multi-population MOGA where individuals can migrate every certain number of generations. The scheduler of the *MDGA* procedure is modified to distribute to each device either complete subpopulations or individuals if there is only one population. This allows up to three levels of parallelism without modifying the CPU and GPU kernels.

- **Distributed Genetic Algorithm** (*DGA*): a fourth level of parallelism is implemented by providing the application with multi-computer processing capacity through a master-worker procedure designed with MPI that dynamically distributes subpopulations among the *cluster* nodes. Each node acts following the *MPGA* model but also adds another level of *migration* that allows individuals to migrate between nodes.

- **Distributed Genetic Algorithm II** (*DGA-II*): this version is very similar to *DGA* but the *OpenCL* kernel for CPU is replaced by *OpenMP* code, so the evaluation of individuals is also part of the host code. In addition to simplifying the source code, the application's performance could increase depending on the host compiler and optimization flags chosen since the compiler can now manage all the CPU code.

Table 4.1: Characteristics of the procedures implemented in this thesis.

Feature	Type	Version									
		SGA	PGA	OPGA	MDGA	MPGA	DGA	DGA-II	ODGA	GAAM	DNN
Sequential	Only CPU	✓									
Parallel	CPU or GPU		✓	✓	✓	✓	✓	✓	✓	✓	✓
	CPU + GPU				✓	✓	✓	✓	✓	✓	
OpenMP	Scheduling				✓	✓	✓	✓	✓	✓	
	CPU compute							✓	✓	✓	
Optimized	Coalescence			✓	✓	✓	✓	✓	✓	✓	
	MPI messages								✓	✓	✓
Multi-population	Synchronous					✓	✓	✓	✓		
	Asynchronous									✓	
MPI	MOGA						✓	✓	✓	✓	
	CNN										✓

- **Optimized Distributed Genetic Algorithm** (*ODGA*): the communication process between nodes is optimized by reducing the number of messages and modifying the distribution scheme so that the master node has direct communication with the computing devices of a worker node. The objective is to reduce execution time and energy consumption by minimizing workload imbalance situations caused by the different parallelism levels.

- **Genetic Algorithm with Asynchronous Migrations** (*GAAM*): the approach based on a master-worker scheme is replaced by an multi-population MOGA that eliminates the idle states created by the *migration* processes between nodes. All nodes implement a hierarchy of I/O buffers and a handler that allows receiving or sending *migration* requests to other nodes asynchronously.

- **Distribution of Neural Network** (*DNN*): this ANN-based approach is presented as an alternative method to MOGA. The procedure reuses the master-worker skeleton of *ODGA* to evaluate multiple CNNs by distributing combinations of hyperparameters among the *cluster* nodes. The objective is to compare its energy-time behavior with respect to the MOGA versions and check if the MPI code can be easily adapted to other applications.

Table 4.2: CPU characteristics of the platforms used in the experiments.

Platform	Model	Cores/Threads	Core (MHz)	RAM (GB)	TDP (W)
Node 1	2x Intel Xeon E5-2620 v2	12/24	2,100	32	80
Node 2					
Node 3	1x Intel Xeon E5-2620 v4	8/16			85
Node 4	2x Intel Xeon E5-2620 v4	16/32			
PC	1x Intel i7 4770K	4/8	3,500	16	84

4.2 EXPERIMENTAL SETUP

To perform the experimental work two different platforms are used: a desktop PC that runs *Ubuntu* (v18.04) and a heterogeneous four-node *cluster* with *CentOS* (v7.4.1708) and a Gigabit Ethernet switch to connect the NUMA nodes. The characteristics of the CPU and GPU devices of these platforms can be found in Tables 4.2 and 4.3, respectively.

The neural networks of *DNN* procedure are executed with the *Python* interpreter (v3.6.5) and developed with *Keras* (v2.2.4) and *TensorFlow* (v1.12). All *C++* source codes are compiled with GCC compiler (v4.8.5), optimization flags `-O2 -funroll-loops`, and use the *OpenMPI* library (v1.10.7). The kernels are compiled with *OpenCL* (v1.2). Since the *cluster* contains heterogeneous CPUs, compiling with `-O3` implies a risk of incompatibility during execution. The `mpirun` command allows to specify a binary file for each MPI process that runs the program, but this can be cumbersome depending on how the application is executed, so the use of `-O2` instead of `-O3` is for simplicity. Even so, for some experiments the optimization flag `-O3` and ICC compiler are also used.

GCC: GNU Compiler Collection

ICC: Intel C++ Compiler

Cluster and PC energy measurements are obtained by two wattmeters based on the *Arduino Mega* board and developed specifically for the experiments. Every second they provide two different values: the instantaneous power, P (W), and the energy consumption, E (W · h). In the case of the *cluster*, the meters include both measurements for each node and the switch, although the energy values of the switch are much lower than those of the devices (below 5 W). Appendix B provides more detailed information on the operation of the wattmeter.

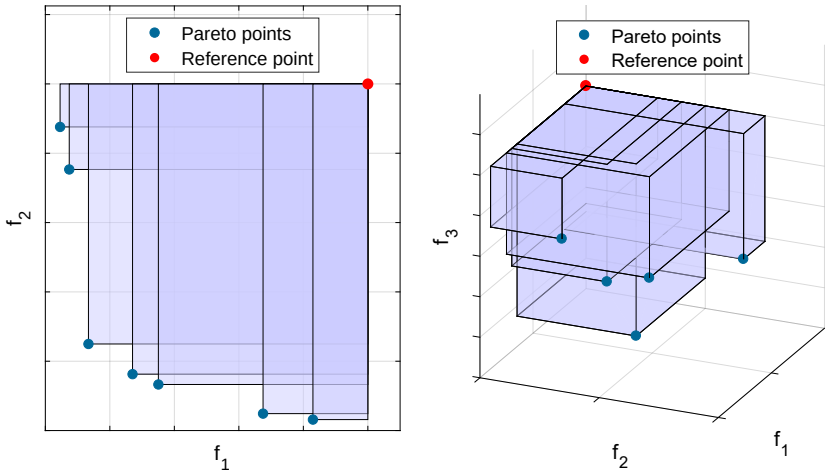
Table 4.3: GPU characteristics of the platforms used in the experiments. The two GPUs *TITAN Xp* of Nodes 3 and 4 are the same as those of the PC, so they rotate according to the experiment.

Platform	Model	CUs/Cores	Core/Memory (MHz)	RAM (GB)	TDP (W)
Node 1	1x Quadro K2000	2/384	954/4,000	2	51
Node 2	1x Tesla K20c	13/2,496	706/5,200	5	225
Node 3	1x Tesla K40m	15/2,880	745/6,008	12	250
Node 4	1x TITAN Xp	30/3,840	1,582/11,408		
PC	2x TITAN Xp	30/3,840	1,582/11,408		

4.3 EVALUATION METRICS

It is necessary to use a comparative metric to identify when one population is better than another. Although there are other measures in the literature to evaluate the quality of solutions, the difference in hypervolume between populations is considered here. The hypervolume consists in calculating the volume covered by the hypercube formed by the vectors of the *Pareto front*, PF, with respect to a reference point, r (see Figure 4.2). This point may be arbitrary, but it must be weakly dominated by all points belonging to PF. The hypervolume metric is calculated using the Fonseca and Zitzler [121–123] algorithms with the origin as reference point. However, to calculate it, all objective functions should be minimization or maximization. As this is not usual, minimization functions must be transformed into maximization or vice versa. The process is simple since maximizing a number is equivalent to minimizing its opposite value. Precisely, the cost functions f_1 and f_2 defined in Section 4.1.1 do not comply with the conditions to calculate the hypervolume, so f_1 is transformed to create a purely maximization problem. In this way, the objective is to obtain populations with the greatest possible hypervolume, and as the cost functions are normalized, the maximum hypervolume value is $hv = 1.0$.

The energy-time performance of the aforementioned approaches is compared by using execution time, instantaneous power, energy consumption, and speedup. The execution time, $t\,(s)$, is obtained by inserting a call to the *OpenMP* function `omp_get_wtime` at the beginning and end of the code and then compute the difference between both calls. Similarly, energy values are obtained by calling the measurement



(a) *2D space* (b) *3D space*

Figure 4.2: Hypervolume metric illustration. The shaded area define the portion of the objective space dominated by the *Pareto* points with respect to a reference point.

function just before and after executing the binary file of the procedure to be evaluated. The speedup, S, is calculated in Equation (4.4) as the quotient of dividing the execution time obtained after a sequential execution, t_S, by the execution time of its corresponding parallel execution, t_P. Although these measures are usually sufficient to compare the procedures, Equation (4.5) defines the energy-time product, Et, as another performance measure for those cases in which it is more difficult to identify the procedure that obtains the best results:

$$S = \frac{t_S}{t_P} \tag{4.4}$$

$$Et = E \cdot t \tag{4.5}$$

All experiments are only repeated 20 times due to the complexity of the algorithms evaluated and the number of experiments. To obtain a comparison between procedures as reliable as possible, the final value of each metric, including that of the hypervolume, is calculated as the average of all values. Also, for some experiments, the Kolmogorov-Smirnov test [124] is applied to determine whether the data follow a standard normal distribution or not. Depending on the result, either the ANOVA test [125] is applied if the data follow a normal distribution or the Kruskal-Wallis test [126] otherwise.

ANOVA: ANalysis Of VAriance

4.4 EEG DATASET

For the experiments, a datasets from the BCI Laboratory of the University of Essex is used, which correspond to a human subject coded as 110. The dataset contains EEG signals that can belong to three different cognitive tasks, or classes. Specifically, three types of movements: left hand, right hand, and feet. The process for recording the EEGs, described in [127], follows the typical protocol of a cue-based approach. The subject is sitting on a chair approximately 1.5 m away from the screen. For each trial, the subject experiences the following sequence shown on the screen:

1. **Blank screen**: the subject has between 2.5-3.5 s to relax and prepare for the next trial.

2. **Fixation cross**: at instant $t = 0$ and for 2 s, a fixation cross appears on the screen indicating the subject that the trial is about to start.

3. **Visual cue**: in addition to the fixation cross, a visual cue is displayed at $t = 2$ s for 6 s indicating the start of the trial, so a trial has a total duration of $t = 8$ s: An auditory cue is also emitted at $t = 2$ s. The relative position of the visual cue with respect to the fixation cross indicates the type of MI that the subject must perform: below for the feet movements and left and right positions for the corresponding movements of the hand.

Each recording session is divided into four runs, providing a total of 357 samples of which the first 178 are used for training and 179 for testing. Nevertheless, for the experiments only the training dataset is used since *K-means* is an unsupervised algorithm. EEG signals are acquired from 32 electrodes at 256 Hz with the *BioSemi system* [128] and processed using the MRA method, although finally only 15 of the 32 electrodes are useful. As a trial is recorded from the fixation cross to the end of the cue, an EEG contains $256 \cdot 8 = 2,048$ samples per channel. The resulting signal is composed by several segments, which are characterized by a set of details and approximation coefficients belonging to different levels of wavelets [129]. Generalizing, the dataset is defined by 15 electrodes, 20 segments, 6 wavelet levels, and 2 detail/coefficient. In total, there are $15 \cdot 20 \cdot 6 \cdot 2 = 3,600$ sets of coefficients whose sizes range from 4 to 128. In addition, as shown in [127], the number of coefficients is reduced to only $3,600$ by computing the second moment of the coefficient distribution (variance) for each set, and normalizing the obtained value in the interval $(0, 1)$. To summarize, the dataset used in this thesis contains 178 EEG signals composed of $3,600$ features.

MRA: MultiResolution Analysis

5

DEVELOPMENT ON SINGLE-COMPUTER SYSTEMS

CONTENTS

In this chapter, eight sets of experiments are carried out to analyze the procedures designed for single-computer systems. Specifically, versions *SGA*, *PGA*, *OPGA*, *MDGA*, and *MPGA*, which are tested in both performance and quality of the solutions under different experimental conditions. The chapter is organized as follows: Section 5.1 presents a first sequential and parallel versions. In Section 5.2, two parallel versions that take advantage of the devices are detailed and analyzed, while in Section 5.3 a multi-population version of the MOGA is exposed. In Section 5.4, an energy-time model is developed to predict and explain the behavior of the best version implemented so far, and finally Section 5.5 shows the conclusions after finish the experimentation.

5.1 THE FIRST IMPLEMENTATIONS

Prior to the development of this thesis, a *MATLAB* implementation of NSGA-II algorithm that can be found in [130] was used to address the EEG classification problem. This implementation was modified to use the *MATLAB* `kmeans` function for the evaluation of individuals. Although *MATLAB* works well and incorporates improvements and new functionalities over time, its performance is far from that offered by conventional programming languages such as *C++* or *Fortran*[1]. As EEG classification has a high computational cost, a low-level programming language could accelerate the application. In this way, *C++* is the ideal candidate to create more efficient implementations since it is also compatible with the main existing parallelism libraries.

5.1.1 *Porting a MATLAB Source Code to C++*

The portability from *MATLAB* to *C++* results in the first version: *SGA*. As stated in Chapter 4, this version is only capable of running on a single CPU core, just like the *MATLAB* implementation. Despite this, it has been proven that under the same experimental conditions and using the desktop PC, the execution time of *SGA* is better by a factor of 177 compared to that of *MATLAB*, confirming that the choice of *C++* was a wise decision. The operation of the program is simple: using the command line interpreter, the binary file is executed with the necessary parameters, which can be obtained from the command line or an XML configuration file (see Appendix C[2] for more information). After reading the parameters and checking that they are correct, the dataset is loaded into RAM, the initial centroids for *K-means* are randomly chosen, and NSGA-II starts. Once completed, the program returns the hypervolume of the final population and ends.

XML: eXtensible Markup Language

> **Experiment 5.1:** *Evaluate the execution time and hypervolume of SGA when increasing the number of individuals and generations. The experimental conditions are described in Table 5.1.*

Using the profiling tool *gprof* [131], it has been observed that the `evaluation` function of Algorithm 3.1, defined by *K-means* algorithm, is

1 High-level compiled language specially designed for numerical computation
2 The appendix describes another version but the information of the parameters is valid

Table 5.1: Experimental setup to analyze *SGA*.

Feature	Description
Platforms	Desktop PC: CPU
Compiler	GCC with -O2 -funroll-loops
Dataset	Only the first 480 features used
Individuals	$\{15, 30, 60, 120, 240, 480, 960$ $2500, 5000, 10000, 15000, 20000, 25000, 30000\}$
Generations	$\{1, \ldots, 50\}$

the part of the code that needs more time. Table 5.2 shows the distribution of execution time in *SGA* when increasing the population size. As can be seen, for moderate sizes the execution time of the `evaluation` function exceeds 99% and is linearly proportional to parameter N. Nevertheless, for larger sizes, the NDS step of NSGA-II gains importance due to its quadratic time complexity in the number of individuals. Therefore, from the perspective of execution speed, it might be better to increase the number of generations rather than the number of individuals. The time required by *K-means* for an iteration is defined as the sum of the time for the clustering step plus the time to update the centroids. However, as the process is repeated for i iterations, and the cost of computing f_1 and f_2 must also be added, the execution time is finally calculated as:

$$t_K = i \cdot (t_D \cdot K \cdot N_P + t_k) + t_W + \frac{K^2 - K}{2} \cdot t_D \tag{5.1}$$

being t_k the time to obtain the new K centroids, t_W the time to calculate the WCSS value, and t_D the time required to compute the Euclidean distance between two points. The expression $t_D \cdot K \cdot N_P$ corresponds to the clustering step, which depends on the number of points, N_P. On the other hand, $\frac{K^2-K}{2} \cdot t_D$ defines the cost of computing the BCSS value as the sum of the distances between each pair of centroids. Thus, in general terms, the execution time of *SGA* to evaluate N individuals along g generations can be estimated as:

$$t_S = g \cdot (t_N + N \cdot t_K + N \cdot t_O) \tag{5.2}$$

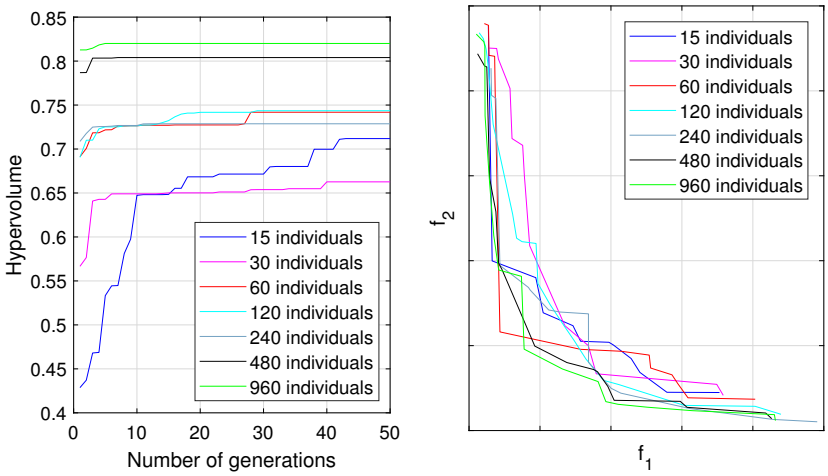
where t_N correspond to the NDS step and t_O is the time required to perform the *mutation*, *crossover*, and *selection* operators.

Table 5.2: Distribution of execution time in *SGA* when increasing the population size, N.

N	Evaluation		NDS		Others		Total
	$t(s)$	%	$t(s)$	%	$t(s)$	%	$t(s)$
120	119.19	99.93	0.01	0.01	0.07	0.06	119.27
240	236.38	99.92	0.07	0.03	0.12	0.05	236.57
480	477.00	99.90	0.14	0.03	0.32	0.07	477.46
960	954.85	99.87	0.70	0.07	0.60	0.06	956.15
2,500	2,492.26	99.72	5.21	0.21	1.87	0.07	2,499.34
5,000	4,973.70	99.48	21.74	0.43	4.69	0.09	5,000.13
10,000	9,984.37	99.05	85.22	0.85	10.11	0.10	10,079.70
15,000	14,946.12	98.60	196.61	1.30	15.40	0.10	15,158.13
20,000	19,518.64	98.00	377.64	1.90	22.53	0.10	19,918.81
25,000	24,961.48	97.78	539.68	2.11	27.40	0.11	25,528.56
30,000	29,959.06	97.36	778.95	2.53	33.00	0.11	30,771.01

Regarding the quality of the solutions, Figure 5.1 shows the hypervolume and *Pareto front* obtained along 50 generations for different population sizes, N. It can be seen that, in general, increasing the number of individuals also causes an increase in the hypervolume value. The maximum value, $hv = 0.82$, is reached for $N = 960$ and seems to be high enough to produce a stagnation in its value, since it has been proven that above that number the value hardly increases even increasing the number of generations.

The figure also clarifies which of the two parameters has most impact on the hypervolume. Although for small number of individuals the hypervolume grows quickly with the number of generations, the value of N has a higher impact on the final value. In addition, as Table 5.2 shows that most of execution time corresponds to the step of evaluation of individuals, the time required to evaluate N individuals for two generations is approximately the same as evaluating $2 \cdot N$ individuals for a single iteration. Taking this into account, it means that to reach a certain hypervolume value, less time is needed if the number of individuals is increased instead of the number of generations. In fact, for some values of N the hypervolume obtained after generation #1 is equal to or greater than that achieved by lower values of N after 50 generations. A more detailed study could find out the optimal value for N and the number of generations to reach a trade-off between them.



(a) *Hypervolume metric* (b) *Pareto front after 50 generations*

Figure 5.1: Hypervolume and *Pareto front* obtained in *SGA* when increasing the number of individuals and generations.

5.1.2 *A Parallel Implementation for CPU and GPU*

Although *SGA* is faster than the *MATLAB*-based implementation, its execution time remains very high. GAs, in addition to efficiently addressing the problem, have the advantage that each individual can be assessed independently. Taking into account that almost all execution time is consumed in the `evaluation` function, the parallel algorithm should focus on that part. Thus, *PGA* version is born, which is composed of a host code and two *OpenCL* kernels that can be launched by the host to CPU or GPU depending on how it was specified in the XML file. Before starting NSGA-II, in addition to the preprocessing step performed by *SGA*, the host code must also carry out other tasks related to the device that will execute *K-means*.

First, the device must be initialized by creating an execution context, a command queue, and the memory objects used by the kernel, which are copied to the device asynchronously. Although this kernel is compiled online, the *OpenCL* API also allows to use a previously compiled binary file. Before the first call to the `evaluation` function, a barrier is set to guarantee that the copy of previous memory objects has finished. For successive calls, only the individuals of the new population are copied since the rest of data, such as the dataset, does not change during the entire execution. Finally, within the function the host launches the kernel once its arguments are properly set.

5.1.2.1 *CPU and GPU Kernels*

The kernels are launched with as many *work-groups* as specified CUs. In case of GPU, a *work-group* is composed of several *work-items*, but in CPU, a *work-group* includes a single *work-item* since the number of CUs matches the number of logical cores. The input parameters of the kernel include the individuals, initial centroids, and dataset. The individuals are stored into global memory so that they can be accessed by all *work-groups*. The initial centroids and the dataset are also accessed by all *work-groups*, but they are stored into constant memory because they do not change during program execution. Although in GPU local memory is faster than the global one, its size is very limited, so there is no room for the dataset. In fact, during device initialization, the host must check possible memory overflows and leave a margin of 1 KB for internal operations of the *OpenCL* runtime (CPU local memory is RAM, so that there is no difference in performance).

The population is evaluated in parallel by assigning to each *work-group* the evaluation of one individual, which implies a full execution of *K-means* by using the dataset and the FS codified by the individual's chromosome. Moreover, in the GPU kernel, as each *work-group* includes several *work-items*, the evaluation of the individual is also parallelized taking advantage of data parallelism present in the Euclidean distances, providing two levels of parallelism.

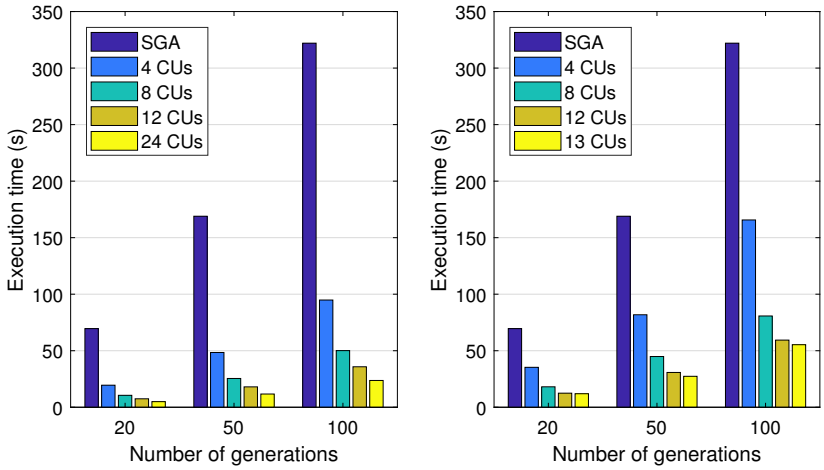
Both kernels use two auxiliary buffers stored into local memory to keep information about the changes in the centroids. One of the buffers stores information of *K-means* iteration i and the other one the results of iteration $i-1$. Once a iteration is over, the content of the buffers is exchanged. Data transfers between buffers requires $3 \cdot N_F$ operations, where N_F is the number of features of the dataset. To increase GPU performance, both centroids and the individual's chromosome are cached to local memory because the on-chip memory is faster. Finally, when *K-means* converges or a maximum number of iterations is reached, WCSS and BCSS values are obtained for the individual.

5.1.2.2 *SGA and PGA Evaluation*

Experiment 5.2: *Compare the execution time and speedup of SGA and PGA when increasing the number of generations and CUs. The experimental conditions are described in Table 5.3.*

Table 5.3: Experimental setup to compare *SGA* and *PGA*.

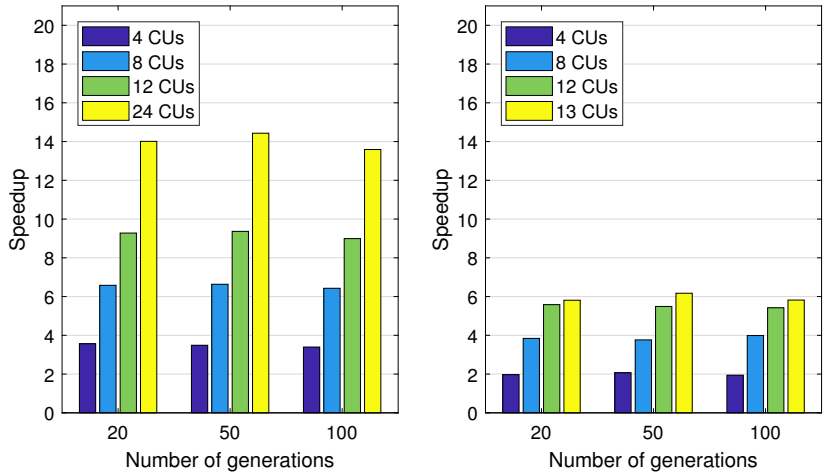
Feature	Description
Platforms	Node 2: CPU and GPU Tesla K20c
CUs / work-items	CPU: {4, 8, 12, 24} / 1 each
	GPU: {4, 8, 12, 13} / 192 each
Compiler	GCC with -O2 -funroll-loops
Dataset	Only the first 480 features used
Individuals	1,000
Generations	{20, 50, 100}



(a) *CPU* (b) *GPU*

Figure 5.2: Execution time of *SGA* and *PGA* when increasing the number of CUs and generations.

Figure 5.2 compares *SGA* and *PGA* along several generations, showing that the execution time increases proportionally when the number of generations increases. This behavior is as expected since the computational cost between generations is approximately the same. Smaller values are also appreciated for *PGA* when using any of the parallel devices. For similar numbers of CUs, i.e., 4, 8, 12, and 13, CPU gets better results. The difference in performance between both devices is even greater when CPU uses its 24 CUs due to GPU only has 13. Similar to execution time of Figure 5.2, Figure 5.3 shows the corresponding speedup achieved by *PGA* with respect to *SGA*. The speedup allows to identify behaviors that are more difficult to observe with the exe-



(a) *CPU* (b) *GPU*

Figure 5.3: Speedup achieved by *PGA* with respect to *SGA* when increasing the number of CUs and generations.

cution times, e.g., the speed gain factor with respect to a sequential execution. In CPU, for 4 CUs the speedup stays close to 4, whose value is the maximum possible for problems that scale linearly. For 8 CUs, the speedup grows but not in the same proportion, a situation that is further aggravated for 12 CUs. This is due to limitations in memory accesses where several CUs compete for RAM access. On the contrary, GPU does not have this problem since it is designed to be very efficient in memory operations and the handling of many *work-items* (threads). In fact, Figure 5.3b shows how the speedup with 8 CUs doubles that achieved with 4, and the ratio of going from 8 to 13 is respected. This means that distributing individuals among CUs is a good approach.

Finally, the case of 24 CUs is special since it seems that the program does not scale correctly. However, the explanation for its behavior is that, in addition to the problem of the memory accesses, it is also necessary to consider that CPU really has 12 physical cores with *Hyper-Threading*[3], so the performance of CUs 13 to 24 is not the same as the first 12. Regarding GPU, although appreciable speedups are provided, the performance is a bit poor. A possible explanation for this could be: (i) the $3 \cdot N_F$ operations of the auxiliary buffers in each *K-means* iteration; (ii) the synchronization barriers, and (iii) the irregularity in the SIMD parallelism due to FS, since the fact that some characteristics are selected and others do not make the performance worse.

3 Each CPU core include two logical cores that share hardware resources when possible

5.2 DEVICE-LEVEL IMPROVEMENTS

PGA has allowed to accelerate the application in several orders of magnitude. Nonetheless, as it is the first parallel version, it has room for improvement. Thus, Section 5.2.1 exposes some optimizations for the GPU kernel and performs a memory management analysis, while Section 5.2.2 deals with the cooperation capacity between CPU and GPU by introducing a scheduler capable of dynamically distributing the evaluation of individuals among both devices.

5.2.1 *Optimizing the CPU and GPU Kernels*

Although GPUs currently include several thousand PEs, one of its best assets is the SIMD processing capacity and memory frequencies higher than those of the CPU RAM. As its performance depends mostly on memory, the management of the different types of memory available and the way in which they are accessed are the key to achieving good results. Thus, this section details the operation of the next developed version, *OPGA*, which is focused on improving the GPU kernel. However, as CPU can benefit from certain optimizations, some of them are also applied to the CPU kernel.

The design of the CPU and GPU kernels is maintained for the rest of the versions developed in this thesis, so that their final pseudocodes can be seen in Algorithms 5.1 and 5.2, respectively. In the algorithms, expression << CUs, work-items >> defines which CUs and *work-items* will execute the following code block. E.g., in the CPU kernel, the expression in Line 2 indicates that the loop in Line 3 is executed by all CUs and their *work-items* #0. This means that each loop iteration is executed by the CU in charge of individual I_i (Line 4).

5.2.1.1 *Memory Management and Coalescing*

The memory optimizations carried out can be grouped into three types: (i) coalesced access to global and local memory; (ii) the proper use of the GPU memory hierarchy taking into account its weaknesses and strengths, and (iii) the reduction of memory consumption, which is important due to local memory scarcity. Table 5.4 compares the memory requirements of *PGA* and *OPGA*. The optimizations and aspects related to kernel components are listed below:

Algorithm 5.1: *OpenCL K-means* pseudocode defined in *OPGA* to evaluate a chunk of individuals in CPU.

```
1  Kernel KmeansCPU(I, N_I, DS, k)
       Input  : Individuals, I
       Input  : Chunk of individuals to be evaluated, N_I
       Input  : Dataset DS: N_P points (samples) with N_F features
       Input  : Set k of K centroids randomly chosen from DS
       Output : Individuals already evaluated, I
2      << All CUs, Work-item #0 >>
3      for i ← 1 to N_I individuals do
4          << CU in charge of I_i , Work-item #0 >>
5          kC ← Create a copy of the centroids
6          Initialization of the mapping table, MT ← 0
7          repeat
8              for j ← 1 to N_P points do
9                  MT_j ← Point p_j in DS is assigned to a cluster
10                 ND_j ← Store the distance for point p_j
11             end
12             kC ← Update centroids using dataset DS
13         until stop criterion is not reached;
14         f_1(I_i) ← wcss(kC, ND) according to Equation (4.1)
15         f_2(I_i) ← bcss(kC) according to Equation (4.2)
16     end
17     return I
18 End
```

1. In both *PGA* and *OPGA*, the N individuals of array I are stored into global memory and cached individually to local memory during the evaluation step. An individual I_i is a one-dimensional array of contiguous N_F 1's and 0's according to the selection or not of the corresponding feature. Thus, the amount of global memory used by I is $N_F \cdot N$ bytes and the local memory for each I_i, N_F bytes, being N_F is the chromosome size. In the GPU kernel of *OPGA*, in addition to dataset DS, a transposed version of DS called DS^T is also used to perform the centroid update step more efficiently. In DS, the points are organized in row-major order while column-major order is used in DS^T. Both datasets include N_P points with N_F features and are also stored into global memory in another one-dimensional array of $N_P \cdot N_F$ elements.

Algorithm 5.2: *OpenCL K-means* pseudocode defined in *OPGA* to evaluate a chunk of individuals in GPU.

```
1  Kernel KmeansGPU(I, N_I, DS, k, DS^T)
       Input  : Individuals, I
       Input  : Chunk of individuals to be evaluated, N_I
       Input  : Dataset DS: N_P points (samples) with N_F features
       Input  : Set k of K centroids randomly chosen from DS
       Input  : Dataset DS^T is DS in column-major order
       Output : Individuals already evaluated, I
2      << All CUs, All their work-items >>
3      for i ← 1 to N_I individuals do
4          << CU in charge of I_i , All its work-items >>
5          kC ← Copy the centroids from global to local memory
6          IC ← Copy individual I_i from global to local memory
7          Initialization of the mapping table, MT ← 0
8          repeat
9              << CU in charge of I_i , 1 work-item >>
10             for j ← 1 to N_P points do
11                 MT_j ← Point P_j in DS^T is assigned to a cluster
12                 ND_j ← Store the distance for point P_j
13             end
14             << CU in charge of I_i , All its work-items >>
15             kC ← Update centroids using dataset DS
16         until stop criterion is not reached;
17         << CU in charge of I_i , Work-item #0 >>
18         f_1(I_i) ← wcss(kC, ND) according to Equation (4.1)
19         f_2(I_i) ← bcss(kC) according to Equation (4.2)
20     end
21     return I
22 End
```

Taking into account that a floating-point data has a size of 4 bytes, each dataset requires $4 \cdot N_F \cdot N_P$ bytes of global memory. Although *OPGA* doubles the *PGA* memory requirement to store both datasets, this is not a problem since the amount of global memory available in current GPUs is generally much greater than that required by EEG datasets. To be more specific, each database requires approximately a total of 2.44 MB, while the global memory of the GPUs used is around several GB.

Table 5.4: Memory (in bytes) used by the GPU kernels of *PGA* and *OPGA*. N_I, K, N_F, and N_P are, in this order, the number of individuals, centroids, features, and points.

Memory		Global		Constant	Local			
Array		I	DS/DS^T	k	KC	I_i	MT	ND
Size	**PGA**	$N_F N$	$4N_F N_P$	$4N_F K$	$4N_F K$	N_F	$3N_P K$	$4N_P K$
	OPGA		$8N_F N_P$	$4K$			N_P	$4N_P$
Total size	**PGA**	$N_F N + 4N_F N_P$		$4N_F K$	$4N_F K + 7N_P K + N_F$			
	OPGA	$N_F N + 8N_F N_P$		$4K$	$4N_F K + 5N_P + N_F$			

2. In *OPGA*, instead of copying the K centroids as in *PGA*, only the indices of these centroids are copied from host memory to GPU constant memory, reducing memory usage from $4 \cdot N_F \cdot K$ to $4 \cdot K$ bytes. This is possible because the centroids are randomly selected from the dataset, which is already available in global memory. On the other hand, as the positions of the centroids are modified along the *K-means* iterations, it is necessary to create a copy of the centroids into local memory whenever a new individual is going to be evaluated (Line 5). In *OPGA*, the copy is executed in parallel by all *work-items* of its corresponding CU through *coalescing*, a technique where consecutive PEs request data stored into global memory, in consecutive logical addresses. This technique aims to minimize the number of transaction segments requested from global memory by taking advantage of the memory bus width to get multiple data in a single transaction. The kernels are able to use *coalescing* because consecutive *work-items* request data stored into consecutive logical addresses of global memory. As Figure 5.4 shows, the memory bank conflicts in local memory are minimized. When the *work-items* process the first chunk of data, the next chunk is requested and processed, and so on until the whole dataset is processed. In the CPU kernel, the unique *work-item* of the CU sequentially performs the copy of centroids. In both *PGA* and *OPGA* the centroids require $4 \cdot N_F \cdot K$ bytes of local memory.

3. The mapping table *MT* contains an `unsigned char` datatype representing the centroid assigned to each point along the *K-means* iterations. Its initialization in Line 7 is carried out by all *work-items* in the same way as the previous initialization of centroids and individuals. This table replaces the two auxiliary buffers included in the *PGA* kernels. In *MT*, each point only stores the index of its corresponding centroid, while in *PGA* a binary coding similar to

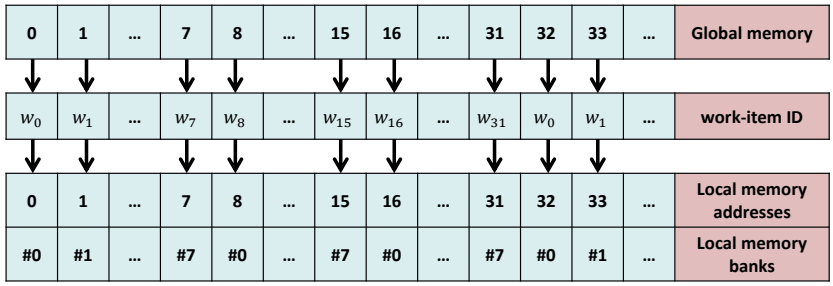


Figure 5.4: *Coalescing* technique illustration. A group of 32 *work-items* (*warp*) copy data from global to local memory providing coalesced access to global memory and minimizing memory bank conflicts.

that of chromosomes is used indicating whether a centroid is assigned to point P_j (1) or not (0). Therefore, the use of MT allows to reduce the local memory consumption from $3 \cdot N_P \cdot K$ to N_P bytes. Also, through this table it is easier to check the algorithm convergence since the number of operations is reduced.

4. Each *work-item* has to find the nearest centroid for a specific point by using the Euclidean distance between its point and the centroids (Line 11). As dataset DS^T is stored in column-major order, the first N_P memory addresses contain the values of the first feature for all points, while the following N_P addresses contain the values of the second feature, and so on. Therefore, as each *work-item* handles a different point in a given time, consecutive *work-items* will request consecutive memory addresses, allowing fully coalesced access to global memory. Moreover, when the nearest centroid to a given point and the corresponding distance are obtained, they can be written into MT and ND, respectively, with the minimum number of memory bank conflicts. Array ND is stored into local memory and includes the Euclidean distances between each point and its closest centroid. As its representation is analogous to that of MT, its memory requirement is reduced in *OPGA* from $4 \cdot N_P \cdot K$ to $4 \cdot N_P$ bytes. Finally, concerning to the CPU kernel, DS^T is not necessary since the *work-item* that calculates the distances access to DS memory addresses consecutively.

5. The most complex *K-means* step in terms of data parallelism is the centroids update (Line 15). Indeed, some approaches [132, 133] directly propose to perform this step sequentially in the host, although the cost per iteration associated with transferring the centroids to the host, processing them, and returning them could be too high, specially in applications with high-dimensional

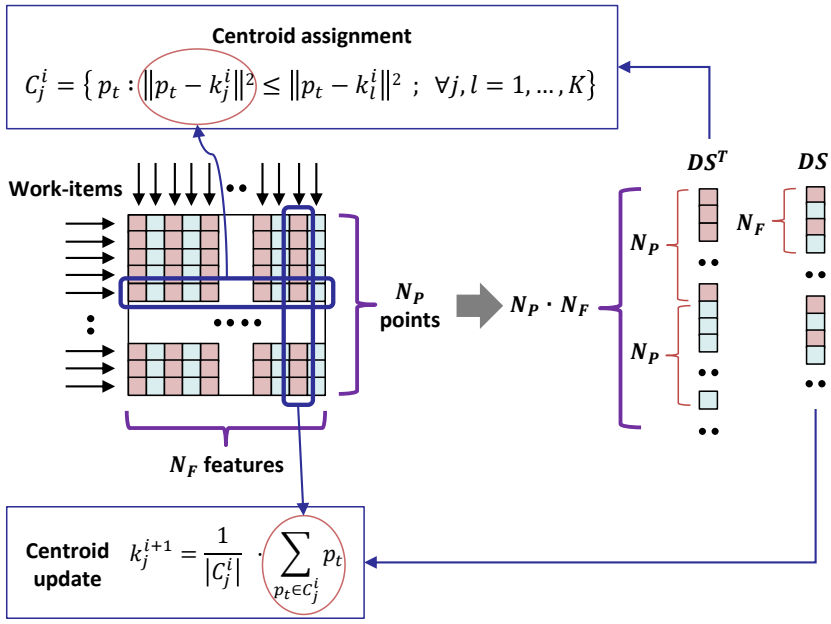


Figure 5.5: Scheme that shows how the *work-items* of the GPU kernel defined in *OPGA* access to *DS* and DS^T during *K-means* execution.

points. To avoid that, in the GPU kernel each *work-item* adds the same feature of all points belonging to the centroid to update. To perform this step, dataset DS^T is not adequate as consecutive *work-items* compute consecutive features. Now, *DS* is used because its first N_F memory addresses contain all features of the first point, the following N_F addresses contain the features of the second point, and so on. Thus, coalesced memory access can be achieved and the memory bank conflicts are minimized. Figure 5.5 shows the relation between *work-items* and datasets *DS* and DS^T during *K-means* execution.

5.2.1.2 *Speedup Analysis*

Experiment 5.3: *Compare the GPU performance in PGA and OPGA and analyze their behaviors when the number of work-items per CU is increased above the maximum that the device can execute simultaneously. CPU-GPU comparisons in both versions are also made. The experimental conditions are described in Table 5.5.*

Table 5.5: Experimental setup to compare *PGA* and *OPGA*.

Feature	Description
Platforms	Node 2: CPU and GPU Tesla K20c
CUs / work-items	CPU: $\{4, 8, 12, 24\}$ / 1 each GPU: $\{4, 8, 12, 13\}$ / $\{192, 256, 512, 1024\}$ each
Compiler	GCC with -O2 -funroll-loops
Dataset	Only the first 480 features used
Individuals	1,000
Generations	50

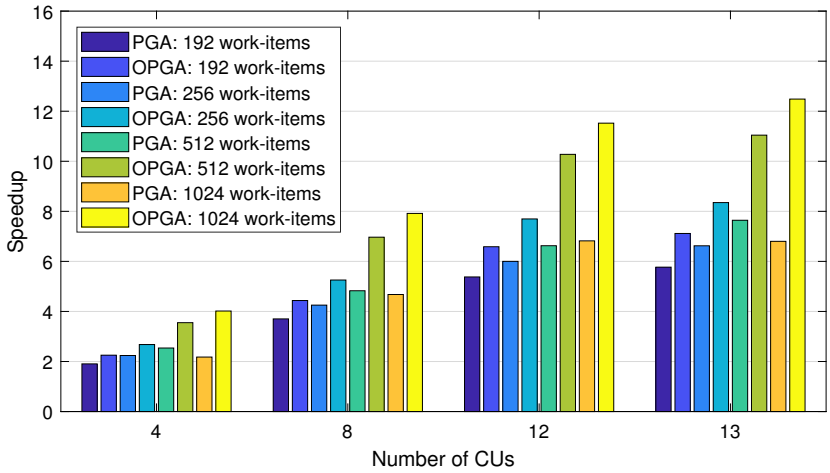
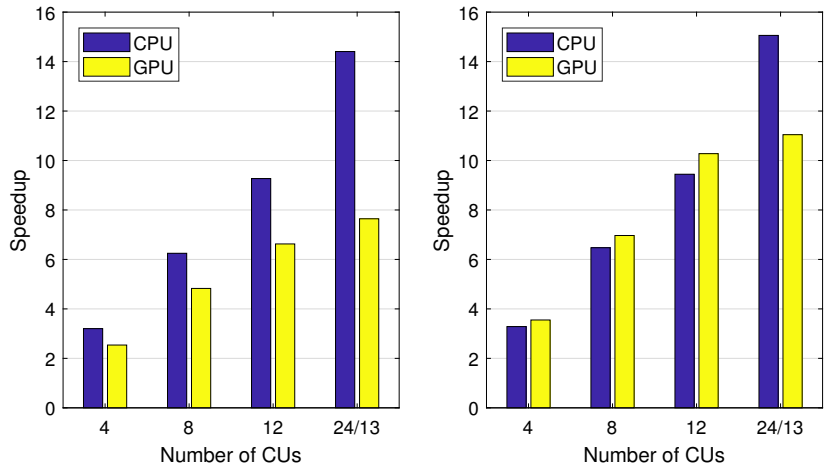


Figure 5.6: GPU speedup achieved by *OPGA* with respect to *PGA* when increasing the number of CUs and *work-items*.

Figure 5.6 shows the speedup achieved by the GPU kernels of *OPGA* and *PGA* when increasing the number of CUs and *work-items*. As can be seen, for any combination of CUs and *work-items* the speedup of the new GPU kernel exceeds that of *PGA*. The differences in all speedups are statistically significant after applying the Kolmogorov-Smirnov and Kruskal-Wallis tests, with p-values[4] lower than 0.009. The figure also reveals a surprise: using more *work-items* than the maximum that GPU can handle simultaneously gives better results in both versions. When *SGA* was compared to *PGA*, 192 *work-items* per CU were used because GPU has a total of 2,496 PEs divided into 13 CUs, that is, $\frac{2{,}496}{13} = 192$ *work-items*. The explanation for the observed behavior is that GPU has

4 A p-value lower than 0.05 is statistically significant



(a) *PGA: 512 work-items per CU in GPU* (b) *OPGA: 1,024 work-items per CU in GPU*

Figure 5.7: CPU-GPU speedup achieved by *PGA* and *OPGA* when increasing the number of CUs.

the ability to hide latencies by enqueuing the leftover *work-items* that are waiting for their turn. However, the number of *work-items* per CU is not infinite since *OpenCL* sets a maximum of 1,024 per dimension[5]. The appropriate value will depend on the specific application and the size of the problem. Another conclusion can also be drawn from the figure: when more than 256 *work-items* are used, the gap between both versions is greater due to the effects of the *coalescing* technique. Even for the specific case of 1,024 *work-items*, version *PGA* gets worse performance than using 512. The opposite effect occurs in *OPGA*.

In both versions, the speedup obtained again is proportional to the number of CUs. On the contrary, when increasing the number of *work-items* the proportion is much smaller. Although more *work-items* are being used than the maximum that the device can handle simultaneously, it has been proven that with values lower than 192 (32, 64, and 128), the rate of increase is not completely proportional. This means that the parallelism level corresponding to the distribution of individuals among CUs is more efficient than the second parallelism level, which takes advantage of data parallelism available in *K-means*. It has to be taken into account that in the application, the amount of parallelism available changes according to the number of features selected in each individual. Given that the GPU kernel of *PGA* gets its best performance with 512 *work-items* per CU, the question is whether in Experiment 5.2

5 *OpenCL* can organize *work-items* up to three dimensions, which are known as *Ranges*

the GPU kernel would have exceeded that of CPU when using that amount of *work-items*. Figure 5.7a answers that question. What is observed is that the difference between both kernels is reduced, but it is not enough to reach the CPU kernel. Figure 5.7b compares the CPU and GPU kernels of *OPGA*. When CPU uses its 24 CUs, the performance is clearly higher to that of GPU, but with the same number of CUs, the optimized GPU kernel is better. Although CPU does not obtain the same benefit from *coalescing* as GPU nor from the use of the memory hierarchy because they are all the same, its speedup is increased by 5.5% thanks to optimizations related to memory reductions. E.g., it avoids the copies of the auxiliary buffers required in each iteration and reduces the number of operations to initialize array ND to 0.

5.2.2 *Dynamic Distribution of Individuals*

While *OPGA* optimizes the kernels, version *MDGA* developed in this section focuses on the host code. Instead performing optimizations, *MDGA* adds the ability to dynamically distribute the evaluation of individuals to CPU and GPU through a scheduler whose pseudocode can be seen in Algorithm 5.3, providing up to three parallelism levels in GPU. Its operation is as follows: through the *OpenMP* `pragma` of Line 3, the host forks a thread for each available device to manage everything related to the kernel (copy individuals, launch the kernel, and store results). The N_I individuals to be evaluated are distributed among the threads in chunks of size C_S using the *OpenMP* `pragma` of Line 11. These threads share the pointer that iterates over the array of individuals, so that when one of them picks up its next chunk, it increases atomically the value of the pointer (Line 13). The value of C_S is conditioned to the number of existing devices (Lines 6 and 8). If there is only one, $C_S = N_I$ since there is no need for distribution (as in *OPGA*). On the contrary, if there are several devices, as the individuals within the kernel are distributed among CUs, C_S will be equal to the number of CUs of the device to avoid workload imbalance.

5.2.2.1 *Scheduler Overhead*

Experiment 5.4: *Measure the overhead time caused by the scheduler of MDGA on different devices when evaluating individuals in multiple chunk sizes. The experimental conditions are described in Table 5.6.*

Algorithm 5.3: Pseudocode of the scheduler defined in *MDGA* to distribute the evaluation of individuals among CPU and GPU.

```
1  Function scheduler(I, N_I, D, N_D)
       Input  : Individuals, I
       Input  : Number of individuals to be evaluated, N_I
       Input  : Object D containing the OpenCL devices
       Input  : Number of devices, N_D
       Output : Individuals already evaluated, I
       // Shared pointer between all OpenMP threads
2      Ptr ← 0
3      #pragma omp parallel num_threads(N_D)
4          D_j ← Copy I to device j // ∀j = 1, ..., N_D
5          if N_D > 1 then
6              C_S ← Number of CUs of device D_j
7          else
8              C_S ← N_I
9          end
10         repeat
11             #pragma omp atomic capture
12                 Priv_Ptr ← Ptr
13                 Ptr ← Ptr + C_S
14             end
               // Datasets and k were previously copied to D_j
15             if D_j is CPU then
16                 I_Priv_Ptr ← kmeansCPU(I, C_S, DS, k)
17             else
18                 I_Priv_Ptr ← kmeansGPU(I, C_S, DS, k, DS^T)
19             end
20         until Ptr ≥ N_I;
21     end
22     I ← normalizeFitness(I)
23     return I
24 End
```

The smaller the value of C_S, the greater the number of times the kernel must be launched to evaluate all individuals. As each device uses a different C_S value, its overhead percentage Ov can be calculated as:

$$Ov\,(\%) = \frac{t_{CPY} + t_{CL}}{t} \cdot 100 \tag{5.3}$$

Table 5.6: Experimental setup to analyze *MDGA*.

Feature	Description
Platforms	Node 2: all devices
CUs / work-items	CPU: $\{1, \ldots, 24\}$ / 1 each GPU: all / 1,024 each
Compiler	GCC with -O2 -funroll-loops
Dataset	All features used
Individuals	$\{120, 240, 480, 960\}$
Generations	50

Table 5.7: Time overhead caused by the scheduler defined in *MDGA* when 120 individuals are evaluated in N_C chunks.

Device	N_C	Kernel (ms)	t_{CPY} (ms)	Bandwidth (GB/s)	t_{CL} (ms)	t (ms)	Ov (%)
Quadro	2	587.09	0.33	2.44	47.91	635.33	7.60
K2000	120	575.89	0.21	6.24	3.05	579.15	0.56
Tesla	13	263.70	0.23	4.85	11.29	275.22	4.18
K20c	120	257.37	0.21	6.26	3.07	260.65	1.26
Xeon	24	230.75	-	-	7.95	238.70	3.33
E5-2620 v2	120	229.05			1.64	230.68	0.71
Sequential	120	2,546.60	-	-	-	2,546.60	0.00

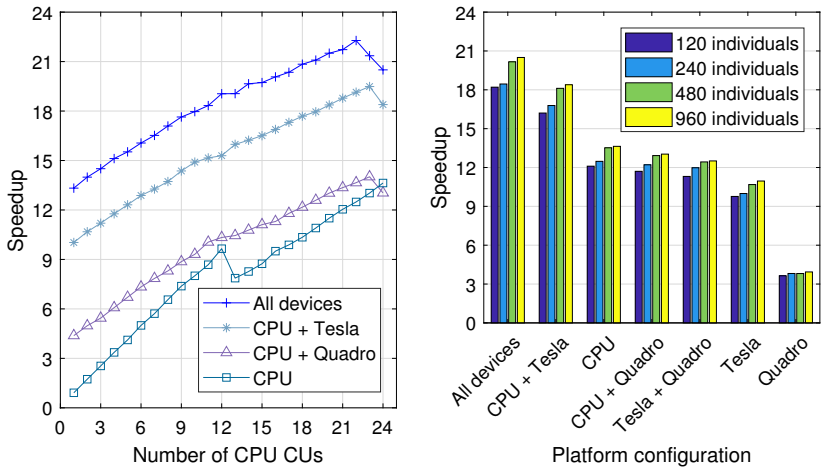
where t_{CPY} is the time to transfer the individuals from host to the corresponding device, t_{CL} the time required to setup and finish the kernel, and t the total execution time. As the measurements for t_{CPY} and bandwidth are provided by the *NVIDIA* profiler, the equivalent values for CPU could not be obtained. Table 5.7 shows the overhead percentage for each device when C_S changes. The table demonstrates that the overhead percentage is higher whenever C_S is lower than the population size (120 in this case). It also reveals that the overhead is higher in GPU *Quadro* than in GPU *Tesla* when the scheduler assigns as many individuals as CUs have the devices (hereinafter, only *Quadro*, *Tesla*, and *Xeon* will be written to simplify reading). The reason is that the kernel time is much higher in *Quadro* than in *Tesla*. However, to compute all individuals, *Quadro* offers more overhead than *Tesla* since it needs 60 kernel executions but *Tesla* only 5.

Indeed, the overhead generated by the multiple calls to the CPU and GPU kernels is quite high. It would be interesting to explore other implementations to reduce such overhead. E.g., a benchmark could be run at the beginning of the program to study the relative performance between devices and apply a static distribution of individuals. However, although in *MDGA* all parameters are known a priori, a static strategy based on the performance of the CPU and GPU CUs could be difficult to apply to huge heterogeneous platforms. Moreover, other applications may have dynamic parameters at runtime where static partitioning is not adequate or cannot be performed. Even in *MDGA*, the stop criterion of *K-means* could be replaced by another where a certain error threshold is reached. For that reason, the dynamic distribution scheme of the scheduler defined in *MDGA* should generally provide better performance in problems with more irregular workloads.

5.2.2.2 *Speedup and Scalability*

> **Experiment 5.5:** *Evaluate the speedup of MDGA when different combinations of devices and number of individuals are used. The experimental conditions are also described in Table 5.6.*

Figure 5.8a shows the speedup achieved by four platform configurations when increasing the number of CUs in CPU and using all CUs in GPU. Starting with the CPU case, the speedup grows when more CUs are used except from 13 to 16 because in that interval the parallelism obtained does not compensate for the cost of multiplexing some physical cores between two CUs. From 16 CUs, the speedup is greater than using only 12 and reaches a peak of 13.63 when CPU uses all its 24 CUs. The figure also depicts the speedup improvement when the GPUs are cooperating with CPU. The best speedup is obtained when all devices are used simultaneously regardless of the number of CPU CUs involved. *Quadro* provides a small speedup improvement when collaborating with CPU since it only has 2 CUs. If *Tesla* is used instead *Quadro*, the opposite occurs because the computing capabilities of *Tesla* and CPU are more balanced, as was verified in Experiment 5.3. As in the CPU case, for any CPU-GPU combination there are also slight speedup reductions for CUs near to 13. In those situations, the reductions are less pronounced than that of the CPU case. Nevertheless, the overhead in configurations involving GPUs is greater. This can be checked by observing that the highest speedups are achieved with 23 CUs in configurations with two devices (CPU + *Tesla* and CPU +



(a) *Scaling the number of CPU CUs and using all GPU CUs to evaluate 960 individuals*

(b) *Each device uses all its CUs*

Figure 5.8: Speedup of *MDGA* for different platform configurations.

Quadro), and with 22 CUs when using all devices. In the latter case, as the scheduler creates two *OpenMP* threads to manage the GPUs, if CPU also uses its 24 CUs to evaluate individuals a total of 26 threads are being executed simultaneously. As this number exceeds the maximum CPU capacity (24), obtaining the best speedup with 22 CUs is the expected behavior. Therefore, in a hypothetical heterogeneous platform in which the scheduler must manage a large number of devices, an approximate value of C_S for a given CPU could be:

$$C_S = \max\left(1, \lambda_C - N_D\right) \tag{5.4}$$

where λ_C is the number of CPU CUs. On the other hand. Figure 5.8b summarizes the highest speedups attained by all possible platform configurations using different population sizes. It clearly shows the improvement achieved when more devices are used to distribute the evaluation of the individuals. In addition, it has been checked that the values for all population sizes and platform configurations are statistically significant. Undoubtedly, the introduction of the scheduler in *MDGA* has caused a greater impact on application performance than that caused by the memory optimizations of *OPGA*. In addition, the difficulty of the scheduler is low since it is implemented with only seven source code lines and some slight changes to the existing code. Despite the good results, it is quite difficult to reach the floating-point peak performance of CPU and GPU. E.g., the GPU performance

is greatly impaired by the conditional branches in the kernel code (`if-then-else` statements), which cause the *work-items* of the same *warp* to be branched between both statements. As the SIMD model only allows the execution of one instruction for all *work-items*, those which execute the `if` statement must wait for those which execute the `else` and vice versa. Moreover, synchronizations between *work-items* and the copy of the individual to be evaluated into local memory also decrease performance. Although almost all of those drawbacks are not present in CPU, the achieved speedups are similar in both devices.

5.3 A MASTER-WORKER MULTI-POPULATION APPROACH

While Section 5.2 presented some optimizations to increase performance, this section extends the existing MOGA to a multi-population model where the subpopulations are distributed among the available devices. This results in the fifth version developed in this thesis: *MPGA*.

First, the GA scheme is modified so that each subpopulation has its own evolutionary process. This can be implemented similar to the *MDGA* scheduler, that is, creating as many CPU threads as available devices through the corresponding *OpenMP* `pragma` to parallelize the loop that iterates over all subpopulations. In this way, each subpopulation Sp_i is assigned to one of these CPU threads, which execute the evolutionary operators for their corresponding subpopulation and performs a call to the `evaluation` function. Within the function, as the scheduler defined in *MDGA* always distributes individuals among the detected devices, its behavior must be modified to adapt it to the new circumstances. Basically, to evaluate the subpopulation received, the scheduler first checks the total number of subpopulations in the application and the number of devices: if only one subpopulation is evolving, the *MDGA* behavior is applied, but if there are several subpopulations, the device managed by the CPU thread that called the function will evaluate all individuals. Thus, *MPGA* offers two dynamic scheduling alternatives to evaluate individuals, which it is baptized as DSSI (Figure 5.9).

DSSI: Dynamic Scheduling of Subpopulations or Individuals

Concerning to the *migration* process, a *migration* implies to build a new set of subpopulations from individuals of other subpopulations after a certain number of generations. The process is performed by the master CPU thread and each subpopulation contributes with the half of individuals present in its *Pareto front* at most. When all generations have been completed, the master CPU thread merges all subpopulations into a single array of size $N_{Sp} \cdot N$ and applies the NDS step. Once the array is sorted, the final subpopulation is obtained by picking up the first N

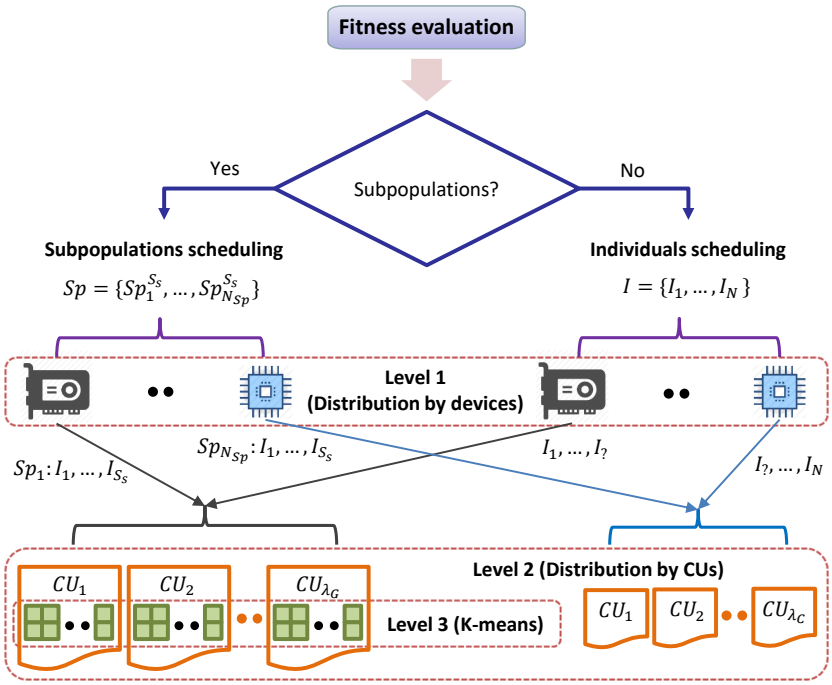


Figure 5.9: Scheme defined in *MPGA* that allows two dynamic scheduling alternatives to evaluate individuals. It distributes individuals if only one subpopulation is detected, or subpopulations otherwise.

individuals regardless of whether the size of the resulting *Pareto front* is larger than N or not. Although that situation is possible, it did not occur in any of the experiments carried out.

5.3.1 *CPU-GPU Performance*

Experiment 5.6: *Compare the performance of MDGA and MPGA when MPGA increases the number of migrations and the total number of individuals is divided into multiple subpopulations. The experimental conditions are described in Table 5.8.*

Figure 5.10 provides the speedup obtained in *MPGA* when the total number of individuals is divided into multiple subpopulations. Specifically, $N = 480$ individuals are divided into 1, 2, 4, 8, and 16 subpopulations which contain 480, 240, 120, 60, and 30 individuals,

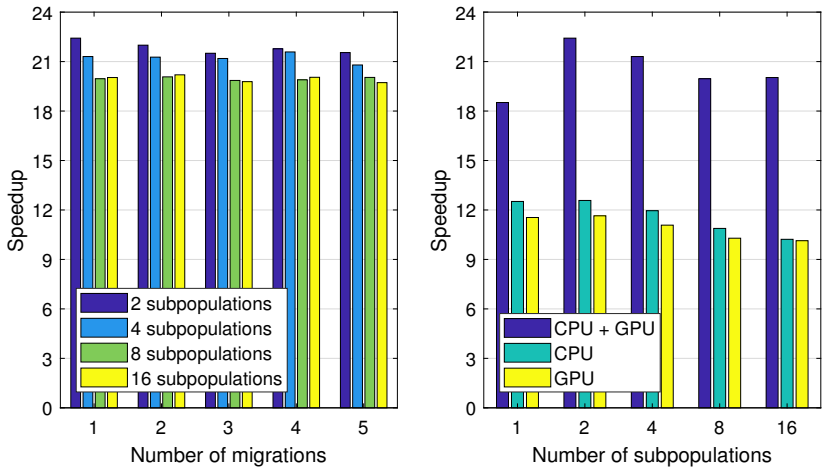
Table 5.8: Experimental setup to compare *MDGA* and *MPGA*.

Feature	Description
Platforms	Node 4: CPU and GPU Tesla K40m
CUs / work-items	CPU: all / 1 each GPU: all / 1,024 each
Compiler	GCC with -O2 -funroll-loops
Dataset	All features used
Individuals	480
Subpopulations	$\{1,\ldots,32\}$
Generations	60
Migrations	$\{1,\ldots,5\}$

respectively. Figure 5.10a shows results from $N_M = 1$ to 5 *migrations* and, as all executions are run along $g = 60$ generations, a *migration* is performed every $\frac{g}{N_M}$ generations. When increasing the number of *migrations*, the speedups remain approximately constant. A *migration* implies the exchange of individuals between subpopulations and therefore a cost in execution time. If increasing the number of *migrations* has no impact on performance, it means that the evaluation of individuals remains the dominant step in terms of computational cost.

In addition, according to the statistical analysis the differences between executions are not significant and therefore would only be fluctuations caused by the workload of the operating system at that time. Thus, the main speedups changes shown in the figure seem to be determined by the number of subpopulations and their size, where the highest value is always reached for $N_{Sp} = 2$ subpopulations and the lowest for 8 or 16, depending on the case. Why this happens is easy to understand: increasing the number of subpopulations also reduces the number of individuals per subpopulation, a situation that was already analyzed in Figure 5.8b of experiment Experiment 5.5. This figure showed the speedup changes for the same population sizes as in the current experiment, which match.

Figure 5.10b evaluates the speedup for different platform configurations without performing *migrations*. The performance of each device (CPU or GPU) is similar to that seen in version *MDGA*, although a bit higher since in this experiment Node 4 of the cluster is being used and it is slightly more powerful than Node 2. Therefore, the interesting thing here is to compare the case of 1 subpopulation with the rest using CPU



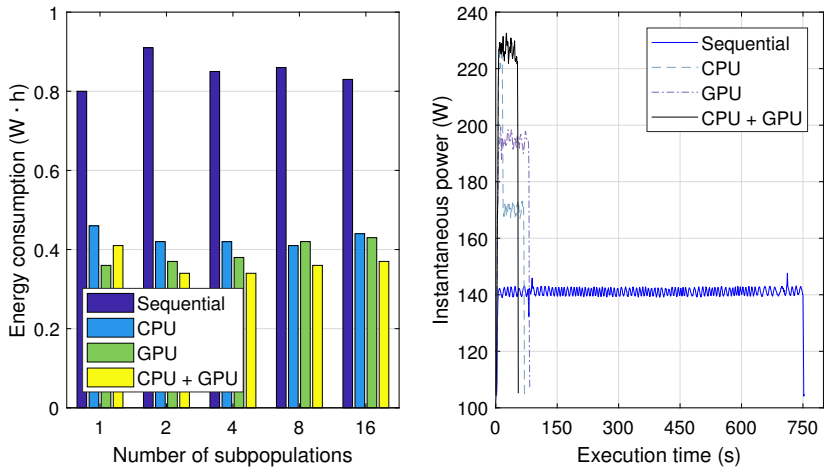
(a) *Increasing the number of migrations*

(b) *Using different platform configurations without performing migrations*

Figure 5.10: Speedup obtained in *MPGA* when the total number of individuals is divided into multiple subpopulations.

and GPU simultaneously. When *MPGA* only evolves 1 subpopulation and does not perform *migrations*, it behaves like *MDGA*. With this in mind, the performance of both versions can be compared. Simplifying, what can be observed is that the speedup for 1 subpopulation is always lower because the scheduling of individuals is applied, and as each device calls its kernel into chunks of C_S individuals, greater overhead is produced, as it was analyzed in Table 5.7.

On the other hand, Figure 5.11 shows some energy measures of executions seen in Figure 5.10b. It is a common mistake to think that using more resources to compute leads to more energy consumption, although it could be true if the application is not parallelized in the right way, of course. This is demonstrated in Figure 5.11a, which shows that sequential execution is the one that causes the highest energy consumption and implicitly also the longest execution time, as shown in Figure 5.11b. Certainly, although using more resources increases the instantaneous power (Figure 5.11b), it is remembered that energy consumption depends not only on instantaneous power but also on execution time, and hence why the best result is obtained for the configurations that allow to reduce computation time. Finally, using CPU and GPU simultaneously produces the lowest energy consumption except in the case of 1 subpopulation. The reason is the same as the one mentioned above: the overload caused by the scheduling of individuals that keeps the devices involved in idle state for longer.



(a) *Energy consumption when the total number of individuals is divided into multiple subpopulations*

(b) *Instantaneous power when using only 1 subpopulation*

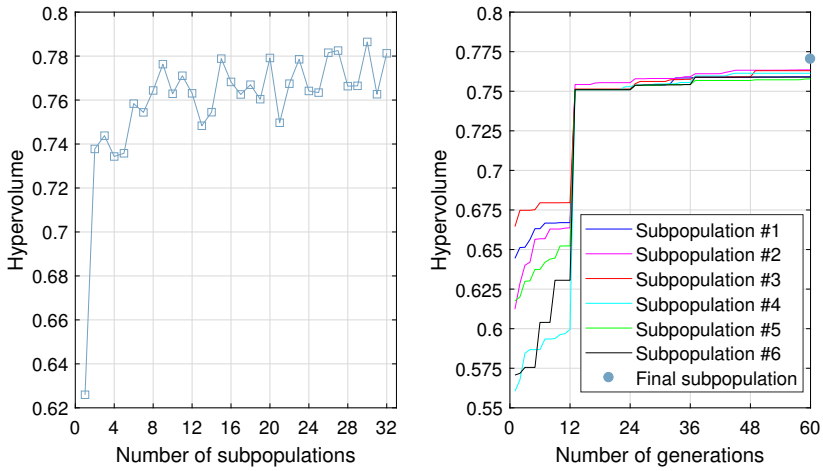
Figure 5.11: Energy measures of *MPGA* for different platform configurations without performing *migrations*.

5.3.2 *Hypervolume Comparison*

Since several parallel and efficient versions are already available at this point, it is time to evaluate the quality of the solutions obtained from the best of them. The reason why hypervolume has not been compared between previous parallel versions[6] is because *PGA*, *OPGA*, and *MDGA* obtain the same hypervolume as in *SGA* since they are based on a master-worker approach that does not alter the behavior of the GA but simply distribute the workload in several ways. However, in *MPGA*, the introduction of the *migration* process and the independent evolution of each subpopulation cause a change in the evolutionary behavior that should have an impact on the hypervolume. Figure 5.12 shows the hypervolume obtained after using multiple subpopulations with and without *migrations*.

Experiment 5.7: *Compare the hypervolume of MDGA and MPGA when MPGA increases the number of migrations and the total number of individuals is divided into multiple subpopulations. The experimental conditions are described in Table 5.8.*

6 The *MATLAB* implementation is excluded due to its poor performance and limitations



(a) *Using 1 to 32 subpopulations without performing migrations*

(b) *Using 6 subpopulations and 5 migrations*

Figure 5.12: Hypervolume obtained in *MPGA* when the total number of individuals is divided into multiple subpopulations.

Figure 5.12a is focused on analyzing the effect of dividing $N = 480$ individuals into 1 to 32 subpopulations, so that there are no *migrations* between subpopulations. The maximum value, $hv = 0.786$, is reached for $N_{Sp} = 30$, and the lowest corresponds to the case in which the 480 individuals compose a single subpopulation, i.e., the same behavior as executing *MDGA*. When more subpopulations are used, the general tendency of hypervolume is to increase, although not progressively but with ups and downs. Since the total number of individuals is the same for any number of subpopulations, the fact that the hypervolume increases demonstrates that evolving subpopulations separately and merging them in the end is the best option.

Moreover, the figure reveals a detail that should not be ignored: regardless of the number of subpopulations. Why is hypervolume worse than that obtained in *SGA* (Figure 5.1a), even if individuals evolve along 60 generations instead of 50? This experiment differs from Experiment 5.1 in that the latter used only the first 480 features of the dataset due to the high computational cost and *MDGA* and *MPGA* use all (3,600). To clarify this, both versions have been executed with only 480 features and the same experimental conditions as those used in *SGA*. The result: *MDGA* and *SGA* get the same hypervolume (as expected) and *MPGA* exceeds *SGA* and *MDGA* when using multiple subpopulations (0.853). Therefore, a plausible explanation could be that as the search space is larger, the GA has more difficulty to find better solutions.

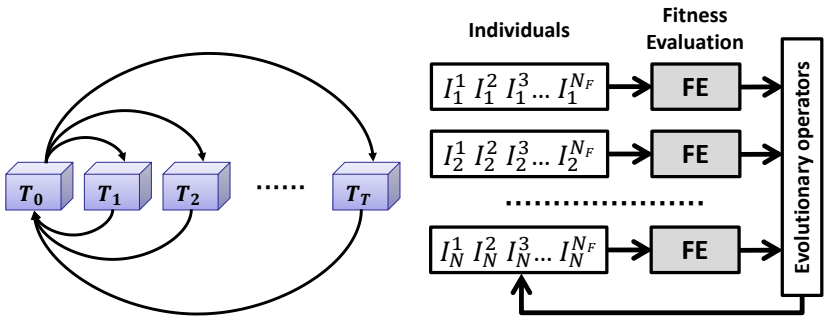
Figure 5.12b checks the effect of using 5 *migrations* for the case in which only 6 subpopulations evolve because including more subpopulations in the graph (lines) can be confusing. As the process runs for 60 generations, *migrations* occur in generations 12, 24, 36, 48, and 60. The final hypervolume is 0.77, a little better than that obtained in Figure 5.12 without *migrations* (0.759). However, what stands out in the figure is the behavior after each *migration*, where a small jump in quality is obtained in almost all subpopulations. Particularly, after the first *migration*, the hypervolume increases greatly due to each subpopulation exchanges many individuals (half of its *Pareto front*). The negative part is that in the rest of generations the subpopulations hardly explore new solutions. The reason is that the *Pareto fronts* of the subpopulations are quite similar. Although it has been proven to reduce the number of exchanges between populations, the hypervolume obtained is always smaller, so despite the behavior observed in Figure 5.12b, it has been decided to keep that scheme.

5.4 ENERGY-TIME MODELING FOR WORKLOAD BALANCING

Due to the availability of mechanisms such as DVFS and heterogeneous systems with different power consumption profiles, it is possible to devise scheduling algorithms for parallel applications aware of both execution time and energy consumption. This section proposes and evaluates a bi-objective cost function to optimize the workload distribution among the CPU and GPU CUs. The objective is to reduce energy consumption without increasing execution time or to reach a trade-off between them. The energy-time model considered correspond to version *MPGA*, although for simplicity only one population of individuals is evolved. The cost function is determined by a multiple linear regression model that requires to assign weights to energy and time objectives [134]. Moreover, the model is built from the experimental energy-time values following the approach described in [35].

5.4.1 *The Bi-objective Cost Function*

The scheduling strategy proposed has to take into account information about execution time and energy consumption. These two objectives usually correspond to opposite goals, since improving execution time could imply the increase of instantaneous power. An alternative to solve this problem is to use a *Pareto*-based approach that searches a



(a) *Example graph* (b) *Wrapper approach presented in this thesis*

Figure 5.13: Task dependence graph.

set of non-dominated solutions among which the user selects the most appropriate one for the given situation. Nevertheless, due to the high execution time required by this *Pareto*-based approach, a cost function whose minimum corresponds to the desired trade-off among time and energy should be implemented. Of course, this trade-off is set offline by manually setting the parameter values of the cost function.

In a sense, the *wrapper* approach of the procedures implemented in this thesis could be interpreted as a set of T_T tasks whose dependence graph is shown in Figure 5.13a. In this graph, tasks $T_1, \ldots, T_T$ can be executed in parallel after task T_0, which are repeated cyclically until the end of the execution. Moreover, the execution time of task T_0 is negligible with respect to that of parallel tasks. Many useful applications can be parallelized according to the dependence graph of Figure 5.13a. Given a task i with a workload equal to W_i clock cycles that is assigned to a device j running at frequency f, it is possible to define two deviations $\Delta t(W_i, f_j)$ and $\Delta E(W_i, f_j)$ with values in the interval $(0, 1)$:

$$\Delta t(W_i, f_j) = \frac{t(W_i, f_j) - t(W_{min}, f_{max})}{t(W_{max}, f_{min}) - t(W_{min}, f_{max})} \tag{5.5}$$

$$\Delta E(W_i, f_j) = \frac{E(W_i, f_j) - E(W_{min}, f_{min})}{E(W_{max}, f_{max}) - E(W_{min}, f_{min})} \tag{5.6}$$

Both equations are related to the relative deviations of the estimated time and energy, $t(W_i, f_j)$ and $E(W_i, f_j)$, respectively. The *min max* configurations in combination with C and f determine the situations of maximum and minimum workload and frequency in which a task could be. E.g., $t(W_{min}, f_{max})$ is the time required by the task with the lowest

workload in a device running at the highest frequency. In the same way, $t(W_{max}, f_{min})$ is the time required by the task with the highest workload in a device running at the lowest frequency. The same logic is applied to energy. Thus, to select a device and the corresponding frequency for a given task, the scheduling algorithm could use a cost function that takes into account both energy and time objectives through deviations $\Delta t(W_i, f_j)$ and $\Delta E(W_i, f_j)$. The bi-objective cost function is defined as:

$$\Delta(W_i, f_j) = a \cdot \Delta t(W_i, f_j) + b \cdot \Delta E(W_i, f_j) \tag{5.7}$$

with positive coefficients a and b verifying that $a + b = 1$. Depending on the values of a and b, it is possible to give more relevance to the optimization of time or energy. Parameters $t(W_i, f_j)$ and $E(W_i, f_j)$ also allow the determination of the maximum and minimum values for time and energy: $t(W_{max}, f_{min})$, $E(W_{max}, f_{max})$, $t(W_{min}, f_{max})$, and $E(W_{min}, f_{min})$. Different approaches have estimated these parameters to quantify the energy-time behavior of an application in a given computing platform. As in [49] and many other works, the energy model can be estimated from equations of instantaneous power corresponding to CMOS circuits, which include the terms capacitance, short-circuit, and leakage power. Assuming the capacitance term as the most significant, the instantaneous power P in a device j can be estimated as:

CMOS: Complementary Metal–Oxide Semiconductor

$$P_j = \alpha \cdot f_j \cdot (V_j)^2 \tag{5.8}$$

where V_j is the supply voltage and parameter α the product of the number of transistors switching in the device per clock cycle and the total capacitance load. On the other hand, the execution time of task i is calculated dividing its workload W_i by the device frequency, f_j:

$$t_j = \frac{W_i}{f_j} \tag{5.9}$$

As energy consumption E depends on instantaneous power and execution time, its value for a time t_j can be defined as follows:

$$\begin{aligned} E_j &= P_j \cdot t_j \\ &= \alpha \cdot f_j \cdot \frac{W_i}{f_j} \cdot (V_j)^2 \\ &= \alpha \cdot W_i \cdot (V_j)^2 \end{aligned} \tag{5.10}$$

Whenever a device j is in (I)dle state, there is a term called indirect energy consumption, which can be estimated as:

$$E_j^I = \alpha \cdot f_j \cdot t_j^I \cdot \left(V_j^I\right)^2 \tag{5.11}$$

being V_j^I the supply voltage of the device in idle state for a time t_j^I. A device j can operate at different SVLs, $V_{j,l}$; $\forall l = 1, \ldots, V_L$, which corresponds to different clock frequencies, $f_{j,l}$. Therefore, using those terms and substituting, the deviations of Equations (5.5) and (5.6) can be redefined as follows:

SVL: Supply Voltage Level

$$\Delta t(W_i, f_{j,l}) = \frac{\frac{W_i}{f_{j,l}} - t(W_{min}, f_{max})}{t(W_{max}, f_{min}) - t(W_{min}, f_{max})} \tag{5.12}$$

$$\Delta E(W_i, f_{j,l}) = \frac{\alpha \cdot W_i \cdot \left(V_{j,l}\right)^2 - E(W_{min}, f_{min})}{E(W_{max}, f_{max}) - E(W_{min}, f_{min})} \tag{5.13}$$

In Equation (5.12), parameters $t(W_{max}, f_{min})$ and $t(W_{min}, f_{max})$ can be obtained from the frequencies of the available devices, $f_{j,l}$, and from the highest and lowest W_i values because they are known in advance. In Equation (5.13), the energy consumption while the devices are idle is not explicitly shown to prevent unnecessary complexities in the mathematical expressions. Taking into account that, the following parameters can be evaluated:

$$t(W_{max}, f_{min}) = \frac{max\,(W_i)}{min\left(f_{j,l}\right)} \tag{5.14}$$

$$t(W_{max}, f_{max}) = \frac{max\,(W_i)}{max\left(f_{j,l}\right)} \tag{5.15}$$

$$t(W_{min}, f_{min}) = \frac{min\,(W_i)}{min\left(f_{j,l}\right)} \tag{5.16}$$

$$t(W_{min}, f_{max}) = \frac{min\,(W_i)}{max\left(f_{j,l}\right)} \tag{5.17}$$

verifying that $t(W_{min}, f_{max}) < t(W_{max}, f_{max}) < t(W_{max}, f_{min})$ and also that $t(W_{min}, f_{max}) < t(W_{min}, f_{min}) < t(W_{max}, f_{min})$. Moreover, it is also possible to define the corresponding energy consumption parameters:

$$E(W_{max}, f_{min}) = \alpha \cdot max\,(W_i) \cdot min\left(V_{j,l}\right)^2 \tag{5.18}$$

$$E(W_{max}, f_{max}) = \alpha \cdot max\,(W_i) \cdot max\left(V_{j,l}\right)^2 \tag{5.19}$$

$$E(W_{min}, f_{min}) = \alpha \cdot min\,(W_i) \cdot min\left(V_{j,l}\right)^2 \tag{5.20}$$

$$E(W_{min}, f_{max}) = \alpha \cdot min\,(W_i) \cdot max\left(V_{j,l}\right)^2 \tag{5.21}$$

which verify that $E(W_{min}, f_{min}) < E(W_{max}, f_{min}) < E(W_{max}, f_{max})$ and also that $E(W_{min}, f_{min}) < E(W_{min}, f_{max}) < E(W_{max}, f_{max})$.

5.4.2 *Energy-aware Procedure for Task Scheduling*

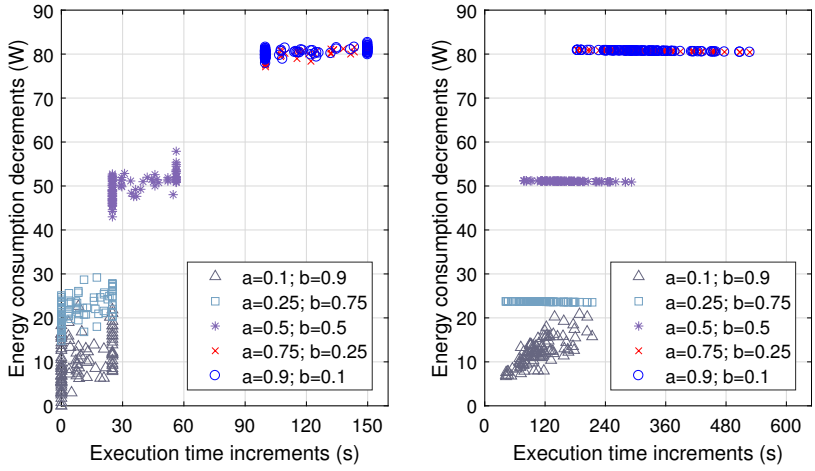
In what follows, a scheduling procedure that assigns tasks to devices and frequencies with the objective of minimize both execution time and energy consumption is described. The procedure takes advantage of the proposed bi-objective cost function seen in Section 5.4.1, which joins the deviations of Equations (5.12) and (5.13). The assignment of tasks is carried out taking into account the available energy-time models and the characteristics of the devices where the tasks are executed. Among these characteristics, the possible frequencies at which the devices can run and the availability of changing frequencies and voltages either by the operating system or the user. Algorithm 5.4 provides the pseudocode of the proposed scheduling procedure. Given a set of T_T tasks with workloads C, the procedure sorts the tasks according to their workloads, verifying that $W_i \leq W_{i+1}$ and therefore prioritizing the execution of the lightest tasks. The frequencies of the N_D devices among which the tasks have to be distributed are also sorted according to their values to obtain f_{min} and f_{max}.

Algorithm 5.4 proceeds as follows. For the first task W_1 in the sorted list of tasks, the cost function $\Delta(W_1, f_{j,l}) = a \cdot \Delta t(W_1, f_{j,l}) + b \cdot \Delta E(W_1, f_{j,l})$ is evaluated in order to assign this task to all possible frequencies in the available devices. The task is assigned to device and frequency for which the lowest value of $\Delta(W_1, f_{j,l})$ is obtained. Then, the device and its operating frequencies are marked as selected and the procedure continues with the next task. The energy-aware scheduling procedure could be implemented either in the runtime system or in the application, depending on whether the changes in the device's frequency are done at system or user level. E.g., it could be implemented inside the application code whenever DVFS is available at user level.

Algorithm 5.4: Pseudocode of the energy-aware procedure for task scheduling.

Function `EnergyAwareProcedure`(T, N_D, W, FL, a, b)
  Input : Number of tasks, T
  Input : Number of devices, N_D
  Input : Set W: workloads of the tasks
  Input : Matrix FL: frequency levels of each device
  Input : Coefficient a, related to execution time
  Input : Coefficient b, related to energy consumption
  $W \leftarrow$ Sort verifying $W_i \leq W_{i+1}$
  W_{max} and $W_{min} \leftarrow \texttt{max}\,(W)$ and $\texttt{min}\,(W)$
  $FL \leftarrow$ Sort by device j, FL_j
  repeat
    `// Select device and frequency to assign` W_i
    repeat
      if device j has not been previously selected **then**
        $F_L = \texttt{size}\left(FL_j\right)$
        f_{max} and $f_{min} \leftarrow \texttt{max}\left(FL_j\right)$ and $\texttt{min}\left(FL_j\right)$
        $t(W_{max}, f_{min}) \leftarrow$ Equation (5.14)
        $t(W_{min}, f_{max}) \leftarrow$ Equation (5.17)
        $E(W_{max}, f_{min}) \leftarrow$ Equation (5.18)
        $E(W_{min}, f_{max}) \leftarrow$ Equation (5.21)
        repeat
          $\Delta\left(W_i, f_{j,l}\right) = a \cdot \Delta t\left(W_i, f_{j,l}\right) + b \cdot \Delta E\left(W_i, f_{j,l}\right)$
        until all $f = F_L$ frequency levels are considered;
      end
    until all $j = N_D$ devices are evaluated;
    $W_i \rightarrow$ Assign to j and f that get the minimum $\Delta\left(W_i, f_{j,l}\right)$
    $j \leftarrow$ Mark device as selected
  until all $i = T$ tasks are assigned;
End

Moreover, the procedure can also be implemented in heterogeneous systems that include devices with different energy consumption profiles, i.e., different operating frequencies or voltages. Given a set of T_T tasks to be distributed among N_D devices, each with F_L possible frequencies,



(a) *Tasks assigned randomly* (b) *Minimum execution time*

Figure 5.14: Time increments and energy decrements with respect to different scheduling procedures.

the computational complexity of the procedure is $\mathcal{O}(T_T \cdot N_D \cdot F_L)$. I.e., the program explores all combinations to obtain the global optimum. However, for very large sizes of T_T, N_D, and F_L, this approach would require an unsuitable computational cost, so that optimization methods such as EAs are required to find a trade-off between time and energy.

On the other hand, the procedure considers that the number of available devices, N_D, matches with the number of tasks, T_T. Nevertheless, the procedure could be used in case of having more tasks than devices, $T_T > N_D$. If this were the case, the remaining $W_R = T_T - N_D$ tasks can be successively assigned once a device j completes the current task. In the same way as for the first N_D tasks, the cost function $\Delta(W_R, f_{j,l})$ is evaluated. The frequency f of device j for which the lowest value of $\Delta(W_R, f_{j,l})$ is obtained is the one used.

Figure 5.14 provides information about the increments in execution time and decrements in energy consumption obtained by simulating Algorithm 5.4. The simulation uses 100 different random task configurations with computing costs in the range of 100 and 3,000. A heterogeneous configuration with devices running at different sets of relative speeds has been considered (see Table 5.9). These speeds are relative with respect to the one achieved at the frequency of 1 GHz (100% in P5, P6, P7, and P8). An idle frequency of 100 MHz has been considered and the pairs of simulated coefficients (a, b) are $(0.1, 0.9)$, $(0.25, 0.75)$, $(0.5, 0.5)$, $(0.75, 0.25)$, and $(0.9, 0.1)$.

Table 5.9: Relative speeds (%) used in the devices for the simulation of task scheduling seen in Algorithm 5.4.

Devices	P1	P2	P3	P4	P5	P6	P7	P8
	80	80	80	80	100	100	100	100
Relative speed (%)	64	64	64	64	80	80	80	80
	40	40	40	40	50	50	50	50

Figure 5.14a compares the scheduling obtained by Algorithm 5.4 with a random assignments of tasks to devices running at their highest frequencies. Figure 5.14b shows the increments with respect to a scheduling that provides the minimum execution time. Figure 5.14 demonstrates that, when coefficient b increases (and thus a consequently decreases), the decrements in energy consumption with respect to the reference scheduling are larger at the cost of increasing execution time.

5.4.3 *A Scheduling Model for CPU-GPU Platforms*

This section deals with the case of a platform including only two different kinds of devices, CPU and GPU, among which it is necessary to distribute a set of tasks. This allows a specific way for applying the proposed bi-objective cost function $\Delta(W_i, f_j)$ described in Section 5.4.1. As previously mentioned, the model to be developed is applied to version *MPGA* but with only one population, so that individuals are distributed among the CPU and GPU CUs. Therefore, as the evaluation of individuals is the heaviest task, the workload scheduling is reduced to determine the rate of individuals assigned to GPU and CPU, x and $1 - x$. A model for execution time could be:

$$t = g \cdot \left[N \cdot t_M + max \left(\left\lceil \frac{x \cdot N}{\lambda_G} \right\rceil \cdot t_G, \left\lceil \frac{(1-x) \cdot N}{\lambda_C} \right\rceil \cdot t_C \right) \right] \tag{5.22}$$

where g is the number of generations, N the number of individuals, and λ_C and λ_G are the number of CPU and GPU CUs, respectively. Parameter t_M corresponds to the time required by the master thread to process task T_0 (Figure 5.13a) at each iteration, while parameters t_C and t_G are the time required by CPU and GPU to evaluate one individual, respectively. Parameters t_M, t_C, and t_G can also be expressed according to the workload and frequency of the corresponding device as:

$$t_M = \frac{W_M}{f_C} \tag{5.23}$$

$$t_C = \frac{W_C}{f_C} \tag{5.24}$$

$$t_G = \frac{W_G}{f_G} \tag{5.25}$$

where f_C and f_G are the frequencies of the CPU and GPU CUs, respectively. W_M, W_C, and W_G are, in this order, the estimation of the cycles for workloads of task T_0, the evaluation of an individual in CPU, and the evaluation of an individual in GPU. If all these terms are substituted in Equation (5.22), the execution time can be modeled as:

$$t = g \cdot \left\lceil \frac{N \cdot W_M}{f_C} + max\left(\left\lceil \frac{x \cdot N}{\lambda_G} \right\rceil \cdot \frac{W_G}{f_G}, \left\lceil \frac{(1-x) \cdot N}{\lambda_C} \right\rceil \cdot \frac{W_C}{f_C} \right) \right\rceil \tag{5.26}$$

A consideration has to be highlighted regarding to the model of Equation (5.26). It has been supposed that the time required to evaluate one individual, t_C and t_G, is the same for all individuals. This circumstance is unusual for other applications, but the MOGA approach of version *MPGA* can be suitably modeled in this way because the number of *K-means* iterations is fixed. Taking that into account, it is possible to fit two linear regressions considering the values of the workload distribution, x, verifying:

$$\left\lceil \frac{x \cdot N}{\lambda_G} \right\rceil \cdot \frac{W_G}{f_G} \leq \left\lceil \frac{(1-x) \cdot N}{\lambda_C} \right\rceil \cdot \frac{W_C}{f_C} \tag{5.27}$$

$$\left\lceil \frac{x \cdot N}{\lambda_G} \right\rceil \cdot \frac{W_G}{f_G} > \left\lceil \frac{(1-x) \cdot N}{\lambda_C} \right\rceil \cdot \frac{W_C}{f_C} \tag{5.28}$$

As the model is based on linear regression, the expressions obtained match with the equation of a line in a 2D space, i.e., $y = z + mx$, where m and z are the slope and y-intercept of the line, respectively:

$$t_{left} = t_{left\_0} + t_{left\_1} \cdot \left\lceil \frac{(1-x) \cdot N}{\lambda_C} \right\rceil \tag{5.29}$$

$$t_{right} = t_{right\_0} + t_{right\_1} \cdot \left\lceil \frac{x \cdot N}{\lambda_G} \right\rceil \tag{5.30}$$

being t_{left} the execution time in case of low values of x, where more workload is assigned to CPU and thus GPU ends its workload earlier. Following this scheme, t_{right} is the execution time for high values of x, where CPU finishes its workload earlier. Thus, for a given platform, from the experimental values of t_{left} and t_{right} when using different x values, it is possible to determine t_C, t_G, and t_M since N, λ_C, and λ_G are known. In addition, the values of W_C, W_G, and W_M can be determined because f_C and f_G are also known.

In what follows, an approximate model for energy consumption is described. Since more terms are involved in the energy model than in the time one, some of them will be grouped into new terms to simplify. In this way, the terms R_{λ_C} and R_{λ_G} are defined as the ratio of individuals assigned to a single CU of CPU and GPU, respectively.

$$R_{\lambda_C} = \frac{(1-x)\cdot N}{\lambda_C} \tag{5.31}$$

$$R_{\lambda_G} = \frac{x \cdot N}{\lambda_G} \tag{5.32}$$

Given a device j with λ_j CUs running at frequency f_j, the energy consumption during the evaluation of its individuals can be expressed as the product of instantaneous power and execution time of each CU plus the energy consumption of the CUs that are idle. Thus, the energy consumption of CPU and GPU in each generation g can be defined as:

$$\begin{aligned} E_C &= P_C \cdot \lfloor R_{\lambda_C} \rfloor \cdot t_C + \frac{P_C}{\lambda_C} \cdot \left[(1-x)\cdot N - \lfloor R_{\lambda_C} \rfloor \cdot \lambda_C\right] \cdot t_C + E_C^I \\ &= P_C \cdot R_{\lambda_C} \cdot t_C + E_C^I \\ &= P_C \cdot R_{\lambda_C} \cdot \frac{W_C}{f_C} + E_C^I \\ &= \frac{P_C \cdot R_{\lambda_C} \cdot W_C}{f_C} + E_C^I \end{aligned} \tag{5.33}$$

$$\begin{aligned} E_G &= P_G \cdot \lfloor R_{\lambda_G} \rfloor \cdot t_G + \frac{P_G}{\lambda_G} \left(x \cdot N - \lfloor R_{\lambda_G} \rfloor \cdot \lambda_G\right) \cdot t_G + E_G^I \\ &= P_G \cdot R_{\lambda_G} \cdot t_G + E_G^I \\ &= P_G \cdot R_{\lambda_G} \cdot \frac{W_G}{f_G} + E_G^I \\ &= \frac{P_G \cdot R_{\lambda_G} \cdot W_G}{f_G} + E_G^I \end{aligned} \tag{5.34}$$

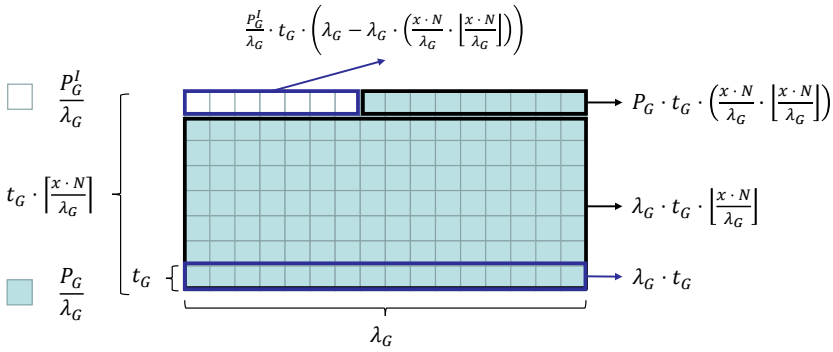


Figure 5.15: GPU energy model expressions.

where P_G is the instantaneous power when the GPU CUs are evaluating individuals, and E_G^I is the energy consumption of the idle CUs. P_C and E_C^I are the analogous terms for CPU. The rationale for the previous Equations (5.33) and (5.34) can be understood from Figure 5.15, which refers to GPU but can also be applied to CPU (substituting x by $1 - x$).

In the figure, the squares in the horizontal dimension corresponds to the number of CUs (λ_G) while the squares in the vertical dimension corresponds to the times that the CUs work in parallel once the $x \cdot N$ individuals are distributed among them. A blue square indicates that the CU works with an instantaneous power of $\frac{P_G}{\lambda_G}$. On the contrary, a white square corresponds to the instantaneous power when the CU are in idle state, $\frac{P_G^I}{\lambda_G}$. Probably, in the last generation not all available CUs work. Taking into account the previous expressions, the total energy consumption along g generations is:

$$\begin{aligned} E &= g \cdot [E_G + E_C + \epsilon] \\ &= g \cdot \left[\frac{P_G \cdot R_{\lambda_G} \cdot W_G}{f_G} + \frac{P_C \cdot R_{\lambda_C} \cdot W_C}{f_C} + E_C^I + E_G^I + \epsilon \right] \end{aligned} \tag{5.35}$$

The last term, ϵ, corresponds to the energy consumed by task T_0 and other elements of the platform such as memory, buses, and I/O. Models for E_C^I and E_G^I can also be obtained from the characteristics of the platform and workload distribution as:

$$\begin{aligned} E_C^I &= P_C^I \cdot (1 - R_{\lambda_C} + \lfloor R_{\lambda_C} \rfloor) \cdot \frac{W_C}{f_C} \\ &= \frac{P_C^I \cdot (1 - R_{\lambda_C} + \lfloor R_{\lambda_C} \rfloor) \cdot W_C}{f_C} \end{aligned} \tag{5.36}$$

$$\begin{aligned} E_G^I &= P_G^I \cdot (1 - R_{\lambda_G} + \lfloor R_{\lambda_G} \rfloor) \cdot \frac{W_G}{f_G} \\ &= \frac{P_G^I \cdot (1 - R_{\lambda_G} + \lfloor R_{\lambda_G} \rfloor) \cdot W_G}{f_G} \end{aligned} \tag{5.37}$$

Summarizing, the parameters of the energy-time models seen in Equations (5.26) to (5.30) and (5.33) to (5.37) could be determined from experiments considering different workload rates (x), number of individuals (N), generations, (g), CUs (λ_C and λ_G), and frequencies (f_C and f_G, respectively). By fitting Equation (5.26) with experimental time measures, it would be possible to obtain parameters W_M, W_C, and W_G. Once these values are substituted in Equations (5.35) to (5.37), the experimental values of energy consumption can be used to determine P_C, P_G, P_C^I, P_G^I, and ϵ after fitting the energy model of Equation (5.35). Thus, according to Equations (5.35) to (5.37) the multiple linear regression model presents the following terms:

$$E = A_0 + A_1 \cdot R_{\lambda_G} + A_2 \cdot \lfloor R_{\lambda_G} \rfloor + A_3 \cdot \lfloor R_{\lambda_C} \rfloor + \epsilon \tag{5.38}$$

where coefficients A_0, A_1, A_2, and A_3 can be related to the parameters of the model from Equations (5.35) to (5.37) as:

$$A_0 = g \cdot \left[P_G^I \cdot \frac{W_G}{f_G} + P_C^I \cdot \frac{W_C}{f_C} \cdot \left(1 - \frac{N}{\lambda_C}\right) + P_C \cdot \frac{N}{\lambda_C} \cdot \frac{W_C}{f_C} \right] \tag{5.39}$$

$$A_1 = g \cdot \left[\left(P_G - P_G^I\right) \cdot \frac{W_G}{f_G} - \left(P_C - P_C^I\right) \cdot \frac{W_C}{f_C} \cdot \frac{\lambda_G}{\lambda_C} \right] \tag{5.40}$$

$$A_2 = g \cdot \frac{W_G}{f_G} \cdot P_G^I \tag{5.41}$$

$$A_3 = g \cdot \frac{W_C}{f_C} \cdot P_C^I \tag{5.42}$$

With all this, it is possible to build the cost function for unknown values of N, g, λ_C, λ_G, etc. This approach for static workload scheduling is depicted in Figure 5.16 and experimentally analyzed in Section 5.4.4. The flow is as follows: multiple experiments with different workload distributions are executed in CPU and GPU at a certain operating frequency. From the experimental results, the energy-time model, and the linear regression, the necessary parameters are obtained to generate the bi-objective cost function, which determines the best static workload distribution for the devices present in the heterogeneous platform.

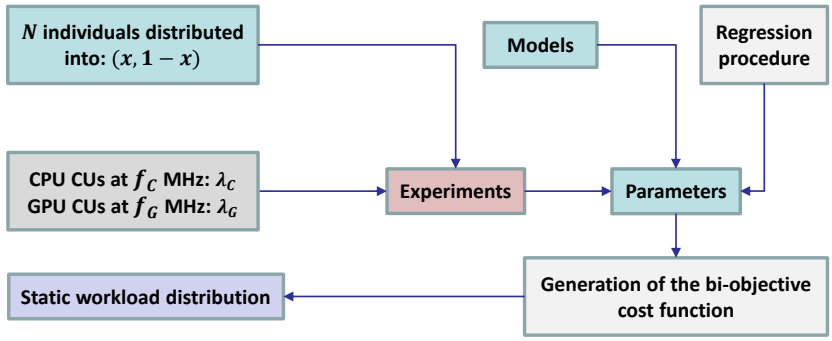


Figure 5.16: Scheme of the static workload distribution procedure for CPU-GPU platforms.

5.4.4 *Experimental Analysis of the Model*

This section analyzes the model quality by running version *MPGA* under different experimental conditions. The frequency of the CPU cores is modified using the *Unix* `cpupower` command, which needs superuser privileges for proper operation. Figures 5.17 and 5.18 provide the time and energy measures, respectively, and curve fitting using the model and multiple linear regression.

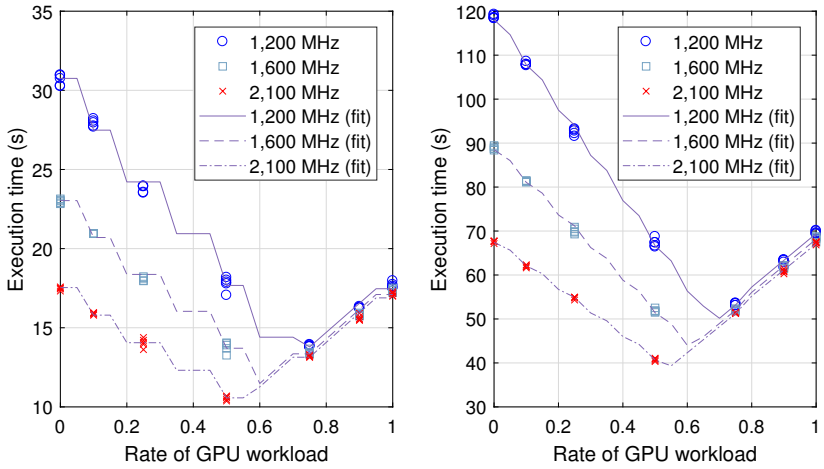
Experiment 5.8: *Evaluate how the model fits the experimental results and determine the optimal workload value for CPU and GPU when different numbers of individuals and CPU frequencies are used. The experimental conditions are described in Table 5.10.*

Looking at Figures 5.17 and 5.18, it can be seen that the accuracy of curve fitting is acceptable. In particular, the minimum of the fitted curves corresponds to that experimentally observed. Indeed, it has been proven that the R-squared (R^2)[7] values are closed to 1 in all alternatives. In addition, p-values and error standard deviations demonstrate that the proposed energy-time model is statistically significant. From the experimental energy measures, it is clear that the curves do not evolve linearly with the rate of GPU workload (x) and therefore ϵ in Equation (5.35) does not either. The rest of terms in that equation are linear with x or, as can be seen in Equations (5.33) and (5.34), bounded by curves that are linear with x.

7 *R*-squared values approaching 1 is better

Table 5.10: Experimental setup to analyze the energy-time model for workload distribution.

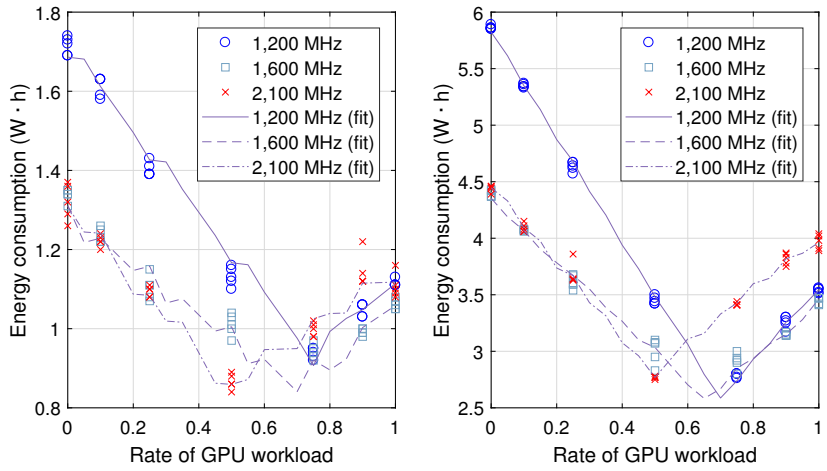
Feature	Description
Platforms	Node 4: CPU and GPU Tesla K40m
CUs / work-items	CPU: all / 1 each
	GPU: all / 1,024 each
Frequency	CPU: {1200, 1600, 2100} MHz
	GPU: maximum MHz
Compiler	GCC with -O2 -funroll-loops
Dataset	All features used
Individuals	{240, 960}
Generations	50



(a) $N = 240$ *individuals* (b) $N = 960$ *individuals*

Figure 5.17: Curve fitting of experimental time measures for different GPU workload rates (x), CPU frequencies, and population sizes, N.

The energy consumption by the application depends on the power dissipated by the CPU and GPU CUs, other on-chip elements such as memory and buses, and the remaining components of the system. It is possible to assume that the power dissipated by all these elements is constant once the computer has executed some workload for a time [45]. Moreover, it seems that ϵ behaves as a constant term up to a certain value of x, from which the GPU memory transfers via PCIe bus implies



(a) $N = 240$ *individuals* (b) $N = 960$ *individuals*

Figure 5.18: Curve fitting of experimental energy measures for different GPU workload rates (x), CPU frequencies, and population sizes, N.

more power consumption. From this value of x, the experimental results seems to correspond to linear increments in ϵ when x grows. It has to be taken into account that, although the time required by memory accesses and bus transfers can be overlapped with CPU or GPU processing without impacting on time, these elements consume energy that is added to the values obtained. Therefore, ϵ is modeled as:

$$\epsilon = \epsilon_0 \cdot (x - x_c) \cdot \left\lfloor \frac{x}{x_c} \right\rfloor \tag{5.43}$$

The value of x_c in Equation (5.43) can be obtained by two linear regressions. The first one is applied to values of x lower than x_c and close to 0, corresponding to workloads much higher in CPU CUs than in GPU. On the contrary, the second linear regression is applied to values of x higher than x_c and close to 1, where GPU has assigned more workload than CPU. The crossing point between both lines estimates the value of x_c. Table 5.11 provides the estimated values of x_c for different values of N and f_C to obtain the minimum execution times.

Table 5.12 provides some parameters of the energy-time model once they are fitted to the experimental data of Figures 5.17 and 5.18. The values obtained, $W_M = 0.46 \cdot 10^6$, $W_C = 78.48 \cdot 10^6$, and $W_G = 13.69 \cdot 10^6$ cycles allow to solve Equation (5.26) and thus the possibility of predicting the curves for the values of f_C and N used. The values provided in

Table 5.11: Estimated values of x_c for different population sizes (N) and CPU frequencies to obtain the minimum execution time, t_{min}.

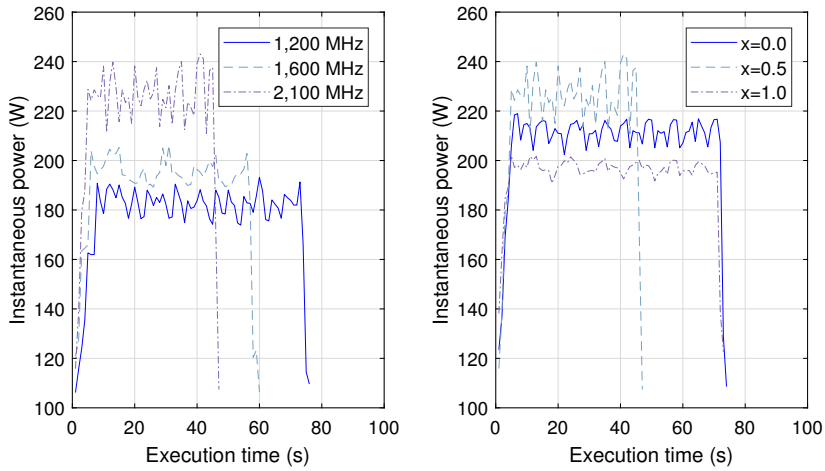
N	f_C (MHz)	$x_c \pm 0.05$	$x\,(t_{min}) \pm 0.05$
240	1,200	0.75	0.75
	1,600	0.75	0.60
	2,100	0.50	0.50
960	1,200	0.70	0.70
	1,600	0.65	0.60
	2,100	0.50	0.55

Table 5.12: Parameters of the energy-time model fitted by multiple linear regression. W_C, W_G, and W_M values are given in (C)ycles $\cdot 10^6$.

N	f_C (MHz)	W_C (C)	W_G (C)	W_M (C)	$\frac{P_C}{\lambda_C}$ (W)	$\frac{P_G}{\lambda_G}$ (W)	$\frac{P_C^I}{\lambda_C}$ (W)	$\frac{P_G^I}{\lambda_G}$ (W)	ϵ_0 (W · h)
240	1,200	78.48	13.79	0.46	7.96	9.62	1.41	2.48	1.82
	1,600	74.62	14.05	0.58	8.23	12.11	1.53	4.46	1.13
	2,100	73.16	14.06	0.63	11.70	6.74	1.37	2.05	1.57
960	1,200	82.30	15.22	0.38	6.44	4.39	2.78	1.38	8.07
	1,600	79.09	15.13	0.48	6.44	6.17	2.99	2.35	4.82
	2,100	74.96	14.88	0.61	9.52	4.44	3.41	1.43	5.75

Table 5.12 for parameters W_C and W_G present standard deviations of only 5.2% and 3.5%, respectively, compared to their respective mean values. This circumstance seems to corroborate the assumption presented in Section 5.4.3 indicating that a similar number of cycles is required to evaluate all W_G individuals in GPU and all W_C in CPU.

Moreover, from the experimental energy values after executing the application it is possible to determine P_C, P_G, P_C^I, P_G^I, and ϵ_0 from the multiple linear regression of Equation (5.38), in which the term ϵ is substituted by its definition given in Equation (5.43). As Table 5.12 shows, these parameters depend on the operating frequency of CPU. The values of P_C are similar for 1,200 and 1,600 MHz but higher for 2,100 MHz in all population sizes. Regarding P_G, the values in all



(a) *Using different CPU frequencies* (b) *Using different GPU workload rates (x)*

Figure 5.19: Instantaneous power of *MPGA* for some f_C and x values using 1 subpopulation with $N = 960$ individuals.

population sizes are higher for 1,600 MHz than for 1,200 and 2,100 MHz, which present similar values. The values of P_C range from 1.4 to 3.7 times higher than those of P_G. With respect to the parameters related to energy consumption, the experimental results of Table 5.11 shows that the values of x_c slightly decrease when the CPU frequency grows and almost do not change with the population size. Also, for each population size, ϵ_0 decreases from 1,200 to 1,600 MHz and slightly increases from 1,600 to 2,100 MHz, as shown in Table 5.12. For each CPU frequency, ϵ_0 clearly grows with the population size, N.

Figure 5.19 draws the temporal evolution of the instantaneous power. On the one hand, Figure 5.19a uses different CPU frequencies (f_C) and the same workload distribution for CPU and GPU ($x = 0.5$). On the other hand, Figure 5.19b uses $f_C = 2,100$ MHz and different workload rates. From Figure 5.19a, it is clear that the instantaneous power grows with f_C and the value for 1,600 MHz is closer to that for 1,200 MHz than for 2,100 MHz. Moreover, it seems that execution time is reduced proportionally to the increase in instantaneous power. Figure 5.19b shows that the instantaneous power also changes with the GPU workload rate. The line with the lowest value corresponds to the situation in which all individuals are evaluated in GPU ($x = 1.0$). This scenario reveals that GPU is a little more efficient than CPU since the required execution time is the same but energy consumption is lower. Finally, the higher instantaneous power is obtained when $x = 0.5$ because both devices are computing at the same time.

5.5 CONCLUSIONS

Throughout this chapter, eight sets of experiments have been carried out to analyze five of the versions implemented under different experimental conditions. Also, to complement the analysis, an energy-time model has been designed capable of identifying and explaining the behavior of the latest version developed so far, *MPGA*, although its application is limited to the use of a single population of individuals.

Experiment 5.1 has shown that choosing a lower level programming language allows to reduce execution time without having to parallelize the application. Although programming with *MATLAB* is fast and intuitive, using *C++* is worth not only for its great performance but also for its versatility and flexibility in managing hardware resources.

Experiment 5.2 has analyzed the first parallel version, which uses *C++* for the host code and the *OpenCL* API to develop the CPU and GPU kernels. The parallelization is based on a *round-robin* strategy in which individuals are distributed statically among CPU or GPU CUs. The results have shown that both devices accelerate the application several orders of magnitude with respect to the sequential version, with CPU being the device that gets the best speedup mainly due to its large number of CUs and the performance per CU. Although appreciable speedups are provided by GPU, its performance is poor compared to the peak performance that its architecture can provide. The same happens in CPU, but to a lesser extent.

For that reason, version *OPGA* has mainly focused on improving the performance of the GPU kernel by reducing memory usage and applying the *coalescing* technique to optimize memory access. The analysis carried out in Experiment 5.3, which compares *OPGA* and the non-optimized version, has shown an improvement in the GPU speedup so that its performance per CU already exceeds that achieved by CPU.

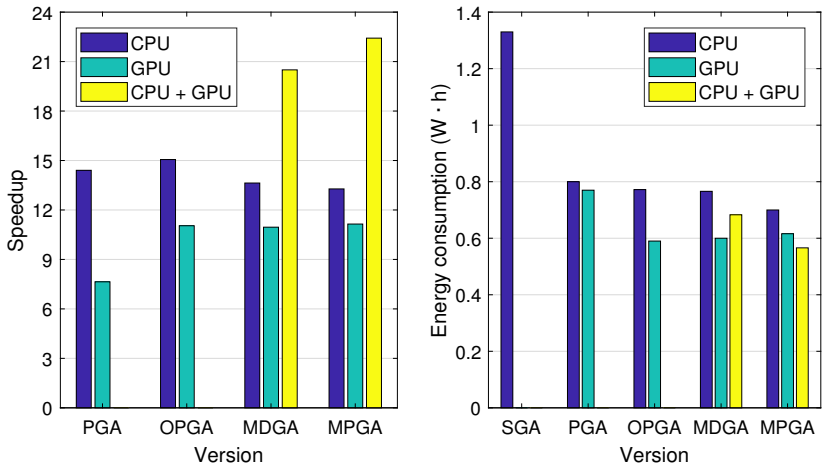
However, in general terms, GPU is outperformed since it has fewer CUs. The analysis has also revealed that using more CUs than GPU can handle simultaneously provides extra performance by hiding latencies. The effect of using *coalescing* has been revealed not only by the increase in speedup, but also by the fact that the non-optimized version gets worse performance when the number of *work-items* increases too much ($1,024$ in this case because the *work-items* are distributed in a single dimension, or range). Despite all optimizations applied, the synchronization barriers and the required memory transfers between host and GPU via PCIe bus, among others, still affect the final execution time.

To increase the application performance, a master-worker scheduler has been developed in version *MDGA* to dynamically distribute the evaluation of individuals by simultaneously launching CPU and GPU kernels. According to the results shown in Experiments 5.4 and 5.5, through this scheduler the speedup obtained is almost equivalent to the sum of the speedup that the devices obtain separately. Therefore, the small loss of performance is conditioned by the times that the *OpenCL* kernels are called, since a call to the kernel implies a certain overhead caused by memory copies, parameter initialization, etc., as seen in Table 5.7. From all combinations of devices evaluated, the best result has been obtained for the combination that includes all devices, demonstrating the efficiency of the scheduler.

One thing to highlight is the speedup behavior when scaling the number of CPU CUs. For a certain range, the speedup reached is worse than using as many CUs as the number of physical cores the CPU has. This, at least, in the case of CPUs with *Hyper-Threading*. Another conclusion that can be obtained is that it is not a good idea to use all CPU CUs in the kernel if CPU must also manage one or more GPUs. As has been specified in Equation (5.3), the optimal number of CUs should be less than the total number of CPU CUs and equal to the chunk of individuals to be evaluated to avoid workload imbalance.

Taking into account that the kernels have been improved and the program already allows to take advantage of all available devices, the difficulty to optimize more is high. To continue improving, version *MPGA* extends the current GA to a multi-population one where the subpopulations are dynamically distributed among the devices, providing up to three levels of parallelism in GPU. As shown in Experiment 5.6, the speedup improves markedly with respect to version *MDGA* when more than two subpopulations and all devices are used. In addition, the use of *migrations* seems not to affect performance.

The figures also show that energy consumption depends on execution time. Therefore, the lowest energy values are also obtained for the situation in which all devices compute even though its instantaneous power is greater. A performance comparison of all versions can be seen in Figure 5.20, whose caption shows the experimental conditions. Regarding hypervolume, as seen in Experiment 5.1, the main factor that affects its final value is the total number of individuals followed by the number of generations. Moreover, in Experiment 5.7 of *MPGA*, the use of *migrations* and the division of individuals into subpopulations provides better results than those achieved by previous versions, reaching maximums of $hv = 0.853$ and 0.786 depending on whether all features of the dataset are used or only first 480, respectively.



(a) *Speedup with respect to version SGA* (b) *Energy consumption*

Figure 5.20: Maximum speedup and energy consumption of the versions presented in this chapter when using 50 generations, 960 individuals, and Node 2 of the cluster. *MPGA* evolves two subpopulations and does not perform *migrations*.

Finally, in Section 5.4 a bi-objective cost function $\Delta = a \cdot \Delta t + b \cdot \Delta E$ is proposed to determine the best workload distribution according to the information of execution time and energy consumption. It also allows the prediction of time and energy for a given workload distribution and computing platform once a suitable model is available. The cost function has been evaluated by simulation and is designed for those cases in which the heterogeneous architecture include DVFS mechanisms that allow different power consumption profiles for the devices.

In Experiment 5.8, the application is executed in a heterogeneous CPU-GPU platform at different CPU frequencies. Moreover, by multiple linear regression, the energy-time model has been fitted to the experimental results with good accuracy and statistical significance. The experiments have shown that by using adequate values for coefficients a and b it is possible to get relevant energy-savings without important increases in execution time. As energy consumption is the product of instantaneous power and execution time, lower energy values can be obtained once the right workload distribution is considered even using higher CPU frequencies. The experiments demonstrate that, for some applications like the MOGA here evaluated, the effect of programming strategies that take into account energy-time profiles is directly observable in the behavior of the computing platforms, which opens the possibility of analyzing, modeling, and optimizing the application.

6

DEVELOPMENT ON DISTRIBUTED SYSTEMS

CONTENTS

Distributed computing systems arise in response to the large number of applications that make use of intensive computing. Although the performance of single-computer systems continues to increase, it does not improve at the same rate as the computing requirements. The depletion of Moore's law, the high frequencies that microprocessors already reach, or the difficulty in continuing to reduce lithography are some of the causes that make single-computer systems not viable for many applications. In Chapter 5, the developed versions already exploited the capabilities that a heterogeneous computer can offer: parallelism in CPU and GPU, optimization of kernels using techniques such as *coalescing*, and the distribution of workload between both devices. The evolutionary scheme was also changed to introduce the multi-population MOGA, providing a slight improvement in performance and hypervolume at the cost of an important algorithmic change. In short, a situation has been reached in which the effort to scratch more performance is not worth it unless the computing paradigm changes.

Thus, in this chapter the application is extended to distributed systems to continue increasing performance. These systems offer new computational paradigms and therefore new energy profiles that must be taken into account when designing new versions. The procedures implemented correspond to versions *DGA*, *DGA-II*, *ODGA*, *DNN*, and *GAAM*, which following the methodology of Chapter 5 are evaluated by eight sets of experiments with different experimental conditions.

The chapter is organized as follows: Section 6.1 describes and analyzes the implementations based on the master-worker scheme. This includes the first distributed version, another one in which the *OpenCL* CPU kernel is replaced by an *OpenMP*-based implementation, and a new version that optimizes the workload distribution and whose skeleton is reused in Section 6.1.4 to distribute CNNs among the *cluster* nodes. On the other hand, a multi-population approach with asynchronous *migrations* is proposed in Section 6.2 as an alternative to the master-worker approach implemented in previous versions. Finally, Section 6.3 shows the conclusions after finishing the experimentation.

6.1 IMPLEMENTATIONS BASED ON A MASTER-WORKER SCHEME

Although having several nodes in the *cluster* provides different parallelization profiles, their computing devices and the characteristics of the application will determine the approach to be used. In distributed systems, multi-population EAs can be treated mainly in two ways: distribute the evaluation of subpopulations following the iterative master-worker approach used in the previous procedures, or distribute all subpopulations among the nodes before starting the evolutionary process. In this chapter, both alternatives are studied. This section tackles the master-worker approach and leaves the other for Section 6.2.

6.1.1 *The First Distributed Version*

The first distributed version that makes use of the *cluster* is called *DGA*, and is presented as an evolution of version *MPGA*. The procedure is developed with the MPI library for communication between nodes, which entails major changes in the source code and therefore also increases its complexity. So, if it has been previously commented that changing the paradigm is partially motivated by the difficulty of continuing to improve, why use an even more complex paradigm now? The

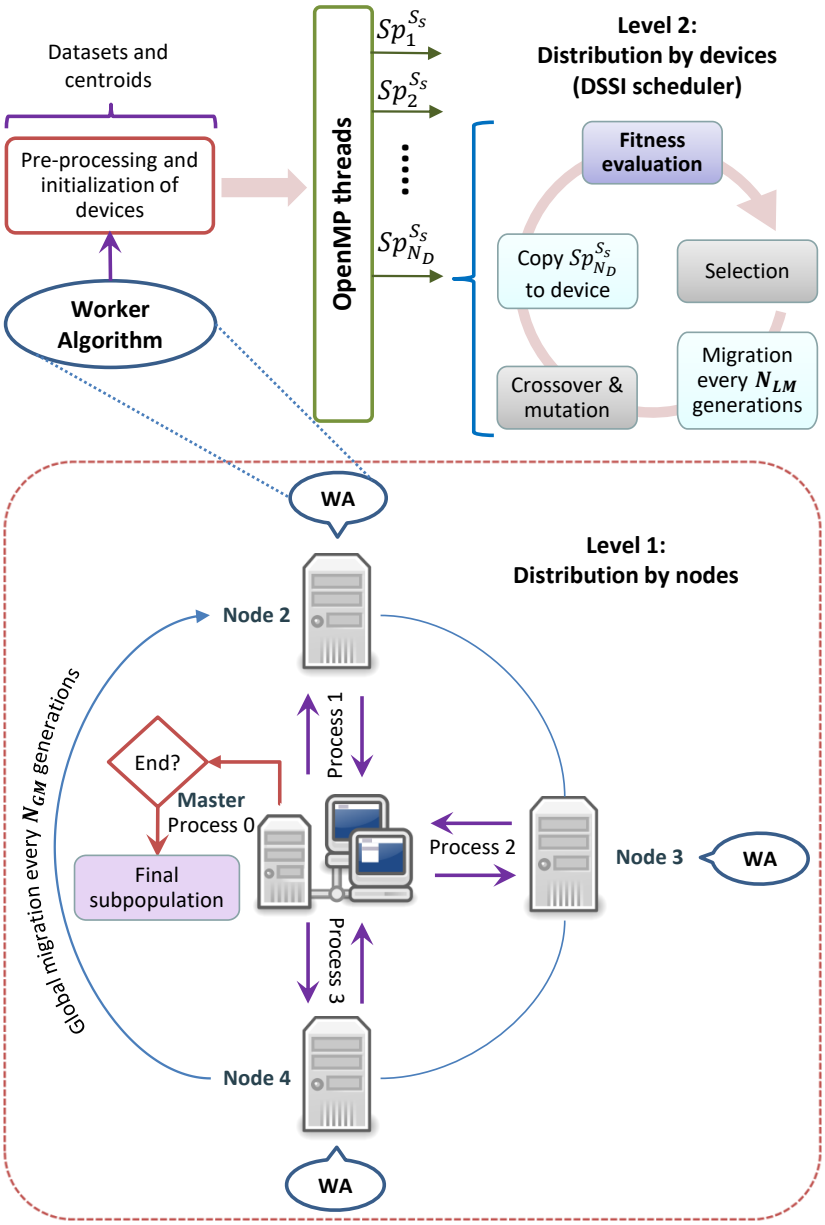


Figure 6.1: MPI-*OpenMP* scheme defined in *DGA*, which constitutes the first and second parallelism levels.

answer lies in the great potential in terms of performance offered by a multicomputer system, making the effort worthwhile. A well-designed source code would allow the program to easily scale according to the number of nodes or computers that the *cluster* has. In case of *MPGA*,

the program could be improved, e.g., using the SIMD model to take advantage of vectorization in certain *OpenCL* kernel operations, but its impact on performance would be slight. However, it all depends on the requirements of the application. In the hypothetical case where the maximum performance is needed all possibilities must be explored.

6.1.1.1 *A New Parallelism Level*

The main advantage of dividing parallelism into multiple hierarchical levels, or layers, is that each of them can be optimized independently. *DGA* adds a new parallelism level in addition to the three levels already offered by version *MPGA*, becoming the first level of parallelism. Therefore, the new hierarchy is as follows:

1. **First level**: distribution of subpopulations among *cluster* nodes.
2. **Second level**: distribution of subpopulations/individuals among *OpenCL* devices.
3. **Third level**: distribution of individuals among CUs.
4. **Fourth level**: GPU data parallelism in *K-means*.

Figure 6.1 depicts how the first level links with the second, starting with the initial distribution of subpopulations in the master node, and ending with the *fitness* evaluation of each individual in the worker nodes. The second, third, and fourth levels correspond to the first, second, and third levels of *MPGA*, which were already shown in Figure 5.9. In the first level, the master distributes subpopulations among the worker nodes through MPI functions. Each node is assigned an MPI process, but the master is always the process with rank #0 and runs on Node 1. To minimize workload imbalance, the subpopulations are distributed dynamically. Observing Algorithm 6.1, the master obtains dataset *DS*, from which it extracts the centroids, and initializes the individuals of all subpopulations (Lines 2 to 4). At this point, master and workers are ready to start communicating.

Firstly, the master broadcasts the centroids to all workers (Line 5), which are necessary to perform *K-means*. Then, the dynamic distribution of subpopulations and the *migrations* are repeated as many times as N_{GM} global *migrations* have been defined (Lines 6 to 17). The master asynchronously begins to attend the requests of each worker and distributes subpopulations until there is no more work to do (Lines 8 to 15). The worker can make two kinds of requests: ask for new subpopulations or return those already evolved. This situation is handled by the master in the `if` conditional of Line 10. In the first case, the master checks if

Algorithm 6.1: Master pseudocode defined in *DGA* to distribute subpopulations among all worker nodes.

```
1  Function MasterDGA(N_Sp, S_S, M, N_GM, N_Wk)
       Input  : Number of subpopulations to be evolved, N_Sp
       Input  : Subpopulation size, S_S
       Input  : Number of objectives, M
       Input  : Number of global migrations, N_GM
       Input  : Number of workers, N_Wk
       Output : Sp, the subpopulations already evolved
2      DS ← getDataset()
3      k ← getRandomCentroids(DS)
4      Sp ← initSubpopulations(N_Sp, S_S, M, DS)
       // Start MPI section
5      MPI::Bcast(k) → To all N_Wk workers
6      for i ← 1 to N_GM global migrations do
           // Dynamic distribution of subpopulations
7          RemainingWork ← N_Sp
8          repeat
9              MPI::Recv(Sp) ← Request from worker Wk_n
10             if worker Wk_n requests subpopulations then
11                 sent ← min(RemainingWork, Requested)
12                 MPI::Send(Sp, sent) → To Wk_n
13                 RemainingWork ← RemainingWork − sent
14             end
15         until all N_Sp subpopulations are evaluated;
16         Sp ← globalMigration(Sp, N_Sp, S_S, M)
17     end
       // End MPI section
18     MPI::Bcast(FINISH) → To all N_Wk workers
19     Sp ← subpopulationMerging(Sp, N_Sp, S_S, M)
20     return Sp
21 End
```

there are still subpopulations to evolve. If so, it sends the worker as many subpopulations as indicated in the request, or less if there are not enough. When all subpopulations have evolved, in Line 16 the master performs a global *migration* between subpopulations in the same way as in *MPGA*. Once all N_{GM} global *migrations* have been completed, the master broadcasts the *FINISH* signal to the workers to notify that

Algorithm 6.2: Worker pseudocode defined in *DGA* to evaluate the subpopulations received from the master.

```
1  Function Worker(S_S, M, N_LM, N_D)
       Input: Subpopulation size, S_S
       Input: Number of objectives, M
       Input: Number of local migrations, N_LM
       Input: Number of devices, N_D
2      DS ← getDataset()
       // Start MPI section
3      MPI::Bcast(k) ← Centroids from the master
4      D ← initDevices(DS, k, N_D)
5      repeat
           // Ask for as many subpopulations as N_D devices
6          MPI::Send(N_D) → Request subpopulations to the master
7          MPI::Recv(Sp) ← rcv subpopulations
8          repeat
9              #pragma omp parallel for num_threads(rcv)
10             for j ← 1 to rcv do
11                 Sp_j ← evolve(Sp_j, S_S, M, D, N_D)
12             end
13         end
14         Sp ← localMigration(Sp, rcv, S_S, PF)
15     until all N_LM local migrations are completed;
16     MPI::Isend(Sp, rcv) → Return rcv subpopulations
17 until the FINISH signal is received;
18 End
```

there are no more subpopulations to be evolved and therefore the MPI communications have ended (Line 18). Then, the master merges all subpopulations in Line 19 to perform the final subpopulation, which is returned to the main function. While the master process schedules the distribution of subpopulations, the workers (Algorithm 6.2) perform all evolutionary steps of each subpopulation. Previously, the worker obtains the dataset, receives the centroids from the master, and initializes the *OpenCL* devices (Lines 2 to 4). Then, after starting the MPI communications with the master, in Line 6 the worker requests the master as many subpopulations as N_D devices are present in the node. However, the number of subpopulations received, *rcv*, could be lower than N_D if there are not enough subpopulations available.

The *rcv* subpopulations received by the worker are evolved in parallel in Lines 9 to 13. Here, the operation is the same as in *MPGA*, i.e., as many *OpenMP* threads as subpopulations are created. Depending on the value of *rcv*, the following may occur: if $rcv = N_D$, the DSSI scheduler will distribute one subpopulation per device. Also if $1 < rcv < N_D$, but in this case $N_D - rcv$ devices will not work. On the contrary, if $rcv = 1$, the DSSI scheduler will distribute the individuals of that subpopulation among all N_D devices. As a reminder, this constitutes the second parallelism level because the DSSI scheduler assigns workload to all devices regardless of the number of subpopulations received.

On the other hand, the worker also performs N_{LM} local *migrations* in the same way the master performs a global *migration*, but between the *rcv* subpopulations received (Line 14). Once the whole process is finished, the worker returns to the master the *rcv* subpopulations already evolved (Line 16). Now, it awaits the assignment of more work or the *FINISH* signal, which means that all N_{GM} global *migrations* have been carried out by the master and therefore the worker can finish.

6.1.1.2 *Hypervolume and Speedup Analysis*

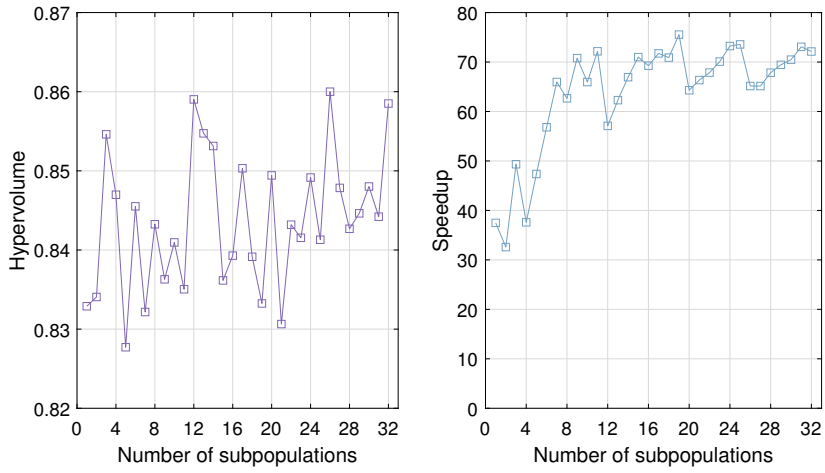
> **Experiment 6.1:** *Evaluate the hypervolume and speedup of DGA when using all cluster nodes and the total number of individuals is divided into multiple subpopulations. The experimental conditions are described in Table 6.1.*

Figure 6.2a shows the hypervolume obtained in *DGA* when 3,840 individuals are divided into multiple subpopulations. The reason for evolving 3,840 individuals in this experiment is to ensure that all nodes have enough workload to correctly evaluate the performance of the new version. What can be seen in the figure is that all values are very similar and are in the range of 0.827 to 0.86, the latter being reached for $N_{Sp} = 26$. Although there seems to be a slight increase from 21 subpopulations, the Kruskal-Wallis test indicates that there is no statistical significance.

Despite this, the important thing is that in all cases the hypervolume is higher than the two maxima reached by *MPGA*, that is, 0.786 and 0.853 depending on whether *MPGA* used 480 or 3,600 dataset features, respectively. As a trend in the hypervolume variation is not appreciated when the number of subpopulations changes, the improvement could

Table 6.1: Experimental setup to analyze *DGA*.

Feature	Description
Platforms	All cluster nodes and desktop PC
Devices	Cluster: all CPUs and Tesla GPUs
	PC: only GPUs
Compiler	GCC with -O2 -funroll-loops
Dataset	All features used
Individuals	3,840
Subpopulations	$\{1, \ldots, 32\}$
Generations	150
Migrations	Global: 5
	Local: 15



(a) *Hypervolume* (b) *Speedup*

Figure 6.2: Hypervolume and speedup obtained in *DGA* when using all *cluster* nodes and the total number of individuals is divided into multiple subpopulations.

be motivated by one or more of the following factors: (i) the increase in the number of individuals per subpopulation; (ii) the increase in the number of generations, and (iii) the local *migrations*. Global *migrations* are not considered because they are equivalent to the *migrations* performed by *MPGA*, which do not alter performance. Summarizing what was seen in the previous chapter about hypervolume:

- Increasing the number of individuals directly affects its value.
- From 20-30 generations the hypervolume tends to stabilize.
- A *migration* rapidly increases the value of the hypervolume, but limits its progression during the rest of the generations.

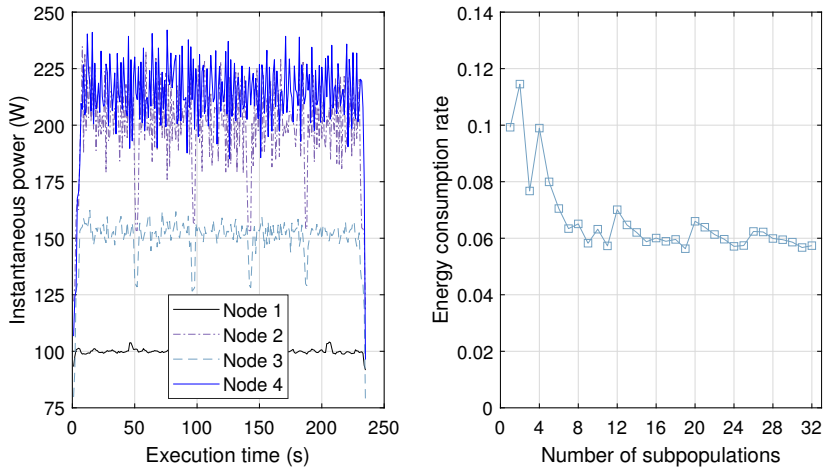
Thus, the cause is probably the number of individuals. To solve the dilemma, *DGA* has been executed with only 60 generations and also without local *migrations*. The result is that hypervolume is the same when removing local *migrations* and hardly decreases when reducing generations. Therefore, it can be concluded that the hypervolume improvement is due to the increase in the total number of individuals.

On the other hand, Figure 6.2b shows the speedup when using all *cluster* nodes. In general, the speedup is higher when the number of subpopulations increases, reaching several peaks above 70. The anomalous behavior observed from 1 to 5 subpopulations is related to the way in which the subpopulations are assigned to the nodes: as there are two *OpenCL* devices working on each node, only one worker node computes when evolving 1 or 2 subpopulations, so that the lower speedups are obtained. In case of 3 and 4 subpopulations, two workers compute. From 5 on, the three worker nodes compute but a better speedup is achieved for 6 subpopulations since this number coincides with the total number of devices. As all *cluster* nodes work, a total of 115 CUs plus the master CPU thread of Node 1 are computing. The increase in speedup shown for more than 6 subpopulations can be explained by taking into account that when the number of subpopulations is increased, more distribution and less workload imbalance occurs.

6.1.1.3 *Energy-time Behavior*

> **Experiment 6.2:** *Evaluate the energy-time behavior of DGA when using all cluster nodes, the desktop PC, and the total number of individuals is divided into multiple subpopulations. The experimental conditions are also described in Table 6.1.*

Figure 6.3a provides the evolution of the instantaneous power of all *cluster* nodes when evolving 32 subpopulations. The measures of the switch are not included in the figures as their values are very low. Due to the heterogeneity of the nodes, the instantaneous power of each one is different, being Node 1 the one that shows the lower values because it only uses a CPU thread to distribute the subpopulations. Another



(a) *Instantaneous power*

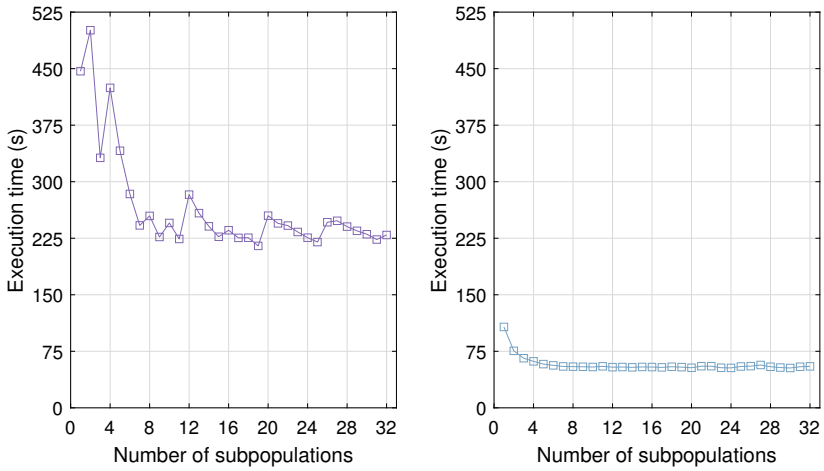
(b) *Rate of energy consumption with respect to a sequential execution of DGA*

Figure 6.3: Energy measures obtained in *DGA* when using all *cluster* nodes and the total number of individuals is divided into multiple subpopulations.

highlight is related to the five instantaneous power drops that occur for several seconds. These falls are a consequence of performing the global *migrations*, which also require communication between nodes. During the *migration* process, the entire workload falls on the master, so the worker nodes do not compute in the meantime.

With respect to energy consumption, Figure 6.3b shows the energy rate necessary by an execution that uses all nodes with respect to a sequential execution of *DGA*. The shape of the lines matches with that shown in Figure 6.4a since the shorter execution time, the lower energy consumption. In fact, the best value of speedup and energy is reached when $N_{Sp} = 19$ subpopulations. In that situation, the energy rate is below 6% and therefore an important energy-saving is achieved. Even in the worst case (2 subpopulations), the energy rate is about 12%. Thus, from Figures 6.2b and 6.3b, it is clear that parallel processing constitutes a valuable alternative not only to reduce execution time but also to decrease energy consumption.

Although *DGA* is designed to obtain the best performance in multi-computer systems, its use is not limited to these platforms. For its correct operation, only two MPI processes are necessary: one that act as a master and another that act as a worker. Therefore, *DGA* could run on a single node of the *cluster* as well as on the desktop PC. The



(a) *Using all cluster nodes* (b) *Using the desktop PC and only the GPUs*

Figure 6.4: Execution time obtained in *DGA* when using different platform configurations and the total number of individuals is divided into multiple subpopulations.

execution time obtained by both platforms is provided in Figure 6.4. For any number of subpopulations, the PC obtains better results even with a smaller number of CUs (60). This occurs for two reasons: the first one, the differences between GPU architectures. The GPUs included in the *cluster* are based on *Kepler*[1] architecture, while those of the PC are based on *Pascal*[1], which is newer and incorporates technological improvements. E.g., the clock frequency of both PEs and memory is much higher, the latter being especially beneficial since the kernel makes intensive use of memory. The second reason is that the PC offers lower workload imbalance. Its configuration for this experiment corresponds more to that of a homogeneous system than a heterogeneous one since both GPUs are the same and CPU is not used to evaluate individuals. As a result, shorter and more stable execution times are achieved. This can be seen in Figure 6.4b, which shows small fluctuations with few subpopulations and a full time stabilization from 6 subpopulations on.

Despite the good performance of the PC, its GPUs also do not offer maximum performance due to the irregular *K-means* characteristics, which were discussed in Chapter 5. In fact, although the TDP of each GPU is about 250 W, using the `nvidia-smi`[2] command it has been observed that the instantaneous power ranges between 135 and 158 W when they are computing, so GPUs are not pushed to the limit.

1 The *Kepler* and *Pascal* architectures date from 2012 and 2016, respectively
2 Command line utility that allows the management and monitoring of *NVIDIA* GPUs

Algorithm 6.3: *OpenMP K-means* pseudocode defined in *DGA-II* to evaluate a chunk of individuals in CPU.

```
1  Function KmeansCPU(I, N_I, DS, k, N_T)
       Input  : Individuals, I
       Input  : Chunk of individuals to be evaluated, N_I
       Input  : Dataset DS: N_P points (samples) with N_F features
       Input  : Set k of K centroids randomly chosen from DS
       Input  : Number of CPU threads, N_T
       Output : Individuals already evaluated, I
2      #pragma omp parallel for num_threads(N_T)
3      for i ← 1 to N_I individuals do
4          kC ← Create a copy of the centroids
5          Initialization of the mapping table, MT ← 0
6          repeat
7              for j ← 1 to N_P points do
8                  MT_j ← Point p_j in DS is assigned to a cluster
9                  ND_j ← Store the distance for point p_j
10             end
11             kC ← Update centroids using dataset DS
12         until stop criterion is not reached;
13         f_1(I_i) ← wcss(kC, ND) according to Equation (4.1)
14         f_2(I_i) ← bcss(kC) according to Equation (4.2)
15     end
16 end
17 return I
18 End
```

6.1.2 *Use of OpenMP to Evaluate Individuals in CPU*

In the versions developed so far, *OpenCL* has been used to build the CPU and GPU kernels. However, the version *DGA-II* proposed in this section replaces the *OpenCL* CPU kernel with the *OpenMP* pseudocode shown in Algorithm 6.3. Its implementation is the same as that of Algorithm 5.1 but it adds two components: (i) the input parameter N_T indicating the number of CPU threads to use, and (ii) the `pragma` necessary to create the parallel region in order to distribute the individuals among the threads. Parameter N_T replaces the term λ_C previously used since the *OpenMP* terminology does not contemplate the term CU.

Depending on the programming language or library, the nomenclature to refer to CPU and GPU PEs is different. In case of *OpenCL*, the right terms are CU and *work-item*. However, as *DGA-II* implements the evaluation of individuals in CPU with *OpenMP* instead of *OpenCL*, for this device the terms CU and *work-item* will be replaced by the terms core and thread, respectively.

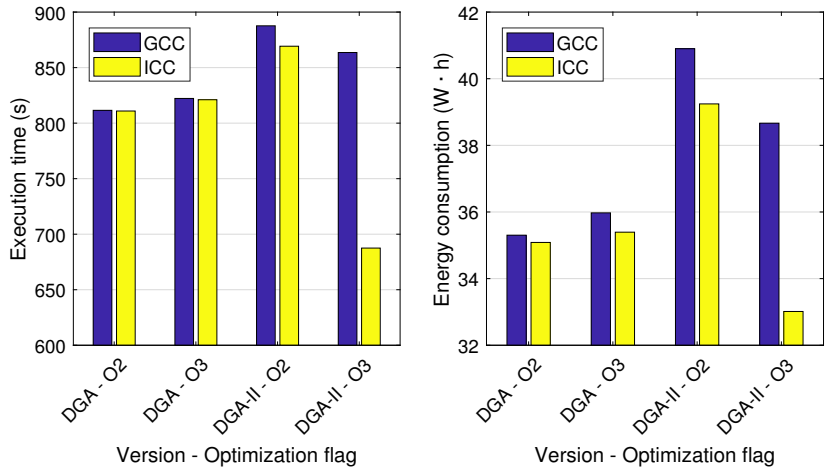
Replacing *OpenCL* with *OpenMP* is not a random decision. Its use is motivated by the ease of implementation, since a single `#pragma omp parallel for` is enough to distribute the loop that iterates over the array of individuals. Moreover, *OpenMP* could offer better performance. Although this can lead to confusion, it should be noted that the *OpenCL* kernels are compiled at runtime by the *OpenCL* driver while the *OpenMP* code is compiled by the corresponding *C++* compiler. Although the *OpenCL* kernels can also be compiled offline, for simplicity they have been compiled at runtime. This circumstance, along with the compiler used and the possible optimizations that each one is able to achieve, can make the difference between choosing one option or the other. To get rid of doubts, the following experiment carries out a small comparison between *DGA* and *DGA-II*.

Experiment 6.3: *Compare the execution time and energy consumption of DGA and DGA-II when different compilers and optimization flags are used. The experimental conditions are described in Table 6.2.*

Figure 6.5 provides the execution time and energy consumption of *DGA* and *DGA-II* when using GCC and ICC compilers, and optimization flags `-O2` and `-O3`. Only CPU is used in the experiment because the compilers act exclusively on the host code, so that GPUs are disconnected. Focusing first on GCC compiler, the figure shows that *DGA-II* does not outperform *DGA*. The CPU driver seems to apply better optimizations on the *OpenCL* code than GCC is able to apply on the *OpenMP* code. Even the use of optimization level `-O3` is not enough to achieve the performance of *DGA*. As discussed in Section 4.2, an inexperienced programmer might think that compilation with `-O3` is not possible in heterogeneous systems since the generated binary file would not be executed correctly. However, MPI allows the user to specify a binary file for each process that runs the program, although depending on the application this can lead to many headaches. E.g., in *DGA-II* the order of the processes when launching the application is important since each one is mapped to a node following the order of the host list.

Table 6.2: Experimental setup to compare *DGA* and *DGA-II*.

Feature	Description
Platforms	Node 3
Devices	CPU
Compiler	{GCC, ICC} with -{O2, O3} -funroll-loops
Dataset	All features used
Individuals	3,840
Subpopulations	1
Generations	150



(a) *Execution time* (b) *Energy consumption*

Figure 6.5: Impact of compilers on *DGA* and *DGA-II* when using different compilers and optimization flags.

On the other hand, when using ICC compiler, what is observed is that all results obtained by GCC have been improved or matched. Moreover, *DGA* gets similar execution times since the compiler cannot act on the *OpenCL* code. The difference between both compilers is even greater when the optimization flag `-O3` is used. This means that ICC applies more aggressive optimizations, allowing version *DGA-II* to reduce execution time and energy consumption of *DGA* by approximately 19%. The behavior is as expected taking into account that both CPU and compiler have been designed by the same vendor (*Intel Corporation*). However, as ICC is a proprietary compiler, for the next experiments only GCC will be used.

Algorithm 6.4: Master pseudocode defined in *ODGA* to distribute subpopulations among all worker nodes.

```
1  Function MasterODGA(N_Sp, S_S, M, N_GM, N_Wk, N_D)
       Input  : Number of subpopulations to be evolved, N_Sp
       Input  : Subpopulation size, S_S
       Input  : Number of objectives, M
       Input  : Number of global migrations, N_GM
       Input  : Number of workers, N_Wk
       Input  : Number of devices, N_D
       Output : Sp, the subpopulations already evolved
2      DS ← getDataset()
3      Sp ← initSubpopulations(N_Sp, S_S, M, DS)
4      if N_Wk == 0 then
5          k ← getRandomCentroids(DS)
6          D ← initDevices(DS, k, N_D)
7          for i ← 1 to N_GM global migrations do
8              Sp ← evolve(Sp, N_Sp, S_S, M, D, N_D)
9              Sp ← globalMigration(Sp, N_Sp, S_S, M)
10         end
11     else
12         for i ← 1 to N_GM global migrations do
13             RemainingWork ← N_Sp
14             repeat
15                 MPI::Recv(Sp) ← Request from worker Wk_n
16                 sent ← min(RemainingWork, Requested)
17                 MPI::Isend(Sp, sent) → To Wk_n
18                 RemainingWork ← RemainingWork − sent
19             until each worker has work || RemainingWork == 0;
20             repeat
21                 MPI::Recv(Sp) ← Request from worker Wk_n^j
22                 sent ← min(RemainingWork, 1)
23                 MPI::Send(Sp, sent) → To Wk_n^j
24                 RemainingWork ← RemainingWork − sent
25             until RemainingWork == 0;
26             Sp ← globalMigration(Sp, N_Sp, S_S, M)
27         end
28         MPI::Bcast(FINISH) → To all N_Wk workers
29     end
30     Sp ← subpopulationMerging(Sp, N_Sp, S_S, M)
31     return Sp
32 End
```

6.1.3 *Optimization of Workload Distribution*

DGA offers great performance and is able to use all *cluster* nodes, so the highest level of parallelism is already reached. However, as it is the first distributed version, there is room for improvement. Therefore, this section presents version *ODGA* as an implementation based on optimization and whose pseudocode can be found in Algorithm 6.4. The goal is to minimize the drawbacks of *DGA* by improving aspects such as the workload distribution and communications between nodes.

The first change is related to the capabilities of the master process. As discussed in Section 6.1.1.3, when *DGA* and *DGA-II* are executed on a single node, an MPI process that acts as a master and another as a worker is mandatory. However, logically this presents a problem with the saturation of CPU and memory resources, since each process is mapped to a thread of the same node and therefore less resources are available to evaluate individuals. In addition, other overheads such as the message passing between both processes or their synchronization requirements have to be taken into account. The issue is fixed by giving the master the worker role without the need for more processes (Line 4).

By using an `if-else` statement, the program checks the total number of workers running the application. If the condition becomes true, no workers are available and therefore the master is the responsible for doing the entire job. As the master has the role of the worker, in addition to centroids and datasets, it also has to initialize the devices (Line 6). On the contrary, if the `if-else` statement becomes false, it means that the subpopulations are computed by the workers and the subpopulations must be distributed by the master. This time, the master does not broadcast the centroids since each worker can obtain them from dataset *DS*. Regarding the distribution of subpopulations, two changes have been introduced. In *DGA*, the distribution is carried out with a single loop, which attends requests, identifies their type, and acts accordingly. In *ODGA*, the distribution is done with two loops:

1. **First loop**: the master sends to all nodes a first chunk of subpopulations less than or equal to the one indicated in the request. The purpose is to make all nodes busy as soon as possible to reduce idle states. In addition, as the type of request is restricted to the workload demand, the master can use the `MPI::Isend` asynchronous function for the sending operation since there is no conflict with the `MPI::Recv` function of Line 15. In *DGA*, the reception and sending operations use the same variable, Sp, so the use of asynchronous communications was not possible.

Algorithm 6.5: Worker pseudocode defined in *ODGA* to evaluate the subpopulations received from the master.

```
1  Function Worker(S_S, M, N_D)
       Input: Subpopulation size, S_S
       Input: Number of objectives, M
       Input: Number of devices, N_D
2      DS ← getDataset()
3      k ← getRandomCentroids(DS)
4      D ← initDevices(DS, k, N_D)
5      MPI::Isend(N_D) → Request subpopulations to the master
6      MPI::Recv(Sp) ← rcv subpopulations
7      repeat
           // Create rcv workers, Wk_n^j; ∀j = 1,...,rcv
8          #pragma omp parallel num_threads(rcv)
9              repeat
10                 Sp_j ← evolve(Sp_j, S_S, M, D, N_D)
11                 MPI::Isend(Sp_j, 1) → Return subpopulation
12                 MPI::Recv(Sp_j) ← New subpopulation
13             until no new subpopulation is received in Sp_j;
14         end
15         MPI::Isend(N_D) → Request subpopulations to the master
16         MPI::Recv(Sp) ← rcv subpopulations
17     until the FINISH signal is received;
18 End
```

2. **Second loop**: once all workers are computing, the master waits for new requests. The loop ends when all subpopulations are evolved. As each worker already has work assigned, a new request involves receiving one subpopulation and sending another if available.

Why only one subpopulation? The reason is as follows: in versions *DGA* and *DGA-II*, worker Wk_n does not ask for more work to the master until its *rcv* subpopulations have been evolved. However, this strategy causes workload imbalance because the devices are not homogeneous, being this imbalance even greater when their computing capabilities differ significantly. Normally, the most powerful devices finish their work first, so they remain idle. As a result, an increase in energy consumption and also a reduction in the acceleration of the application. Thus, in *ODGA*, the thread Wk_n^j that handles device j now has the capacity to return its subpopulation directly to the master and ask for

Table 6.3: Experimental setup to compare *DGA*, *DGA-II*, and *ODGA*.

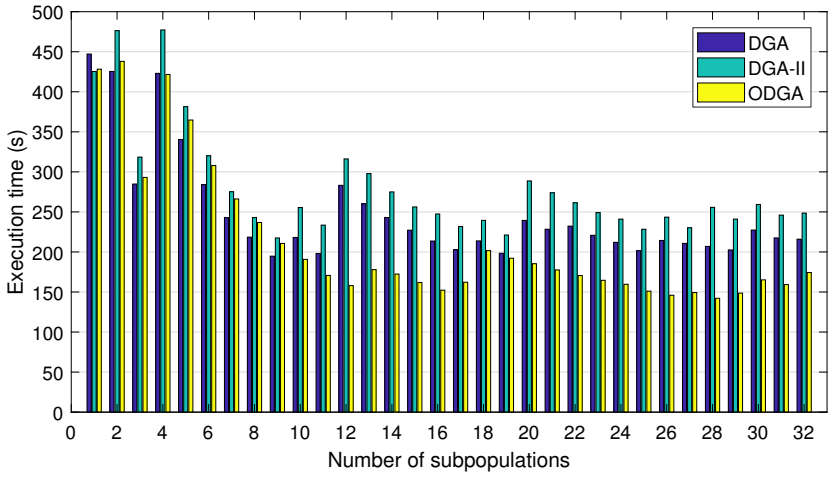
Feature	Description
Platforms	All cluster nodes
Devices	All CPUs and Tesla GPUs
Compiler	GCC with -O2 -funroll-loops
Dataset	All features used
Individuals	3,840
Subpopulations	{1,...,32}
Generations	150
Migrations	Global: 5

a new subpopulation (Lines 11 and 12 of Algorithm 6.5). The cost of introducing this improvement is the elimination of the local *migrations* implemented in version *DGA*, which allowed *migrations* of individuals between the subpopulations of each device. However, as shown in Experiment 6.1, these *migrations* do not provide any improvement in the quality of the hypervolume, so its elimination is not a problem.

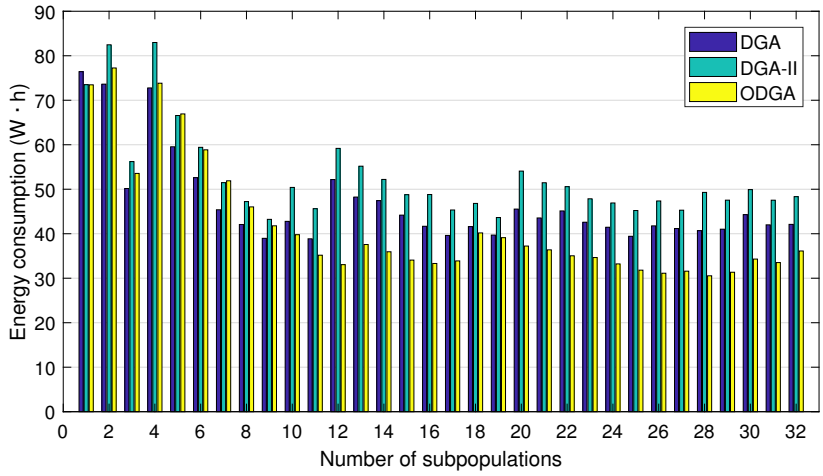
6.1.3.1 *Energy-time Analysis*

Experiment 6.4: *Compare the execution time and energy consumption of DGA, DGA-II, and ODGA when using all cluster nodes and the total number of individuals is divided into multiple subpopulations. The experimental conditions are described in Table 6.3.*

Figure 6.6 provides the execution time and energy consumption of *DGA*, *DGA-II*, and *ODGA* when using all *cluster* nodes. From these figures, it is clear that version *ODGA* provides the best energy-time values from 10 to 32 subpopulations. For lower number of subpopulations, *ODGA* always exceeds *DGA-II* but is outperformed by *DGA* in both time and energy. The reasons is that with few subpopulations, only exist the initial distribution and at most three more, so the effect of the optimizations cannot be appreciated. In addition, as *DGA-II* and *ODGA* use *OpenMP* for the evaluation of individuals in CPU, the execution time is worse than that shown by *DGA* if ICC with `-O3` is not used, as was demonstrated in Experiment 6.3. The best execution time is



(a) *Execution time*



(b) *Energy consumption*

Figure 6.6: Performance obtained in *DGA*, *DGA-II*, and *ODGA* when using all *cluster* nodes and the total number of individuals is divided into multiple subpopulations.

achieved by *ODGA* when using 28 subpopulations. For this number, a sequential execution has been carried out in order to obtain the speedup and the corresponding energy consumption. Speedup measures are not provided for all subpopulations because an execution in sequential mode is very slow. The energy-time values obtained by the sequential execution are $E_S = 626.2\,\mathrm{W \cdot h}$ and $t_S = 11,840.26$ s. Thus, *ODGA* provides a speedup of 83 with only 4.9% of the energy consumption shown by the sequential execution.

6.1.3.2 *Fitting the Time Model to ODGA*

In this section, a time model is built to explain the execution time shown in Figure 6.6. The purpose is not to achieve an accurate prediction of the experimental data but to extract conclusions about the best alternative to select according to the power-performance goals. The model starts from Equation (5.26), which was already seen in Section 5.4. To facilitate reading it is shown again:

$$t = g \cdot \left\lceil \frac{N \cdot W_M}{f_C} + max\left(\left\lceil \frac{x \cdot N}{\lambda_G} \right\rceil \cdot \frac{W_G}{f_G}, \left\lceil \frac{(1-x) \cdot N}{N_T} \right\rceil \cdot \frac{W_C}{f_C} \right) \right\rceil \quad (6.1)$$

The meaning of each term is also remembered:

- g: number of generations.
- N: total number of individuals.
- λ_G: number of GPU CUs.
- N_T: number of CPU threads. In previous versions known as λ_C.
- x: rate of individuals assigned to GPU.
- $1 - x$: rate of individuals assigned to CPU.
- W_M: workload assigned to the master.
- W_C: evaluation of an individual in CPU.
- W_G: evaluation of an individual in GPU.
- f_C **and** f_G: operating frequencies of CPU and GPU, respectively.

Equation (6.1) determines the execution time in a single-computer system when only a population of individuals is used. This means that it must be adapted to the operation of *ODGA*. Summarizing, the master process distributes a total of N_{Sp} subpopulations among the workers, which evolve one subpopulation in each device. When a subpopulation is evolved, it is returned to the master and another one is requested. Every certain number of generations, the master performs a global *migration* and when it finishes the subpopulations are distributed again. The whole process is repeated as many times as N_{GM} global *migrations* have been defined. Taking into account the operation mode, for *ODGA* the following terms are redefined:

$$\begin{aligned} R_{\lambda_G} &= \frac{x \cdot \frac{N}{N_{Sp}}}{\lambda_G} \\ &= \frac{x \cdot N}{\lambda_G \cdot N_{Sp}} \end{aligned} \quad (6.2)$$

$$R_{\lambda_{N_T}} = \frac{(1-x)\cdot\frac{N}{N_{Sp}}}{N_T} = \frac{(1-x)\cdot N}{N_T\cdot N_{Sp}} \tag{6.3}$$

Terms $R_{\lambda_{N_T}}$ and R_{λ_G} correspond to the ratio of individuals assigned to a single CPU thread and GPU CU, respectively. As with λ_C, $R_{\lambda_{N_T}}$ replaces the term R_{λ_C} used in previous versions. In this way, the execution time required by a worker node to compute the generations between each global *migration* can be estimated as:

$$t_{Wk} = \frac{g}{N_{GM}}\cdot\left[\frac{N\cdot W_{Wk}}{f_C} + max\left(\left\lceil R_{\lambda_G}\right\rceil\cdot\frac{W_G}{f_G}, \left\lceil R_{\lambda_{N_T}}\right\rceil\cdot\frac{W_C}{f_C}\right)\right] \tag{6.4}$$

where W_{Wk} is the workload for the sequential part of the worker. Thus, the total execution time for the distributed system is as follows:

$$t = N_{GM}\cdot\left[t_M + max\left(t_{Wk_n}\right) + t_{Com}\right];\ \forall n = 1,\ldots,N_{Wk} \tag{6.5}$$

being t_M the time required by the master, t_{Wk_n} the time of worker Wk_n, and t_{Com} the cost of communications between workers.

Experiment 6.5: *Evaluate how the time model fits the experimental results of version ODGA when the total number of individuals is divided into multiple subpopulations. The experimental conditions are also described in Table 6.3.*

Figure 6.7 compares the experimental and fitted execution time when increasing the number of subpopulations. Although there are significant deviations in the predictions for some cases, the model fits correctly the changes shown from 1 to 6 subpopulations, which are more irregular due to the workload imbalance. The model quality is evaluated with the NRMSE[3] metric, a standard criteria for regression whose values are in the interval $[0,1]$ depending on whether the model fits the experimental data properly or not. The NRMSE value obtained is 0.787, which is an acceptable fit taking into account the model complexity.

NRMSE: Normalized Root-Mean-Squared Error

3 A value of NRMSE higher than 0.2 is considered in [135] as a good result

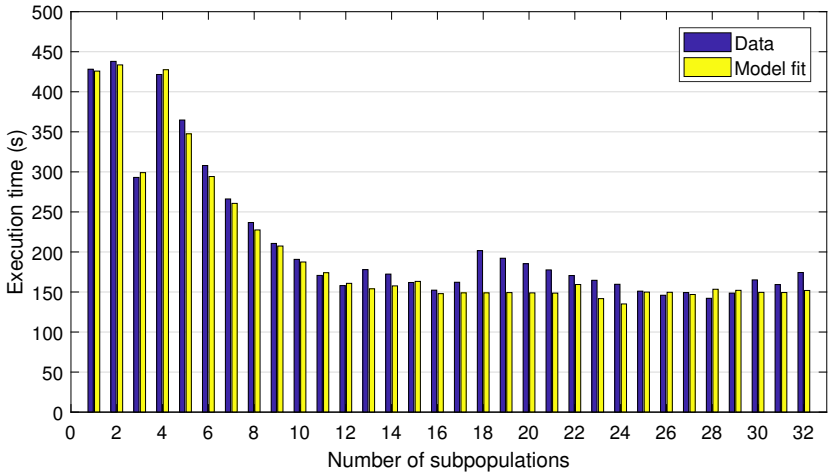


Figure 6.7: Execution time and model fit obtained in *ODGA* when the total number of individuals is divided into multiple subpopulations.

6.1.3.3 *Fitting the Energy Model to ODGA*

The energy model can also be adapted to *ODGA*. As energy consumption depends on instantaneous power and execution time, its value for a worker node can be estimated as:

$$E_{Wk} = P_{Wk} \cdot t_{Wk} + P^I_{Wk} \cdot (t - t_{Wk}) \tag{6.6}$$

where P_{Wk} and P^I_{Wk} are the instantaneous power of the worker when it is computing workload or in idle state, respectively. Expression $(t - t_{Wk})$ corresponds to the time that the worker is in idle state. In this way, the energy consumption for the distributed system is obtained as the sum of energy consumption of all workers plus that of the network:

$$E = E_{Net} + \sum_{n=1}^{N_{Wk}} E_{Wk_n} \tag{6.7}$$

As a switch is required for communications between workers, the energy consumption of the network, E_{Net}, could be defined taking into account its instantaneous power and execution time:

$$E_{Net} = P_{Sw} \cdot t_{Com} + P^I_{Sw} \cdot (t - t_{Com}) \tag{6.8}$$

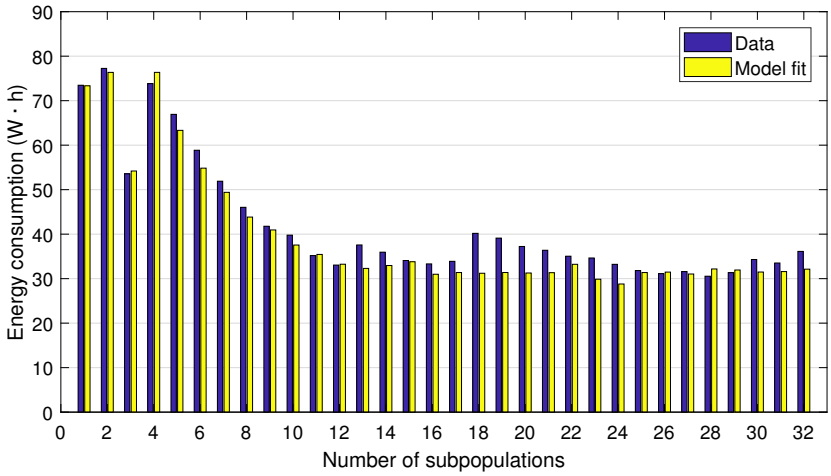


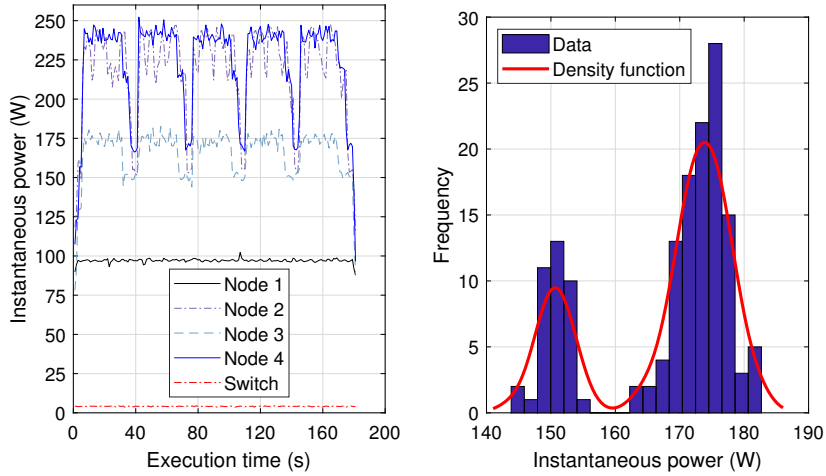
Figure 6.8: Energy consumption and model fit obtained in *ODGA* when the total number of individuals is divided into multiple subpopulations.

being P_{Sw} and P_{Sw}^{I} the instantaneous power of the switch when there are or there are no communications, respectively. Expression $(t - t_{Com})$ corresponds to the time in which there are no communications.

Experiment 6.6: *Evaluate how the energy model fits the experimental results of version ODGA when the total number of individuals is divided into multiple subpopulations. The experimental conditions are also described in Table 6.3.*

The precision in the estimation of energy consumption depends on the behavior of the instantaneous power. The model always assumes two possible average instantaneous power values for each element of the *cluster*: one when it is working and another when it is in idle state. Of course, this situation does not happen frequently, but if the experimental values are quite close to two different average values, the approach could be useful to estimate energy consumption.

In this way, Figure 6.8 compares the experimental values of energy consumption with those estimated by the model. The NRMSE value obtained is 0.744, which is similar to that shown by the time model and therefore corresponds to good model approximations. This means that the two average instantaneous power values used in the model are sufficient to provide an acceptable estimate of energy consumption.



(a) *Using all cluster nodes* (b) *Histogram for the values of Node* 3

Figure 6.9: Instantaneous power obtained in *ODGA* when using different platform configurations and 32 subpopulations.

Figure 6.9a shows the evolution of instantaneous power when using all *cluster* nodes and 32 subpopulations. The highest values correspond to Nodes 2, 3, and 4 because they compute the highest workload. The figure also shows a behavior similar to that of *DGA* and clearly reveals the two instantaneous power states in which a worker node can be. The existence of these states is partly motivated by the use of performance profiles that the devices offer. E.g., the standard ACPI provides information about energy profiles to optimize energy consumption or control the temperature. Similarly, the *Linux* kernel implements the `Cpufreq` infrastructure [136]. It allows the operating system to change the operating frequency of the processor either automatically through events generated by the ACPI interface or through user program calls.

ACPI: Advanced Configuration and Power Interface

Figure 6.9b illustrates the histogram of instantaneous power and the corresponding density function for Node 3. The number of bins in the histogram is equal to the square root of the number of experimental samples. As it can be seen, the density function depicts two maxima: one for 150 W and the other for approximately 174 W. The highest value corresponds to the moment in which the worker is computing; the lowest to when the worker is waiting for the master to perform the global *migration* (idle state). The maxima can be used to estimate the parameters P_{Wk} and P^I_{Wk} required by Equation (6.6). Table 6.4 provides the estimate of these parameters for all *cluster* nodes. As the table shows small variations in the values of the standard deviation, it can be concluded that the estimate is acceptable.

Table 6.4: Standard deviation of P_{Wk} and P^I_{Wk} for all *cluster* nodes.

Node	P_{Wk} (W)	P^I_{Wk} (W)
1	234.72 ± 2.65	156.58 ± 1.47
2	177.36 ± 3.21	154.38 ± 3.41
3	244.58 ± 2.93	170.14 ± 2.16

Table 6.5: CNN hyperparameter values. N_{Ft}: number of filters for the convolution layers with a kernel size of K_S x K_S; H_L: number of hidden layers; FC: number of fully-connected layers with H_N hidden neurons.

N_{Ft}	K_S	H_L	FC	H_N	Epochs	K-folds
16	3	1	1	100	50	4
32	9	2	2	200	100	6

6.1.4 *Distribution of Neural Networks*

There are two methods commonly used to parallelize the training of a neural network with *clusters*: (i) model parallelism [137], where different nodes of the distributed system are responsible for computing different parts of the network, and (ii) data parallelism [138], in which the nodes have a complete copy of the model and each one handles a different portion of data. In both cases, the result of each part is somehow combined. These approaches are focused on the parallelization of a single neural network. However, the classification problem considered requires the training of multiple CNNs with different hyperparameters, so either of these two methods is not efficient given the large number of synchronization operations required by the nodes. Thus, version *DNN* provides a more efficient approach where each CNN is trained independently in each node of the *cluster*.

6.1.4.1 *CNN Structure and Operation Mode*

The implementation of *DNN* is similar to the master-worker scheme of *ODGA* but eliminates the evolutionary steps and reuses the MPI functions for workload distribution and communication between nodes. The master process runs on Node 1 again. Also, a two-dimensional CNN (2D-CNN) is adopted as an EEG classifier. $N_C = 128$ different combinations of hyperparameters are considered to build different CNN architectures, which are obtained from combining two possible

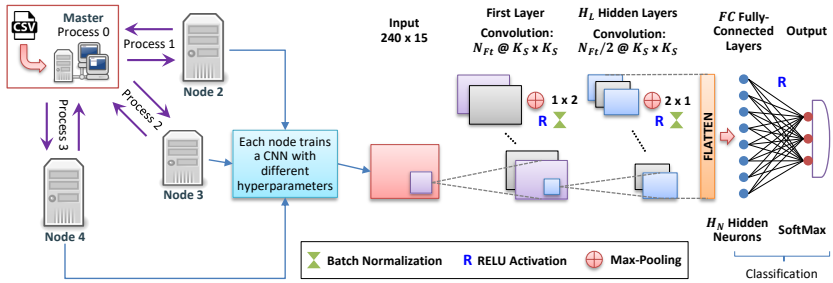


Figure 6.10: Scheme defined in *DNN* and 2D-CNN topology. The master distributes combinations of hyperparameters among the worker nodes.

values for each of the seven hyperparameters (see Table 6.5). The values are arbitrary, although they have been chosen to obtain valid outputs in the networks. Figure 6.10 shows a general scheme of the proposed procedure and the CNN topology, which is built as follows:

1. The first layer is a 2D-convolutional operation whose input shape is a 240 x 15 matrix since an EEG signal is composed of 240 samples from 15 electrodes. The convolution applies N_{Ft} filters with a kernel size of K_S x K_S.

2. A batch normalization operation is applied to normalize the activations of the first layer at each batch.

3. A *max-pooling* operation of size 2 x 1 is applied to reduce the size of the samples' dimension.

4. H_L hidden layers are applied, which are also convolutional operations. Each convolution applies half of the filters used by the previous layer but keeps the kernel size. After each hidden layer, a batch normalization operation and a *max-pooling* of size 1 x 2 are applied to reduce the electrode's dimension. All convolution layers compose the FS part, which should be able to select the most relevant spatio-temporal features of the EEGs.

5. A *flattening* operation is used to reshape the output of the last convolutional layer to vectors.

6. *FC* fully-connected layers, together with the output layer compose the classification part. The first fully-connected layer contains H_N hidden neurons. The next one, half of the neurons of the previous layer, and so on. The output layer contains three units since an EEG can belong to one of the three possible classes.

Algorithm 6.6 details the master-worker scheme to distribute and evaluate combinations of hyperparameters. Certainly, the procedure is a mixture between *DGA* and *ODGA* because it takes advantage of the simplicity of *DGA* but also some of the *ODGA* optimizations. The workload distribution is again dynamic to avoid workload imbalance. It must be taken into account that execution time depends not only on the heterogeneity of the nodes but also on each combination of hyperparameters. The `Main` function is divided into two sections: one for the master process (Line 2) and the other for the workers (Line 20).

Firstly, the function receives the input parameters necessary to operate correctly: the number of hyperparameter combinations, the CSV filename with these combinations, the *Python* file with the CNN implementation, and the number of workers available. The CSV file is read by the master in Line 3. Each row of the file contains a different combination, which is composed of multiple columns, i.e., the hyperparameters. As in *ODGA*, if no workers are available the master will perform all the job by using the CPU or GPU devices. If the `if-else` statement becomes false, the master asynchronously attends the requests of each worker. It dynamically distributes the rows of the CSV file until there is no more work to do. Once all N_C combinations (rows) have been evaluated, the master sends the *FINISH* signal to all workers to notify that the job has finished (Line 18).

CSV: Comma-Separated Values

In the case of workers, each of them requests to the master a combination of hyperparameters at the beginning, which is stored into C if the $FINISH$ signal has not been received (Lines 22 and 23). Then, the worker cyclically trains the CNN, asks the master for more work, and awaits the response (Lines 25 to 28). Regardless of whether the training of the network is done by the master or a worker, the operation mode is as follows: from the application, a call to the *Python* interpreter is made by invoking the command line interpreter through the `system` function (Line 8 and Line 26). This function requires as an argument the command to be executed, which is composed mostly from the hyperparameters. Its use is equivalent to executing it through a terminal of the operating system. The command is a concatenation of, in this order: name of the *Python* interpreter, name of the *Python* file with the CNN implementation, the hyperparameters, and the device to use (CPU or GPU). Taking into account the possible combinations of seven hyperparameters that could be obtained from the values shown in Table 6.5, an example of the command to be executed is:

```
python3 cnn.py 2 9 4 100 1 100 32 GPU
```

Algorithm 6.6: Master-worker pseudocode defined in *DNN* to distribute and evaluate combinations of hyperparameters. If no workers are detected, the master will perform all combinations.

```
1  Function Main(N_C, CSVFile, PyFile, N_Wk)
       Input: Number of hyperparameter combinations, N_C
       Input: CSVFile: file containing all N_C combinations
       Input: PyFile: Python file with the CNN implementation
       Input: Number of workers, N_Wk
2      if Master then
           // Each row is a string like "<ARG_1>...<ARG_N>"
3          C ← readCSV(N_C, CSVFile)
4          if N_Wk == 0 then
5              D ← getDevice()
6              for i ← 1 to N_C combinations of hyperparameters do
7                  CmdExec ← append("python3 ", C_i, D)
8                  Acc ← system(CmdExec)
9              end
10         else
11             RemainingWork ← N_C
12             repeat
13                 MPI::Recv(Acc) ← Request from worker Wk_n
14                 sent ← min(RemainingWork, 1)
15                 MPI::Isend(C, 1) → To Wk_n
16                 RemainingWork ← RemainingWork − sent
17             until RemainingWork == 0;
18             MPI::Bcast(FINISH) → To all N_Wk workers
19         end
20     else
21         D ← getDevice()
22         MPI::Isend(1) → Request one combination to the master
23         MPI::Recv(C) ← 1 combination from the master
24         repeat
25             CmdExec ← append("python3 ", C, D)
26             Acc ← system(CmdExec)
27             MPI::Isend(Acc) → Return the CNN accuracy
28             MPI::Recv(C) ← 1 combination from the master
29         until the FINISH signal is received;
30     end
31 End
```

Table 6.6: Experimental setup to analyze *DNN*.

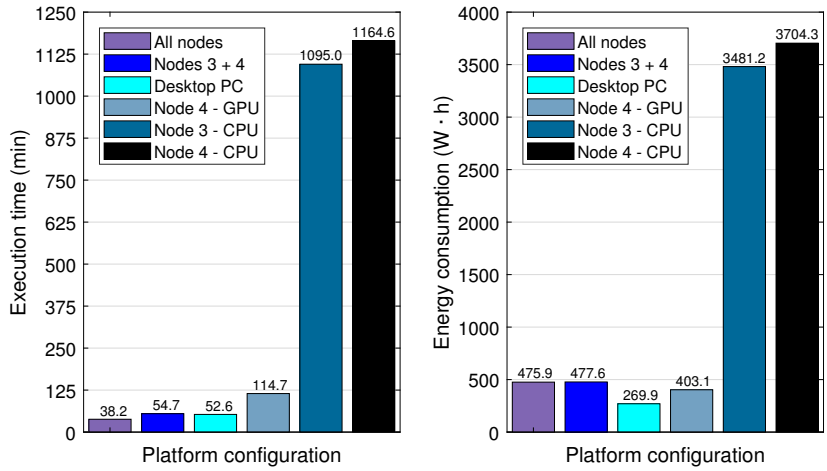
Feature	Description
Platforms	All cluster nodes and desktop PC
Devices	Cluster: CPU of Nodes 3 and 4 and all Tesla GPUs
	PC: only one GPU
Compiler	GCC with -O2 -funroll-loops
Dataset	All features used
CNN	128 combinations of hyperparameters (see Table 6.5)

6.1.4.2 *Node Scalability and CPU-GPU Issues*

Experiment 6.7: *Evaluate the energy-time performance of DNN when using all cluster nodes and the desktop PC to train 128 CNNs. The experimental conditions are described in Table 6.6.*

Figure 6.11a shows the average execution time obtained after training all possible CNNs. Observing the *cluster* measures, it can be seen that the more nodes are used, the less execution time. In theory, the execution time when using Nodes 3 and 4 should be half the time obtained when only Node 4 computes, but what happens is that the value is even less. As *TensorFlow* executes some operations on CPU, the difference is due to the fact that the CPU of Node 3 is more efficient than that of Node 4 despite having fewer cores/threads. This can be verified in the figure by observing the CPU execution times for both nodes. The behavior can be explained taking into account that the number of CPU sockets of Node 3 is 1, while Node 4 has 2. It has been checked that during the program execution the threads constantly migrate between the cores of both sockets, which introduces a considerable overhead. Probably, if two homogeneous CPUs had been used, the speedups would be very close to 2. Unfortunately, each node of the *cluster* has a different CPU.

On the other hand, using all worker nodes provides a time reduction equal to 3. This is pure coincidence, since continuing with the previous logic the speedup should be greater than 3. However, it seems that the lower capabilities of the GPU *Tesla K20c* of Node 2 along with the overhead caused by its 2 CPU sockets derive in a performance reduction. In any case, it seems that the algorithm scales correctly since the execution time necessary to train a network is almost 100%.



(a) *Execution time* (b) *Energy consumption*

Figure 6.11: Performance obtained in *DNN* when using different platform configurations to train 128 CNNs.

Executions on GPU have been compared with the equivalent ones on CPU. The result confirms that CPUs are not currently the best devices for this type of problems. The time to perform the task on these devices is around 10 times slower than the worst possible scenario when using GPUs. Although the CPU of Node 4 contains 16/32 cores/threads, during executions the CPU usage does not exceed 35%, so CPU is not being exploited (45% for Node 3). This issue could be motivated by one or several factors: (i) that neural networks are not highly parallelizable in CPU; (ii) *TensorFlow* is not optimized for these devices, perhaps derived from point (i); (iii) the network topology used for the experiment is not complex enough to take advantage of the full architecture. For GPU, the percentage of use ranges 68-85% when the program is executed.

Figure 6.12 shows that in all configurations the instantaneous power goes up and down twice and the general tendency is to increase with the passage of time. This is because the hyperparameters that create a more complex neural network model are evaluated at the end of each stretch. If the instantaneous power is higher when the model is more complex, and given that the device usage is related to energy consumption, it can be concluded that at least point (iii) is affecting performance. Although this also affects GPU, the CPU is negatively affected not only by its low usage, but also because it does not have the same data parallelism capacity as GPU architectures. Summarizing, all this makes the GPU far outperform CPU, so that it is the best device to accelerate neural networks.

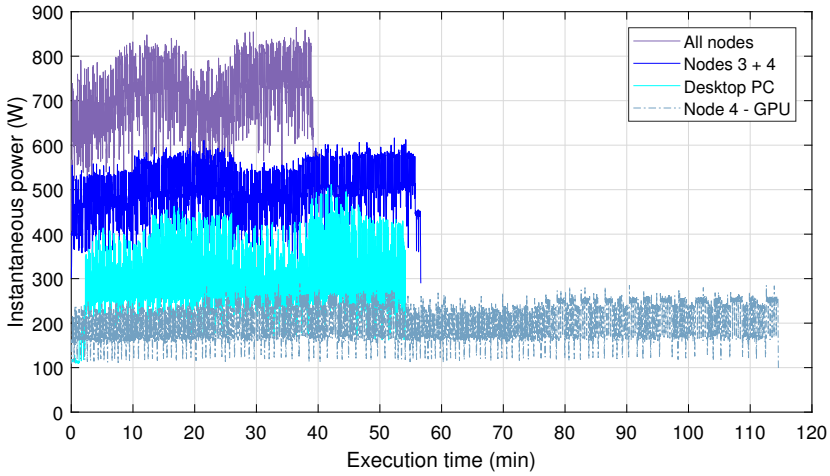


Figure 6.12: Instantaneous power obtained in *DNN* when using different platform configurations to train 128 CNNs.

From Figure 6.12, it can also be observed that the use of more nodes not only entails a reduction in execution time but also an increase of instantaneous power, as it is logical. However, the instantaneous power of the desktop PC can be highlighted. It is able to perform the task in a slightly shorter time than needed by Nodes 3 + 4 but with the instantaneous power to just over half. The reasons were already commented in Experiment 6.2: the differences between the GPU architecture of the *cluster* and that of the PC. Moreover, it must be taken into account the energy consumption of the three nodes (Nodes 1, 3, and 4), which include RAMs, CPUs, hard disks, etc. Based on the results obtained, the GPU architecture acquires great importance. Thus, it could be better to acquire more modern architectures and amortize their cost by decreasing the number of nodes and the monetary cost associated with energy consumption.

Figure 6.11b shows the energy consumption after training all networks. As expected, its behavior is similar to that of execution time and the shape of the bars for CPU remains constant. However, the interesting thing is to see how the number of nodes and the type of architecture affects energy consumption. Using all nodes simultaneously provides almost the same energy consumption than using only Nodes 3 + 4 but with the difference that the execution time is shorter. Using only Node 4 leads to a reduction in energy consumption of approximately 15.6% with respect to the other two *cluster* configurations, but its execution time is 3x slower than when using all nodes, and approximately 2x if Nodes 3 + 4 are used. Again, GPU *TITAN Xp* allocated in the desktop

PC provides the most remarkable results because it provides the lowest energy consumption while offering good times. Moreover, it is the simplest platform. Analyzing the data, it seems that there is a certain tie between using the desktop PC and all *cluster* nodes. The energy-time product metric, presented in Section 4.3, should clarify the situation:

$$\begin{aligned} Et_{Cluster} &= 38.23 \cdot 475.89 = 1.819327 \cdot 10^4 \\ Et_{PC} &= 52.61 \cdot 269.87 = 1.419786 \cdot 10^4 \end{aligned} \tag{6.9}$$

As the lowest value of Et corresponds to the desktop PC, it can be concluded that this option provides the best results. Nevertheless, depending on the context, it may be preferred the option with the lowest energy consumption or the one that provides the fastest execution.

6.2 AN IMPLEMENTATION WITH ASYNCHRONOUS MIGRATIONS

This section presents the latest version developed in this thesis: *GAAM*. Unlike previous versions, *GAAM* does not use the master-worker approach to distribute subpopulations among nodes. Instead, all working nodes have the same number of subpopulations unless the total number of them is not entirely divisible by the number of workers, so that one or more workers would have one more subpopulation. Also, in the case where there are more nodes than subpopulations, some of them will not compute. In this way, the first level of parallelism is preserved since in a certain way the subpopulations are distributed among nodes. The master-worker scheme is not completely ruled out, since the operation mode of the workers remains governed by the same principles of *ODGA*. This means that each worker distributes all its subpopulations among the devices iteratively.

6.2.1 *Buffers Hierarchy Design*

The new implementation allows the *migration* of individuals from one subpopulation to another at each generation. The whole *migration* process is performed asynchronously thanks to a hierarchy of I/O buffers and a handler whose function is to receive and send *migration* requests, manage buffers, and decide how many individuals to migrate. Communication between the handlers of each node is done through MPI library. The *GAAM* pseudocode is shown in Algorithm 6.7.

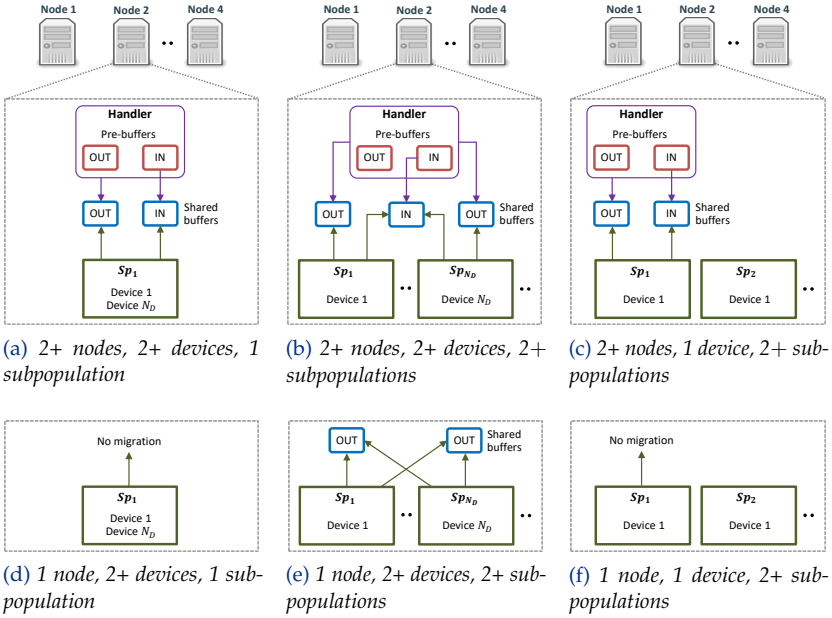
Algorithm 6.7: Procedure pseudocode defined in *GAAM* that evolves subpopulations using asynchronous *migrations*.

```
1  Function Main(N_Sp, S_S, M, N_Wk, N_D)
       Input  : Number of subpopulations to be evolved, N_Sp
       Input  : Subpopulation size, S_S
       Input  : Number of objectives, M
       Input  : Number of workers, N_Wk
       Input  : Number of devices, N_D
       Output : hv, the hypervolume metric
2      N_Sp_W ← N_Sp / N_W
3      if Wk_n < (N_Sp mod N_W) then
4          N_Sp_W ← N_Sp_W + 1
5      end
6      if N_Sp_W > 0 then
7          DS ← getDataset()
8          Sp ← initSubpopulations(N_Sp_W, S_S, M, DS)
9          k ← getRandomCentroids(DS)
10         D ← initDevices(DS, k, N_D)
11         B ← initBuffers(S_S)
12         #pragma omp parallel num_threads(2)
13             if omp_get_thread_num() == 0 then
14                 th ← min(N_Sp_W, N_D)
15                 #pragma omp parallel for num_threads(th)
16                     Sp ← evolve(Sp_W, N_Sp_W, S_S, M, D, N_D, B)
17                 end
18             else
19                 handler(S_S, B)
20             end
21             if Wk_n == 0 then
22                 MPI::Gatherv(Sp) ← Sp_W from all Wk_n
23                 Sp ← subpopulationMerging(Sp, N_Sp, S_S, M)
24                 hv ← getHypervolume(Sp, N_Sp, S_S, M)
25             else
26                 MPI::Gatherv(Sp_W, N_Sp_W) → to root MPI process
27             end
28         end
29     end
30     return hv
31 End
```

Although the master-worker scheme is not present in this version, for simplicity each node will continue to be known as a worker. This means that all nodes are workers and the master role does not exist. At the beginning of the procedure, each worker determines how many subpopulations it has to evolve. This is calculated by dividing the total number of subpopulations by the number of workers. If the remainder of the division is nonzero, those workers with MPI rank lower than the remainder will evolve one more subpopulation (Lines 3 to 5).

One of the drawbacks of *ODGA* is that all workers are active throughout the execution. This is because after requesting subpopulations to the master, they remain waiting for an answer. On the contrary, in *GAAM* the workers previously calculate the number of subpopulations to evolve. This allows them to know in advance if they have to continue with the execution or can finish (Line 6). In case of continuing, each worker performs the usual steps, i.e., obtaining the dataset and centroids, initializing subpopulations, devices, and buffers necessary for the asynchronous *migrations* (Lines 7 to 11). Then, through the *OpenMP* `pragma` of Line 12, a parallel region with two threads is created, where the thread #0 will be in charge of the evolutionary steps and the thread #1 will be in charge of the handler. In turn, thread #0 creates another parallel region with as many threads as indicated by the operation of Line 14 to distribute the evolution of subpopulations among the devices. The aim is not to create more threads than necessary and therefore not to waste resources. Finally, once the workers complete the evolution of their subpopulations, the root MPI process performs the subpopulation merging in Line 23. To do this, it combines its populations with those received by the rest of workers through the `MPI::Gatherv` operation of Line 22. This step could be considered as the only one in which there is synchronization among the workers, although it occurs at the end of the execution.

With respect to the *migration* step, this is carried out within the `handler` function. The handler continually awaits the receipt of a *migration* request from another worker's handler, although it can also randomly initiate a *migration* request. The handler and certain buffers will only exist if they are necessary, which avoids the use of unnecessary CPU resources. Figure 6.13 illustrates the six possible scenarios in what a worker can be. Its situation depends on the number of workers, devices, and subpopulations used during the program execution. E.g., if only one node is used, the handler is expendable since communications with other nodes are not required. If there is also only one subpopulation, I/O buffers are not created either. Assuming the case of Figure 6.13b for being the most complete, to make the entire process asynchronous a hierarchy of buffers with the following elements is used:



(a) *2+ nodes, 2+ devices, 1 subpopulation*

(b) *2+ nodes, 2+ devices, 2+ subpopulations*

(c) *2+ nodes, 1 device, 2+ subpopulations*

(d) *1 node, 2+ devices, 1 subpopulation*

(e) *1 node, 2+ devices, 2+ subpopulations*

(f) *1 node, 1 device, 2+ subpopulations*

Figure 6.13: Different *migration* scenarios and buffers hierarchy of the nodes depending on their number of devices, assigned subpopulations, and number of nodes used during program execution.

- The handler has an input prebuffer to store the individuals received from other handlers and an output prebuffer with individuals ready to send. These prebuffers can only be accessed by the handler and can store a quarter of the subpopulation size ($\frac{S_S}{4}$).

- Each thread that evolves a subpopulation has an output buffer on which it can add individuals after each generation. When the handler decides to migrate, it selects one of the output buffers and copy some individuals to its output prebuffer. An output buffer can only be accessed by its corresponding thread and the handler except in the case of Figure 6.13e. Here, since there is no handler, the *migration* is performed directly between the threads of each subpopulation. The storage of these buffers is also $\frac{S_S}{4}$, and the `pragma omp critical` has been used to control shared access.

- There is an intermediate input buffer that acts as a link between the handler's input prebuffer and the threads of each subpopulation. This buffer has a capacity of $\frac{S_S}{2}$ because it is shared between all threads and the handler. After receiving individuals from other workers, the handler copies some individuals from

Table 6.7: Experimental setup to compare *ODGA* and *GAAM*.

Feature	Description
Platforms	All cluster nodes
Devices	All CPUs and all GPUs Tesla and TITAN Xp
Compiler	GCC with -O2 -funroll-loops
Dataset	All features used
Individuals	3,840
Subpopulations	32
Generations	150
Migrations	Global: 5

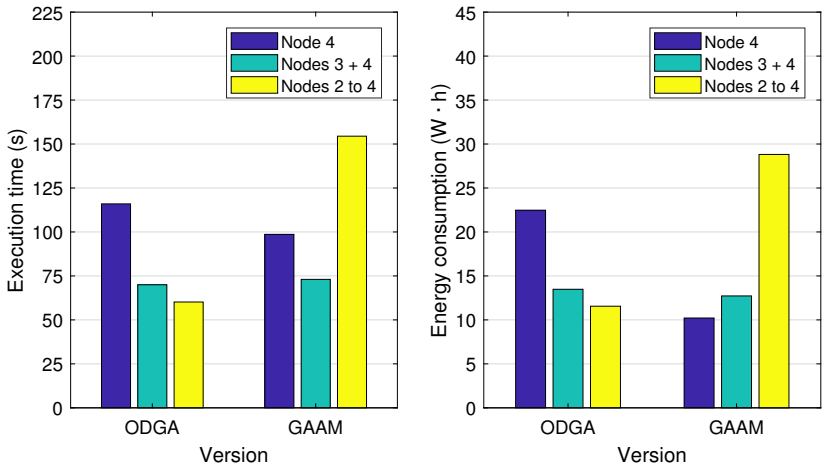
its input prebuffer to the intermediate buffer. At each generation, the threads check the buffer and may randomly pick up some individuals, completing the *migration*. In a sense, migrating in this way can be seen as a mix between the global and local *migrations* implemented in *DGA*. As this buffer is shared, the individuals that a thread obtains can come from the same node or another.

All decisions, both of threads and handler are random with a probability of 0.25. The number of individuals to copy between buffers ranges from 1 to the square root of the origin buffer size. The intention is to avoid excessive *migrations* between subpopulations.

6.2.2 *ODGA and GAAM Comparison*

Experiment 6.8: *Compare the energy-time performance of ODGA and GAAM when using different platform configurations. The experimental conditions are described in Table 6.7.*

Figure 6.14 shows the performance of *ODGA* and *GAAM* when using different platform configurations. It should be noted that for this experiment the two GPUs *TITAN Xp* of the desktop PC have been added to Nodes 3 and 4 of the *cluster*. The extra performance for incorporating both GPUs can be seen indirectly in Figure 6.14a. The data show execution times 2.5x faster than those shown in Figure 6.6a, denoting the great impact that *TITAN Xp* has on performance.



(a) *Execution time* (b) *Energy consumption*

Figure 6.14: Energy-time performance obtained in *ODGA* and *GAAM* when using different platform configurations.

The figure also reveals that *GAAM* runs a little faster when using Node 4. Although both versions use a single node to compute, the master process of *ODGA* runs on Node 1, so the cost of communications reduces performance. On the other hand, if Node 3 also computes, *GAAM* reduces its execution time by 28% but is slightly worse than that shown by *ODGA*. Its scalability is not so good because the CPU of Node 3 has half the cores than that of Node 4. In addition, as a consequence there is a workload imbalance that *ODGA* does not show. It is remembered that *GAAM* assigns the same number of subpopulations to each node regardless of their computing capabilities. This is demonstrated by observing the behavior of *GAAM* when Node 2 is added. This node does not have a *TITAN Xp*, its GPU *Tesla* is slightly lower than *Teslas* of Nodes 3 and 4, and its CPU is somewhere in between. In summary, Node 2 is much less powerful in terms of computing capabilities and as a result the execution time is degraded.

Regarding energy consumption, in Figure 6.14b it can be seen that *GAAM* shows an energy profile different from that of *ODGA*. Of all experiments carried out so far, it is the first time that energy consumption does not completely follow the same trend as execution time. With a single node, *GAAM* is a little faster but the energy value is reduced to less than half. The reason lies in the absence of the master of Node 1 and therefore its energy consumption is not taken into account. When two nodes compute, the energy value increases, although it does so to a lesser extent than time reduction. This behavior differs from those

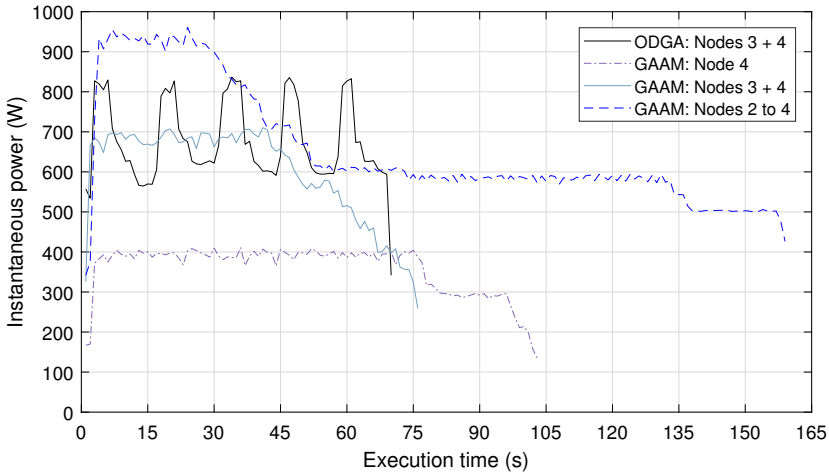


Figure 6.15: Instantaneous power obtained in *ODGA* and *GAAM* when using different platform configurations.

shown by previous versions where the more parallelism, the less consumption. Finally, when three nodes are used, energy consumption increases since more nodes are used and time reduction is not obtained.

The workload imbalance of *GAAM* is depicted in more detail in Figure 6.15, which shows the instantaneous power for different platform configurations. It can be seen that in *GAAM* there are no power drops as a result of global *migrations*. Instead, falls occur every time a node finishes its work, giving clues about how long a node or device could be in idle state. Thus, in the case of Nodes 2 to 4, all nodes and devices work for 28 s. In the interval of 28-52 s, some devices stop working and until second #132 only two nodes compute. From here, only one node is active. Based on the moment the first fall occurs, a redesign of *GAAM* that incorporates workload balancing may provide better results than *ODGA* since full parallelism is only achieved for 17.7% of the time.

6.3 CONCLUSIONS

In this chapter, eight sets of experiments have been carried out to evaluate the versions designed for distributed systems. The first version is *DGA*, which adds local *migrations* and a new parallelism level in which a master node distributes subpopulations among worker nodes. It has allowed not only to reach speedup peaks above 70 but also an increase in hypervolume as a result of increasing the problem size.

However, *DGA* states some issues related to communications between nodes and workload distribution, which have been improved in version *ODGA* by giving the devices the ability to communicate directly with the master. Local *migrations* have also been eliminated, since according to Experiment 6.1 they have no impact on hypervolume. The analysis performed in Experiment 6.4 demonstrates that the optimizations implemented in this version improve performance, raising the speedup to a maximum of 83 with only 4.9% of the energy consumption shown by a sequential execution. Therefore, this version is presented as the most balanced and efficient developed throughout this thesis. To complement the analysis, in Experiments 6.5 and 6.6 an energy-time model for this version has been developed and evaluated. The results are promising, since the model allows predicting the behavior of the procedure with an acceptable error. Moreover, in an attempt to improve *ODGA*, version *GAAM* is created in order to remove synchronizations and idle states derived from global *migrations*. For this, the master-worker approach is replaced by a scheme based on asynchronous *migrations* and hierarchical buffers. However, Experiment 6.8 shows that its performance is worse as a result of the workload imbalance. Even so, this version has room for improvement, so its approach should not be easily discarded.

On the other hand, version *DGA-II* is born as a variant of *DGA* that uses *OpenMP* instead of *OpenCL* to implement *K-means*. The premise is that a good compiler could extract from the *OpenMP* code more performance than the *OpenCL* driver could obtain. This has been checked in Experiment 6.3, which analyzes the impact on performance of GCC and ICC compilers. The analysis determines that the best result in both execution time and energy consumption is obtained when using the proprietary compiler (ICC) along with the `-O3` optimization flag.

Finally, a procedure based on neural networks has been developed as an alternative to MOGA, which reuses the master-worker scheme of *ODGA* to distribute the workload among the *cluster* nodes, if any. Instead of subpopulations, the master distributes combinations of hyperparameters, where each one is used to train a different CNN. As the procedure is based on *ODGA*, it is mainly coded with *C++*, although CNNs are implemented with *Python*, *Keras*, and *TensorFlow*. The proposed algorithm has been analyzed in Experiment 6.7, which shows that when more nodes are used, the procedure scales linearly and the lowest execution time is obtained. However, the desktop PC provides the best energy results since the efficiency of the GPUs *TITAN Xp* is far superior to that shown by GPUs *Tesla*. The experiment also states that CPUs, although nowadays include many processing cores, are not the most suitable devices for training neural networks. This demonstrates that each task should be carried out with the appropriate device.

7

CONCLUSIONS

CONTENTS

Although many works in the literature have shown important advances in the development of efficient *metaheuristics*, less details have been reported about the benefits of parallel architectures for energy-saving. For this reason, the main contribution of this thesis has been to provide efficient and parallel methods capable of taking advantage of the computing resources that heterogeneous systems provide. The application considered here corresponds to an EEG classification problem for BCI tasks that has been addressed with neural networks and a multi-objective GA where the individuals codify different FSs. Both approaches have been parallelized for single-computer and distributed systems, which include platforms such as multi-core CPUs and GPUs accelerators. More specifically, a heterogeneous four-node *cluster* and a desktop PC have been used.

However, the use of heterogeneous systems with different energy profiles and computing capabilities has shown other problems, mainly related to the workload imbalance. In addition, GAs also add irregularities in memory accesses during the *fitness* evaluation caused by the FS characteristics. To identify the impact of these problems on performance, the procedures have been analyzed under different experimental conditions. Through experimentation it is possible to find optimizations that allow to reduce execution time and therefore energy consumption. In fact, each procedure is developed taking into account the drawbacks of its predecessor and keeping those functionalities that provide good results. Another contribution of this thesis has been the development of energy-time models to approximate the behavior of the algorithms in both single-computer and distributed systems, which is not very common to find today. Although they are fitted to the profile of the application proposed here, a generalization of the models would not be a problem since most of the terms in the equations refer to the characteristics of the platforms rather than the application.

7.1 CONCLUSIONS ABOUT THE PROPOSED OBJECTIVES

In Section 1.2, a total of nine objectives were proposed, which have been satisfactorily completed throughout the document. The first objective is essential as it consists in creating at least one full version of the application. Finally, a *MATLAB* version and another one in *C++* have been created and compared. In general, the remaining objectives are related to performance improvement through the use of parallelism, new computing paradigms, optimization techniques, and strategies for workload distribution. The term performance refers to execution time, energy consumption, and hypervolume.

Parallelism has allowed the application to accelerate several orders of magnitude using the computing capabilities of CPU and GPU devices. In distributed systems, it has been possible to implement up to four parallelism levels in GPU, providing in the best case speedup peaks of 83 that reduce energy consumption to 4.9%. Within the optimization options, the *coalescing* technique can be found, which has allowed improving access to GPU memory and take advantage of local memory. The cooperation between CPU and GPU for the *fitness* evaluation can also be considered as a way of optimization, although it is close to the domains of workload balancing. Regarding the latter, there are several versions focused on this objective. Experiment 6.8 demonstrates that workload balancing can have the same impact on final performance as parallelism. In fact, a bad strategy can spoil everything, as has been the case of *GAAM*. While the idea is correct and really achieves what it proposes (eliminating idle states), the equitable allocation of subpopulations to all workers has impaired its performance.

A comprehensive analysis of each implemented version was also proposed as an objective. This has been done through 16 sets of experiments distributed among Chapters 5 and 6. Thanks to them, the problems of each version have been identified and improved. In addition to the usual performance analyzes, energy-time models have also been proposed to understand the behavior of the procedures. However, no model has been designed for version *DNN* because *TensorFlow* acts as a black box and it is difficult to identify how it works internally. This version is the only one that is not based on the MOGA approach. Instead, it uses neural networks for EEG classification. Although *DNN* has been presented as an alternative method, no classification metrics have been provided since the scope of this thesis is more related to the energy-time analysis, so it remains as future work. Even so, some preliminary results show that the classification accuracy obtained from the trained CNNs is between 61.81 and 79.13%, and 72.55% on average.

7.2 FUTURE WORKS AND OUTLOOK

A lot of research work is still possible since this thesis deals with multiple aspects. On the one side, it would be very useful to have more accurate energy-time models that take into account other system elements, as well as measure the energy consumption of each device. E.g., The CUDA library of *NVIDIA* GPUs provides methods to measure in real-time the instantaneous power of its devices. *Intel* CPUs also have similar functionalities. In addition, the energy measurement system must be improved because with current accuracy (1 s) information is lost, especially in executions with relatively small execution time.

On the other hand, this thesis has focused on the performance of the procedures and has relegated to the background the EEG classification problem. Therefore, this should be the first issue to address. Although *K-means* is a good method, it is not the most appropriate to solve the problem. The successor could be the KNN supervised algorithm for its similar implementation and high parallelism. Another possibility is to continue the line of neural networks since they currently obtain good results. In order to evaluate the performance of *DNN*, the topology of the CNNs has been previously fixed. However, the correct way to train a neuronal network is by optimizing its hyperparameters. Thus, as future work a hybrid method that combines neural networks with evolutionary procedures is proposed to optimize these hyperparameters.

Concerning energy-time performance, it is clear that heterogeneous *clusters* are essential to advance in the field of energy-aware computing. The different parallelism levels they provide open new opportunities to improve performance. One of them is to analyze the scalability of the procedures in larger *clusters*, something that has not been possible in this thesis due to economic factors. Although it is possible to continue optimizing at the device level, scaling the number of nodes offers performance with no apparent limits. However, this is not entirely true because the massive use of nodes implies solving challenges related to communication costs and high energy consumption. In addition, the workload balancing issues seen in this thesis must be added, which could be aggravated in larger *clusters*. In this sense, it is proposed the creation of a new procedure that unifies the advantages of the master-worker approach of *ODGA* and the philosophy of *GAAM*, despite its poor results. However, probably the most important factor to consider in the coming years is not performance, but energy consumption due to environmental reasons such as the problem of climate change. Although luckily it is already working on it, there is still a long way to go in this regard and this thesis has tried to take a small step forward.

Part III

APPENDICES & BIBLIOGRAPHY

A

PUBLICATIONS

CONTENTS

A.1 INTERNATIONAL JOURNALS WITH IMPACT FACTOR

[1] J. J. Escobar, J. Ortega, J. González, M. Damas, and A. F. Díaz. "Parallel high-dimensional multi-objective feature selection for EEG classification with dynamic workload balancing on CPU-GPU". In: *Cluster Computing* 20.3 (2017), pp. 1881–1897. DOI: 10.1007/s10586-017-0980-7.

[2] J. J. Escobar, J. Ortega, A. F. Díaz, J. González, and M. Damas. "A Power-Performance Perspective to Multiobjective Electroencephalogram Feature Selection on Heterogeneous Parallel Platforms". In: *Journal of Computational Biology* 25.8 (2018), pp. 882–893. DOI: 10.1089/cmb.2018.0080.

[3] J. J. Escobar, J. Ortega, A. F. Díaz, J. González, and M. Damas. "Energy-aware load balancing of parallel evolutionary algorithms with heavy fitness functions in heterogeneous CPU-GPU architectures". In: *Concurrency and Computation: Practice and Experience* 31.6 (2019), e4688. DOI: 10.1002/cpe.4688.

[4] J. J. Escobar, J. Ortega, A. F. Díaz, J. González, and M. Damas. "Time-energy Analysis of Multi-level Parallelism in Heterogeneous Clusters: The case of EEG Classification in BCI Tasks". In: *The Journal of Supercomputing* 75.7 (2019), pp. 3397–3425. DOI: 10.1007/s11227-019-02908-4.

[5] J. J. Escobar, J. Ortega, A. F. Díaz, J. González, and M. Damas. "A Parallel Master-worker GA for CPU-GPU Clusters: An Approach to Energy-time Analysis". In: *The International Journal of High Performance Computing Applications* (2020). **Under review**.

A.2 INTERNATIONAL CONFERENCES

[1] J. J. Escobar, J. Ortega, J. González, and M. Damas. "Assessing Parallel Heterogeneous Computer Architectures for Multiobjective Feature Selection on EEG Classification". In: *Proceedings of the 4th International Conference on Bioinformatics and Biomedical Engineering*. IWBBIO'2016. Granada, Spain: Springer, April 2016, pp. 277–289. DOI: 10.1007/978-3-319-31744-1_25.

[2] J. J. Escobar, J. Ortega, J. González, and M. Damas. "Improving Memory Accesses for Heterogeneous Parallel Multi-objective Feature Selection on EEG Classification". In: *Proceedings of the 4th International Workshop on Parallelism in Bioinformatics*. PBIO'2016. Grenoble, France: Springer, August 2016, pp. 372–383. DOI: 10.1007/978-3-319-58943-5_30.

[3] J. J. Escobar, J. Ortega, J. González, M. Damas, and B. Prieto. "Issues on GPU Parallel Implementation of Evolutionary High-Dimensional Multi-objective Feature Selection". In: *Proceedings of the 20th European Conference on Applications of Evolutionary Computation, Part I*. EVOSTAR'2017. Amsterdam, The Netherlands: Springer, April 2017, pp. 773–788. DOI: 10.1007/978-3-319-55849-3_50.

[4] J. Ortega, J. J. Escobar, A. F. Díaz, J. González, and M. Damas. "Energy-aware scheduling for parallel evolutionary algorithms in heterogeneous architectures". In: *Proceedings of the 2nd International Workshop on Power-Aware Computing*. PACO'2017. Schloss Ringberg, Germany: Zenodo, July 2017. DOI: 10.5281/zenodo.814806.

[5] J. J. Escobar, J. Ortega, A. F. Díaz, J. González, and M. Damas. "Power-Performance Evaluation of Parallel Multi-objective EEG Feature Selection on CPU-GPU Platforms". In: *Proceedings of the 5th International Workshop on Parallelism in Bioinformatics*. PBIO'2017. Helsinki, Finland: Springer, August 2017, pp. 580–590. DOI: 10.1007/978-3-319-65482-9_43.

[6] J. J. Escobar, J. Ortega, A. F. Díaz, J. González, and M. Damas. "Multi-objective Feature Selection for EEG Classification with Multi-Level Parallelism on Heterogeneous CPU-GPU Clusters". In: *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*. GECCO'2018. Kyoto, Japan: ACM, July 2018, pp. 1862–1869. DOI: 10.1145/3205651.3208239.

[7] J. J. Escobar, J. Ortega, A. F. Díaz, J. González, and M. Damas. "Speedup and Energy Analysis of EEG Classification for BCI Tasks on CPU-GPU Clusters". In: *Proceedings of the 6th Interna-*

tional Workshop on Parallelism in Bioinformatics. PBIO'2018. Barcelona, Spain: ACM, September 2018, pp. 33–43. DOI: 10.1145/3235830.3235834.

[8] J. J. Escobar, J. Ortega, M. Damas, R. Savran Kızıltepe, and J. Q. Gan. "Energy-time Analysis of Convolutional Neural Networks Distributed on Heterogeneous Clusters for EEG classification". In: *Proceedings of the 15th International Work-Conference on Artificial Neural Networks*. IWANN'2019. Gran Canaria, Spain: Springer, June 2019, pp. 895–907. DOI: 10.1007/978-3-030-20518-8_74.

[9] J. J. Escobar, J. Ortega, A. F. Díaz, J. González, and M. Damas. "A Parallel and Distributed Multi-population GA with Asynchronous Migrations: Energy-time Analysis for Heterogeneous Systems". In: *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*. GECCO'2020. **Under review**. Cancun, Mexico: ACM, July 2020.

A.3 NATIONAL CONFERENCES

[1] J. J. Escobar, J. Ortega, J. González, M. Damas y C. Gil. «Acceso eficiente a memoria para selección de características multi-objetivo en GPUs». Español. En: *Actas de las XXVII Jornadas de Paralelismo*. JP'2016. Salamanca, España: Zenodo, septiembre de 2016. DOI: 10.5281/zenodo.3463163.

[2] J. J. Escobar, J. Ortega, A. F. Díaz, J. González y M. Damas. «Consumo Energético y Velocidad en Plataformas CPU-GPU de Algoritmos Paralelos Multi-objetivo para Selección de Características de EEGs». Español. En: *Actas de las XXVIII Jornadas de Paralelismo*. JP'2017. Málaga, España: Zenodo, septiembre de 2017. DOI: 10.5281/zenodo.900339.

[3] J. J. Escobar, J. Ortega, A. F. Díaz, J. González y M. Damas. «Selección Multi-objetivo de Caracteríısticas para Clasificación de EEGs con Paralelismo Multi-nivel en Clusters CPU-GPU». Español. En: *Actas de las XXIX Jornadas de Paralelismo*. JP'2018. Teruel, España: Zenodo, septiembre de 2018. DOI: 10.5281/zenodo.1309243.

[4] J. J. Escobar, J. Ortega, M. Damas, R. Savran Kızıltepe y J. Q. Gan. «Análisis Energía-Tiempo de Redes Neuronales Convolucionales Distribuidas en Clusters Heterogéneos para Clasificación de EEGs». Español. En: *Actas de las XXX Jornadas de Paralelismo*. JP'2019. Cáceres, España: Zenodo, septiembre de 2019. DOI: 10.5281/zenodo.3401528.

B

WATTMETER FOR ENERGY MEASUREMENTS

CONTENTS

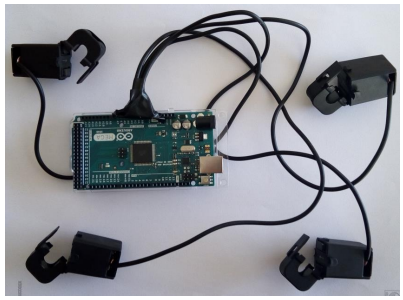
The proposed wattmeter is designed for *Linux* distributions and allows real-time energy measures of one or several platforms. The measurements are made at intervals of one second and determine, for each platform, both instantaneous power $P\,(\mathrm{W})$ and accumulated energy consumption $E\,(\mathrm{W}\cdot\mathrm{h})$. For the experiments carried out in this thesis, one wattmeter has been used for the *cluster* and another one for the desktop PC. Both platforms were described in Section 4.2.

The measurements are made by sensors capable of obtaining the electric current flowing through the cable that supply the platform. Measuring at this point allows to determine its real consumption, including all active components as well as the conversion losses of the power supply. Among its different utilities, measuring energy consumption can be used to identify the energy profile of an application and make the necessary improvements according to the observations.

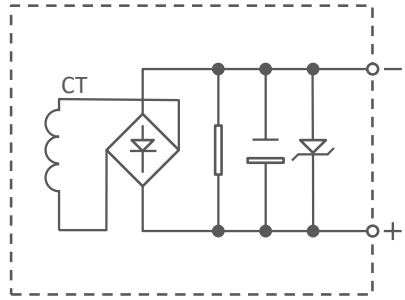
B.1 WATTMETER COMPOSITION

The wattmeter consists of an *Arduino Mega* [139] board and a set of sensors connected directly to the analog inputs. The power cable of each equipment is placed in a different sensor and the clamp is closed to fix it, detecting the current flowing through the cable. Then, the *Arduino* board is connected to the USB port of the platform, which supplies power and allows data to be transmitted through the serial port created in the */dev/ttyACM0* interface. In the case of the *cluster*, it is simply connected to the board to one of the nodes. The *Arduino* obtains four measurements of instantaneous power and energy consumed per second from each sensor, calculates the average, and transmits the result through the serial port at intervals of one second.

USB: Universal Serial Bus



(a) *Arduino Mega board and current sensors*



(b) *Internal diagram of the current sensor*

Figure B.1: Wattmeter based on *Arduino Mega* for energy measurements.

The sensor model used is known as *SCTD010T-5A*, and has been supplied by *YHDC Electronics Co., Ltd*. The sensor is capable of measuring up to 5 A and provides an output voltage between 0 and 5 V. In addition, its accuracy is around $\pm 2\%$ and the supported working temperature is in the range of -20 to $+50°$ Celsius. The analog signals emitted by the sensors are processed by the *Arduino* through its internal 10-bit analog to digital converter. In addition, to take better advantage of its dynamic range, the internal reference of 2.56 V is used, which is only available in the *Mega* model. Although this reduces the measurement to a maximum input of 2.56 V, in practice it allows instantaneous power values of up to 588 W. Since the platforms present lower values, the reduction of the maximum voltage does not represent any limitation when obtaining the measurements. Figures B.1a and B.1b show the wattmeter and the internal diagram of the current sensors, respectively.

B.2 SOFTWARE DESCRIPTION

The software is developed with *Python* and allows different users to obtain energy measurements independently and simultaneously. As the *Arduino* board sends the information to the serial port through USB, it is necessary to share the data received between the different users. For this, the *ZeroMQ*[1] message system is used under the *publish-subscribe* pattern. A master process called *master_meter.py* opens the serial port and activates with *ZeroMQ* a data publishing process through TCP port 5214, in which the received data is retransmitted. Each user who wants to receive data executes a process in *Python* named *show_consumption* to subscribe to the system, so that data is received from *Arduino* in

TCP: Transmission Control Protocol

1 Asynchronous messaging library used in distributed or concurrent applications

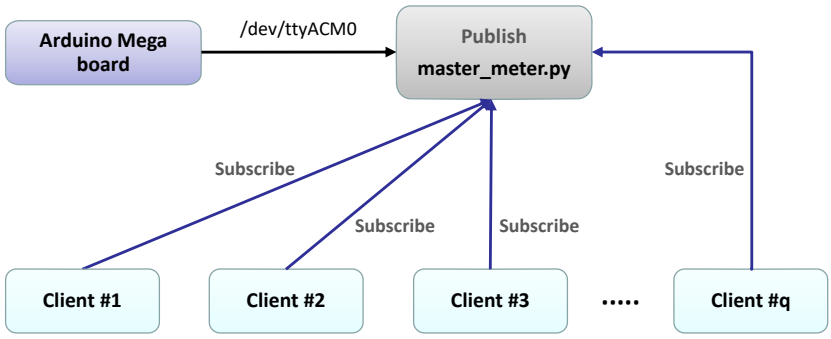


Figure B.2: Scheme of the *publish-subscribe* pattern.

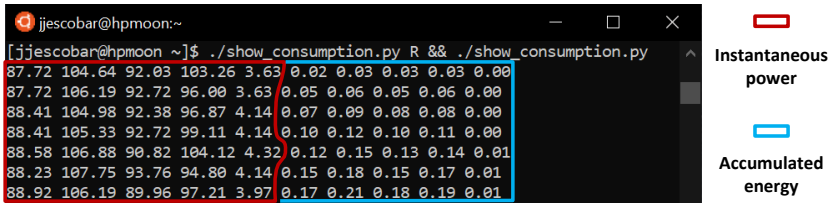


Figure B.3: Energy values obtained by the wattmeter when the *cluster* nodes are in idle state. The first four columns within each color box represent Nodes 1 to 4, while the fifth column correspond to the switch.

real-time. Since multiple processes can be connected to the sending system, all receive the information simultaneously. A general scheme of this process can be seen in Figure B.2.

In addition, each user has in his `$HOME` directory a file with the accumulated energy consumption after the first call to *show_consumption.py*. In the next call, the user will obtain the difference between the current value and the one stored in the file. However, the values can be set to 0 at any time to re-estimate the accumulated energy for a period of time. In the experiments performed in Chapters 5 and 6, the values were always reset before each measurement. To do this, The option `R` of *show_consumption.py* can be used to rewrite the file with the values of that moment. To reset the values and start the measurement with the shortest time interval the following command is used:

```
./show_consumption R && ./show_consumption
```

The output when using the *cluster* nodes can be seen in Figure B.3.

C

GETTING STARTED GUIDE TO HPMOON

CONTENTS

This appendix presents a user guide designed for people interested in using version *ODGA* as it is the most efficient version developed to date. The program can be found as a tool of *e-hpMOBE* project [140] or directly in *Github* repository [141]. Before using the software it is recommended to read the documentation, whose information also is presented in the following sections.

The program provides a subpopulation-based GA for accelerating an EEG classification problem and includes multi-level parallelism to take advantage of parallel architectures such as multi-core CPUs and GPUs. The procedure is mainly developed with MPI to distribute subpopulations among the *cluster* nodes. Moreover, it implements two dynamic scheduling alternatives to evaluate individuals according to the number of existing subpopulations (one or more). Inside of each node, *OpenMP* is used to distribute dynamically either subpopulations or individuals among devices. The *fitness* evaluation of the individuals is performed using *OpenMP* in CPU and *OpenCL* in GPU. This way, by taking into account the devices characteristics, the procedure provides three parallelism levels in CPU and up to four levels in GPU.

C.1 PROGRAM COMPILATION AND USE OF PARAMETERS

To build the project a *Makefile* is provided. The program is compiled by running the following command in a *Unix shell*:

```
make -j N_FEATURES=NF COMP=COMPILER
```

where *NF* is the number of features of the dataset to use (columns), which must be in the range of 4 and the total number of features of the dataset. *COMPILER* set the MPI compiler (`mpic++` by default). The executable file, named *hpmoon*, is generated in the *bin* folder. To run it, in the *shell* the next command must be executed:

```
mpirun --bind-to none --map-by node --host
node1,...,nodeX ./bin/hpmoon -conf config.xml
```

where *config.xml* is the necessary configuration file for the correct performance of the program, which is specified by the option `-conf` and located in the root folder of the project. In addition, the user can indicate separately through command line most of the parameters of the XML file. Table C.1 summarizes the list of parameters, their possible values, and how to use them in the command line. In any case, the special option `-h` displays the available options and examples to use. The option `--map-by node` is mandatory because it is necessary to guarantee that the MPI processes and the nodes are mapped correctly. In the XML file, the information of the devices for each node is sorted according to the MPI process ID. The option `--bind-to none` is also mandatory since the program uses *OpenMP* threads to evaluate the *fitness* of the individuals. This option avoids the mapping of all threads to the same CPU core. On the other hand, the *Makefile* contains a rule to generate *Doxygen* documentation in the *doc/html* folder. This can be done by running the following command:

```
make documentation
```

Finally, the files and documents generated when compiling the project can be deleted. There are two types of cleaning depending on the content to be deleted. The command:

```
make clean
```

deletes the following contents:

- **Binary files**.
- **.o files**.
- $\sim$ **files**.

For a complete cleaning, run the following command:

```
make eraseAll
```

which also removes the following content:

- ***gnuplot* files**.
- **Documentation files generated by *Doxygen***.

The *gnuplot* files contain the *fitness* of the individuals belonging to the *Pareto front* and the necessary source code for the *gnuplot* program.

C.2 THE XML CONFIGURATION FILE

The XML configuration file is required to run the program. Its parameters are read and used at runtime. On the contrary, the parameters for the `make` command are read at compile time to avoid dynamic memory. Table C.1 shows the value ranges of all input parameters, while the XML parameters are described below:

- ***NSubpopulations***: total number of subpopulations.
- ***SubpopulationSize***: size of the subpopulation.
- ***NGlobalMigrations***: number of *migrations* between nodes.
- ***NGenerations***: number of generations in each subpopulation.
- ***MaxFeatures***: maximum number of features initially set to 1.
- ***DataFileName***: name of the file which will contain the *fitness* of the individuals belonging to the *Pareto front*.
- ***PlotFileName***: name of the file which will contain the *gnuplot* code to generate a figure.
- ***ImageFileName***: name of the file which will contain the figure after using the *gnuplot* program to generate it.
- ***TournamentSize***: number of individuals in the tournament step.
- ***NInstances***: number of instances of the dataset to use (rows).
- ***FileName***: name of the file containing the dataset.
- ***Normalize***: if the dataset must be normalized or not.

Table C.1: Value range of the *HPMoon* input parameters and how to use them from the command line (if available).

Parameter	Value range	Command line option
N_FEATURES (NF)	$4 \leq NF \leq N_F$	-
NSubpopulations	$1 \leq N_{Sp}$	-ns
SubpopulationSize	$1 \leq S_S$	-ss
NGlobalMigrations	$1 \leq N_{GM}$	-ngm
NGenerations	$0 \leq g$	-g
MaxFeatures	$1 \leq M_F \leq NF$	-maxf
DataFileName	Valid filename	-plotdata
PlotFileName	Valid filename	-plotsrc
ImageFileName	Valid filename	-plotimg
TournamentSize	$2 \leq TS$	-ts
NInstances	$4 \leq NI \leq N_P$	-trni
FileName	Existing dataset	-trdb
Normalize	1 or 0	-trnorm
NDevices	$0 \leq N_D$	-
Names	Existing device name	-
ComputeUnits	$1 \leq \lambda$	-
WiLocal	$1 \leq WL \leq 1,024$	-
CpuThreads	N_T	-
KernelsFileName	Existing kernel file	-ke
Display usage	-	-h
List devices	-	-l

- ***NDevices***: number of *OpenCL* devices that will run the program in a specific node.
- ***Names***: name of each previous *OpenCL* device. The values must be separated by commas and in the same order than their corresponding devices.
- ***ComputeUnits***: number of CUs (λ) for each previous *OpenCL* device. The values must be separated by commas and in the same order than their corresponding devices.
- ***WiLocal***: number of *work-items* per CU for each previous *OpenCL* device. The values must be separated by commas and in the same order than their corresponding devices.

- ***CpuThreads***: number of *OpenMP* threads to use in CPU (N_T) during the *fitness* evaluation. If this parameter is set to 1 and *NDevices* to 0, the program will run in a sequential mode.
- ***KernelsFileName***: name of the file containing the GPU kernel.

C.3 CONSIDERATIONS FOR USE

The following points should be considered to obtain good performance when running the program:

The evaluation function for each individual is parallelized in GPU with *OpenCL*. A CU evaluates one individual and it is composed by *WiLocal work-items*. Thus, *WiLocal* should be a multiple of 32 (*warp*) to improve performance. The user can approximate the optimal value of *WiLocal*. This value is calculated as the total number of PEs divided by the number of CUs. E.g., the GPU *NVIDIA GeForce GTX 1080* has 2,560 PEs and 20 CUs, so that $\frac{2,560}{20} = 128$ *work-items*. However, for this program it is recommended to use 1,024 to hide latencies. In CPU, *CpuThreads* should have a value equal to the number of logical cores.

The NDS step of NSGA-II contains a loop of quadratic time complexity that is related to the number of individuals. For good quality results it is not necessary to increase the number of individuals so much. The best option is to increase the number of generations or subpopulations.

On GPU, the program performs better with values of *N_FEATURES* and *NInstances* higher than the number of *work-items*. However, the dataset is stored into local memory and its capacity is very limited (approximately 49 KB depending on the device). The program will abort if the dataset is too large.

If multiple devices are specified, and only one subpopulation is present in the node, the evaluation of individuals is distributed dynamically among the devices by using *OpenMP* `pragmas`. Each *OpenMP* thread handles one device, which computes chunks of individuals equal to their number of CUs or CPU threads until all individuals are evaluated.

Only one MPI process is mandatory to run the program. This is the best scenario for single-computer systems. However, the program can be run using more MPI processes. In this case, the MPI process #0 distributes subpopulations among the workers (nodes), which are the MPI processes #1, #2, and so on. This situation should be considered when the program is run on a cluster containing multiple nodes.

D

GRANTS AND SPECIAL ACKNOWLEDGEMENTS

This thesis has been supported by the following projects and grants:

- National research project **TIN2012-32039**, funded by the Spanish MINECO: *Optimización Multiobjetivo de Altas Prestaciones y Aplicaciones en Neuroingeniería y Técnicas para Rehabilitación*. In English: *High-performance Multi-objective Optimization and Applications in Neuroengineering and Rehabilitation Techniques*.

MINECO: MINistry of Economy and COmpetitiveness

- National research project **TIN2015-67020-P**, funded by the Spanish MINECO and ERDF funds: *Optimización Multiobjetivo de Altas Prestaciones y Energéticamente Eficiente en Arquitecturas de Computador Heterogéneas. Aplicaciones en Ingeniería Biomédica*. In English: *Energy-efficient and High-performance Multi-objective Optimization in Heterogeneous Computer Architectures. Biomedical Engineering Applications*.

ERDF: European Regional Development Fund

- National research project **PGC2018-098813-B-C31**, funded by the Spanish MICIU and ERDF funds: *Nuevos Paradigmas de Cómputo y Arquitecturas Heterogéneas Paralelas para la Mejora en Velocidad y Energía de Tareas de Optimización y Clasificación en Aplicaciones Biomédicas*. In English: *New Paradigms of Computation and Parallel Heterogeneous Architectures for the Improvement in Speed and Energy of Tasks of Optimization and Classification in Biomedical Applications*.

MICIU: MInistry of sCience, Innovation, and Universities

- *NVIDIA* grant program, which has donated two GPUs *TITAN Xp*.

The following people are also thanked for their collaborative work:

- Special thanks to **Dr. Antonio F. Díaz** and **Mr. Francisco M. Illeras**, both from Department of Computer Architecture and Technology of University of Granada, for providing the wattmeter and the constant technical support in the *HPMoon cluster*.

- Thanks to the BCI laboratory of the University of Essex, especially **Dr. John Q. Gan**, for allowing the use of their EEG datasets.

BIBLIOGRAPHY

[1] Statista Online Portal. *Volume of data/information created worldwide from 2010 to 2025 (in zetabytes)*. URL: https://www.statista.com/statistics/871513/worldwide-data-created/ (visited on 10/18/2019) (Cited on pages 4, 5).

[2] A. Mukhopadhyay, U. Maulik, S. Bandyopadhyay, and C. A. Coello Coello. "A Survey of Multiobjective Evolutionary Algorithms for Data Mining: Part I". In: *IEEE Transactions on Evolutionary Computation* 18.1 (2014), pp. 4–19. DOI: 10.1109/TEVC.2013.2290086 (Cited on page 4).

[3] A. Mukhopadhyay, U. Maulik, S. Bandyopadhyay, and C. A. Coello Coello. "A Survey of Multiobjective Evolutionary Algorithms for Data Mining: Part II". In: *IEEE Transactions on Evolutionary Computation* 18.1 (2014), pp. 20–35. DOI: 10.1109/TEVC.2013.2290082 (Cited on page 4).

[4] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. "A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II". In: *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*. PPSN'2000. Paris, France: Springer, September 2000, pp. 849–858. DOI: 10.1007/3-540-45356-3_83 (Cited on pages 6, 48).

[5] G. E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (1965), pp. 114–117 (Cited on page 12).

[6] M. M. Waldrop. "The chips are down for Moore's law". In: *Nature News* 530.7589 (2016), pp. 144–147. DOI: 10.1038/530144a (Cited on page 12).

[7] R. W. Keyes. "Physical limits of silicon transistors and circuits". In: *Reports on Progress in Physics* 68.12 (2005), pp. 2701–2746. DOI: 10.1088/0034-4885/68/12/r01 (Cited on page 12).

[8] National Academies of Sciences, Engineering, and Medicine. *Quantum Computing: Progress and Prospects*. National Academies Press, 2019 (Cited on page 12).

[9] N. H. T. Ghany, S. A. Elsherif, and H. T. Handal. "Revolution of Graphene for different applications: State-of-the-art". In: *Surfaces and Interfaces* 9 (2017), pp. 93–106. DOI: 10.1016/j.surfin.2017.08.004 (Cited on page 12).

[10] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* 21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071 (Cited on pages 13, 14).

[11] OpenMP Community. *OpenMP specifications*. URL: http://www.openmp.org/specifications/ (visited on 11/21/2016) (Cited on page 19).

[12] The Open MPI Project. *OpenMPI documentation*. URL: https://www.open-mpi.org/doc/ (visited on 12/03/2019) (Cited on page 19).

[13] J. D. Owens et al. "A Survey of General-Purpose Computation on Graphics Hardware". In: *Computer Graphics Forum* 26.1 (2007), pp. 80–113. DOI: 10.1111/j.1467-8659.2007.01012.x (Cited on page 20).

[14] Khronos Group. *Khronos OpenCL Registry*. URL: https://www.khronos.org/registry/cl/ (visited on 11/30/2015) (Cited on pages 20, 22).

[15] S. Mittal and J. S. Vetter. "A Survey of CPU-GPU Heterogeneous Computing Techniques". In: *ACM Computing Surveys* 47.4 (2015), 69:1–69:35. DOI: 10.1145/2788396 (Cited on page 26).

[16] TOP500 list. *The 500 Most Powerful Supercomputers*. URL: https://www.top500.org/lists/top500/ (visited on 12/15/2019) (Cited on pages 26, 27).

[17] K. Raju and N. C. Niranjan. "A survey on techniques for cooperative CPU-GPU computing". In: *Sustainable Computing: Informatics and Systems* 19 (2018), pp. 72–85. DOI: 10.1016/j.suscom.2018.07.010 (Cited on page 26).

[18] IEEE 802.15.1-2002. *IEEE Standard for Telecommunications and Information Exchange Between Systems - LAN/MAN - Specific Requirements - Part 15: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs)*. URL: https://standards.ieee.org/standard/802_15_1-2002.html (visited on 10/30/2019) (Cited on page 26).

[19] Green500 list. *The 500 Most Energy-efficient Supercomputers*. URL: https://www.top500.org/green500/ (visited on 12/16/2019) (Cited on page 27).

[20] G. Teodoro, T. Kurc, G. Andrade, J. Kong, R. Ferreira, and J. Saltz. "Application performance analysis and efficient execution on systems with multi-core CPUs, GPUs and MICs: a case study with microscopy image analysis". In: *The International Journal of High Performance Computing Applications* 31.1 (2015), pp. 32–51. DOI: 10.1177/1094342015594519 (Cited on page 28).

[21] Y. Wang and N. Ranganathan. "An Instruction-Level Energy Estimation and Optimization Methodology for GPU". In: *Proceedings of the 11th International Conference on Computer and Information Technology*. CIT'2011. Paphos, Cyprus: IEEE, August 2011, pp. 621–628. DOI: 10.1109/CIT.2011.69 (Cited on page 28).

[22] J. M. Cebrín, G. D. Guerrero, and J. M. García. "Energy Efficiency Analysis of GPUs". In: *Proceedings of the 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IPDPSW'2012. Shanghai, China: IEEE, May 2012, pp. 1014–1022. DOI: 10.1109/IPDPSW.2012.124 (Cited on page 28).

[23] S. Mittal and J. S Vetter. "A Survey of Methods for Analyzing and Improving GPU Energy Efficiency". In: *ACM Computing Surveys* 47.2 (2014), 19:1–19:23. DOI: 10.1145/2636342 (Cited on page 28).

[24] C. Gregg and K. Hazelwood. "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer". In: *Proceedings of the 2015 International Symposium on Performance Analysis of Systems and Software*. ISPASS'2011. Austin, TX, USA: IEEE, April 2011, pp. 134–144. DOI: 10.1109/ISPASS.2011.5762730 (Cited on page 28).

[25] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso. "A Survey of Performance Modeling and Simulation Techniques for Accelerator-Based Computing". In: *IEEE Transactions on Parallel and Distributed Systems* 26.1 (2015), pp. 272–281. DOI: 10.1109/TPDS.2014.2308216 (Cited on page 28).

[26] S. Hong and H. Kim. "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness". In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA'2009. New York, NY, USA: ACM, June 2009, pp. 152–163. DOI: 10.1145/1555754.1555775 (Cited on page 28).

[27] T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee. "A Performance Model for GPUs with Caches". In: *IEEE Transactions on Parallel and Distributed Systems* 26.7 (2015), pp. 1800–1813 (Cited on page 28).

[28] A. Gainaru, E. Slusanschi, and S. Trausan-Matu. "Mapping Data Mining Algorithms on a GPU Architecture: A Study". In: *Proceedings of the 19th International Symposium. Foundations of Intelligent Systems*. ISMIS'2011. Warsaw, Poland: Springer, June 2011, pp. 102–112. DOI: 10.1007/978-3-642-21916-0_12 (Cited on page 28).

[29] J. Hestness, S. W. Keckler, and D. A. Wood. "GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors". In: *Proceedings of the 2015 International Symposium on Workload Characterization*. IISWC'2015. Atlanta, GA, USA: IEEE, October 2015, pp. 87–97. DOI: 10.1109/IISWC.2015.15 (Cited on page 28).

[30] T. Allen and R. Ge. "Characterizing Power and Performance of GPU Memory Access". In: *Proceedings of the 4th International Workshop on Energy Efficient Supercomputing*. E2SC'2016. Salt Lake City, UT, USA: IEEE Press, November 2016, pp. 46–53. DOI: 10.1109/E2SC.2016.8 (Cited on page 29).

[31] A. Marowka. "Energy Consumption Modeling for Hybrid Computing". In: *Proceedings of the 18th International Conference on Parallel Processing, Euro-Par 2012*. Euro-Par'2012. Rhodes Island, Greece: Springer, August 2012, pp. 54–64. DOI: 10.1007/978-3-642-32820-6_8 (Cited on page 29).

[32] E. M. Garzón, J. J Moreno, and J. A. Martínez. "An approach to optimise the energy efficiency of iterative computation on integrated GPU–CPU systems". In: *The Journal of Supercomputing* 73.1 (2017), pp. 114–125. DOI: 10.1007/s11227-016-1643-9 (Cited on page 29).

[33] B. Pérez, E. Stafford, J. L Bosque, and R. Beivide. "Energy efficiency of load balancing for data-parallel applications in heterogeneous systems". In: *The Journal of Supercomputing* 73.1 (2017), pp. 330–342. DOI: 10.1007/s11227-016-1864-y (Cited on page 29).

[34] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang. "GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures". In: *Proceedings of the 41st International Conference on Parallel Processing*. ICPP'2012. Pittsburgh, PA, USA: IEEE, September 2012, pp. 48–57. DOI: 10.1109/ICPP.2012.31 (Cited on page 29).

[35] D. De Sensi. "Predicting Performance and Power Consumption of Parallel Applications". In: *Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. PDP'2016. Heraklion Crete, Greece: IEEE, February 2016, pp. 200–207. DOI: 10.1109/PDP.2016.41 (Cited on pages 29, 96).

[36] K. Pruhs, R. Stee, and P. Uthaisombut. "Speed Scaling of Tasks with Precedence Constraints". In: *Theory of Computing Systems* 43.1 (2008), pp. 67–80. DOI: 10.1007/s00224-007-9070-1 (Cited on page 29).

[37] G. L. Valentini et al. "An overview of energy efficiency techniques in cluster computing systems". In: *Cluster Computing* 16.1 (2013), pp. 3–15. DOI: 10.1007/s10586-011-0171-x (Cited on pages 29, 30).

[38] E. Rotem, U. C. Weiser, A. Mendelson, R. Ginosar, E. Weissmann, and Y. Aizik. "H-EARtH: Heterogeneous Multicore Platform Energy Management". In: *IEEE Computer Magazine* 49.10 (2016), pp. 47–55. DOI: 10.1109/MC.2016.309 (Cited on page 29).

[39] S. Nesmachnow, B. Dorronsoro, J. E. Pecero, and P. Bouvry. "Energy-Aware Scheduling on Multicore Heterogeneous Grid Computing Systems". In: *Journal of Grid Computing* 11.4 (2013), pp. 653–680. DOI: 10.1007/s10723-013-9258-3 (Cited on pages 29, 30).

[40] J. Balladini, R. Suppi, D. Rexachs, and E. Luque. "Impact of parallel programming models and CPUs clock frequency on energy consumption of HPC systems". In: *Proceedings of the 9th International Conference on Computer Systems and Applications*. AICCSA'2011. Sharm El-Sheikh, Egypt: IEEE, December 2011, pp. 16–219. DOI: 10.1109/AICCSA.2011.6126618 (Cited on page 30).

[41] L. Ponciano, A. Brito, L. Sampaio, and F. Brasileiro. "Energy Efficient Computing through Productivity-Aware Frequency Scaling". In: *Proceedings of the 2nd International Conference on Cloud and Green Computing*. CGC'2012. Xiangtan, China: IEEE, November 2012, pp. 191–198. DOI: 10.1109/CGC.2012.59 (Cited on page 30).

[42] R. Barik, N. Farooqui, B. T. Lewis, C. Hu, and T. Shpeisman. "A black-box approach to energy-aware scheduling on integrated CPU-GPU systems". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO'2016. Barcelona, Spain: ACM, March 2016, pp. 70–81. DOI: 10.1145/2854038.2854052 (Cited on page 30).

[43] S. Hong and H. Kim. "An Integrated GPU Power and Performance Model". In: *SIGARCH Computer Architecture News* 38.3 (2010), pp. 280–289. DOI: 10.1145/1816038.1815998 (Cited on page 30).

[44] R. Ge, X. Feng, M. Burtscher, and Z. Zong. "PEACH: A Model for Performance and Energy Aware Cooperative Hybrid Computing". In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. CF'2014. Cagliari, Italy: ACM, May 2014, 24:1–24:2. DOI: 10.1145/2597917.2597948 (Cited on page 30).

[45] J. I. Aliaga, M. Barreda, M. F. Dolz, A. F. Martín, R. Mayo, and E. S. Quintana-Ortí. "Assessing the impact of the CPU power-saving modes on the task-parallel solution of sparse linear systems". In: *Cluster Computing* 17.4 (2014), pp. 1335–1348. DOI: 10.1007/s10586-014-0402-z (Cited on pages 30, 109).

[46] Y. Zhang, X. Hu, and D. Z. Chen. "Task Scheduling and Voltage Selection for Energy Minimization". In: *Proceedings of the 39th Annual Design Automation Conference*. DAC'2002. New Orleans, LA, USA: ACM, June 2002, pp. 183–188. DOI: 10.1145/513918.513966 (Cited on page 30).

[47] G. Park, B. Shirazi, J. Marquis, and H. Choo. "Decisive path scheduling: a new list scheduling method". In: *Proceedings of the 1997 International Conference on Parallel Processing*. ICPP'1997. Bloomington, IL, USA: IEEE, August 1997, pp. 472–480. DOI: 10.1109/ICPP.1997.622682 (Cited on page 30).

[48] S. Baskiyar and R. Abdel-Kader. "Energy aware DAG scheduling on heterogeneous systems". In: *Cluster Computing* 13.4 (2010), pp. 373–383. DOI: 10.1007/s10586-009-0119-6 (Cited on page 30).

[49] Y. C. Lee and A. Y. Zomaya. "Energy Conscious Scheduling for Distributed Computing Systems Under Different Operating Conditions". In: *IEEE Transactions on Parallel and Distributed Systems* 22.8 (2011), pp. 1374–1381. DOI: 10.1109/TPDS.2010.208 (Cited on pages 31, 98).

[50] B. Dorronsoro, S. Nesmachnow, J. Taheri, A. Y. Zomaya, E. Talbi, and P. Bouvry. "A hierarchical approach for energy-efficient scheduling of large workloads in multicore distributed systems". In: *Sustainable Computing: Informatics and Systems* 4.4 (2014), pp. 252–261. DOI: 10.1016/j.suscom.2014.08.003 (Cited on page 31).

[51] J. Taheri, A. Y. Zomaya, and S. U. Khan. *Grid simulation tools for Job Scheduling and Datafile Replication in Scalable Computing and Communications: Theory and Practice*. Wiley, 2013 (Cited on page 31).

[52] K. O'brien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou. "A Survey of Power and Energy Predictive Models in HPC Systems and Applications". In: *ACM Computing Surveys* 50.3 (2017), 37:1–37:38. DOI: 10.1145/3078811 (Cited on page 31).

[53] R. Mishra, N. Rastogi, D. Zhu, D. Mosse, and R. Melhem. "Energy aware scheduling for distributed real-time systems". In: *Proceedings of the 17th International Parallel and Distributed Processing Symposium*. IPDPS'2003. Nice, France: IEEE, April 2003, pp. 9–16. DOI: 10.1109/IPDPS.2003.1213099 (Cited on page 31).

[54] Center for Machine Learning and Intelligent Systems. *Machine Learning Repository*. URL: https://archive.ics.uci.edu/ml/datasets.php (visited on 11/11/2019) (Cited on page 33).

[55] N. Forbes. *Imitation of Life: How Biology is Inspiring Computing*. MIT Press, 2005 (Cited on page 33).

[56] P. E. Hart, N. J. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136 (Cited on page 34).

[57] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957 (Cited on page 34).

[58] I. Boussaïd, J. Lepagnot, and P. Siarry. "A Survey on Optimization Metaheuristics". In: *Information Sciences* 237.2 (2013), pp. 82–117. DOI: 10.1016/j.ins.2013.02.041 (Cited on page 35).

[59] Y. Sawaragi, H. Nakayama, and T. Tanino. *Theory of Multiobjective Optimization*. Academic Press, 1985 (Cited on page 38).

[60] J. A. Richards. "Supervised Classification Techniques". In: *Remote Sensing Digital Image Analysis: An Introduction*. Springer, 2013, pp. 247–318. DOI: 10.1007/978-3-642-30062-2_8 (Cited on page 38).

[61] J. A. Richards. "Clustering and Unsupervised Classification". In: *Remote Sensing Digital Image Analysis: An Introduction*. Springer, 2013, pp. 319–341. DOI: 10.1007/978-3-642-30062-2_9 (Cited on page 39).

[62] S. Lloyd. "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: 10.1109/TIT.1982.1056489 (Cited on page 39).

[63] D. Pelleg and A. Moore. "Accelerating Exact K-means Algorithms with Geometric Reasoning". In: *Proceedings of the 5th International Conference on Knowledge Discovery and Data Mining*. KDD'1999. San Diego, CA, USA: ACM, August 1999, pp. 277–281. DOI: 10.1145/312129.312248 (Cited on page 39).

[64] E. Forgy. "Cluster Analysis of Multivariate Data: Efficiency versus Interpretability of Classification". In: *Biometrics* 21.3 (1965), pp. 768–769 (Cited on page 39).

[65] A. Vattani. "k-means Requires Exponentially Many Iterations Even in the Plane". In: *Discrete & Computational Geometry* 45.4 (2011), pp. 596–616. DOI: 10.1007/s00454-011-9340-1 (Cited on page 40).

[66] R. Rupp, S. C. Kleih, R. Leeb, J. del R. Millan, A. Kübler, and G. R. Müller-Putz. "Brain-Computer Interfaces and Assistive Technology". In: *Brain-Computer-Interfaces in their Ethical, Social and Cultural Contexts*. Springer, 2014, pp. 7–38. DOI: 10.1007/978-94-017-8996-7_2 (Cited on page 41).

[67] J. S. Brumberg, J. D. Burnison, and K. M. Pitt. "Using Motor Imagery to Control Brain-Computer Interfaces for Communication". In: *Proceedings of the 10th International Conference on Augmented Cognition*. AC'2016. Toronto, Canada: Springer, July 2016, pp. 14–25. DOI: 10.1007/978-3-319-43659-3_9 (Cited on page 41).

[68] R. Wei, X. H. Zhang, X. Dang, and G. H. Li. "Classification for Motion Game Based on EEG Sensing". In: *ITM Web Conferences* 11 (2017). DOI: 10.1051/itmconf/20171105002 (Cited on page 41).

[69] N. Birbaumer and L. G. Cohen. "Brain-computer interfaces: communication and restoration of movement in paralysis". In: *The Journal of Physiology* 579.3 (2007), pp. 621–636. DOI: 10.1113/jphysiol.2006.125633 (Cited on page 41).

[70] R. E. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961 (Cited on page 41).

[71] R. P. W Duin. "Classifiers in almost empty spaces". In: *Proceedings of the 15th International Conference on Pattern Recognition*. ICPR'2000. Barcelona, Spain: IEEE, September 2000, pp. 1–7. DOI: 10.1109/ICPR.2000.906006 (Cited on page 41).

[72] G. Pfurtscheller. "EEG event-related desynchronization (ERD) and synchronization (ERS)". In: *Electroencephalography and Clinical Neurophysiology* 103.1 (1997), p. 26 (Cited on page 41).

[73] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing, 1989 (Cited on page 43).

[74] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, 1992 (Cited on page 45).

[75] K. Deb and A. Kumar. "Real-coded genetic algorithms with simulated binary crossover: Studies on multimodel and multiobjective problems". In: *Complex Systems* 9.6 (1995), pp. 431–454 (Cited on page 45).

[76] N. Srinivas and K. Deb. "Muiltiobjective Optimization Using Nondominated Sorting in Genetic Algorithms". In: *Evolutionary Computation* 2.3 (1994), pp. 221–248. DOI: 10.1162/evco.1994.2.3.221 (Cited on page 47).

[77] K. Deb and H. Jain. "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints". In: *IEEE Transactions on Evolutionary Computation* 18.4 (2014), pp. 577–601. DOI: 10.1109/TEVC.2013.2281535 (Cited on page 48).

[78] Y. Gong et al. "Distributed evolutionary algorithms and their models: A survey of the state-of-the-art". In: *Applied Soft Computing* 34.C (2015), pp. 286–300. DOI: 10.1016/j.asoc.2015.04.061 (Cited on page 49).

[79] G. Luque and E. Alba. *Parallel Genetic Algorithms: Theory and Real World Applications*. Springer, 2013 (Cited on page 49).

[80] D. Kimovski, J. Ortega, A. Ortiz, and R. Baños. "Parallel alternatives for evolutionary multi-objective optimization in unsupervised feature selection". In: *Expert Systems with Applications* 42.9 (2015), pp. 4239–4252. DOI: 10.1016/j.eswa.2015.01.061 (Cited on page 50).

[81] T. V. Luong, N. Melab, and E. Talbi. "GPU-based Island Model for Evolutionary Algorithms". In: *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. GECCO'2010. Portland, OR, USA: ACM, July 2010, pp. 1089–1096. DOI: 10.1145/1830483.1830685 (Cited on page 51).

[82] M. Caudill. "Neural Networks Primer: Part I". In: *AI Expert* 2.12 (1987), pp. 46–52. DOI: 10.1007/s00454-011-9340-1 (Cited on page 51).

[83] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Representations by Back-propagating Errors". In: *Neurocomputing: Foundations of Research*. MIT Press, 1986, pp. 696–699 (Cited on page 53).

[84] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791 (Cited on page 53).

[85] C. Emmanouilidis, A. Hunter, and J. MacIntyre. "A Multiobjective Evolutionary Setting for Feature Selection and a Commonality-based Crossover Operator". In: *Proceedings of the 2000 Congress on Evolutionary Computation*. Vol. 1. CEC'2000. La Jolla, CA, USA: IEEE, July 2000, pp. 309–316. DOI: 10.1109/CEC.2000.870311 (Cited on page 54).

[86] J. Handl and J. Knowles. "Feature Subset Selection in Unsupervised Learning via Multiobjective Optimization". In: *International Journal of Computational Intelligence Research* 2.3 (2006), pp. 217–238 (Cited on page 54).

[87] L. S. Oliveira, R. Sabourin, F. Bortolozzi, and C. Y. Suen. "A Methodology for Feature Selection Using Multiobjective Genetic Algorithms for Handwritten Digit String Recognition". In: *International Journal of Pattern Recognition and Artificial Intelligence* 6.1 (2003), pp. 903–929. DOI: 10.1142/S021800140300271X (Cited on page 54).

[88] Y. Kim, W. N. Street, and F. Menczer. "Feature Selection in Unsupervised Learning via Evolutionary Search". In: *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining*. KDD'2000. Boston, MA, USA: ACM, April 2000, pp. 365–369. DOI: 10.1145/347090.347169 (Cited on page 54).

[89] I. Mierswa and M. Wurst. "Information Preserving Multi-objective Feature Selection for Unsupervised Learning". In: *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*. GECCO'2006. Seattle, WA, USA: ACM, 2006, pp. 1545–1552. DOI: 10.1145/1143997.1144248 (Cited on page 54).

[90] M. Morita, R. Sabourin, F. Bortolozzi, and C. Y. Suen. "Unsupervised Feature Selection using Multi-objective Genetic Algorithms for Handwritten Word Recognition". In: *Proceedings of the Seventh International Conference on Document Analysis and Recognition*. ICDAR'2013. IEEE, August 2003, pp. 666–670. DOI: 10.1109/ICDAR.2003.1227746 (Cited on page 54).

[91] I. Guyon and A. Elisseeff. "An Introduction to Variable and Feature Selection". In: *The Journal of Machine Learning Research* 3 (2003), pp. 1157–1182 (Cited on page 54).

[92] M. Charikar, V. Guruswami, R. Kumar, S. Rajagopalan, and A. Sahai. "Combinatorial feature selection problems". In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. FOCS'2000. Redondo Beach, CA, USA: IEEE, November 2000, pp. 631–640. DOI: 10.1109/SFCS.2000.892331 (Cited on page 54).

[93] J. T. de Souza, S. Matwin, and N. Japkowicz. "Parallelizing Feature Selection". In: *Algorithmica* 45.3 (2006), pp. 433–456. DOI: 10.1007/s00453-006-1220-3 (Cited on page 54).

[94] E. Alba, G. Luque, and S. Nesmachnow. "Parallel Metaheuristics: Recent Advances and New Trends". In: *International Transactions in Operational Research* 20.1 (2013), pp. 1–48. DOI: 10.1111/j.1475-3995.2012.00862.x (Cited on page 54).

[95] P. Fazendeiro, C. Padole, P. Sequeira, and P. Prata. "OpenCL Implementations of a Genetic Algorithm for Feature Selection in Periocular Biometric Recognition". In: *Third International Conference on Swarm, Evolutionary and Memetic Computing*. SEMCCO'2012.

Bhubaneswar, India: Springer, December 2012, pp. 729–737. DOI: 10.1007/978-3-642-35380-2_85 (Cited on page 54).

[96] L. Jian, C. Wang, Y. Liu, S. Liang, W. Yi, and Y. Shi. "Parallel Data Mining Techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA)". In: *The Journal of Supercomputing* 64.3 (2013), pp. 942–967. DOI: 10.1007/s11227-011-0672-7 (Cited on page 54).

[97] M. Marinaki and Y. Marinakis. "An Island Memetic Differential Evolution Algorithm for the Feature Selection Problem". In: *Proceedings of the 6th International Workshop on Nature Inspired Cooperative Strategies for Optimization*. NICSO'2013. Canterbury, UK: Springer, September 2014, pp. 29–42. DOI: 10.1007/978-3-319-01692-4_3 (Cited on page 54).

[98] S. Limmer and D. Fey. "Comparison of common parallel architectures for the execution of the island model and the global parallelization of evolutionary algorithms". In: *Concurrency and Computation: Practice and Experience* 29.9 (2017). DOI: 10.1002/cpe.3797 (Cited on page 54).

[99] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001 (Cited on page 54).

[100] D. Kimovski, J. Ortega, A. Ortiz, and R. Baños. "Leveraging Cooperation for Parallel Multi-objective Feature Selection in High-Dimensional EEG Data". In: *Concurrency and Computation: Practice and Experience* 27.18 (2015), pp. 5476–5499. DOI: 10.1002/cpe.3594 (Cited on page 54).

[101] F. Fernández-de-Vega et al. "A Cross-Platform Assessment of Energy Consumption in Evolutionary Algorithms". In: *Proceedings of the 14th International Conference on Parallel Problem Solving from Nature*. PPSN'2016. Edinburgh, UK: Springer, September 2016, pp. 548–557. DOI: 10.1007/978-3-319-45823-6_51 (Cited on page 54).

[102] C. A. Coello Coello and M. Sierra. "A Study of the Parallelization of a Coevolutionary Multi-objective Evolutionary Algorithm". In: *Proceedings of the 3rd Mexican International Conference on Artificial Intelligence*. MICAI'2004. Mexico City, Mexico: Springer, April 2004, pp. 688–697. DOI: 10.1007/978-3-540-24694-7_71 (Cited on page 54).

[103] P. Vidal, E. Alba, and F. Luna. "Solving Optimization Problems Using a Hybrid Systolic Search on GPU Plus CPU". In: *Soft Computing* 21.12 (2017), pp. 3227–3245. DOI: 10.1007/s00500-015-2005-x (Cited on page 54).

[104] P. Collet. "Why GPGPUs for Evolutionary Computation?" In: *Massively Parallel Evolutionary Computation on GPGPUs*. Springer, 2013, pp. 3–14. DOI: 10.1007/978-3-642-37959-8_1 (Cited on page 54).

[105] P. Jähne. "Overview of the current state of research on parallelisation of evolutionary algorithms on graphic cards". In: *GI-Jahrestagung*. INFORMATIK'2016. Bonn, Germany: LNI, September 2016, pp. 2163–2174 (Cited on page 54).

[106] S. Debattisti, N. Marlat, L. Mussi, and S. Cagnoni. "Implementation of simple genetic algorithm within the CUDA architecture". In: *In GPUs for Genetic and Evolutionary Computation*. GECCO'2009. Montreal, CAN, USA: ACM, July 2008 (Cited on page 54).

[107] P. Pospichal, J. Jaros, and J. Schwarz. "Parallel Genetic Algorithm on the CUDA Architecture". In: *Proceedings of the 13th European Conference on the Applications of Evolutionary Computation*. EvoApplications'2010. Istambul, Turkey: Springer, April 2010, pp. 442–451. DOI: 10.1007/978-3-642-12239-2_46 (Cited on page 55).

[108] D. Sharma and P. Collet. "Implementation Techniques for Massively Parallel Multi-objective Optimization". In: *Massively Parallel Evolutionary Computation on GPGPUs*. Springer, 2013, pp. 267–286. DOI: 10.1007/978-3-642-37959-8_13 (Cited on page 55).

[109] J. J. Moreno, G. Ortega, E. Filatovas, J. A. Martínez, and E. M. Garzón. "Improving the Energy Efficiency of Evolutionary Multiobjective Algorithms". In: *Proceedings of the 16th International Conference on Algorithms and Architectures for Parallel Processing*. ICA3PP'2016. Granada, Spain: Springer, December 2016, pp. 62–75. DOI: 10.1007/978-3-319-49956-7_5 (Cited on page 55).

[110] M. L. Wong and G. Cui. "Data Mining Using Parallel Multiobjective Evolutionary Algorithms on Graphics Processing Units". In: *Massively Parallel Evolutionary Computation on GPGPUs*. Springer, 2013, pp. 287–307. DOI: 10.1007/978-3-642-37959-8_14 (Cited on page 55).

[111] P. P. Baramkar and D. B. Kulkarni. "Review for K-Means On Graphics Processing Units (GPU)". In: *International Journal of Engineering Research & Technology* 3.6 (2014), pp. 1911–1914 (Cited on page 55).

[112] E. Kijsipongse and S. U-ruekolan. "Dynamic Load Balancing on GPU Clusters for Large-scale K-Means Clustering". In: *Proceedings of the 9th International Joint Conference on Computer Science and Software Engineering*. JCSSE'2012. Bangkok, Thailand: IEEE,

May 2012, pp. 346–350. DOI: 10.1109/JCSSE.2012.6261977 (Cited on page 55).

[113] F. Farivar, D. Rebolledo, E. Chan, and R. Campbell. "A Parallel Implementation of K-Means Clustering on GPUs". In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*. PDPTA'2008. Las Vegas, NV, USA: CSREA Press, July 2008, pp. 340–345 (Cited on page 55).

[114] R. Wu, B. Zhang, and M. Hsu. "Clustering Billions of Data Points using GPUs". In: *Proceedings of the Combined Workshops on UnConventional High Performance Computing workshop plus Memory Access Workshop*. UCHPC-MAW'2009. Ischia, Italy: ACM, May 2009, pp. 1–6. DOI: 10.1145/1531666.1531668 (Cited on page 55).

[115] M. Zechner and M. Granitzer. "Accelerating K-Means on the Graphics Processor via CUDA". In: *Proceedings of the First International Conference on Intensive Applications and Services*. INTENSIVE'2009. Valencia, Spain: IEEE, April 2009, pp. 7–15. DOI: 10.1109/INTENSIVE.2009.19 (Cited on page 55).

[116] F. Lotte et al. "A review of classification algorithms for EEG-based brain–computer interfaces: a 10 year update". In: *Journal of Neural Engineering* 15.3 (2018). DOI: 10.1088/1741-2552/aab2f2 (Cited on page 55).

[117] J. Ortega, J. Asensio-Cubero, J. Q. Gan, and A. Ortiz. "Classification of motor imagery tasks for BCI with multiresolution analysis and multiobjective feature selection". In: *BioMedical Engineering OnLine* 15.1 (2016), pp. 149–164. DOI: 10.1186/s12938-016-0178-x (Cited on page 55).

[118] Y. R. Tabar and U. Halici. "A novel deep learning approach for classification of EEG motor imagery signals". In: *Journal of Neural Engineering* 14.1 (2016). DOI: 10.1088/1741-2560/14/1/016003 (Cited on page 55).

[119] J. Ortega, A. Ortiz, P. Martín-Smith, J. Q. Gan, and J. González. "Deep Belief Networks and Multiobjective Feature Selection for BCI with Multiresolution Analysis". In: *Proceedings of the 14th International Work-Conference on Artificial Neural Networks*. IWANN'2017. Cádiz, Spain: Springer, June 2017, pp. 28–39. DOI: 10.1007/978-3-319-59153-7_3 (Cited on page 55).

[120] O. Arbelaitz, I. Gurrutxaga, J. Muguerza, J. M. Pérez, and I. Perona. "An Extensive Comparative Study of Cluster Validity Indices". In: *Pattern Recognition* 46.1 (2013), pp. 243–256. DOI: 10.1016/j.patcog.2012.07.021 (Cited on page 60).

[121] C. M. Fonseca, M. López-Ibáñez, L. Paquete, and A. P. Guerreiro. *Computation of the Hypervolume Indicator*. URL: http://lopez-ibanez.eu/hypervolume (visited on 11/30/2015) (Cited on page 65).

[122] E. Zitzler and L. Thiele. "Multiobjective Optimization Using Evolutionary Algorithms - A Comparative Case Study". In: *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*. PPSN V. Amsterdam, The Netherlands: Springer, September 1998, pp. 292–301. DOI: 10.1007/BFb0056872 (Cited on page 65).

[123] E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. Shaker Verlag Germany, 1999 (Cited on page 65).

[124] F. J. Massey Jr. "The Kolmogorov-Smirnov Test for Goodness of Fit". In: *Journal of the American Statistical Association* 46.253 (1951), pp. 68–78. DOI: 10.1080/01621459.1951.10500769 (Cited on page 66).

[125] S. F. Sawyer. "Analysis of Variance: The Fundamental Concepts". In: *Journal of Manual & Manipulative Therapy* 17.2 (2009), pp. 27–38. DOI: 10.1179/jmt.2009.17.2.27E (Cited on page 66).

[126] E. Ostertagová, O. Ostertag, and J. Kováč. "Methodology and Application of the Kruskal-Wallis Test". In: *Applied Mechanics and Materials* 611 (2014), pp. 115–120. DOI: 10.4028/www.scientific.net/AMM.611.115 (Cited on page 66).

[127] J. Asensio-Cubero, J. Q. Gan, and R. Palaniappan. "Multiresolution Analysis over Simple Graphs for Brain Computer Interfaces". In: *Journal of Neural Engineering* 10.4 (2013), pp. 21–26. DOI: 10.1088/1741-2560/10/4/046014 (Cited on page 67).

[128] BioSemi. *BioSemi System*. URL: https://www.biosemi.com/products.htm (visited on 12/02/2019) (Cited on page 67).

[129] I. Daubechies. *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics, 1992 (Cited on page 67).

[130] A. Seshadri. *NSGA - II: A multi-objective optimization algorithm*. URL: https://mathworks.com/matlabcentral/fileexchange/10429-nsga-ii-a-multi-objective-optimization-algorithm (visited on 11/28/2019) (Cited on page 70).

[131] Free Software Foundation. *GNU Gprof Manual*. URL: https://ftp.gnu.org/pub/old-gnu/Manuals/gprof-2.9.1/html_node/gprof_toc.html (visited on 12/03/2019) (Cited on page 70).

[132] T. Gunarathne, B. Salpitikorala, A. Chauhan, and G. Fox. "Optimizing OpenCL Kernels for Iterative Statistical Algorithms on GPUs". In: *Proceedings of the Second International Workshop on GPUs and Scientific Applications*. GPUScA'2011. Galveston Island, TX, USA, October 2011, pp. 33–44. DOI: 10.1007/978-3-642-35380-2_85 (Cited on page 81).

[133] B. Dhanasekaran and N. Rubin. "A new method for GPU based irregular reductions and its application to k-means clustering". In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU'2011. Newport Beach, CA, USA: ACM, March 2011, pp. 729–737. DOI: 10.1145/1964179.1964182 (Cited on page 81).

[134] R. Ge, X. Feng, and K. W. Cameron. "Improvement of Power-Performance Efficiency for High-End Computing". In: *Proceedings of the 19th International Parallel and Distributed Processing Symposium*. IPDPS'2005. Denver, CO, USA: IEEE, April 2005, pp. 233–240. DOI: 10.1109/IPDPS.2005.251 (Cited on page 96).

[135] A. Sîrbu and O. Babaoglu. "Power Consumption Modeling and Prediction in a Hybrid CPU-GPU-MIC Supercomputer". In: *Proceedings of the 22nd International Conference on Parallel Processing, Euro-Par 2016*. Euro-Par'2016. Grenoble, France: Springer, August 2016, pp. 117–130. DOI: 10.1007/978-3-319-43659-3_9 (Cited on page 137).

[136] CPUFreq Governors. *Information for users and developers*. URL: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt (visited on 11/30/2018) (Cited on page 140).

[137] J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan. "Spert-II: a vector microprocessor system". In: *Computer* 29.3 (1996), pp. 79–86. DOI: 10.1109/2.485896 (Cited on page 141).

[138] Y. Zou, X. Jin, Y. Li, Z. Guo, E. Wang, and B. Xiao. "Mariana: Tencent Deep Learning Platform and Its Applications". In: *VLDB* 7.13 (2014), pp. 1772–1777. DOI: 10.14778/2733004.2733082 (Cited on page 141).

[139] Arduino. *Arduino Mega Information*. URL: https://www.arduino.cc/en/Main/arduinoBoardMega2560/ (visited on 01/21/2020) (Cited on page 167).

[140] J. J. Escobar. *e-hpMOBE Project Deliverables*. URL: http://atcproyectos.ugr.es/ehpmobe/index-en.php?menu=deliverables (visited on 12/09/2019) (Cited on page 171).

[141] J. J. Escobar. *Github Repository. HPMoon Software*. URL: https://github.com/rotty11/TIN2015-67020-P/tree/master/HPMoon_v8 (visited on 12/23/2019) (Cited on page 171).