UCLouvain

École polytechnique de Louvain

Optimising and parallelising instructions in ARM Cortex-M processor for ultra-low-power arrhythmia detection algorithm on electrocardiogram data

Author: **Olivier Monzibila Ndju**
Supervisor: **David Bol**
Readers: **Rémi Dekimpe, Julien Verecken, Jean-Didier Legat**
Academic year 2021–2022
Master [120] in Electrical Engineering

Abstract

Heart rhythm disorders and related conditions are becoming increasingly common, due to the ageing population. Electrocardiogram (ECG) is the main diagnostic method used to detect arrhythmia and determine the causes. Research has focused on devices that allow data to be taken regularly and allow physicians to track patients over longer periods. The device on which this thesis is based, SleepRider, is a solution that runs an ultra-low power (ULP) detection and classification software, written in C. This master thesis is focused on the pre-processing part of the algorithm. In this work, ways to optimise the performance have been explored, with the vectorisation having been identified as the ideal solution as it fitted well the characteristics offered by microprocessors compatible with SleepRider, the Cortex-M4, and Cortex-M33, after a careful study of the principles of parallelisation and vectorisation. These core, designed by ARM, offer hardware features such as SIMD intrinsics or vector processing extension. The master thesis studied the gradual implementation of SIMD intrinsics, displaying a 45.47% decrease of cycle output over the pre-processing algorithm. It highlights the potential of the parellising algorithm and the perspective it offers for future works.

Acknowledgements

This work and the completion of this master's thesis would not have been possible without my supervisor, Prof. David Bol, whom I thank for his support and encouragements throughout the year. He managed to make himself available to follow my work despite his hectic schedule.

Next, I would like to thank Rémi Dekimpe and Julien Verecken, for their advices and expertise in embedded programming and for their excellent availability. They spent time to offer me precious explanations. They helped me in bettering my understanding of many concepts and my work would have been much more difficult without them.
I would like to thank Prof. Jean-Didier Legat for introducing me to the field of electricity back in my first year and for accepting to be a reader for this work.
I also want to show gratitude to my friends who accepted to review my work. François and Lucas, your feedback greatly helped me.

Finally, I would like to thank my parents and my sisters, Léa & Vanessa, for doing their best to support me during this stressful year and always being understanding.

Contents

List of Abbreviations

BPF Band pass filter

BPM Beats per minute

CPU Central processing unit

CT Computed tomography

CVS Cardiovascular system

ECG Electrocardiography

FIR Finite impulse response

FN False negative

FP False positive

HPC High performance computing

HPF High pass filter

IDE Integrated development environment

IIR Infinite impulse response

LD Linear discriminant

LPF Low pass filter

LSB Least significant bits

MLP Multilayer perceptrons

MSB Most significant bits

RBF Radial basis function

SIMD Single Instruction Multiple Data

SoC System on chip

SVM Support vector machine

TP True positive

ULP Ultra-low power

WT Wavelet transform

List of Figures

List of Tables

Introduction

Cardiac arrhythmia is a condition that occurs when the heartbeats are no longer regular or when their frequency speeds up or slows down abnormally. The normal heart rate is 60 to 100 heartbeats per minute on a regular basis. [1] It is normal for the number of heartbeats to increase naturally when the body is under stress, from any type of physiological effort. However, they can cause a variety of troublesome symptoms, such as dizziness or chest pain. Furthermore, if it is a phenomenon that is re-occurring often, it can lead to severe consequences such as heart attack, heart failure or sudden death. [2] Arrhythmia treatments exist. They include medication or surgery to implant devices that control the heartbeat. [2]

The main problem lies with the diagnostic, or the lack thereof. Because it is a phenomenon that happens naturally, arrhythmia can be easily overlooked. Apart from that, the lengthening of the lifespan and the inversion of the age pyramid means that the problems of cardiac arrhythmia will surely become more and more present. [3] Therefore, the emphasis on being able to detect these arrhythmia early and quickly is a major issue that must be addressed.
Electrocardiogram (ECG) recording is a tool of paramount importance when it comes to diagnosing heart conditions. When the cardiac impulse passes through the heart, electrical current spreads from the heart into the adjacent tissues: they generate electrical potentials that can be recorded as an ECG, giving important clinical information to the physician. For every beat, a normal ECG recording has a P wave, a QRS complex and a T wave. Each type of arrhythmia is associated with a pattern, and as such, it is possible to identify and classify its type. [4]

The detection of cardiac arrhythmia is the main motivation behind this thesis. ECG recordings can last up to 24 hours. Manual detection of arrhythmia could take a lot of time. Being able to recognize automatically those abnormalities in the cardiac rhythm using only data from ECG would allow doctors to establish quick diagnosis with minimal work. The detection algorithms that aim to measure the heart rate by recognizing the QRS complexes consist of several stages. [5]

- **Pre-processing**: detection can be difficult due to multiple source of noise, inter-patient variability and variation of the heartbeat morphology along time.

Different techniques are used to attenuate P and T waves as well as noise and properly discriminate the QRS complexes.

- **Segmentation**: segmentation is necessary to allow the detector to make the decision whether the data under study is a heartbeat or not.
- **Learning**: Knowing that an heartbeat is detected is not enough. The goal remains arrhythmia detection. As such, learning algorithms grants the detector the ability to determine what kind of beat has just been detected.

The SleepRider is a microcontroller unit (MCU) developed by an UCLouvain team led by Rémi Dekimpe. They proposed an ultra-low power (ULP) microcontroller, with a Cortex-M4 core, designed for arrhythmia classification system, aimed at wearable monitoring device. [6] It runs a detection and classification software that reaches a sensitivity of 82.6% and 88.9% for supraventricular and ventricular arrhythmia respectively on the MIT-BIH arrhythmia database. [7]. The ULP software implementation is based off the work of Pan and Tompkins [8]. They described a detection algorithm that is tailored to run on system with low power consumption constraints. It has then been adapted into a C software by Hamilton, [9] which is the implementation used on the SleepRider.
This master thesis stemmed from the Dekimpe team work, since they aimed to port their application to a new, more efficient core: the Cortex-M33. Therefore, this study focused on the algorithm running on the SleepRider and made it the center of its research question: how can the ULP implementation can be optimized?

Parallel computing is a type of computing architecture in which these problems are broken down into smaller tasks and all of them are processed at the same time. [10] It is almost necessary for applications where time and performance constraints are important. This turns out to be an adequate answer to the research question. The Cortex-M4 and Cortex-M33 cores offer opportunities to apply the principles of parallelism and vectorisation to the algorithm through the use of SIMD instructions for the former, and vector registers and operations for the latter. [11, 12] The title of this thesis takes on its full meaning.

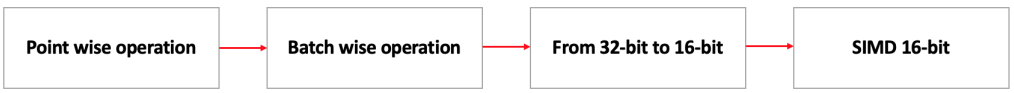

Figure 1: Implementation workflow

The purpose of this work is to implement SIMD intrinsics into the algorithm, precisely on the pre-processing part because the digital filters used are a very potent target to vector operations. The evolution toward this solution will be explained

thoroughly, with results and analysis being presented for every intermediary steps. This work made it possible to obtain a final algorithm which allows a saving of clock cycles of **45.47%** globally on the pre-processing functions.

The contribution of this master thesis is divided in 5 chapters:

- In Chapter 1, contextualisation of the arrhythmia condition, through a general description of the cardiovascular system, the actual condition and the principles of ECG.
- In Chapter 2, principles of arrhythmia are presented and followed by a state of the art of the works centered around QRS detector. SleepRider is then described, as well as the algorithm it is running.
- In Chapter 3, the concept of parallelisation is studied.
- In Chapter 4, the implementation of the SIMD, and all the steps towards that goal, is described. Results are provided and analysed.
- In Chapter 5, the results from the implementations are summarized and the final performances are then analyzed. Limitations, problems encountered and further optimization possibilities are discussed.

The work is summarized in its conclusion, with a reminder of the final results. Future perspectives are given.

Chapter 1

Cardiac arrhythmia : fundamentals

This chapter gives an overview of the fundamental elements required to understand the master thesis. First, the context of cardiac arrhythmia is presented, through a summary of the anatomy and the physiology of the cardiovascular system (CVS). Arrhythmia is then properly presented, followed by an explanation on electrocardiography (ECG) and its role in arrhythmias detection is highlighted. Finally, the treatments available to date are then mentioned.

1.1 Cardiovascular system

The CVS is a subdivsion of the circular system, the other part being the lymphatic system. Its goal is to ensure that blood can reach organs throughout the body in order to proceed to vital biochemical exchanges (nutrients, proteins and, most importantly, oxygen) or collecting metabolic waste. [13]

1.1.1 Structure

The two main components of the CVS are the heart and the blood vessels. The former serves as a pump while the later transport blood throughout the human body.
The CVS has two main subdivisions as shown on figure 1.1:

- Pulmonary circulation loops from the right heart, taking deoxygenated blood to the lungs where it is oxygenated and returned to the left heart. Pulmonary arteries carry the blood from the heart to lungs.

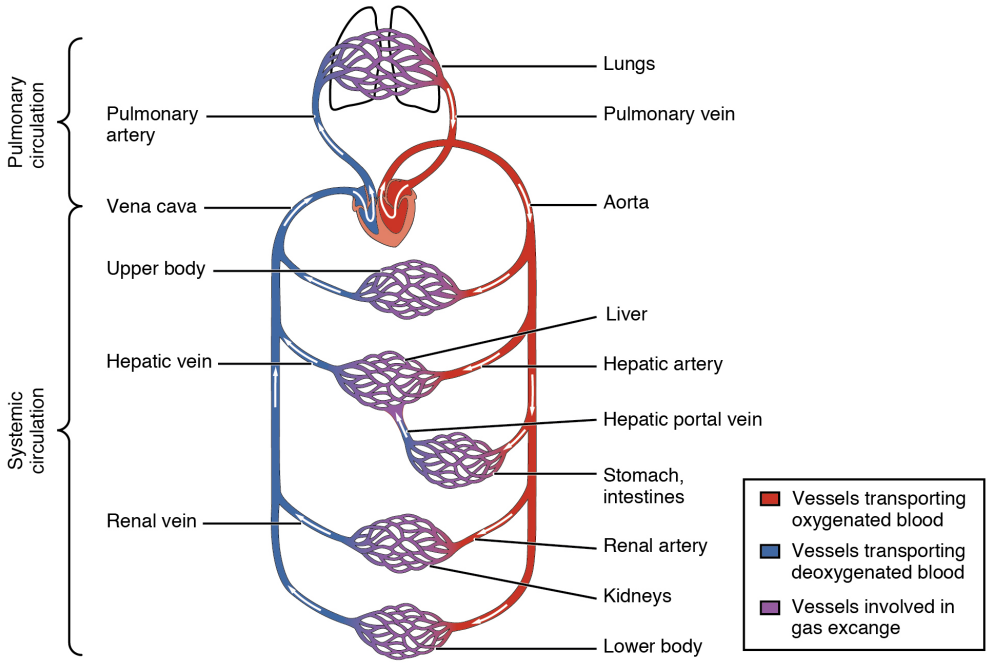

Figure 1.1: Cardiovascular system & subdivisions [13]

- Systemic circulation loops from the left heart to the rest of the body, delivering the oxygenated blood via the aorta and returning deoxygenated blood back to the right heart via the upper and lower vena cava.

1.1.2 Anatomy of the heart

As seen above, the heart is the key component in the CVS, as it fulfills both the role of replacing deoxygenated blood with oxygenated blood and handling the distribution to the rest of the body.
The heart is a muscular organ that contracts to force blood through the body, using the vessels as pathways. It sits behind the sternum, between the lungs, and slightly to the left in the chest cavity: this compartment is called the pericardial cavity, protected by the surrounding rib cage and the diaphragm.
The heart has 4 chambers: the upper chambers are called the left and right atria, and the lower chambers are called the left and right ventricles. Each pair is separated by a muscle called septum.
The blood flow within the heart is regulated by a system made up of one-way valves. There are two atrioventricular valves (between atrium and ventricle): the tricuspid valve (controls blood flow from the right atrium into the right ventricle) and the mitral valve (allows oxygen-rich blood from the lungs to pass from the left

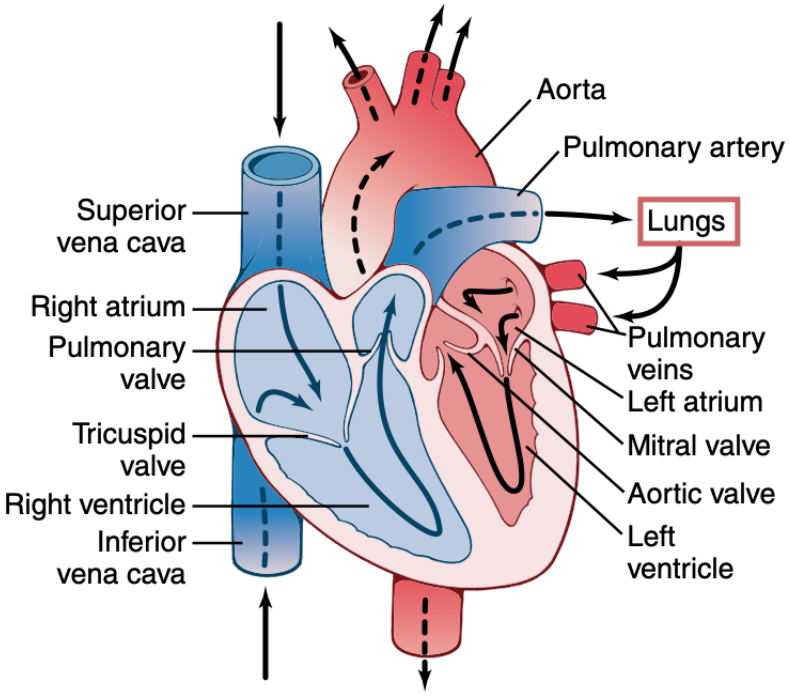

Figure 1.2: Anatomy of the heart [4]

atrium into the left ventricle). There are two semilunar valves (between ventricles and vessels): the pulmonary valve (controls blood flow from the right ventricle into the pulmonary arteries) and the aortic valve (allows oxygen-rich blood to pass from the left ventricle into the aorta).
The heart wall is composed of three layers: the outer epicardium, protecting the inner heart layers and assists in the production of pericardial fluid; the myocardium, a muscular middle layer wall of the heart, which enables the contractions; the inner endocardium — inner layer of the heart that lines the chambers and covers the heart valves. It is surrounded by a double-layered membrane called the pericardium: the outer layer surrounds the roots of major blood vessels and is attached by ligaments to the spinal column, diaphragm, and other bodyparts while the inner layer attached to the heart muscle. [4] [14]

1.1.3 Physiology of the heart

The heart essentially works as a pump. Contraction from the myocardium are caused by electric impulses. The heart's intrinsic regulating system is called the conduction system: it generates and distributes the impulses over the heart to stimulate myocardium fibers or cells to contract. The sinoatrial node (the heart's

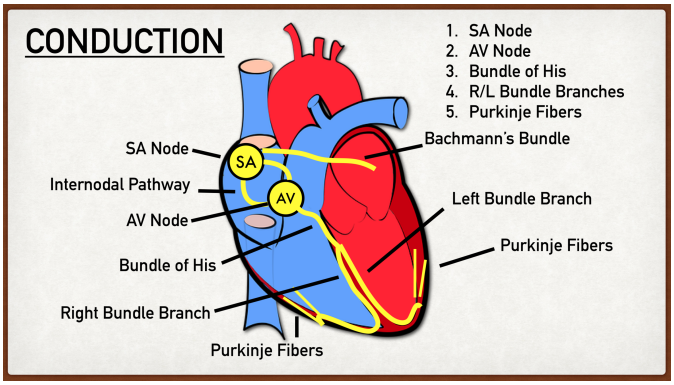

Figure 1.3: Conduction system [15]

"natural pacemaker") initiates each cardiac cycle with an excitation and sets the pace for the heart rate. [15] It is located in the superoposterior wall of the right atrium. The excitation signal travels to the atria, inducing a contraction. The atrioventricular node delays the signal until the atria are empty of blood. The signal is then carried to the Purkinje fibers through the bundle branches. They cause ventricles to contract. The pathway is shown in figure 1.3.

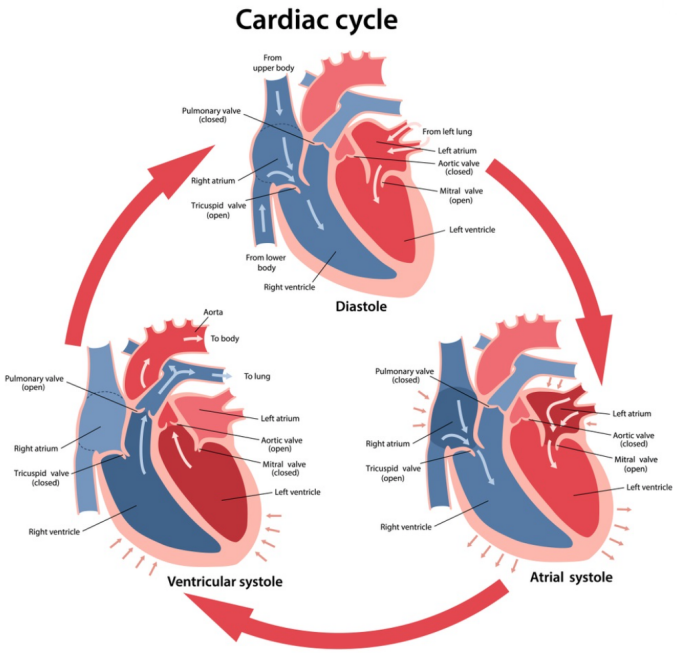

Figure 1.4: Cardiac cycle [16]

The cardiac cycle (shown on figure 1.4) consists of a period of relaxation called

diastole, during which the ventricles fill with blood, followed by a period of contraction called systole. The total duration of the cardiac cycle (including systole and diastole) is the reciprocal of the heart rate. It is measured in beats per minute (BPM). [16]

1.1.3.1 Diastole

During diastole, the ventricles relax and fill with blood returning from the circulatory system. Then the atria contract, forcing more blood into the ventricles. The incoming bloodflow is facilitated by both atrioventricular valves.

1.1.3.2 Systole

During systole, the ventricles contract and pump blood out of the heart, and the atria relax and begin filling with blood again. During atrial systole, both atria contract and force the blood from the atria into the ventricles. During ventricular systole, both ventricles contract, blood is forced to the lungs via the pulmonary artery, and the rest of the body via the aorta. [4, 17, 18]

1.2 Electrocardiography

When the cardiac impulse passes through the heart, electrical current spreads from the heart into the adjacent tissues. A small portion of the current spreads all the way to the surface of the body. Electrical potentials generated by the current can be recorded by placing electrodes on the skin on opposite sides of the heart. The subsequent recording is called electrocardiogram (ECG).
The ECG signal is a valuable clinical tool, giving key information about the cardiac cycle. Changes in the normal ECG pattern occur in numerous cardiac abnormalities, including cardiac rhythm disturbances, inadequate coronary artery blood flow and electrolyte disturbances. Equipment such as Holter monitors are used to continuously record a patient's ECG. [4, 19] The normal electrocardiogram (show on figure 1.5) is composed of a P wave, a QRS complex (almost always three separate waves: the Q wave, the R wave, and the S wave), and a T wave. The ECG is composed of both depolarization (the loss of resting membrane potential as a result of the alteration of the polarization of cell membrane) and repolarization (the restoration of the resting membrane potential after every depolarization event) waves.
The P wave is caused by a depolarization of the atria before atrial contraction begins. The QRS complex occurs when the ventricles depolarize before contraction, as the depolarization wave spreads through the ventricles. The T wave is caused by the repolarization of the ventricles, hence it is known as the repolarization

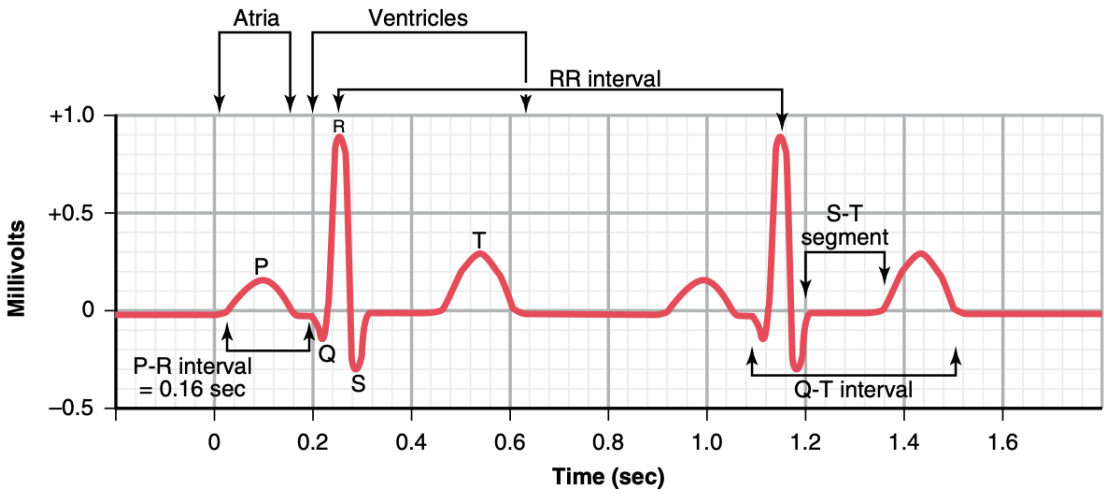

Figure 1.5: Normal ECG [4]

wave. This process normally occurs in ventricular muscle 0.25 to 0.35 second after depolarization. [4]

A sinus rhythm is any cardiac rhythm in which depolarisation of the cardiac muscle begins at the sinus node. It is a necessary (but not sufficient) criterion of a normal electrical activity and is characterised by the presence of correctly oriented P waves on the electrocardiogram (ECG). [2] A Wiggers diagram (shown in figure 1.6) reports the events of the cardiac cycle. Blood pressure changes (aortic pressure, left atrial pressure and left ventricular pressure) are represented in the first three curves. The fourth curve depicts the changes in left ventricular volume, the fifth the electrocardiogram, and the sixth a phonocardiogram (recording of the sounds produced by the heart, especially the valves). The diagram explains the causes of all the events shown. [20]

1.3 Arrhythmia

Arrhythmia are irregularities in the heartbeat, including when it is too fast (tachyarrhytmia) or too slow (brachyarrhytmia).

1.3.1 Symptoms

The most common symptom of tachyarrhythmia is palpitation: an awareness of an abnormally fast heartbeat. These may be infrequent, frequent, or continuous. Some of arrhythmia are harmless. It is normal for the heart rate to speed up during

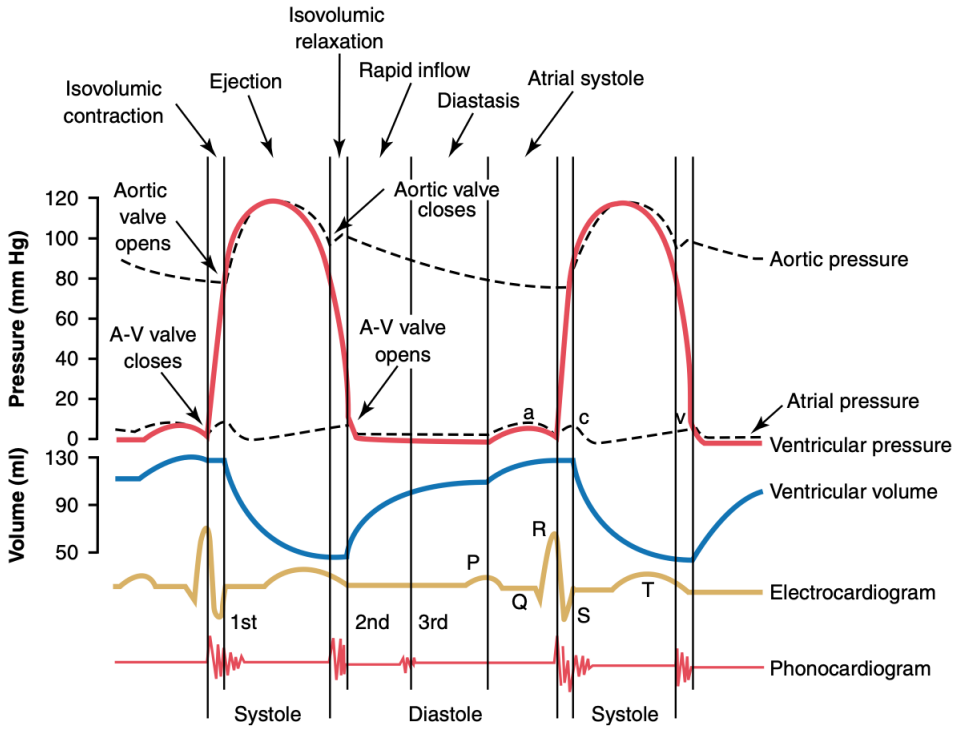

Figure 1.6: Wiggers diagram [4]

physical activity and to slow down while resting or sleeping. Some others, however, predispose to adverse outcomes: some types even result in cardiac arrest, or sudden death. [1]
Other symptoms of arrhythmia include anxiety, chest pain or discomfort, difficulty breathing, dizziness or tiredness. [2]

1.3.2 Types

Some arrhythmia cause irregular heartbeat: tachycardia is when the heart rate is too fast (above 100 BPM in adults) while bradycardia is when the heart rate is too slow (below 60 BPM).
Supraventricular arrhythmias starts in the atria or the gateway to the lower chambers. In the case of atrial fibrillation (most common type of arrhythmia), for example, the lower chambers do not fill completely or pump enough blood to the lungs and body. Ventricular arrhythmias start in the ventricles. [2]
The two types of arrhythmia that will be considered in this work are sinus rhythm with supraventricular ectopic beat and ventricular ectopic beat. Figure 1.7 shows

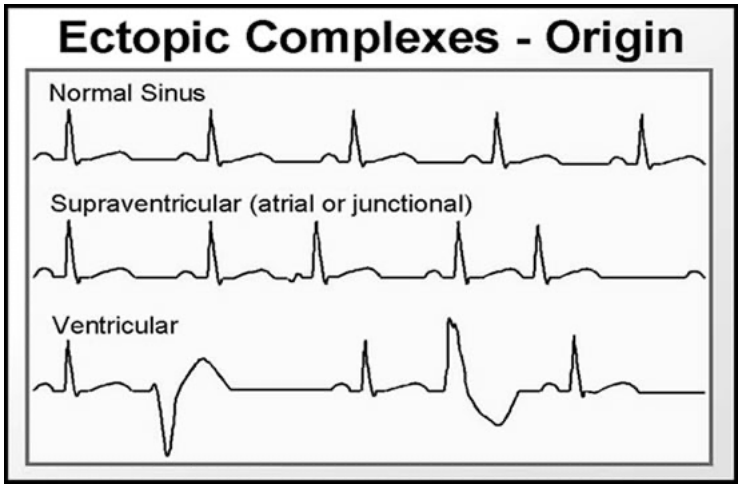

Figure 1.7: Supraventricular ectopic beat & ventricular ectopic beat [2]

both arrhythmia types disruption in comparison to the normal sinus rhythm.

1.3.2.1 Supraventricular ectopic beats

The supraventricular ectopic beat is characterized by its prematurity and the lack of a P-wave (that can be masked by the T-wave of the previous beat). It may indicate atrial irritability. Therefore, an increasing trend in supraventricular ectopic beats may be an indicator or sign for atrial fibrillation.

1.3.2.2 Ventricular ectopic beats

The ventricular ectopic beat is characterized by its prematurity and a broad irregular QRS complex. It may indicate ventricular irritability: an increasing trend in ventricular ectopic beats may lead to ventricular tachycardia.

1.3.3 Diagnosis

Since the heart's electrical signals control how fast your heart beats, a problem with these electrical signals can cause an irregular rhythm. However, this also means that such problem can be tracked down with electrical signals: this is where the ECG reading can give physicians key diagnosis information.
Other methods can be used to track heart conditions. A cardiac computed tomography (CT) scan is a non-invasive imaging test using X-rays to take many detailed pictures of your heart and its blood vessels. It helps physicians to see problems with the heart or blood vessels. [19, 21, 22]

1.4 Treatment

Common arrhythmia treatments include medication, surgery to implant devices that control the heartbeat and other procedures to treat problems with the conduction system. [2]

1.4.1 Medication

The type of medication will depend on the diagnosis. In the case of a bradycardia, atropine is given by emergency medical services. In the case of tachycardia, there is a wider range of medication available, with their own averse effects: adenosine (can cause some chest pain, flushing, shortness of breath, and atrial fibrillation), beta blockers (can cause fatigue, stomach or sleep problems, and sexual dysfunction) or various channel ions blockers (can cause digestive trouble, low blood pressure or even increase the risk of cardiac arrest).
Most of the time, however, medicines are used together with other treatments, in order to limit the secondary effects. [2, 23]

1.4.2 Procedures & devices

Several procedures not involving medication exist. They range from non-invasive treatments to procedure requiring surgical intervention.
Cardioversion, for example, is a procedure that uses external electric shocks to restore a normal heart rhythm. It is also known as defibrillation when it is done in an emergency to prevent death. As a further attempt to treat abnormal electrical signals, the pacemaker is a small device that sends electrical pulses in order to reset the heartbeat to a normal frequency. It can be either temporary or permanent.
Catheter ablation is a procedure to stop abnormal electrical signals from moving through your heart and causing an irregular heartbeat. Since it is an invasive procedure, the risk included are bleeding, infection or heart damages. [24, 25]

Chapter 2

ECG-base detection algorithm

Each type of arrhythmia is associated with a pattern, and as such, it is possible to identify and classify its type. As ECG recordings can range from few seconds to several hours, manual detection of arrhythmia would be a tedious work. Years of research have yielded algorithm able to recognize heartbeats and classify using ECG recording as source material. Being able to recognize those abnormalities in the cardiac rhythm is pivotal in order to treat the disease.
The detection of cardiac arrhythmia is the keystone of this thesis. The medical problem and the methods used to detect this cardiac condition have been presented. Now there is one more constraint to consider: how can we continuously obtain heartbeat data that will allow doctors to make a rapid diagnosis? This chapter will introduce the research question which is, starting from an ultra low power algorithm for arrhythmia detection, finding a way to optimize it.

This chapter will first present the general principles behind ECG-based detection and provide state of the art of result given by prominent research groups. Then, existing arrhythmia detection devices will be presented, with a focus on the SleepRider, which is the basis of this master thesis. Finally, the ultra-low power (ULP) software implementation relevant to this master thesis will be presented.

2.1 State of the art

Detection algorithms aim to measure the heart rate, which remains the first way to assess the heart health state. The detection techniques that will be described are centered around recognizing the QRS complexes, central element of the cardiac cycle. The energy of heartbeats is mainly located in the QRS complex, so an accurate QRS detector is the most important part of ECG analysis. [5]
Generally, a detection algorithm has several stages, from pre-processing the raw signal to being able to classify it.

2.1.1 Databases

In order to evaluate the performance of an algorithm, there are available databases. The most utilized, and recommended by for the validation of medical equipment, is the Massachusetts Institute of Technology – Beth Israel Hospital Arrhythmia Database (MIT-BIH) [7] database for arrhythmia analysis. This database contains 48 records of heartbeats at $360Hz$ for approximately 30 min of 47 different patients. However, several other databases have been widely used in the development of QRS detection such as QT [26], or The American Heart Association Database (AHA) [27].

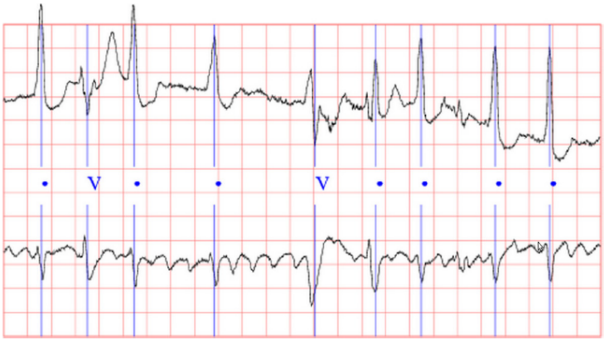

Figure 2.1: Example of annotations in a MIT-BIH database [28]

Due to the various number of databases, a standardization was developed by AAMI, the ANSI/AAMI EC57:1998/(R) 2008 standard. [29] The majority of the heartbeats in MIT-BIH have annotations associated with the type of heartbeat (class and fiducial points). So, for example, the standards specify how annotations should be done in the databases, as we can see in figure 2.1. The annotations are explained in table 2.1.
AAMI recommends that only some types of arrhythmia should be detected, although various types exist. Therefore, 5 classes have been defined, subdivised into 15 sub-classes that each stand for the recommended arrhythmia types, as illustrated in table 2.1.
The measures recommended by AAMI for evaluating methods are: Sensitivity (Se), Positive predictivity (+P), False positive rate (FPR) and Overall accuracy (Acc). Because the last measure can be easily distorted by the results of the majority class, the first three measure are the most relevant for comparison.

Group	Symbol	Class
N Any heartbeat not categorized as SVEB, VEB, F or Q	N or . L R e j	Normal beat Left bundle branch block beat Right bundle branch block beat Atrial escape beat Nodal (junctional) escape beat
SVEB Supraventricular ectopic beat	A a j S	Atrial premature beat Aberrated atrial premature beat Nodal (junctional) premature beat Supraventricular premature beat
VEB Ventricular ectopic beat	V E	Premature ventricular contraction Ventricular escape beat
F Fusion beat	F	Fusion of ventricular & normal beat
Q Unknown beat	P or / f U	Paced beat Fusion of paced & normal beat Unclassifiable beat

Table 2.1: Principal types of heartbeats present in the MIT-BIH. Adapted from [30]

2.1.2 Pre-processing

Because the heartbeat morphology varies with time and different sources of noise can be present, QRS detection can be difficult, the first stage of a QRS detector is the pre-processing stage. It uses different techniques to attenuate P and T waves as well as noise. The choice of method to use depends on final the decision method used. Algorithm focusing on the heartbeat segmentation from the ECG signal tend to require a pre-processing that is different from the methods focusing on the automatic classification of arrhythmias.[30]

2.1.2.1 Finite impulse response (FIR) filters

The most popular technique is the implementation of recursive digital filters of the finite impulse response (FIR). [31]. It works well for the attenuation of the known frequency bands (50/60Hz noise) but requires applying different filters when the frequency of the noise is not known, which can distorts the morphology of the signal and make it unusable for diagnosing cardiac diseases. One way to solve this problem: the adaptive filter. It is a digital filter that has self-adjusting characteristics. It can adjust its filter coefficients to adapt the input signal via an adaptive algorithm. [32]

2.1.2.2 Wavelet transform

A wavelet is a waveform localized in time, with limited duration and zero average value. It has two basic properties: scale and location. Scale (or dilation) defines how stretched or squished a wavelet is. This property relates to frequency as defined for waves. Location defines where the wavelet is positioned in time. There are several type of mother wavelets from which baby wavelets can be derived by tuning the scale and location.
The equation for the Continuous Wavelet Transform (CWT) is [33]

$$T(a,b) = \frac{1}{\sqrt{a}} \int_{\infty}^{-\infty} x(t)\psi^* \frac{(t-b)}{a} dt \tag{2.1}$$

The equation for the Discrete Wavelet Transform (DWT) is [33]

$$T_{m,n} = \int_{\infty}^{-\infty} x(t)\psi_{m,n} dt \tag{2.2}$$

where a is the dilation factor, b is the translation factor and *psi* is the wavelet function.
The wavelet transform (WT) is a time-frequency transform: it is the convolution of an input signal with a set of wavelets at a variety of scales. It is useful in extracting features from non-linear and non-stationary signals in selected sub-bands. The main challenge in wavelet transform domain is to select the correct number of decomposition levels and to select a correct mother wavelet. The WT addresses a major shortcoming of the Fourier transform: its zero time resolution (despite the existence of alternative solutions like the Short Time Fourier transform).
Detection algorithm based on wavelet transforms have been employed to remove noise, since they preserve ECG signal properties.

2.1.2.3 Other methods

Other methods have also presented promising results on noise attenuation, such as the use of nonlinear Bayesian filters [34] or an extended Kalman Filter based algorithm [35], which incorporates the parameters of the ECG dynamic model .

2.1.3 Segmentation

The second stage is the segmentation stage. Through segmentation techniques, the most widely used method for ECG data ([30]), it handles the most important task: the determination of thresholds for the detection of the R peaks. Two measures are considered for the evaluation of the accuracy: sensitivity, defined as

$$Sensitivity = TP/(TP + FN) \tag{2.3}$$

and positive predictivity, defined as

$$PositivePredictivity = TP/(TP + FP) \tag{2.4}$$

where True Positive (TP) indicate the number of heartbeats correctly segmented, False Positive (FP) indicate the number of segmentations that do not correspond to the heart-beats and False Negative (FN) indicate and the number of segmentations that were not performed. [30] Although research groups have used several

Algorithm	**Year**	**Methods**	**Sensitivity**	**PositivePredictivity**
Pan & Tompkins [8]	1985	Pre-processing: FIR Segmentation: adaptive threshold	99.75%	99.54%
Hamilton & Tompkins [36]	1986	Pre-processing: FIR Segmentation: adaptive threshold	99.69%	99.77%
Li [37]	1995	Pre-processing: WT Segmentation: adaptive threshold	99.89%	99.94%
Poli [38]	2002	Pre-processing: FIR Segmentation: genetic algorithm	99.60%	99.50%
Martinez [39]	2004	Pre-processing: WT Segmentation: adaptive threshold	99.80%	99.86%
Bahoura [40]	2007	Pre-processing: WT Segmentation: adaptive threshold	99.83%	99.88%

Table 2.2: Performance of some segmentation methods, results provided by authors. MIT-BIH database is used in all methods. Adapted from [30]

different methods, the most popular segmentation one uses adaptive thresholds to discriminate the locations of the QRS complexes, developed by Pan and Tompkins. [8] The use of adaptive threshold in subsequent works are all based on this original algorithm.

2.1.4 Feature extraction & selection

Any information extracted from the heartbeat used to discriminate its type can be considered as a feature. Feature extraction is defined as the stage that involves the description of a heartbeat, while feature selection consists in choosing a class for this heartbeat; the one with most representative features with the objective to improve the following classification stage.

2.1.5 Learning

Arrhythmia heartbeat classification, the final step, is based upon the set of features defined from the heartbeats. Classical artificial intelligence algorithms from machine learning and data mining domain are used for this task. [30]
Learning algorithms are out of the scope of this master thesis, which focused on optimizing the pre-processing and segmentation parts, although they remain a pivotal part of the detection scheme.

Features	Normal value	Normal variation
P wave	110 ms	± 20 ms
PQ/PR interval	160 ms	± 40 ms
QRS width	100 ms	± 20 ms
QT interval	400 ms	± 40 ms
Amplitude of P	0.115 mV	± 0.05 mV
Amplitude of QRS	1.5 mV	± 0.5 mV
ST level	0 mV	± 0.1 mV
Amplitude of T	0.3 mV	± 0.2 mV

Table 2.3: Typical features of a normal ECG signal (Healthy adult). Adapted from [41]

2.1.5.1 Support vector machine (SVM)

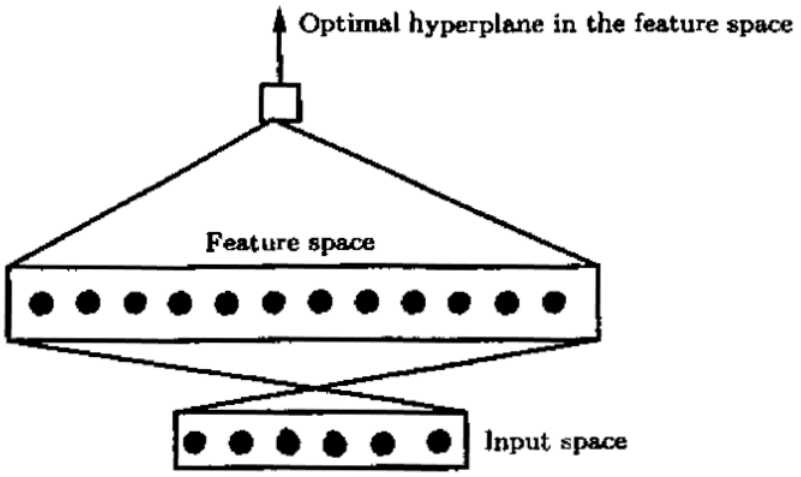

Figure 2.2: SVM mapping of the input space into an high-dimensional feature space [42]

Support Vector Machines (SVM)s) are a family of supervised machine learning algorithms that can be used to solve classification, regression and anomaly detection problems. It is popular due to its computational efficiency and robustness.
Given input vectors x, SVM maps them into an high-dimensional feature space Z through pre-chosen a non-linear mapping. In this space, an optimal separating hyperplane (a subspace whose dimension is one less than its ambient space [43]) is constructed, usually a maximum margin hyperplane, and serves as a decision

boundary. Support vectors are the data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. [42]
SVMs are the most popular classifiers found in literature for ECG-based arrhythmia classification methods. [30] The source code used for this master thesis uses it as well. SVM, however, presents a negative behavior for imbalanced classes. Thus, using balanced databases for the training phase is vital.

2.1.5.2 Multilayer perceptrons (MLP)

In the context of arrhythmia detection, the most used artificial neural network framework are multilayer perceptrons (MLP). [30] A MLP is a neural network using features as an input vector and giving decisions as an output vector. In between those two vectors are multiples hidden layers.
A perceptron in itself is a linear classifier, described in the following equation.

$$f(w^T w + b) = f(\sigma w_j x_j + b) \tag{2.5}$$

, where x_j are the inputs, w_j are the weight of each inputs, b is the bia associated to the layer and f is the activation function. It computes its input on basis of linear combination of all the inputs from the previous layers. Thus, contrary to a perceptron, a MLP uses nonlinear activation functions, granting the network the ability to distinguis data that are not linearly separable.

2.1.5.3 Linear discriminant (LD)

The linear discriminant (LD) is a statistic classification method based on discriminant functions, which are estimated from a training set and try to linearly separate the feature vector (with a weight vector and a bias for adjustment). The criteria for calculating the weight vector varies according to the model adopted, with maximum-likelihood criteria being a common choice, such as in the following equation. [44]

$$j = argmax_i f_i(x) \tag{2.6}$$

where j is the class and f is the discriminant function.
It is a classifier that prevails for its simplicity and for the fact that they did not want to emphasis the classifier, but the proposed features instead. It requires less training time, if compared to SVM and MLP, as it is not iterative. That is, its simply calculates statistics from the training data and then,the classification model is defined. LD can also easily overcome the problem of imbalance in the training set mentioned in section 2.1.5.1. [45]

Work	Year	Feature set	Classifier	Evaluation	Se	+P
Dekimpe [6]	2019	RR intervals DWT Raw signal samples	SVM	Inter-patient	SEVB: 82.6% VEB: 88.9%	SEVB: 34.4% VEB: 80.8%
Villa [46]	2020	RR intervals Value mode decomposition	SVM	Inter-patient	SEVB: 83% VEB: 88%	SEVB: - VEB: 84%
Chen [47]	2019	RR intervals Raw signal samples	MLP	Inter-patient	SEVB: 66.4% VEB: 90.7%	SEVB: 66.4% VEB: 85.5%
Mondéjar [48]	2019	RR intervals DWT High order statistic Morphological indicators	SVM	Inter-patient	SEVB: 78.1% VEB: 94.7%	SEVB: 49.7% VEB: 93.9%
Guo [49]	2019	Raw signal samples	MLP	Inter-patient	SEVB: 62.7% VEB: 91.3%	SEVB: 61.2% VEB: 88.3%
Wu [50]	2019	Raw signal samples	MLP	Intra-patient	SEVB: 78.4% VEB: 92.1%	SEVB: 84.4% VEB: 95.5%

Table 2.4: Performance of arrhythmia classification algorithms. MIT-BIH database is used in all methods. Adapted from [6]

2.2 Existing arrhythmia detection devices

Table 2.4 summarizes some reviewed references of methods aiming at heartbeat classification. The cited methods use different preprocessing approaches.

2.2.1 SleepRider

The SleepRider is a MCU developed by Dekimpe et al that proposed an ultra-low power microcontroller designed for arrhythmia classification system, aimed at wearable monitoring device. It includes a single-channel analog front-end (AFE) for ECG signal acquisition and a digital back-end (DBE) to execute SVM classification. The system has been prototyped in a 28-nm fully-depleted silicon on insulator $3.1 - mm^2$ chip. [6]
It is relevant to this master thesis because the software application is the same as the one used in the SleepRider. The heartbeat detection algorithm used is from Pan and Tompkins [8] while the classifier used is a hierachical SVM with radial basis function (RBF) kernel, with the potential classes being normal beat, supraventricular ectopic beat, or ventricular ectopic beat.

2.3 Ultra-low power software implementation

The algorithm used in the SleepRider, which will be the one explored in this master thesis, can be seen in two parts: beat detection and beat classification.

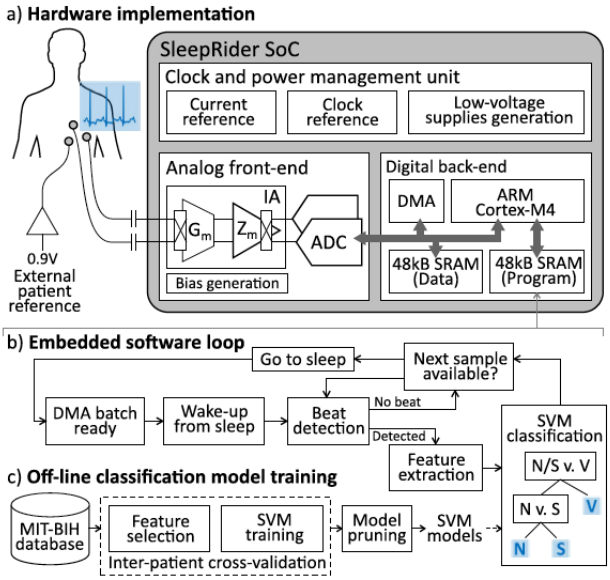

Figure 2.3: System architecture of the SleepRider SoC [6]

The first part is based upon an improved version of the Pan and Tompkins algorithm, proposed by Hamilton and Tompkins. [36]
For the second part, each time a beat is detected, a vector of features is extracted, based on the beat interval, the downsampled signal, and the wavelet transform coefficients. Then a SVM with a radial basis function (RBF) kernel classifies the heartbeat as a normal beat, a supraventricular ectopic beat or a ventricular ectopic beat.
The ECG signal used to provide the plots are from a normal (without noise) ECG from the MIT-BIH database. It has 4000 samples.

2.3.1 Beat detection

The Pan and Tompkins algorithm is a real-time algorithm for detection of the QRS complexes of ECG signals. It applies a series of filters to highlight the frequency content of this rapid heart depolarization and removes the background noise. Then, it squares the signal to amplify the QRS contribution, which makes identifying the QRS complex identification more straightforward. All of this work provides a time limited estimate of the energy in the QRS frequency band. Then, thresholds are calculated and used to only consider the signal peaks and eliminate the noise peaks. [8] In the case of the Hamilton and Tompkins algorithm, they share the same pre-processing step but differ in the decision stage. The focus is on optimizing decision rules by testing the performance of three estimators (mean,

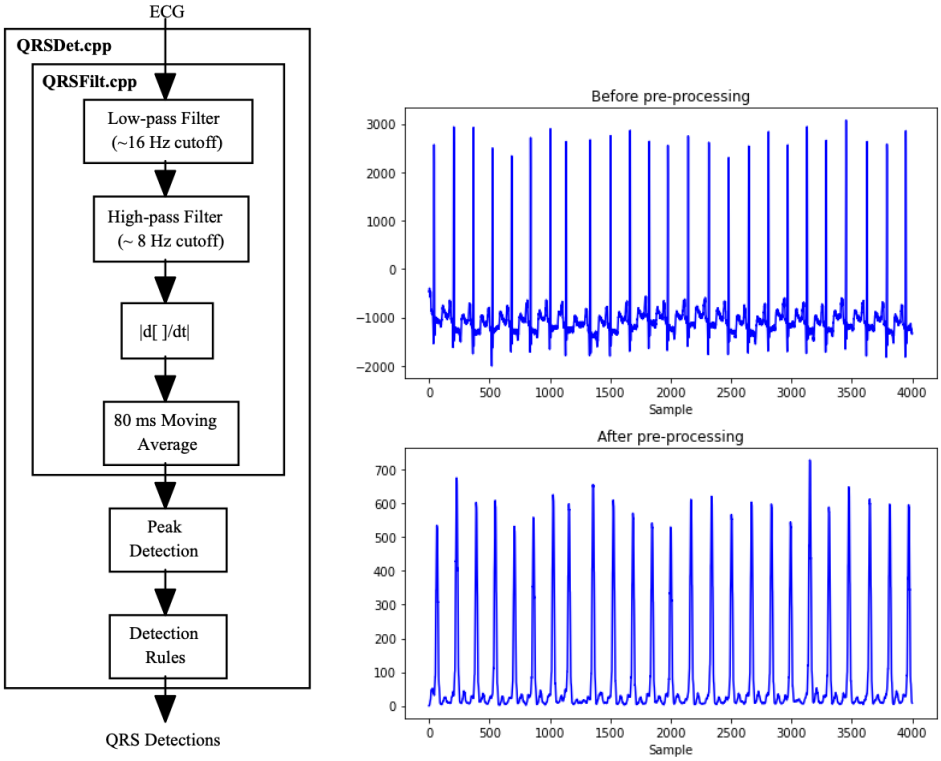

Figure 2.4: Process flow of the beat detection [9] & input vs output of the pre-processing step

median and an iterative peak level) to place the adaptive threshold. stage. [36] The software implementation is based upon specifications described in [9]. The pre-processing filters use a ring buffer [51] in order to store the previous samples and re-use according to each filter specification. It is first in, first out logic fixed-size buffer , interpreted as an array in the C implementation. The size of each ring buffer is equal to the filter it is used for.

2.3.1.1 Band pass filter

The first operation of the neat detection algorithm is the band pass filter (BPF). It reduces the influence of muscle noise, 60 Hz interference, baseline wander, and T-wave interference.
As shown in figure 2.4, instead of using a BP filter, a cascade of a low pass filter (LPF) and a high pass filter (HPF) has been used. The result is effectively a filter with a pass band from 5 to 12 Hz, which is close to the desirable passband to maximize the QRS energy of approximately 5-15 Hz. [8] The filters used are real-time recursive filter in which poles are located to cancel zeros on the unit circle

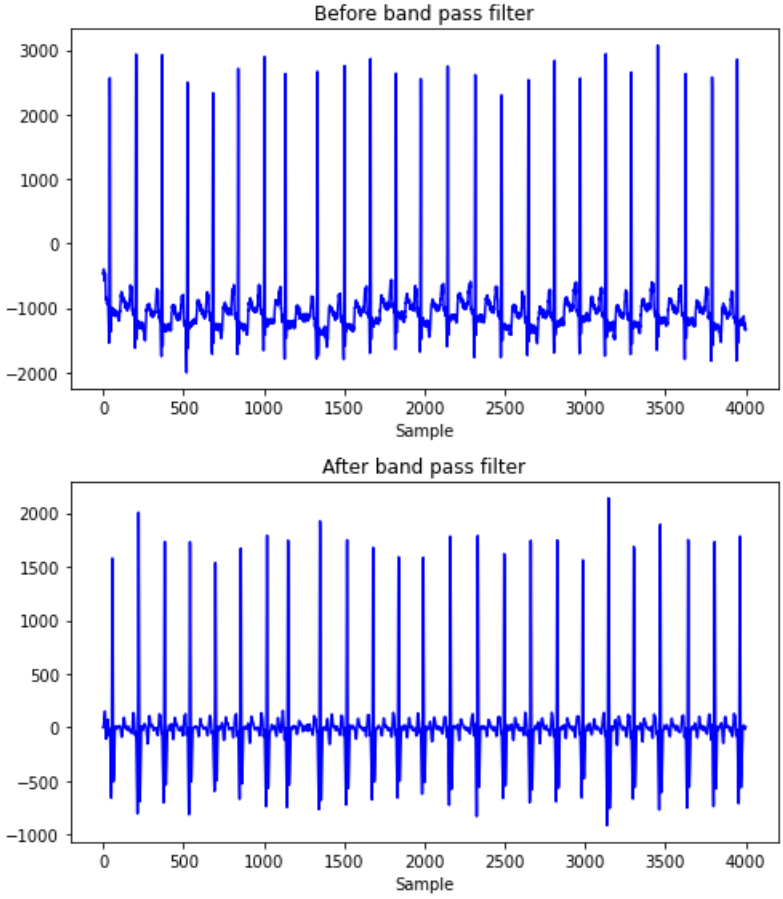

Figure 2.5: Before and after the BPF

of the z plane. For the chosen sample rate of 200 samples per second, a band pass filter could not be designed, hence the use of cascaded LP and HP filter.

Low pass filter The theoretical transfer function of the second-order low-pass filter is [8]

$$H(z) = \frac{(1 - z^{-6})^2}{(1 - z^{-1})^2} \tag{2.7}$$

with an amplitude response of

$$|H(\omega T)| = \frac{sin^2(3\omega T)}{sin^2(\frac{\omega T}{2})} \tag{2.8}$$

where T is the sampling period. The difference equation of the filter is

$$y(nT) = 2y(nT - T) - y(nT - 2T) + x(nT) - 2x(nT - 6T) + x(nT - 12T) \tag{2.9}$$

The cutoff frequency is around $11Hz$ and the gain is 36, with a 6 samples delay. The implementation of the filter as a digital infinite impulse response (IIR) filter gives the following equation (the implementation is explained in [52])

$$y[n] = 2 * y[n-1] - y[n-2] + x[n] - 2 * x[n-24ms] + x[n-48ms] \quad (2.10)$$

The delay is now equal to half the length of the low pass filter, minus 1. [9] chose 10 as the length of the LPF, so there is a 4 samples delay. The length of the ring buffer is 10.

High pass filter The design of the high-pass filter is based on subtracting the output of a first-order low-pass filter from the samples of the original signal. The transfer function is [8]

$$H(z) = \frac{-1 + 32z^{-16} + z^{-32}}{1 + z^{-1}} \quad (2.11)$$

with an amplitude response of

$$|H(\omega T)| = \frac{(256 + sin^2(16\omega T))^{\frac{1}{2}}}{cos(\frac{\omega T}{2}} \quad (2.12)$$

where T is the sampling period. The difference equation of the filter is

$$y(nT) = 32x(nT - 16T) - (y(nT - T) + x(nT) - x(nT - 32T)) \quad (2.13)$$

The cutoff frequency is around $5Hz$ and the gain is 32, with a 16 samples delay. The implementation of the filter as a digital IIR filter gives the following equation (the implementation is explained in [52])

$$y[n] = y[n-1] + x[n] - x[n-128ms] \quad (2.14)$$

$$z[n] = x[n-64ms] - y[n] \quad (2.15)$$

The delay is now equal to half the length of the high pass filter minus 1. [9] chose 25 as the length of the LPF, so there is a 12 samples delay. The length of the ring buffer is 25.

2.3.1.2 Derivative

The resulting signal is then differentiated to provide the QRS complex slope information. The theoretical implementation uses a five-point derivative, with the following transfer function [8]

$$H(z) = \frac{1}{8T}(-z^{-2} - 2z^{-1} + 2z^{1} + z^{2}) \quad (2.16)$$

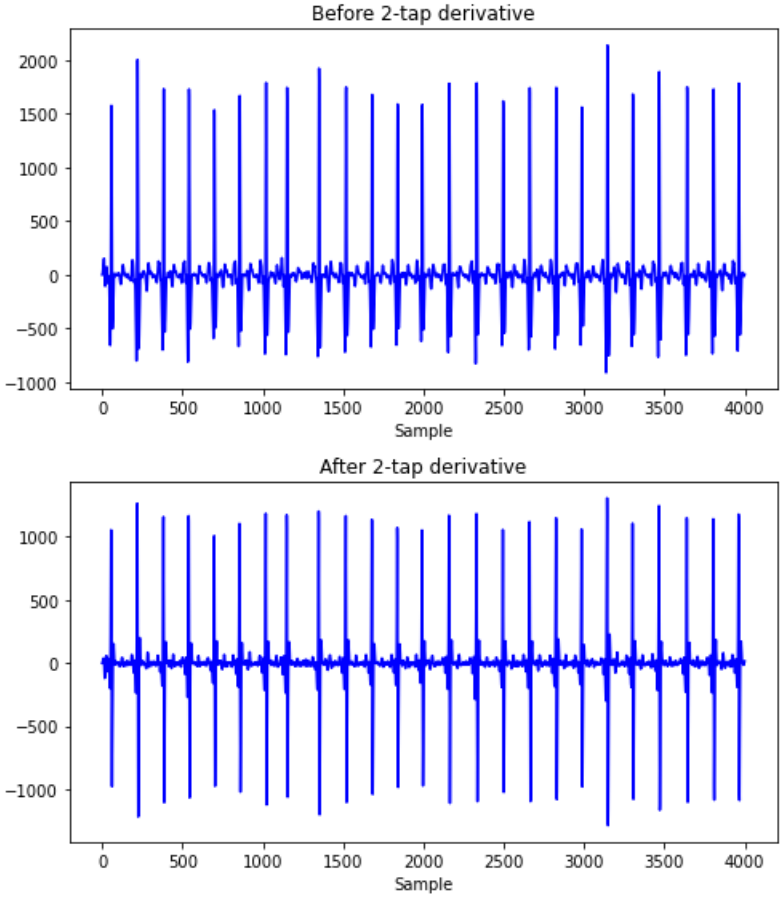

Figure 2.6: Before and after the derivation

where T is the sampling period. The amplitude response is

$$|H(\omega T)| = \frac{1}{8T}(sin(2\omega T) + 2sin(\omega T)) \tag{2.17}$$

The frequency response of this derivative is nearly linear between $0Hz$ and $30Hz$ and there is a two samples delay.
The implementation of the filter as a digital IIR filter is a two-tap filter (the implementation is explained in [52])

$$y[n] = x[n] - x[n - 10ms] \tag{2.18}$$

The delay is now equal to half the length of the derivative filter. [9] chose 2 as the length of the filter, so there is a 1 sample delay. The length of the ring buffer is 2.

2.3.1.3 Absolute value

In [8], the filtered signal was squared rather than rectified with an absolute value operation. This operation caused the detector to be gain sensitive. In [9] implementation, the absolute value was used. It reduced the gain sensitivity, improving the performance of the algorithm.

2.3.1.4 Moving-window integration

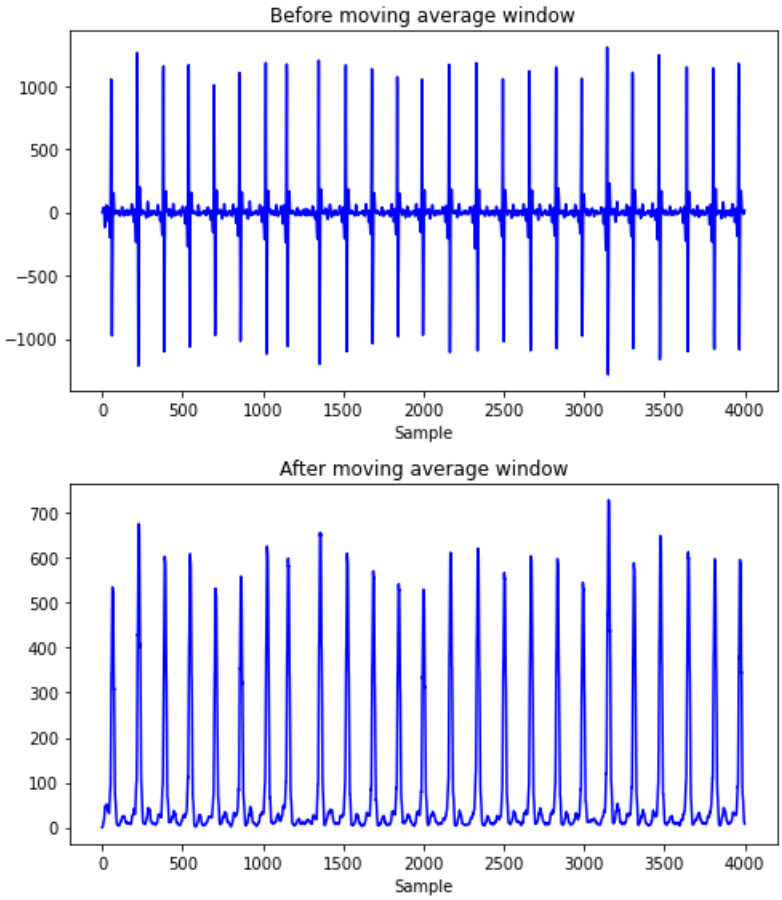

Figure 2.7: Before and after the rectified moving average window

The purpose of moving-window integration is to obtain waveform feature information in addition to the slope of the R wave. The relationship between the moving-window integration waveform and the QRS complex is shown in figure 2.8: the QRS complex corresponds to the rising edge of the integration waveform., such as the time duration of the rising edge is equal to the width of the complex.

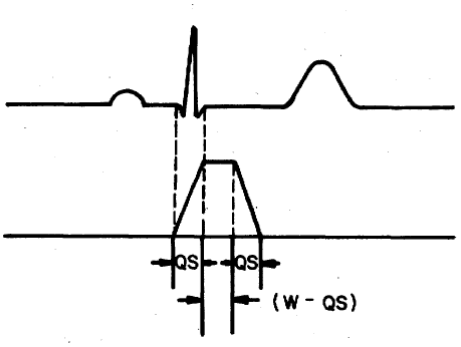

Figure 2.8: Relationship of a QRS complex (above) to the moving integration waveform (under) [8]

It is calculated from [8]

$$y(nT) = \frac{1}{N}(x(nT - (N-1)T) + x(nT - (N-2)T) + ... + x(nT)) \tag{2.19}$$

where N is the number of samples in the width of the integration window and T is the sampling period. The number of samples in the moving window is important: it should be approximately the same as the widest QRS complex possible. If the window is too wide, the integration waveform will merge the QRS and T complexes together and if it is too narrow, some QRS complexes will produce several peaks in the integration waveform. For a sample rate of 200 samples/s, the window is 30 samples wide ($150ms$) in [8].
The digital filter implementation is computed from [52]

$$y[n] = x[n-N] + x[n-(N-1)] + ... + x[n-1] + x[n] \tag{2.20}$$

$$z[n] = y[n]/N \tag{2.21}$$

The implementation from [9] uses a 16 (N) samples wide window ($80ms$).

2.3.1.5 Detection

Each time a peak is detected it is classified as either a QRS complex or noise, or it is saved for later classification. The algorithm uses the peak height, peak location and maximum derivative to classify peaks.

Threshold estimation The detection thresholds are calculated using estimates of the QRS peak and noise peak heights. They are automatically adjusted to float over the noise. [8] computes two thresholds, one high and one low, as follow

$$SPK = 0.125 * PEAK + 0.875 * SPK \tag{2.22}$$

$$NPK = 0.125 * PEAK + 0.875 * NPK \tag{2.23}$$

$$THRESHOLD_H = NPK + 0.25 * (SPK - NPK) \tag{2.24}$$

$$THRESHOLD_L = 0.5 * THRESHOLD_h \tag{2.25}$$

where $PEAK$ is the overall peak, SPK is the current estimate of the signal peak, NPK is the current estimate of the noise peak, $THRESHOLD_H$ is the high threshold and $THRESHOLD_L$ is the low threshold. The low threshold is possible thanks to the improvement made by the BPF.

Two R-to-R intervals are computed in [8]. The first one is the average of the eight most-recent beats and the second one is the average of the eight most-recent beats having R-to-R intervals that fall within certain limits. The reason for maintaining these two separate averages is to be able to adapt to quickly changing or irregular heart rates.

$$RRAVERAGE_1 = 0.125 * (RR_{n-7} + RR_{n-6} + ... + RR_n) \tag{2.26}$$

$$RRAVERAGE_2 = 0.125 * (RR'_{n-7} + RR'_{n-6} + ... + RR'_n) \tag{2.27}$$

where RR_n is the most-recent R-to-R and RR'_n is the most recent R-to-R interval that fell between the acceptable low and high limits. Since [9] only uses the first R-to-R interval, those limits calculation won't be detailed but can be found in [8].

Initialisation In [9], the beat detector must begin with some initial threshold estimate. The maximum peaks in eight consecutive 1-second intervals and are used as the initial eight values in the QRS peak buffer,while the initial eight noise peaks are set to 0. Then, the initial threshold accordingly. The eight most recent R-to-R intervals are set to 1 second.
When no beats are detected for eight seconds, the detection thresholds are reset in the same way that they were initialized.

Detection rules Using the calculated thresholds and R-to-R interval, the detection rules from the ULP implementation are the following, described in [9]:

1. Peaks are held for $200ms$ to avoid multiple detection on one peak. Everything else in this time window is ignored

2. If a peak occurs, check if it represents a baseline shift. It is the case if the raw signal contained both positive and negative slopes

3. If a peak occurs, check if it represents a T-wave. It is the case if it has occurred within $360ms$ of a previous detection and the maximum derivative in the raw signal is at least half the maximum derivative of the previous detection

4. If a peak is under the detection threshold ($THRESHOLD_H$), it is treated as noise.

5. There is an emergency rule in case no peak has been detected in 1.5 R-to-R intervals ($RRAVERAGE$): the lower threshold ($THRESHOLD_L$) is used and the peak followed the preceding detection by at least 360 ms is classified as a QRS complex

Detection delay When a QRS complex is detected, the beat is stored in a buffer and a detection delay is computed. Delay will determine if a sufficient time window as passed to proceed to feature extraction Delay is the number of samples that have occurred since the last QRS complex occurred. If a peak is detected, the delay will be the sum of the filter delays, the moving window integration width, and the $200ms$ delay required for rule 1 for a total delay of $395ms$. If the peak is found using rule 5, the delay will be this delay plus half the average R-to-R interval. Consequently, the detection delay can easily vary.

2.3.2 Beat classification

The classification starts when the detection delay is sufficient and if at least two beats are stored in the beat buffer. First the features will get extracted, then the SVM classification will output the class of the beat. Finally, amplitude normalisation occurs for each classified beat.
The 3 classes considered are the normal heartbeats with sinus rhythm (N), supraventricular ectopic beats (S) and ventricular ectopic beats (V). As explained in 1.3.2, the characteristics of these arrhythmia are quite different: the V beat has a different QRS-complex shape and S beat is morphologically close to the N beat, only a missing a P-wave. Thus, the features used to distinguish those classes must therefore be different in each case.

2.3.2.1 Features extraction

Automated feature selection is performed for each classifier stage based on the same starting feature set. The feature vector components is summarized in table 2.5: the initial feature set includes 122 elements, from which the features have been selected for each class. [6]

- R-to-R intervals, R-to-R interval average and the difference to the average are used to highlight the heart rate variability

- Morphological information of the beat is extracted using downsampled raw signal samples at the location of the P and T waves and the QRS complex

	Feature description	Initial	V	S
R-to-R	RR_i	3	0	2
	$RR_{average}$	1	0	0
	$RR_i - RR_{average}$	3	0	2
Samples	P wave	12	0	0
	QRS complex	12	1	0
	T wave	6	1	0
DWT	L2 detail	35	2	0
	L3 detail	21	4	2
	L4 detail	14	2	2
	L4 approximation	14	0	1

Table 2.5: Initial and selected feature sets distribution, adapted from [6]

- Spectral information is obtained using the coefficients of a DWT obtained with a Daubechies 4-level decomposition [33]

2.3.2.2 SVM classification

RBF kernel has been chose because it performs better than linear and polynomial kernels, especially for discriminating ventricular arrhythmia. [6]

2.3.2.3 Amplitude normalisation

The purpose of amplitude normalisation is to standardise the variability of amplitudes that may occur from patient to patient. An adaptive filter is used on the input sample in order to get a relatively constant amplitude.

Chapter 3

Parallelisation

This chapter approaches the concept of parallelisation, which is a central point behind the optimizations that will be presented in chapter 4. Parallelisation is almost necessary for applications where time and performance constraints are important. The application on which this thesis focuses needs to consume as little energy as possible while maintaining an adequate level of performance: it is therefore required to adopt the necessary optimisation techniques compatible with the hardware architecture on which the algorithm runs. Thus, the concept of parallelisation, and the resulting vectorization, are very relevant; they are the answer to the research question about how to optimize an ULP arrhythmia detection algorithm.
The first part of this chapter introduces to the motivations behind the use of parallel programming and its paradigms. Then, the concept of vectorization is widely explored from different points of view. Finally, two ARM architectures are presented: the current architecture upon which the application is built and the target architecture where optimization and parallelisation could prove to most fruitful.

3.1 Parallelism programming

Large problems can often be divided into smaller ones, which can then be solved at the same time. Parallel computing is a type of computing architecture in which these problems are broken down into smaller tasks and all of them are processed at the same time, instead of running all the tasks one at a time. [10]

3.1.1 Motivations

The main benefits of parallel computing is the increase of performance. A slow, sequentially executing program, can run much faster when it is built using the concepts of parallel computing. It is increasingly relevant for all computing platforms; most electronic devices sold today include multiple processing cores and require parallel programs to yield the best performance. Moreover, high performance computing (HPC) required the use of computing paradigms able to provide answer to challenges the resources at hand. While HPC is commonly associated with scientific research at supercomputing facilities (meteorology and such), it also encompasses applications requiring a significant amount of data processing or low latency responses. [10]

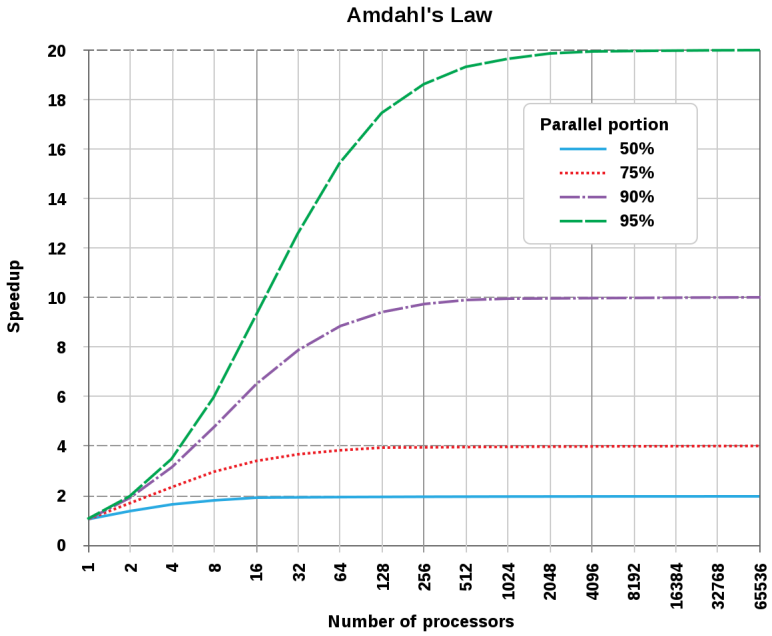

Figure 3.1: Graphical representation of Amdhal's law [53]

Assuming that a task can be divided into a parallelizable part and a serial, non-parallelizable part, Amdhal's law (figure [53]) expresses the theoretical evolution of the speedup in latency that can be expected of a system whose resources are improved.

$$S_{latency}(n) = \frac{T_{serial}}{T_{parallel}} = \frac{(1-p)+p}{(1-p)+\frac{p}{n}} = \frac{1}{(1-p)+\frac{p}{n}} \tag{3.1}$$

where T is the execution time, n is the number of available cores, p is the fraction of the execution time of the task that can be made parallel and $(1-p)$ is the

fraction of the execution time of the task that remains serial. [53]
Ideally, the speedup from parallelisation should be linear: doubling the number of processors should halve the runtime. Amdahl's Law implies that there's a limit on how much faster the original task can be processed by using additional cores: the speedup potential is limited by the serial part of the task, since it does not benefits from the increase of available cores. Indeed, as the number of cores available increases, the speedup can be rewritten as follow

$$S_{latency}(n) = \frac{1}{(1-p)} \tag{3.2}$$

Therefore, the execution time will be ultimately limited by the unparallelizable part of the task. [53]

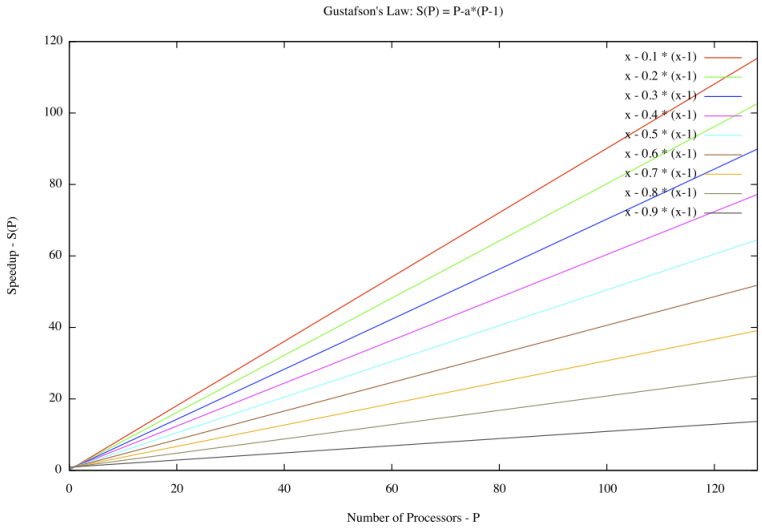

Figure 3.2: Graphical representation of Gustafson's law [54]

However, Amdahl's law only applies to cases where the problem size is fixed. In contrast, Gustafson's law (figure [54]) shows that scaling up a problem is primarily limited by the number of available cores n, giving us a more realistic and optimistic assessment of parallel computing performance. It originates from Gustafson's observation that problems to scale up to match the available computing power, meaning that the parallel portion of a program increases as the number of cores available increases.

$$S_{latency}(n) = \frac{T_{serial}}{T_{parallel}} = \frac{(1-p)+np}{(1-p)+\frac{np}{n}} = (1-p)+np \tag{3.3}$$

The unparallelizable portion of the program limits performance in both cases, but it is more detrimental when the problem size cannot scale with the number of cores.

[54]
Another reason to parallelize a task is to use more memory than is available on a single device. Parallel processing allows programs to use the massive amounts of memory available on supercomputers. [55]

3.1.1.1 SleepRider benefits

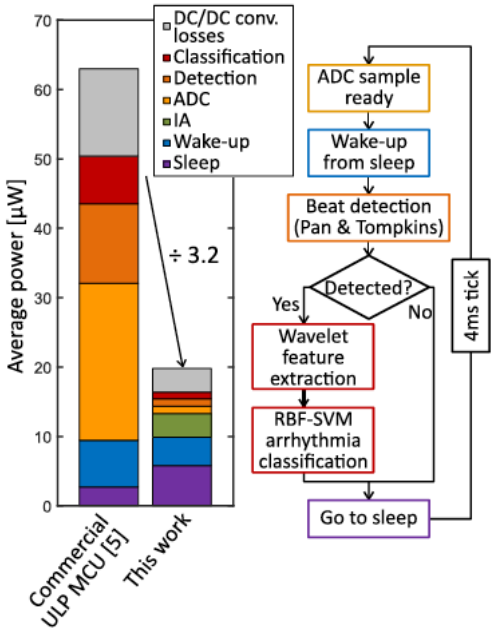

Figure 3.3: SleepRider tasks workflow & energy breakdown [6]

In the SleepRider arrhythmia detection application, there is wake-up interrupt every time a sample of an ECG is sent to the beat detection algorithm. As shown in 3.3, the wake-up routine accounts for around 20% of the power consumption. By applying the concepts of parallel computing, there is a possibility to reduce the amount of wake-up interrupts, which in turn may reduce the power consumption.

3.1.2 Paradigms

Instruction-level parallelism, thread-level parallelism and data-level parallelism are three types of parallelism that can be exploited by computers to gain higher performance. The first approach attempts to reduce the runtime of a program by executing multiple instructions from the same program concurrently, while the second approach executes independent programs or parts of a single program

simultaneously using different channel of execution, called threads. In the third approach, instructions from a single program operate concurrently on several data. Flynn's taxonomy is a classification of computer architectures which shows different

	Single datum	Multiple data
Single instruction	*SISD* typical processor	*SIMD* vectorization
Multiple instructions	*MISD*	*MIMD* multicore processor

Table 3.1: Flynn's taxonomy

types of parallel computing architectures. [56]

SISD A SISD processor is sequential and exploits no parallelism in either the instruction or data streams. A single instruction is executed at a time, and data elements are also dealt with one by one.

SIMD A SIMD processor differs from the SISD by incorporating vector processing into its operations: the same instructions is executed in parallel in different data. SIMD processing being the main field of application this master thesis, it will be detailed further in section [**section:simd**].

MISD MISD is an uncommon architecture. One example of a MISD processor is a set of digital filters operating in parallel on a stream of data.

MIMD MIMD computing consist of separates nodes operating independantly while being connected with efficient and high speed communication network. Memory space is either shared or distributed.

3.2 Vectorization

Vectorization is the process by which mathematical operations found in loops are executed in parallel on special vector hardware found in central processing unit (CPUs) and coprocessors. Vector instructions are a primary example of SIMD parallelism in modern CPU, where parallelism is achieved by executing operation on shorter operands (8-bit, 16-bit, 32-bit operands depending of the architecture). [56]
This section will discuss vectorization from three perspectives. The hardware

perspective will analyze how special registers and functional units allow parallelism for vector operations on arrays of data. Then, the compiler perspective will explore how compilers see and try to vectorize computations from software codes. Finally, the user perspective will analyze how a code can be written in a manner that allows the compiler to deduce that vectorization is possible.

3.2.1 Hardware perspective

3.2.1.1 Vector processing

A vector processor implements an instruction set that operates on one dimensional arrays, called vectors. Coincidentally, the size of its registers is adapted to these vector operations. Number of data elements per vector is referred to as the vector length. In a vector processor, instruction operates on multiple data elements in consecutive time steps. Thus, vector functional units are pipelined and each pipeline stage operates on a different data element. The architecture of a vector processor answers constraints that are inherent to vectors computations.

- It must be able to load and store vectors.
- It must be able to operate on vectors of different lengths.
- It must be able to track vector elements stored apart from each other in memory.

In order to solve the first and most important constraint, a typical vector processor has between 8 and 32 registers, either 64- or 128-bits long, called vector registers. They differ from scalar registers and are only relevant when vector instructions are used. A vector length register solves the second issue while the third issue is solved by a vector stride register. The main drawback of this type of processing is that memory can easily become a bottleneck.

3.2.1.2 SIMD processing

A SIMD processor performs the same instruction on multiple data points concurrently. It aims to take advantage of instruction-level parallelism. It can be seen as the successor of vector processors, because instruction operates on multiple data elements at the same time. In array processor, each parallel processing unit has its own separate and distinct memory and registers. In a pipelined processor, the parallel processing units share the same memory space. [56]
This type of processor requires explicit SIMD instructions from the programmer, so the code must be written with SIMD considerations in mind.

3.2.2 Compiler perspective

Most compilers are capable of vectorizing code automatically. However, knowing how the compiler will perform it is a huge advantage when designing a software application. The relevant compiler is the ARM Compiler, since the target hardware are built around ARM microcontroller. The ARM Compiler for embedded generates code for running fast, compact, and energy-efficient applications on ARM cores. [57]
The optimization level is the degree to which the compiler will optimize the code it generates and is controlled by the $-O$ flag. For the ARM compiler, automatic vectorization occurs at an optimization level of $-O2$ or higher. [58] The assembly instructions available depends on the target architecture: both ARMv7 and ARMv8 vectorization capabilities will explained in section 3.3.
Unlike GCC (the very popular GNU compiler), ARM Compiler does not provide compilation reports. It means that assessing whether vectorization happened requires analysis of the assembly code both with auto-vectorization on and off.

3.2.3 User perspective

Enabling automatic vectorization capabilities of compilers does not mean that nothing can be done to vectorize code other than making sure that the right flag is enabled when compiling. It is important to be aware of the challenges the compiler is facing in order to make it easier.

3.2.3.1 Vectorizable loops

Loops are the key target of the compiler when it is trying to vectorize a code. There are basics requirements for vectorizable loops for most compilers. [59]

Countable loop The total number of loop iterations must be known immediately before the loop executes, in order to be unrolled. As a consequence, break statement should be avoided. Exiting the loop should not be an option, otherwise, the compiler won't know the number of iterations at runtime.

No function calls From an assembly point of view, a function call is implemented as a series of complicated steps: a new stack has to be created, variables have to be pushed on it, jumping to a location in memory and executing instructions present at that location. None of these operations are vector operation, so function calls makes a loop ineligible - as well as potentially harming performance.
There is an exception: function calls that the compiler can replace with inline vector instructions such as math functions $sin()$, $cos()$, etc.

Straight control flow There should be no jumps or branches such as switch statements. Branching and conditionals often cannot be represented as vector instructions. Masked assignment is allowed: in that case, the compiled code follows both branches of the conditional (the then and the else clauses) using vector instructions. [55]

3.2.3.2 Data dependencies

Data dependencies occur when a statement refers to a variable of a preceding statement. When executing loop unrolling, the compiler may alter the original ordering of the loop and problems may arise when an operation in one iteration depends upon the result of a previous iteration. The compiler has to make sure that there are no data dependencies between iterations to make sure that the result would be exactly the same as the sequential execution of the code.
There are four types of dependencies.

Read after write (RAW) dependency It occurs when the values of variables involved in a loop iteration are determined in a previous loop iteration. RAW dependency is not vectorizable.

```
1 int a[3] = {0,1,2,3};
2 int b[3] = {4,5,6,7};
3 for (int i=1; i<4; i++) {
4       a[i] = a[i-1] + b[i];
5 }
```

Code 3.1: Example loop 1

```
a[1]= a[0] + b[1] =  0 + 5 -> a[1]=  6
a[2]= a[1] + b[2] =  5 + 6 -> a[2]= 11
a[3]= a[2] + b[3] =  11 + 7 -> a[3]= 18

a={0, 5, 11, 18}
```

Code 3.2: Example loop 1 - sequential execution

The code 3.1 bears no data dependency when executed sequentially. The first loop iteration performs an addition and writes a value to $a[1]$. The second iteration then reads the value of $a[1]$. The execution is as shown in 3.2.

```
a[i-1]= {0,1,2,3}
b[i]= {4,5,6,7}
a[i]= a[i-1] + b[i] = {0,1,2,3} + {4,5,6,7}

a={0, 5, 8, 10}
```

Code 3.3: Example loop 1 - parallel execution

Now, if we execute all the *for* loop operations simultaneously, the result would be different as shown in 3.3. This demonstrates why read after write dependencies are not vectorizable, as computation on vector hardware would result in incorrect results.

Write after read (WAR) dependency It is the opposite of RAW: a value is read in an operation in an earlier iteration, then written to in a later iteration. WAR dependency is vectorizable.

```
int a[3] = {0,1,2,3};
int b[3] = {4,5,6,7};
for (int i=0; i<3; i++) {
     a[i] = a[i+1] + b[i];
}
```

Code 3.4: Example loop 2

```
a[0]= a[1] + b[0] =  1 + 4 -> a[0]= 5
a[1]= a[2] + b[1] =  2 + 5 -> a[1]= 7
a[2]= a[3] + b[2] =  3 + 6 -> a[2]= 9

a={5, 7, 9, 3}
```

Code 3.5: Example loop 2 - sequential execution

```
a[i+1]= {1,2,3}
b[i]= {4,5,6}
a[i]= a[i-1] + b[i] = {1,2,3} + {4,5,6}

a={5, 7, 9, 3}
```

Code 3.6: Example loop 2 - parallel execution

The results from 3.4 are the same sequentially (3.5) and parallely (3.6). Hence why RAW is vectorizable.

Write after write (WAW) dependency It occurs when multiple loop iterations can writer over the value of a variable. WAW is not vectorizable because if an identical address exists for any two vector store operations, the result to be stored is indeterminate.

Read after read (RAR) dependency Despite its name, RAR is not a dependency in the sense that if a variable is not written, it does not matter how often it is read. RAR is vectorizable.

3.2.3.3 Pointer aliasing

Pointer aliasing yield dependencies to pointer specific language like C or C++. Pointer aliasing is using pointers for array addresses in arithmetic operations. Array data identified by pointers in C can overlap, because the C puts few restrictions on pointers. Because of this, the compiler often take a conservative approach when trying to vectorize a code including pointer aliasing.

```
1  void test(int *a, int *b, int *c) {
2      for (i=0; i<5; i++) {
3          a[i] = b[i] + c[i];
4      }
5  }
6
7  void main() {
8      int a[5], c[5];
9      test(*a, *(a-1), *c);
10 }
```

Code 3.7: Example loop 3

Example from 3.7 shows a case of a valid C code where the referenced pointers are overlapping: it yields a RAW dependency and, as such, is not vectorizable. This problem can be fixed marking up the code with the *restrict* keyword: it assures the compiler that the memory regions addressed by the pointers do not overlap with each other. Then, the compiler does not need to store a copy of the source array when computing and writing in the destination array. [60]

3.3 ARM architectures

The microprocessor targeted is a Cortex-M processor, as Cortex-M processors are specifically designed for microcontrollers, thanks to their optimized cost and energy-efficient characteristics. [57] [58]

3.3.1 ARMv7: Cortex-M4

The Cortex-M4 (CM4) has many specifications (see figure 3.4) and is designed for high-efficiency signal processing functionality with the low-power, low cost and ease-of-use benefits. It runs on 32-bit ARMv7-M architecture. The most important characteristics that made CM4 the right choice is to use CMSIS-SIMD library. [61] This library allow to use of 8 or 16-bit SIMD arithmetic. This SIMD extension increase the processing capability without materially increasing the power consumption is completely transparent to the operating system. Performance is achieved with a "near zero" increase in power consumption on a typical implementation. It

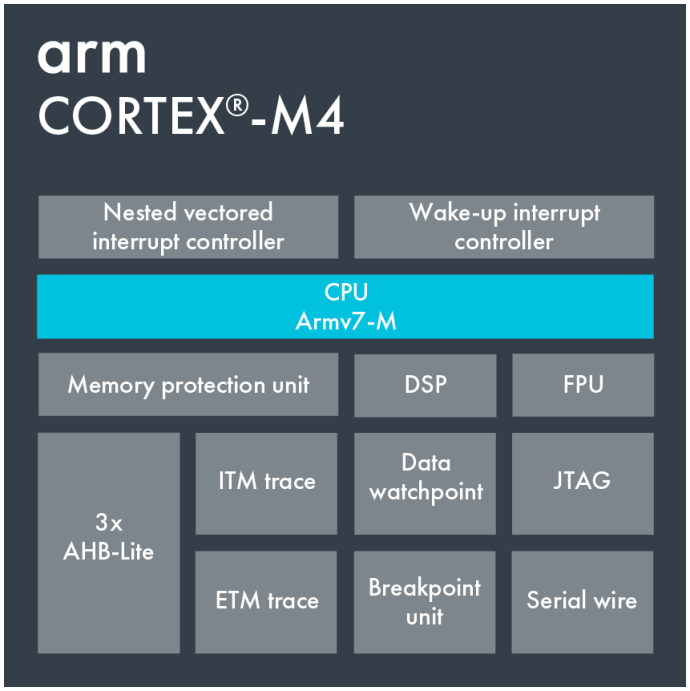

Figure 3.4: Cortex-M4 specifications [11]

provides intrinsics, which are functions call that the compiler replaces with the corresponding assembly SIMD statement. It offers the best compromise between the ease of coding of high-level language like C and the fine-tuning ability of writing in assembly.

The current version of the SleepRider algorithm runs on a microcontroller with a Cortex-M4 core which means the SIMD extension is readily available to optimize the software implementation.

3.3.2 ARMv8: Cortex-M33

The Cortex-M33 (CM33) has many upgrades (see figure 3.5) over the CM4 core. It includes the same SIMD extension as the CM4. However, because it runs on ARMv8.1-M architecture, the CM33 core benefits from a powerful extension: Helium.

Ideally, the SleepRider could benefit from a port, going from its current CM4 processed microcontroller to a CM33 processed one.

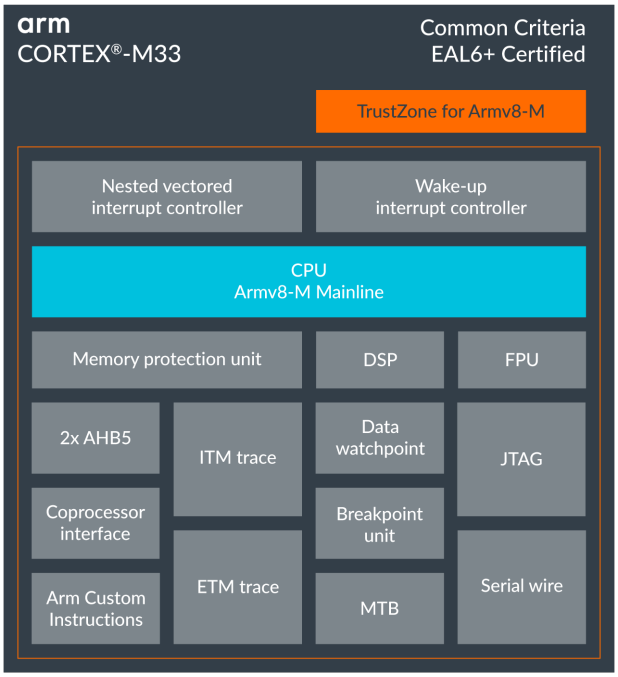

Figure 3.5: Cortex-M33 specifications [12]

3.3.2.1 Helium extension

Arm Helium is the M-Profile Vector Extension for the Arm Cortex-M processor series. It provides significant performance boost for machine learning and digital signal processing applications. [62] Helium provides additional vector registers designed to do calculations simultaneously (as seen in figure 3.6), which increases performance and throughput. They are 128-bits registers, meaning they can contain two 64-bit integers, four 32-bit integers or single precision float, eight 16-bit integers or half precision float or sixteen 8-bit integers. It is a major upgrade on the SIMD instructions provided in the CMSIS library.

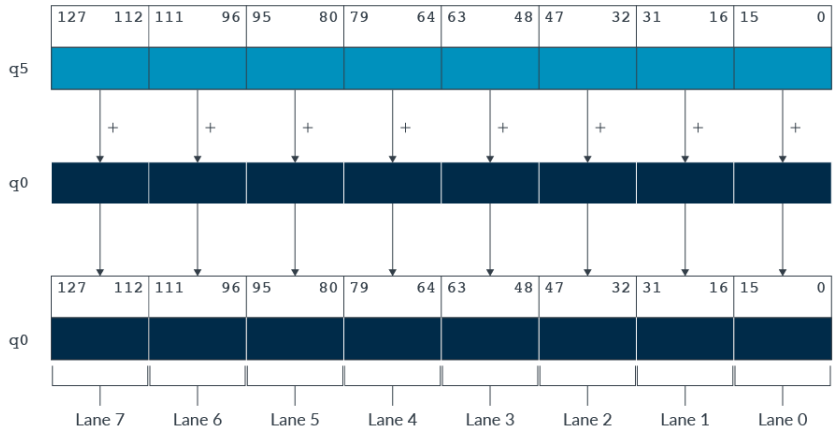

Figure 3.6: Operations on the Helium vector registers [62]

Chapter 4

Optimisation: batch implementation

A thorough analysis of the beat detection and classification algorithm in chapter 2 built a full understanding of the workflow of the application, while chapter 3 presented the key motivations behind parallelisation, as well as the paradigms that need to be taken into account when applying it to a software application. It is now time to try to answer the research question by applying the principles of parallelisation to the algorithm in order to achieve the stated goal of improving performance under ULP consumption constraints.

This chapter will present the main contribution from this master thesis. Although the final version provides the most conclusive answers to the analysis provided by the thesis, all version will be explored due to their respective importance in highlighting the specific features of coding while having hardware considerations in mind. The complete C implementation of all versions are available online on Forge UCLouvain, the reader can message the author at olivier.monzibila@student.uclouvain.be or the supervisor at david.bol@uclouvain.be.

The integrated development environment (IDE) used to run the C code and simulate the Cortex-M4 behaviour is the Keil μ Vision IDE. The debugger runs on a $12MHz$ clock. All the data have been acquired from the performance analyzer, which provides running time information for every function. The Cortex-M33 behaviour has not been simulated because of practical limitations that will be discussed in chapter 5.

The goal of this master thesis is to optimize the performance of an ULP arrhythmia detection algorithm through the use of parallelisation. Specifically, vectorization is the method that this work focused on. In this chapter, the role of profiling will be first highilighted. Characterizing the performance of an application

and pinning down the area that could benefits the most from vectorization is an essential step. After that, the different intermediary methods applied to optimize will be explained, while results will be shown and analysed. Finally, the stage is set for parallelisation to be applied to the code using SIMD intrinsics provided in ARMv7 architecture.

4.1 Profiling

Profiling measure performance characteristics of a running application, for the purpose of identifying compute-intensive areas that may be worth optimizing, in our case vectorizing. [63] It provides figure of merits, quantitative criteria, upon which the choice of target module to optimize can be made. This section describes how the choice of the functions to improve has been made.

4.1.1 Global profile

The goal here is to have a global view of the cycle consumption of the functions involved in the algorithm. Two separates approach have been made in order to gather profiling data. It is going to provide us with our first criterion.

4.1.1.1 Cycle counting

While observing the execution of the application, three cases have been highlighted

- **Case A**: no beat detected
- **Case B**: beat detected, no classification
- **Case C**: beat detected, classification

For each case, the amount of cycles that have passed has been counted (by deducting the cycle number at the end of a function with the cycle number at the start of the function). This method provides an overview of what part of the workflow is compute-extensive. Although these data are computed manually and thus are not totally accurate, they still give valuable tendencies and information. All the data and percentage can be found in appendix A.
Figure 4.1 shows us that cases A and C largely dominate the consumption breakdown. Detailed results from appendix A shows that case A amount for 51.5% of the cycle consumption while case C amount for 46.6% - bringing us to a total of 98.1%. We can dig deeper into the analysis of both cases, in order to pinpoint the most impactful part of the algorithm. The analysis narrows down the potential target

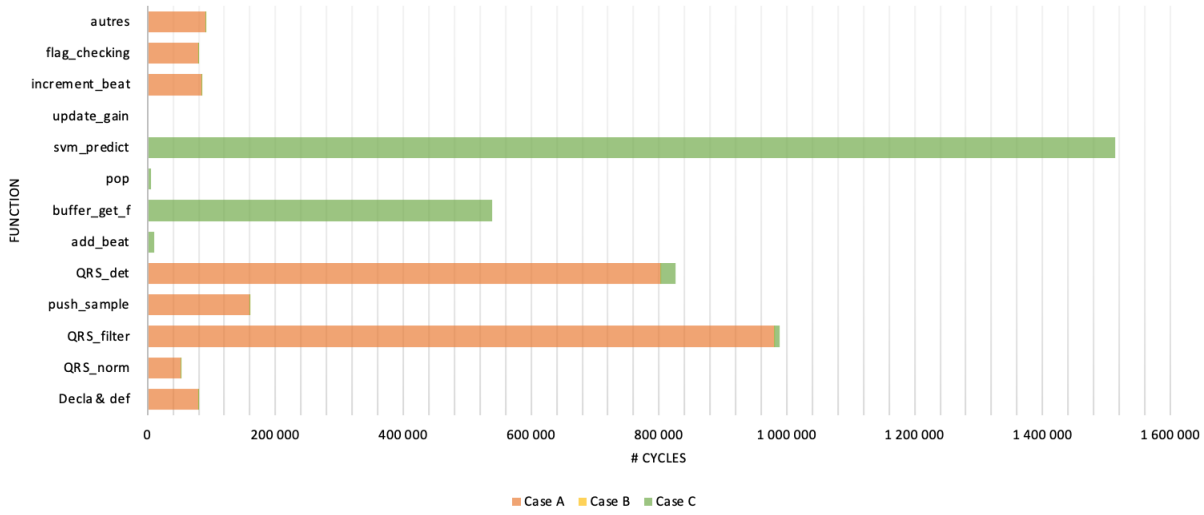

Figure 4.1: Cycle breakdown with cycle counting

functions to four functions only. Yet, they still consume 85.5% of total number of cycles.

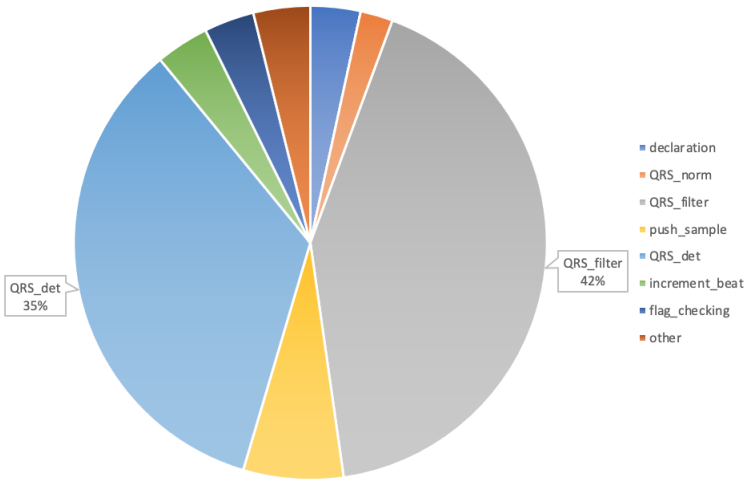

Figure 4.2: Cycle distribution - case A

Case A While this case has a relatively low amount of cycles per occurrence, the high amount of time the *for* loop is accessed makes up for it. Figure 4.2 shows the cycle distribution within this case. The two main beat detection functions, pre-

processing and detection, are the most expensive, amounting to 42.1% and 34.5% of the case A consumption respectively and 21.7% and 17.7% of total consumption respectively. The two main functions in Case A therefore consume a total of 39.4% of the total number of cycles of the application.

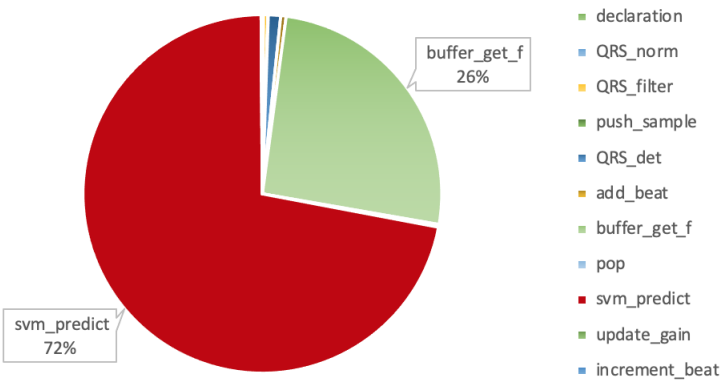

Figure 4.3: Cycle distribution - case C

Case C This case, on the contrary, occurs only a few times but to a great impact: each classification has a significant impact. Figure 4.3 shows the cycle distribution within this case. The main beat classification functions, feature extraction and SVM prediction, are the most expensive, amounting to 25.6% and 71.9% of the case C consumption respectively and 11.9% and 33.5% of total consumption respectively. When adding the beat detection (0.7%) to these two main functions, Case C therefore consume a total of 46.1% of the total number of cycles of the application.

4.1.1.2 μVision performance analyzer

μVision provides a performance analyzer that displays the information collected during debug execution. This method provides an overview of which module (which is a group of functions) is compute-extensive, as well as a much more precise data. All the data and percentage can be found in appendix B.
Figure 4.4 displays the performance of each module (for a total of 72 functions). From data gathered, the five most expensive modules ($> 5\%$ of total cycle consumption) are the SVM (33.4%), the pre-processing (23.5%), the threshold detection (16.1%), the direct wavelet transform (9.5%) and the wrapper that encompasses the whole process for each sample of ECG data (6.0%). That gives a total of 85.3% for 40 functions. Going down further in the analysis, the most expensive functions of each module allow us to narrow down the candidate for optimisation

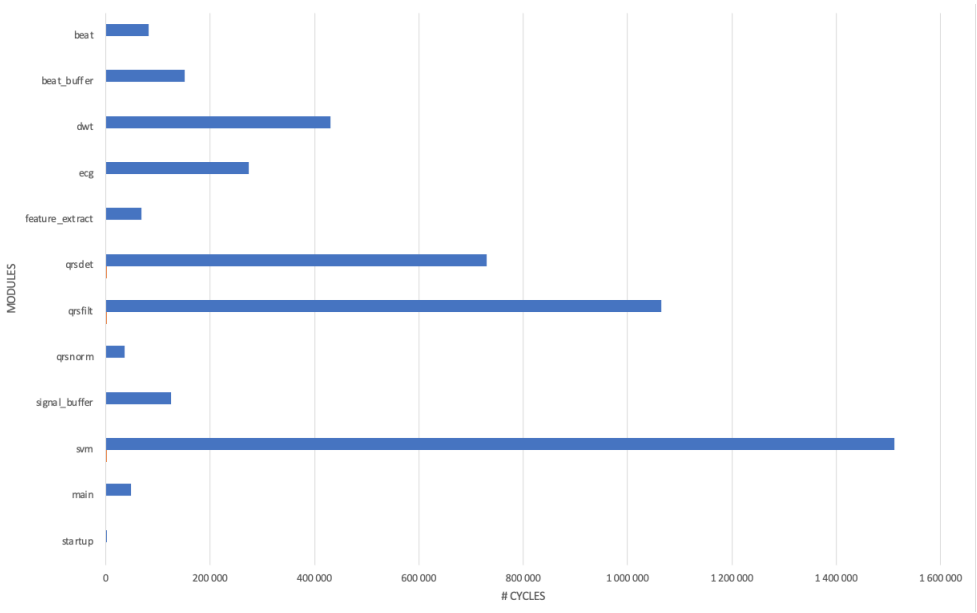

Figure 4.4: Cycle breakdown with performance analyzer

to 10 functions while still consuming 80.1% of the total cycle, as shown in figure 4.5.

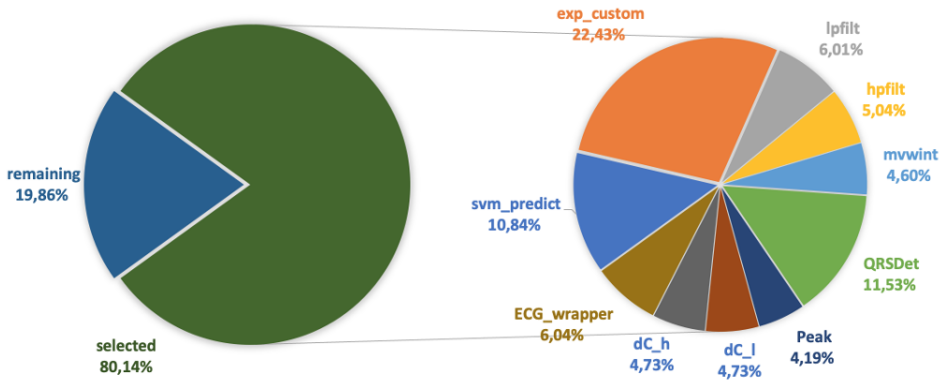

Figure 4.5: Selected functions impact on overal performance

4.1.2 Compiler optimisation profile

The second method is to try to profile the algorithm on different optimisation level (see section 3.2.2). Observing where the compiler focus its optimisation efforts gives the programmer an hint on where he should try to make an improvement.

It is even more relevant in our case, because our goal is to implement parallelism through vectorization.

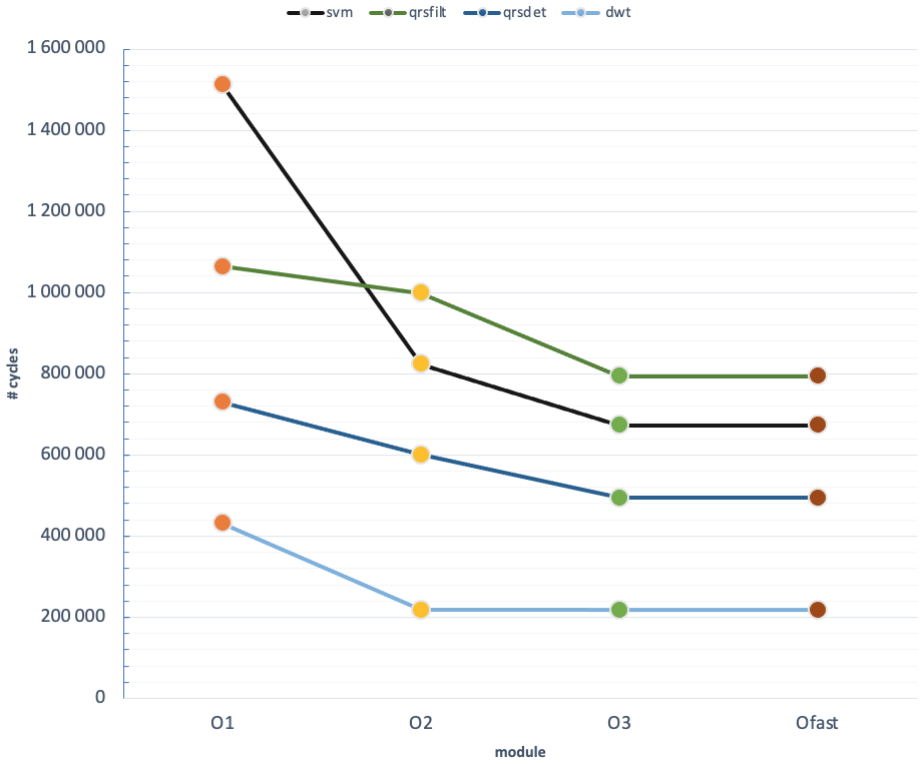

Figure 4.6: Comparison of module consumption (on selected functions) depending on optimisation level

Figure 4.6 displays how the compiler improves the performance, in respect to the optimisation level. The four chosen modules have been selected because they are the only four that showed significant normalized standard deviation (see C) through the different optimisation levels.
Those modules, that where already highlighted as the best potential targets earlier in the thesis, are greatly impacted.

4.1.3 Parallelisation potential

As mentioned in the prior discussion of Amdahl's law [53], the overall speedup due to vectorization is largely determined by the percentage of a code that is vectorizable. This provides a last criterion, qualitative this time, that can guide the selection of the module to optimize. The key to analyze here is to explore the

functions that were made candidate by the two previous methods and pinpoints those with the most "parallelizable" instructions.

- From these four modules, only the pre-processing module, qrsfilt is the most suitable candidate to parallelisation, because its digital filters uses a lot of arithmetic operations that could benefit from vectorization, depending on the architecture. Indeed, register size and available SIMD operations in the will tell if there is a possibility to proceed to vectorization.
- The SVM module could prove difficult to parallelize because it uses 64-bits long data word. On a processor such as the CM4, there is nothing that can be done on this module.
- Operations from the detection module are mostly tied to the calculation of threshold.
- The DWT module uses FIR filter, so vectorization is a viable option. However, the implementation from [9] uses *while* loop and thus violate a key rule about vectorization in loops (mentioned in section 3.2.3.1): the compiler cannot predict the length of the loop.

4.2 Preparation to parallelisation

There are several steps between the original source code and a vectorization-friendly code. These steps will be discussed in this section. The batcherisation is a first step that aimed to reduce the amount of pointwise operation and thus reducing the amount of wake-up interrupt. Then, the ping-pong approach is discussed. Its goal is to remove the necessity to use the ring buffer. Finally, two concurrent implementation for removing branches and their performance overhead are explained.
All data used for analysis can be found in appendix C.

4.2.1 Batcherisation

The qrsfilt module is initially designed to operate on a sample-to-sample basis, introducing a delay and a circular buffer for the filter functions. The goal is to send the filter functions a batch so that they can process on an array. The main benefits from this is the reduction of the wake-up interrupts frequency, which is a huge part of the power consumption (see 3.1.1.1).
The diagram from figure 4.7 shows us the difference the pointwise implementation and the batch implementation.

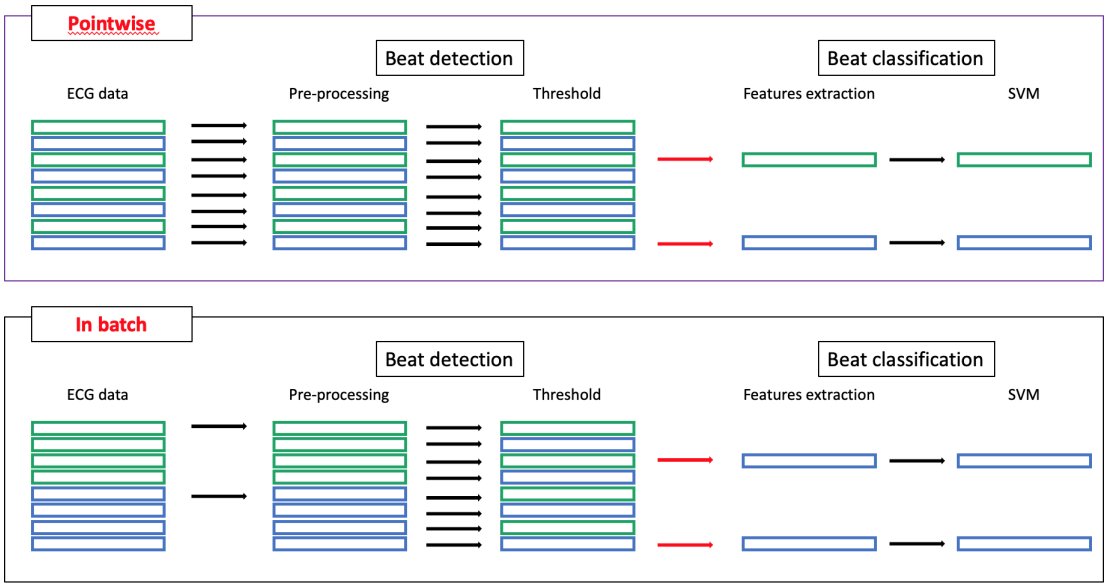

Figure 4.7: Difference between pointwise implementation and batch implementation

- It shows that only the pre-processing step has been targeted. The reason is that all the steps beyond pre-processing (from detection to classification) are designed to operate on a pointwise basis (because they have to compare the points to each other). Applying batcherisation on these other operations would come at the cost of accuracy: for example, the thresholds would be calculated after every batch instead of after every sample. Therefore, only the filters have gone through batcherisation.

- The amplitude normalisation step (see 2.3) occurs before the pre-processing filters. However, the normalisation factor is calculated only after a beat is detected, which can occur in the middle of a batch. Thus, there is gonna be an error between the results due this inevitable bottleneck.

4.2.1.1 Results

Figure 4.12 displays the result from the batcherisation and figure 4.9 displays the error induced by the normalisation step.

- The overall performance is worse than the base implementation of the filter (pointwise and with buffers), which is expected so far because there is an overhead to create and fill the batches. This step is preparation step, so the algorithm has not been improved yet.

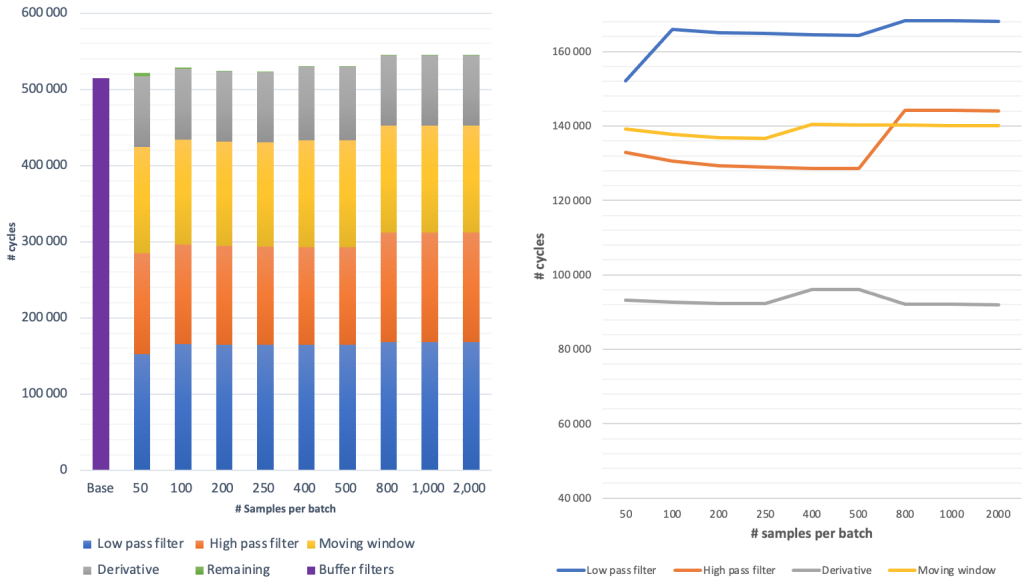

Figure 4.8: Results of batcherisation for different batch size

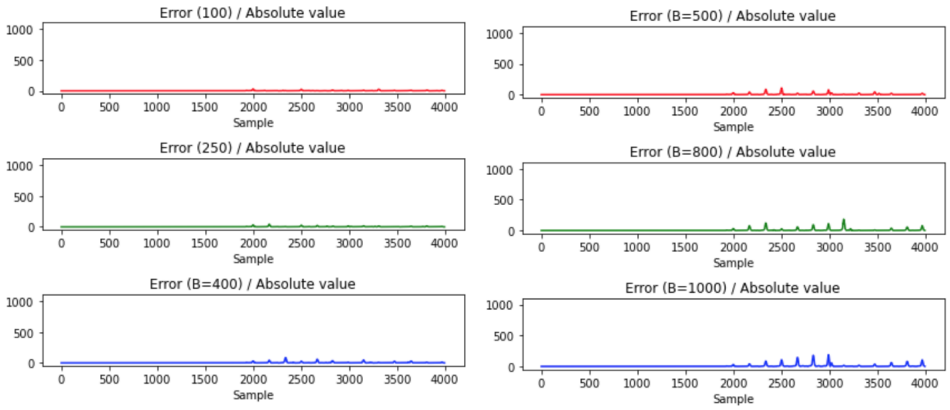

Figure 4.9: Absolute value of the error due to the normalisation step

- There is a monotonic increase of the error due to the normalisation step when the size of the batch is going up. It is expected because there are fewer "batch breaks" when the batch is bigger; that means there are less opportunities to update the amplitude with the normalisation function.

4.2.2 Ping-pong batches

Now that batches of sample are sent to the filters instead individual samples, the next step is to make the most of it by removing the circular buffers. These

buffers were only relevant in a pointwise operation because the was a need to keep old input values in memory.
The approach that has been implemented is the ping-pong buffers. It is a concept borrowed from the hardware side of a microcontroller, which use a peripheral called DMA (Direct memory access) to access memory without using CPU resources [64]. Ping pong buffers is a way of moving memory back and forth so that different parts of the system can use it without colliding. When one buffer is being written (role "A"), the data can be accessed from the second one (role "B"). Once the first buffer is completely filled, a flag is waived and both buffers switch role. [65]

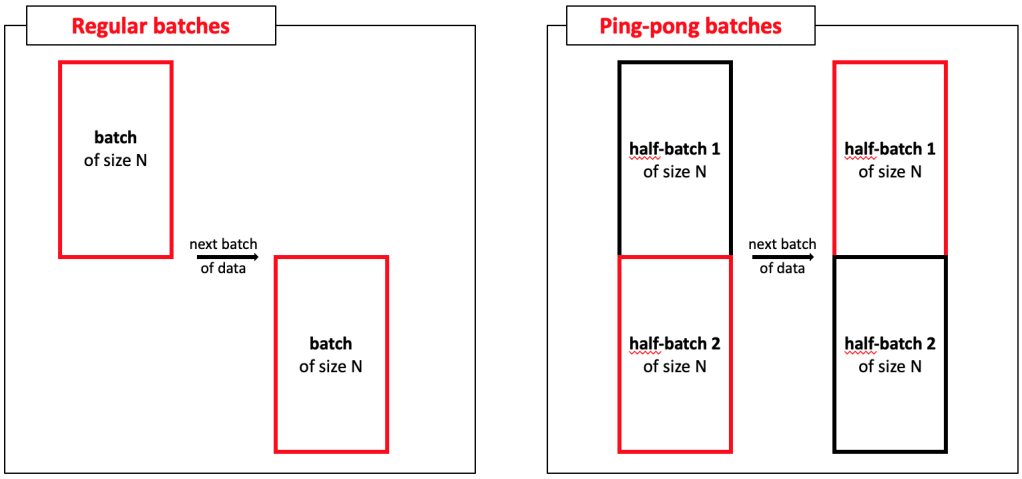

Figure 4.10: Difference between regular batch implementation and ping-pong batches implementation

In the current context, the ping-pong approach goal is to use batches twice as large as required in order to keep an access to older value. Therefore, filters write safely on one half of the batch while the other half can be accessed without the risk of data dependency hazard.
Figure 4.10 shows the difference between regular batch implementation and ping-pong batches implementation. The features of this step

- Because all the data required is now available in the double-sized batch, the circular buffers become obsolete.
- The algorithm must keep track of the half of the buffer it is currently writing in.
- Each filter of the pre-processing step has two cases: one if it is currently working in the first half, the other if it is working in the second half.

- If the half-batch being operated on is the first half, the previous data computed in the previous iteration sit at the end of the second half. The situation is shown in figure 4.11. As it stand, a conditional statement - checking in what half-batch must the filter look to get the required data - is necessary.
- Initialisation fills the first half of the batch with zeroes and the writing start in the second half of the batch.

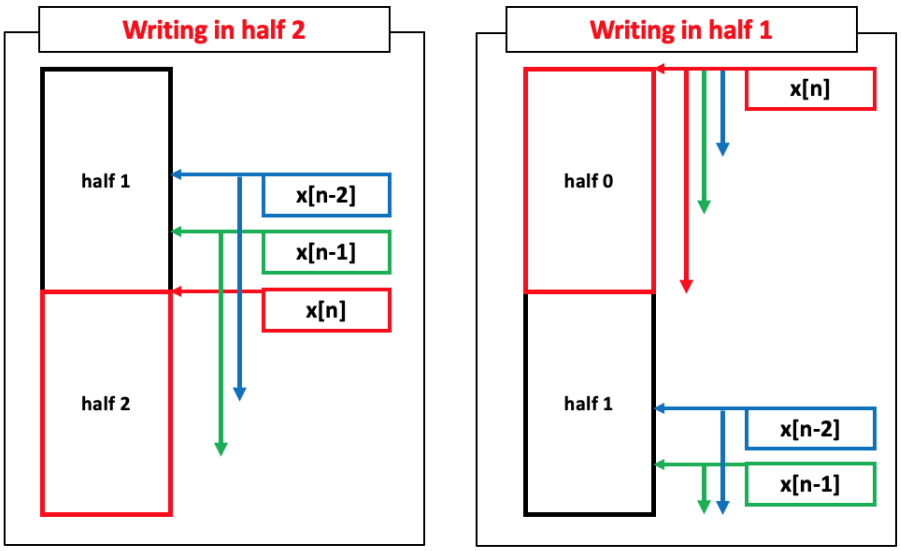

Figure 4.11: Difference in writing in the first half or the second half

4.2.2.1 Results

Figure 4.12 displays the result from the ping-pong batches and figure 4.13 displays the performance in each halves of the buffer.

- The overall performance is better than the original implementation of the filter. Now that the circular buffer usage has been removed, it was the expected outcome.
- Size of the batch is limited to 1000 because the batch are now twice as long to be able to operate on half batches.
- There is a monotonic increase of the performance when the size of the batches is going up. This is the expected behaviour, this time.

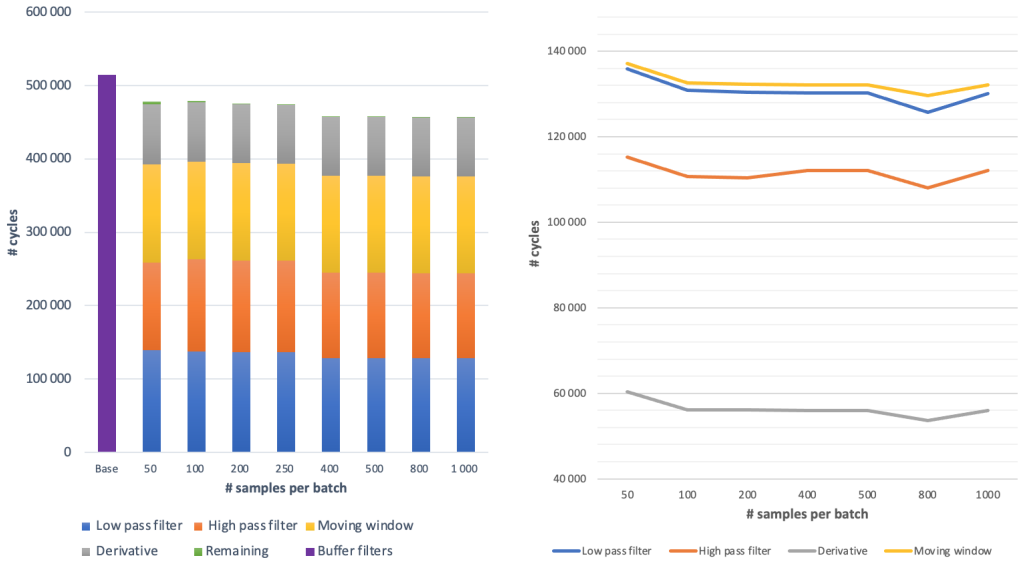

Figure 4.12: Results of ping-pong batches implementation

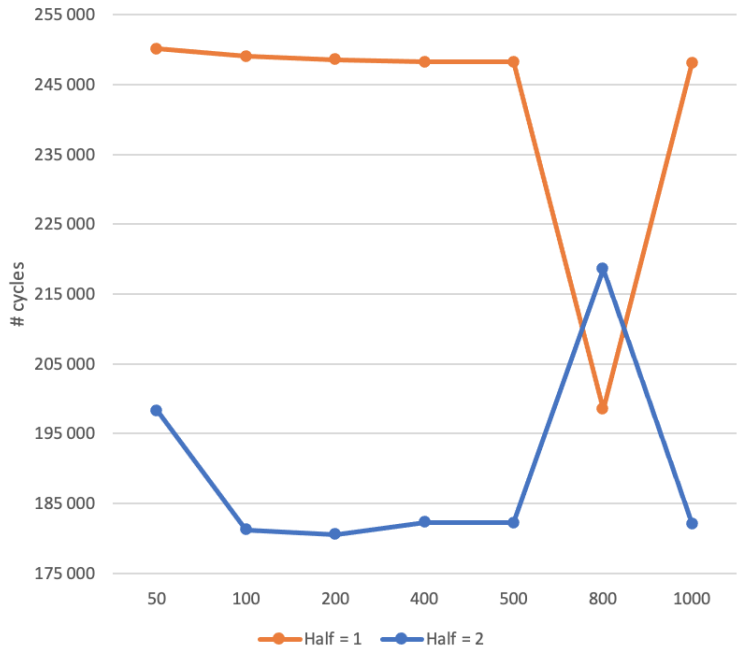

Figure 4.13: Performance in each halves of the buffer

- The performance in the first half is worse than in the second half. This was expected because the first half includes conditional statement to fetch data

in the end of the second half at the beginning of the loop.

- There is a monotonic increase of the performance when the size of the batches is going up, apart when the size of the batch is equal to 800.

4.2.2.2 Analysis

Difference of performance in halves In order to understand the difference in execution between the two halves, an analysis of the assembly is in order. It will show what instructions the compiler has interpreted for each halves. The analysis on the derivative (with batch size of 1000) because it is the simplest function and is given without context. Only the instructions relevant for the comparison have been kept.

```
1  SUBS R2, R2, #0x01 ; advance in the loop
2  LDR R12, [R0, R3, LSL #2] ; load sample[i]
3  ; conditional branch start
4  MOV R2, R3
5  CMP R3, #0x02
6  IT CC
7  ADDCC R2, R2, #0x7D0 ; 0x7D0 = 1000
8  ; conditional branch end
9  LDR R2, [R2, #-0x08] ; load sample[i-2]
10 SUB R2, R12, R2 ; compute
11 STR R2, [R1, R3, LSL #2] ; store the answer
```

Assembly code 4.1: First half - cycle count= 12/13

```
1  SUBS R2, R2, #0x01 ; advance in the loop
2  LDR R3, [R0, #0x08] ; load sample[i]
3  LDR R12, [R0, #0x04]! ; load sample[i-2]
4  SUB R3, R3, R12 ; compute
5  STR R3, [R1, #0x04]! ; store the answer
```

Assembly code 4.2: Second half - cycle count= 8

The two assembly codes show the difference in execution between the halves. The first half has to apply the conditional statement to check if the previous data are

at the end of at the start of the batch.
A theoretical estimate of the cycle consumption can be made thanks to data from [66]. The first half count is 12-13 (because the IT instruction can take 0 cycle or 1 cycle, depending on the preceding instruction) and the second half is 8. The theoretical ratio between the two is between $\frac{12}{8} = 1.5$ and $\frac{13}{8} = 1.625$. So the mean theoretical ratio is **1.563** while the experimental ratio is equal to **1.545**. This demonstrates that the analysis of this hypothesis is correct and therefore explains where the difference in performance between the two halves comes from.

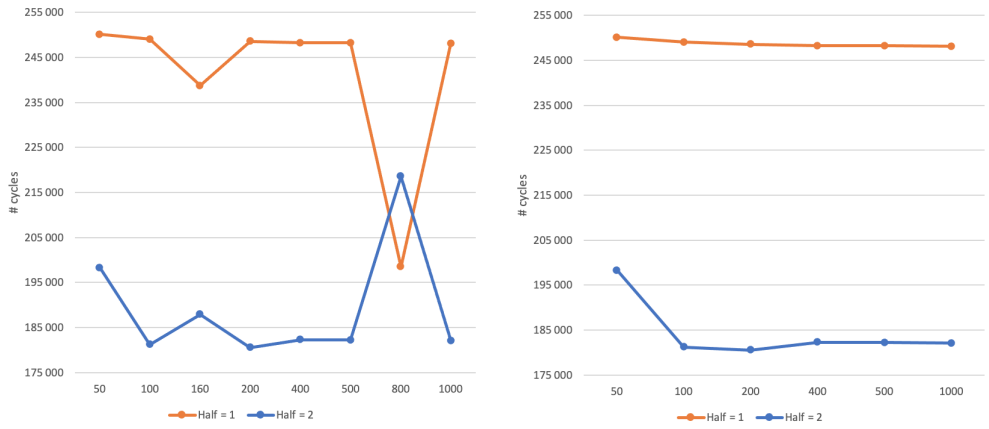

Figure 4.14: Performance in each halves of the buffer: with 160- & 800-samples batches vs without

Case of 800-samples batch The number of iterations N depends on the batch size B.

$$N = \frac{4000}{B} \tag{4.1}$$

If N is odd, the number of accesses to the first and second half-buffer are not equal.

$$B = 800 \longrightarrow N = 5 \tag{4.2}$$

In the case of 800-samples batch, since writing starts in the second half, there are 3 accesses in the second half and 2 in the first half.
To prove this point further, measurements have been taken on a batch: with 160-800 vs without 160-800. Results are shown in figure 4.18.

$$B = 160 \longrightarrow N = 25 \tag{4.3}$$

There are 13 accesses in the second half and 12 in the first half.

To explain this, one must look at the performances in each halves, explained and analysed previously. For the 800-samples case for example:

- In order to follow the monotonic trend, the first half should be accessed 2.5 times. Because $2.5 > 2$, it is accessed less that it should and the performance is accordingly better than it should be according to trend.
- In order to follow the monotonic trend, the second half should be accessed 2.5 times. Because $2.5 < 3$„ it is accessed more that it should and the performance is accordingly worse than it should be according to trend.

While the performance does not show monotonic increase when removing those two special batch size (see figure 4.18), the major bumps are removed and the behaviour is stable.

4.2.3 Branches removal

Conditional statement, on the software level, produces branches, on an assembly level. The way pipelined processor handle these branches is called branch prediction. It attempts to guess whether a conditional jump will be taken or not. [67] While it improves the way the processor uses its pipeline, it has a performance cost that is greater than a regular arithmetic operation. Aside from this cost, branches are not instructions that can vectorized (see section 3.1.1.1), hence the need to get rid of conditional statement if possible.

Conditional statements appear in each pre-processing filter due to the structure of the ping-pong batch, as explained above. Two methods have been implemented to deal with this.

4.2.3.1 Bit masking

The first way to deal with branches is to replace them with operations that do the same job. Conditional branches are based on whether a statement is true or false, so if this can be turned into some sort of mathematical operation, there is a work-around to a branch.
Figure 4.15 explains how the bit mask works and how it is computed.

4.2.3.2 *For* loop separation

The second way to deal with branches is to analyze when the branch will be accessed or not. The whole loop is then separated into several loops, making the job of the compiler much easier and removing the need of branches.

```
1 for(i=0;i<BATCH_LENGTH;i++) {
2       //Get the samples + apply index correction
3       s0= sample[i];
```

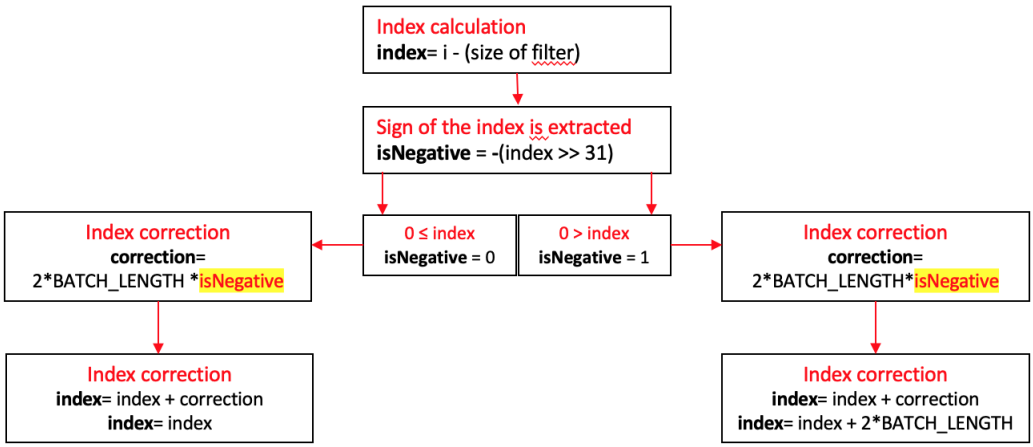

Figure 4.15: How the bit mask works

```
        s10=sample[i-DERIV_LENGTH + 2*BATCH_LENGTH*(i<DERIV_LENGTH)];

        //Proceed to the computation
        out[i] = s0 - s10;
    }
```

Code 4.1: Derivative before separating

```
for(i=0;i<DERIV_LENGTH;i++) {
        //Get the samples + apply index correction
        s0= sample[i];
        s10=sample[i-DERIV_LENGTH + 2*BATCH_LENGTH];

        //Proceed to the computation
        out[i] = s0 - s10;
    }
    for(i=DERIV_LENGTH;i<BATCH_LENGTH;i++) {
        //Get the samples + apply index correction
        s0= sample[i];
        s10=sample[i-DERIV_LENGTH];

        //Proceed to the computation
        out[i] = s0 - s10;
    }
```

Code 4.2: Derivative after separation

Code bits 4.1 and 4.2, shows how the separation is done and allows to get rid of the conditional statement, with the derivative being taken as example.

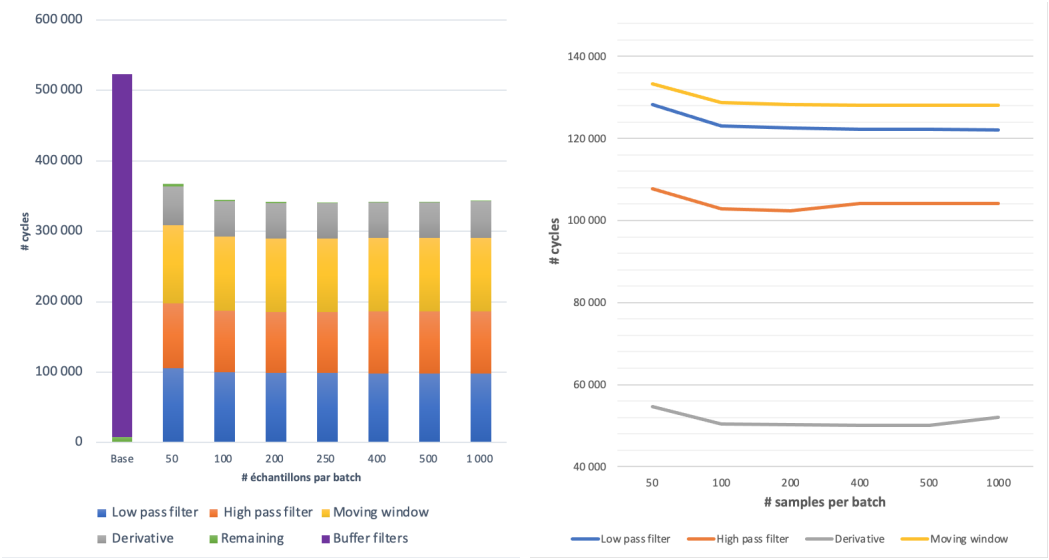

Figure 4.16: Results of branch removal through bit masking

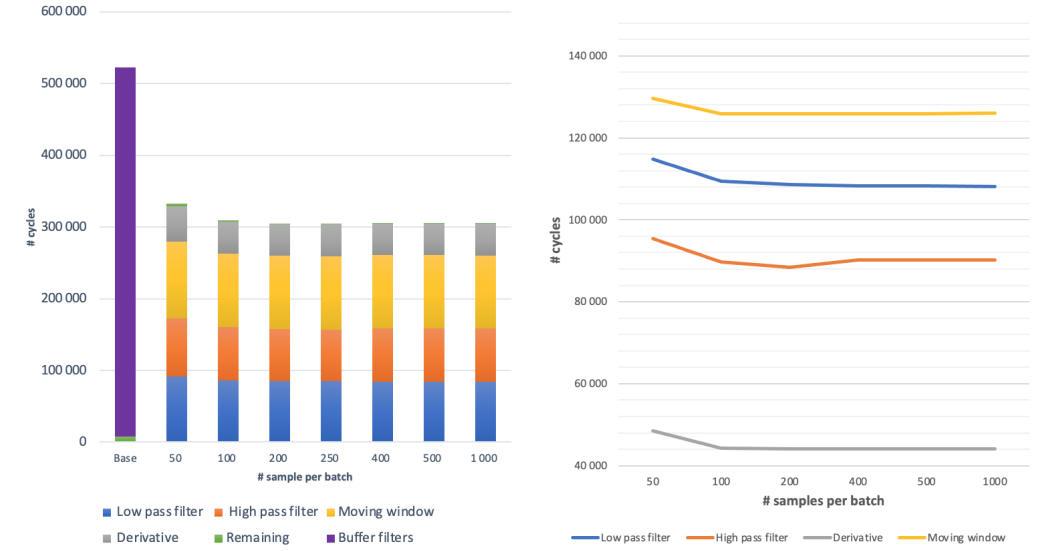

Figure 4.17: Results of branch removal through loop separation respectively

4.2.3.3 Results

Figures 4.16 and 4.17 display the result from the bit masking and the loop separation, respectively. Figures 4.18 displays the performance in each halves of the buffer.

- Since the analysis above has proved that 160 and 800 are not ideal batch size,

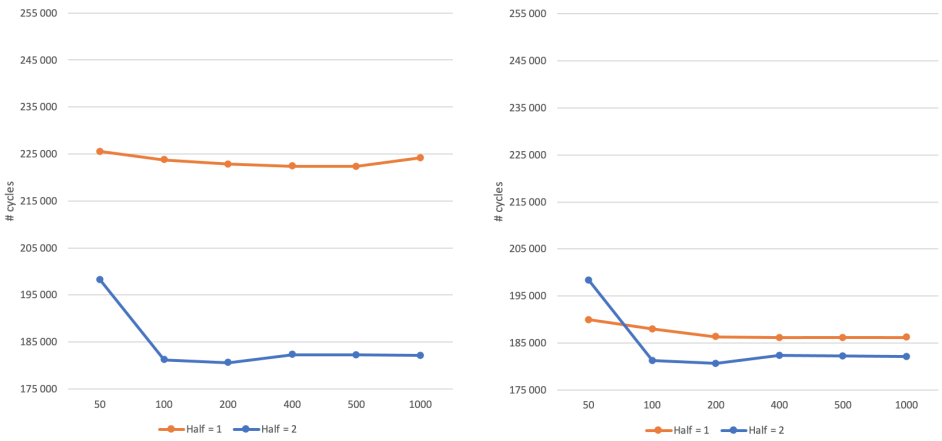

Figure 4.18: Performance in each halves of the buffer for bit masking & loop separation respectively

they have been removed from the measurements. Therefore the evolution of the performance when the batch size increases is monotonic.

- The overall performance is better than the original implementation of the filter, as expected to a greater impact than the ping-pong batches step. It is expected because there is a great overhead in branch prediction.
- Performance in the second half is exactly the same as in the previous step for both methods, as it should because nothing changes in the execution of this half.
- For the bit masking the performance in the first half is worse than in the second half, just as before, for the same reason.
- For the loop separation, the performances between halves are roughly the same. Both behaviour were expected.

4.2.3.4 Analysis - bit masking

Difference of performance in halves Analysing the assembly will allow to compute the theoretical ratio between the halves. Once again, the context is not important here. The derivative for a batch of 1000 samples is analysed.
From codes 4.3 and 4.4, the theoretical ratio [66] is $\frac{11}{8}$ =**1.375** and the experimental ratio is **1.364**.

```
ADDS R3, R3, #0x01 ; advance in the loop
; bit masking start
SUBS R2, R3, #2
ASR R2, R2, #31
SMLABB R2, R2, R12, LR
; bit masking end
LDR R4, [R0, R3, LSL #2] ; load sample[i]
LDR R2, [R2, R3, LSL #2] ; load sample[i-2]
SUB R2, R4, R2 ; compute
STR R2, [R1, R3, LSL #2] ; store the answer
```

Assembly code 4.3: First half (bit masking) - cycle count= 11

```
SUBS R2, R2, #0x01 ; advance in the loop
LDR R3, [R3, #0x08] ; load sample[i]
LDR R12, [R3, #0x04]! ; load sample[i-2]
SUBS R3, R3, R12 ; compute
STR R3, [R1, #0x04]! ; store the answer
```

Assembly code 4.4: Second half (bit masking) - cycle count= 8

Effect of branch removal To understand how this version of branch removal improves on the previous step, comparing the assembly execution of the first halves is gonna give us once again the theoretical tools to validate the method. In order to do so, a comparison between the ping-pong batches (see code 4.2) and the bit masking branch removal (see code 4.3).
The theoretical decrease of cycle output [66] is between $\frac{11}{12} = 0.917$ and $\frac{11}{13} = 0.846$. So, the mean theoretical decrease **88.2%** is while the experimental decrease is **88.3%**.

4.2.3.5 Analysis - loop separation

Difference of performance in halves The conditions are the same as above (see 4.2.3.4).
From codes 4.5, 4.6 and 4.7, the theoretical ratio [66] is $\frac{8}{8}$ =**1** and the experimental ratio is **1.001**.

```
1 ADDS R3, R3, #0x01 ; advance in the loop
2 LDR LR, [R0, R3, LSL #2] ; load sample[i]
3 LDR R2, [R2, #-0x08] ; load sample[i-2]
4 SUB R2, LR, R2 ; compute
5 STR R2, [R1, R3, LSL #2] ; store the answer
```

Assembly code 4.5: First half (loop separation **when i $<$ 2**) - cycle count= 8

```
1 SUBS R2, R2, #0x01 ; advance in the loop
2 LDR R12, [R0, #0x04]! ; load sample[i]
3 LDR R3, [R0, #-0x08] ; load sample[i-2]
4 SUB R3, R12, R3 ; compute
5 STR R3, [R1, #0x04]! ; store the answer
```

Assembly code 4.6: First half (loop separation **when i $\geq$ 2**) - cycle count= 8

```
1 SUBS R2, R2, #0x01 ; advance in the loop
2 LDR R3, [R3, #0x08] ; load sample[i]
3 LDR R12, [R3, #0x04]! ; load sample[i-2]
4 SUBS R3, R3, R12 ; compute
5 STR R3, [R1, #0x04]! ; store the answer
```

Assembly code 4.7: Second half (loop separation) - cycle count= 8

Effect of branch removal The conditions are the same as above (see 4.2.3.4). The theoretical decrease of cycle output [66] is between $\frac{8}{12} = 0.667$ and $\frac{8}{13} = 0.615$. So, the mean theoretical decrease **64.1%** is while the experimental decrease is **64.7%**.

4.2.3.6 Comparison

Although these two methods achieve the same goal, their performances can be compared, as well as their respective potential to be parallelised.
Figure 4.19 shows the difference in results between the two removals methods: better results are obtained with the loops separation. From what we could gather from the assembly codes, the *for* loop separation only uses basic arithmetic operations (besides the loads and stores). This is a huge advantage for the final step of

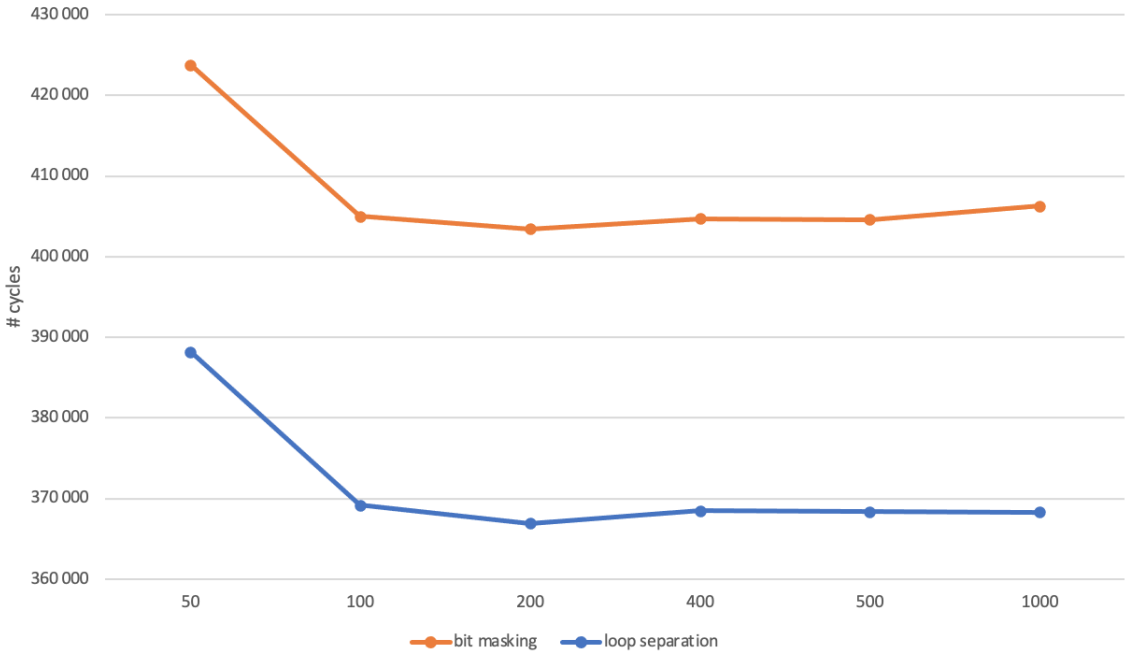

Figure 4.19: Comparison of branch removal methods

parallelisation, because the SIMD instructions available for the Cortex-M4 only handle addition and subtraction.

In conclusion, the loop separation method is selected as a basis to proceed to the parallelisation.

4.3 Parallelisation

After all the previous steps, the data pre-processing part of the algorithm is ready to be parallelized. Two preparation steps are going to be described: data type change and *for* loop unrolling. Finally, the implementation of SIMD intrinsics is going to be explained and the results will be analysed.

The SIMD instructions on the Cortex-M4 only allow vectorisation on 8-bit or 16-bit data, so data type must be converted. Furthermore, *for* loops have to modified in order to make the most of the SIMD instructions: the goal is to make them go two steps at a time instead of one.

Function	Input variables	Range	Output variables	Range
Low pass filter	x[n]	16-bit	y[n]	32-bit
	x[n-24ms]	16-bit	y[n-1]	32-bit
	x[n-48ms]	16-bit	y[n-2]	32-bit
High pass filter	x[n]	16-bit	y[n]	32-bit
	x[n-64ms]	16-bit	y[n-1]	32-bit
	x[n-128ms]	16-bit		
Derivative	x[n]	16-bit	y[n]	16-bit
	x[n-10ms]	16-bit		
Moving average	x[n]	16-bit	y[n]	32-bit
	x[n-16ms]	16-bit		

Table 4.1: Necessary data range for the different variables used by the filters

4.3.1 32-bit to 16-bit

The first thing to do in order to implement the SIMD intrinsics of the CMSIS library is to change the data types. Indeed, before doing so, it is important to make sure that there will be no impact on the validity of the answer. Here are the signed data type ranges [67]

- *signed* int_8: $[-128; 127]$
- *signed* int_{16}: $[-32,768; 32,767]$
- *signed* int_{32}: $[-2,147,483,648; 2,147,483,647]$

Because of certain operations done to make the algorithm less gain sensitive [9], some variables must be 32-bit wide. The table 4.1 encompasses the necessary data ranges.

4.3.1.1 Results

So far, no improvements have been implemented to the algorithm, so there should be no improvement expected from a simple change of data type. Therefore, the increase in performance is unexpected. To understand what happened, looking at the assembly execution is the way to go.
Another observation is that batch size above 200 samples per batch seems to give more predictable results (which means similar to those in figure 4.17).

4.3.1.2 Analysis

In order to have an idea of what to look for, we have to analyse how the change impacted all the filters. The data used for the figures and the assembly codes come

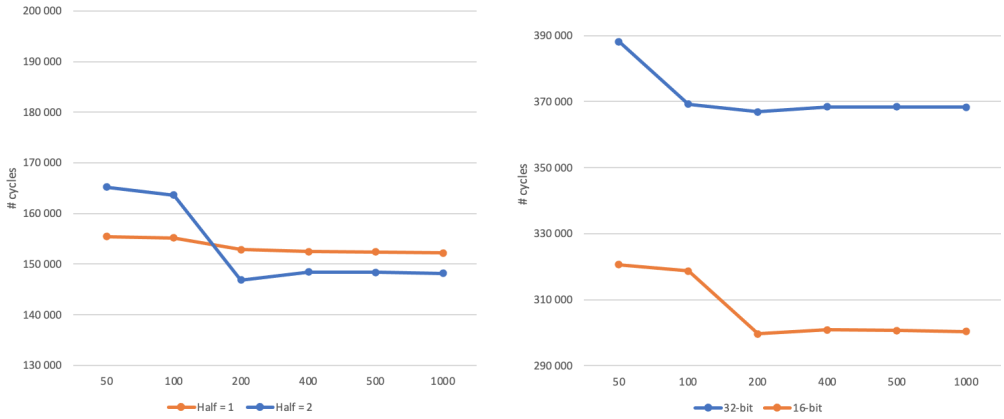

Figure 4.20: Results of transition to 16-bit / Halves comparison (left) & comparison with 32-bit

from a simulation with batches of size 1000.

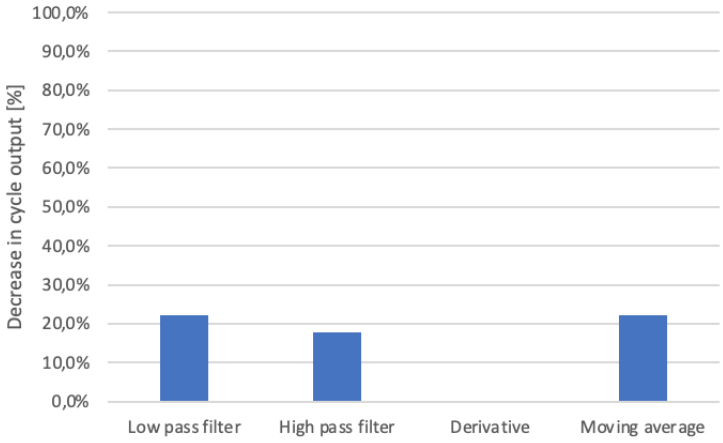

Figure 4.21: Decrease in cycle output from each filter from 32-bit to 16-bit

As seen in 4.21, the derivative performance is exactly the same. The three other filters, however, show a significant improvement in performance. For example, the low pass filter can be analysed.
The theoretical decrease of cycle output [66] is $\frac{18}{24} = 0.75$, which is **75%**. The experimental decrease is **77.8%**. The difference of performance stems from the treatment of values $y1$ and $y2$. In order to store and fetch them, the 32-bit implementation uses LDR and STR while the 16-bit uses MOV, which consumes less cycles. A similar study can be done for both high pass filter and moving average: the difference of performance can be explained now.

```
SUBS R2, R2, #0x01 ; advance in the loop
LDR R4, [R0, #0x04]! ; load sample[i]
LDR R6, [R4, -#0x14] ; load sample[i-5]
LDR R5, [R4, -#0x28] ; load sample[i-10]
LDR R3, [R8, #0x00] ; load the value of y2
LDR R7, [R12, #0x00] ; load the value of y1
SUB R4, R4, R6, LSL #1 ; computation start
ADD R4, R4, R5
SUBS R3, R4, R3
ADD R3, R3, R7, LSL #1 ; computation end
STR R3, [R12, #0x00] ; store the value of y0 in y1
STR R7, [R8, #0x00] ; store the value of y1 in y2
SMMUL R3, R3, LR ; amplitude adjustment start
ASRS R4, R3, #3
ADD R3, R4, R3, LSR #31 ; amplitude adjustment end
STR R3, [R1, #0x04]! ; store the answer
```

Assembly code 4.8: LPF 32-bit (1000 samples) - cycle count= 24

```
SUBS R3, R3, #0x01 ; advance in the loop
LDRSH R7, [R0, #0x02]! ; load sample[i]
MOV R5, R6 ; store the value of y2
MOV R6, R2 ; store the value of y1
LDRSH R2, [R0, #-0x0A] ; load sample[i-5]
LDRSH R4, [R0, #-0x14] ; load sample[i-10]
SUB R2, R7, R2, LSL #1 ; computation start
ADD R2, R2, R4
SUBS R2, R2, R5
ADD R2, R2, R6, LSL #1 ; computation end
SMMUL R4, R2, R8 ; amplitude adjustment start
LSRS R5, R4, #3
ADD R4, R5, R4, LSR #31 ; amplitude adjustment end
STRH R4, [R1, R2, LSL #1] ; store the answer
```

Assembly code 4.9: LPF 16-bit (1000 samples) - cycle count= 18

4.3.2 *For* loop unrolling

To make the most of SIMD instructions, the structure of the *for* loops must be adapted to proceed two computations at a time. Therefore, the second step of the preparation for the intrinsics requires to unroll the *for* loops inside each filter function. Here, we will halve the amount of iteration by going for two steps increments for every iteration of a loop.
One detail that must be taken into account when analysing the results is that manual loop unrolling will reduce the loop overhead [68], so it will always produce better performance. Likewise, unrolling the whole loop would provide the best cycle output but is a very impractical way of coding. The reader must thus keep in mind that the main reason for undergoing loop unrolling is to create an SIMD-friendly environment in the algorithm, not improve cycle output.

4.3.2.1 Results

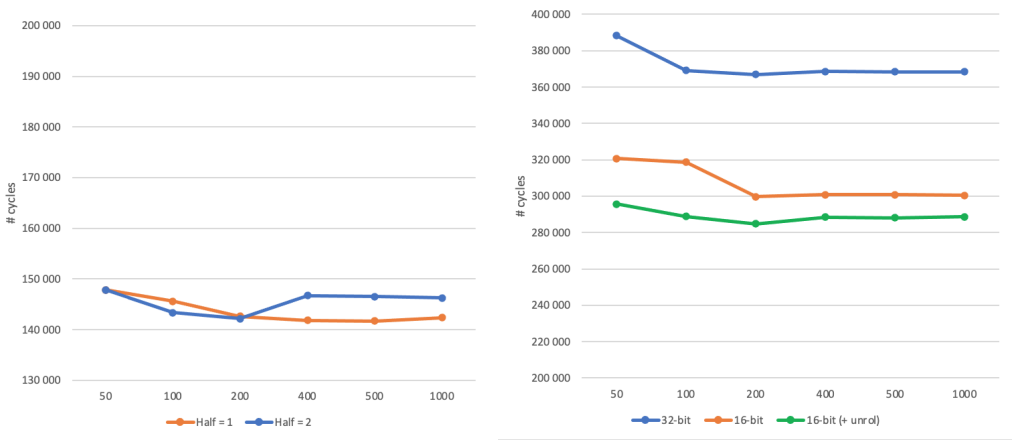

Figure 4.22: Results of loop unrolling / Halves comparison (left) & comparison with previous steps

As mentioned previously, manual loop unrolling will always provide a performance gain due to reduced loop overhead. The increase in performance was therefore expected.

4.3.3 Intrinsics

Once this transition has been made, the instructions can be implemented and the impact on performance studied and discussed. The intrinsics of interest are the following (the way they work is described in details in [61]).

- **PKHBT** & **PKHTB**: the bit packing operations are very important. Its goal is to concatenate two 16-bit words in a 32-bit word. This is a necessary step because the SIMD instructions use the 16 most significant bits (MSB) as the first operand and the 16 least significant bits (LSB) as the second operand. In order to unpack words, bit shift and type cast are used.
- **SADD16** & **SSUB16**: the addition and subtract operations, operating on two 16-bits words at a time. This is the main intrinsic that can be used because the digital filters operations are mostly arithmetic ones.
- **SSAT**: the saturate operation. It allows to saturate a value to a certain bit (for example, if the bit at position 2 is selected, the value will saturate at 4).

While various other operations could have been useful (such has dual multiplication), the range of data limits the potential of what can be used. This will be discussed in the last chapter.

- The LPF and HPF are not compatible to 16-bit SIMD instructions so no further optimization has been done on them.
- The derivative and the moving average window are eligible to SSUB16 intrinsic.
- The moving average window is also eligible to the SSAT intrinsic. This will be used to replace the conditional statement that essentially does a saturation at 32000. With the intrinsic, a saturation can be set at 32767 with a single instruction.

4.3.3.1 Results

The expectation would be to see an increase in performances because two operations can be processed as one instruction thanks to the SIMD instructions. However, results show that it is not the case: the results are roughly the same as before. The hypothesis is that the overhead needed to pack the words and then unpack them. The SSAT intrinsic, in contrast, should have a positive impact because it allows to get rid of branching. To see the impact of the SSAT intrinsic, measurements have been taken with only the SSAT intrinsic (without the SSUB16) applied in the moving average windpw. The results show that the performance is better without SSUB16 for this filter.

4.3.3.2 Analysis

Figure 4.25 shows how to cycle output evolves depending on what SIMD instructions are implemented (data taken on batch of 1000 samples).

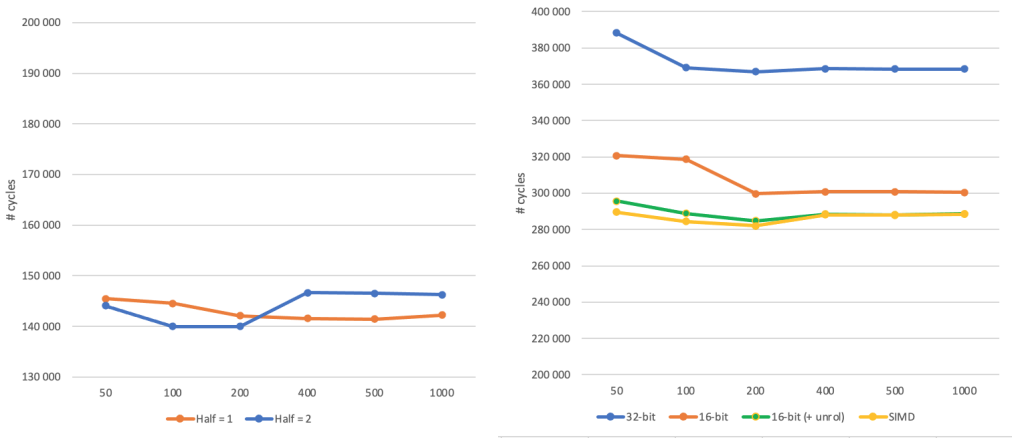

Figure 4.23: Results of SIMD intrinsics implementation / Halves comparison (left) & comparison with previous steps

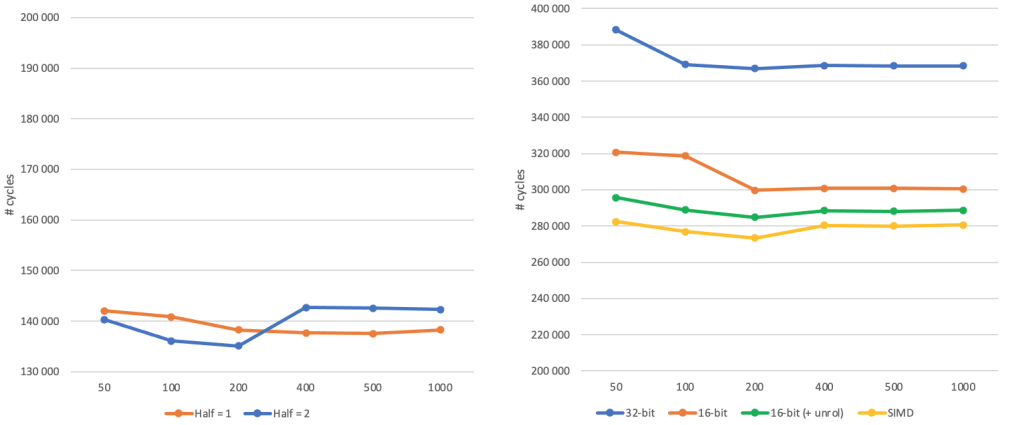

Figure 4.24: Results with only SSAT implemented / Halves comparison (left) & comparison with previous steps

- In the case of the derivative, the SIMD instructions have little to no impact.
- In the case of the moving average window, the SIMD arithmetic have a negative impact on the performance, while the saturation intrinsic has a positive impact.

To understand how the SIMD work in motion, let's analyze the derivative function, with batches of 1000 samples. The cycle count from both assembly exhibits (4.10 and 4.11) shows that the theoretical ratio ($\frac{15}{15}$ =**1**) is the same as the experimental ratio, hence why there is no performance gain or loss from using the SIMD intrinsic. The SIMD version has to concede an overhead due to bit packing

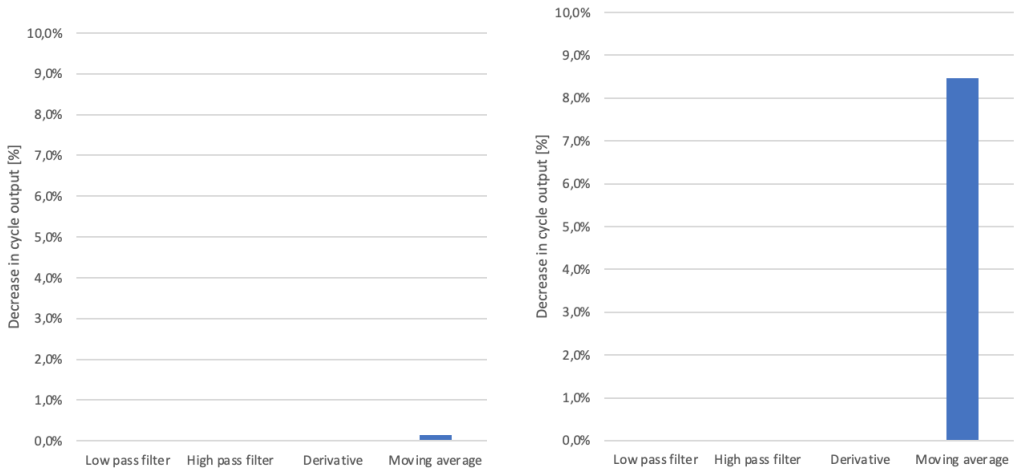

Figure 4.25: Decrease in cycle output from each filter: from no SIMD to full SIMD (left) and only SSAT (right)

```
1 ADDS R2, R2, #0x02 ; advance in the loop
2 LDRH LR, [R0, #0x04]! ; load sample[i-1]
3 LDRH R3, [R0, #0x02] ; load sample[i]
4 LDRH R4, [R0, #-0x02] ; load sample[i-2]
5 LDRH R5, [R0, #0x04] ; load sample[i+1]
6 SUBS R3, R3, R4 ; computation 1
7 STRH R3, [R1, #0x04]! ; store answer 1
8 SUB R3, R5, LR ; computation 2
9 STRH R3, [R1, #0x02] ; store answer 2
```

Assembly code 4.10: Derivative 16-bit + loop unrolled (1000 samples) - cycle count= 15

and unpacking but makes up for it by needing only one store for two answers. The compiler does not take advantage of the fact that the four loads are stored consecutively in the memory. A vector processor could have used only the starting address and would have fetched multiple words at once. Furthermore, the potential gain is limited because only two operations can be done at a time. Wider registers (such as the 128-bits Helium registers) could allow more parallel operations. These observations and hypothesis will be discussed in chapter 5.

```
1  ADDS R2, R2, #0x02 ; advance in the loop
2  LDRH LR, [R0, #0x04]! ; load sample[i]
3  LDRH R3, [R0, #0x04] ; load sample[i-2]
4  LDRH R4, [R0, #0x02] ; load sample[i-1]
5  LDRH R5, [R0, #-0x02] ; load sample[i+1]
6  ORR R3, R3, R4, LSL #16 ; bit packing 1
7  ORR R5, LR, R5, LSL #16 ; bit packing 2
8  SSUB16 R3, R3, R5 ; computation
9  ROR R3, R3, #16 ; bit unpacking
10 STR R3, [R1, #0x04]! ; store both answers
```

Assembly code 4.11: Derivative 16-bit + loop unrolled + SIMD (1000 samples) - cycle count= 15

Chapter 5

Discussion

In this chapter, the results of the previous chapter are compiled to give an overview of all the work that has been done. As the technical and functional analysis has already been done, the aim here is to step back and analyse the relevance of the optimisations that have been implemented, as well as to reflect on the future possibilities opened up by this work.

5.1 Results summary

Figures 5.1 and 5.2 are compilations of the key results from all the successive step from the baseline pre-processing algorithm towards the SIMD-improved algorithm. Each version is summarised below.

- **Base**: this is the initial version of the pre-processing algorithm, used in the SleepRider and developed by Pan and Tompkins [8]
- **Batcherization**: the pre-processing algorithm adapted to process samples in batches instead of one by one, making the ring buffer obsolete
- **Ping-pong**: adaptation have been made to ease the data fetching of old values
- **Branches removal**: branches and their consequent performance overhead have been replaced; the loop separation method has been chosen as its results are better
- **16-bit (without SIMD)**: in preparation for the SIMD intrinsics, the pre-processing algorithm has been adapted to work with 16-bit values as much as possible without impeding the results; loop unrolling has also been performed

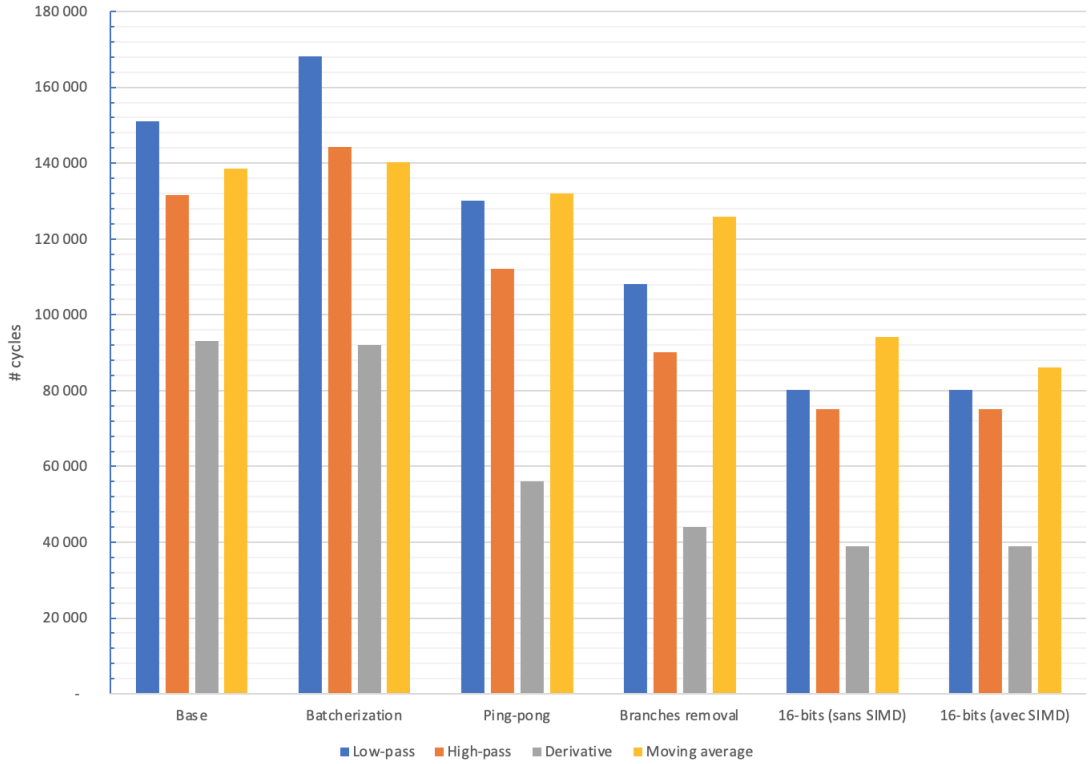

Figure 5.1: Evolution of performance for each version of the algorithm - batch size: 1000

- **16-bit (with SIMD)**: SIMD intrinsics have been implemented in the derivative and moving average window functions

Figure 5.3 shows how each filter performance have evolved compared to the initial algorithm. The 16-bit SIMD version is the final version and shows the greater improvements all round: the overall cycle output reduction is 45.47%. The filters final results, in term of cycle output decrease, are:

- 46.89% for the LPF
- 42.93% for the HPF
- 58.09% for the derivative
- 37.86% for the moving average window

Figure 5.4 shows the memory usage evolution through all the steps towards batcherization. A big part of the zero-initialization are due to the ECG data, the read-only (RO) data are the constants. Then, the total RAM Size = RW Data + ZI Data and the total ROM Size = Code + RO Data + RW Data.

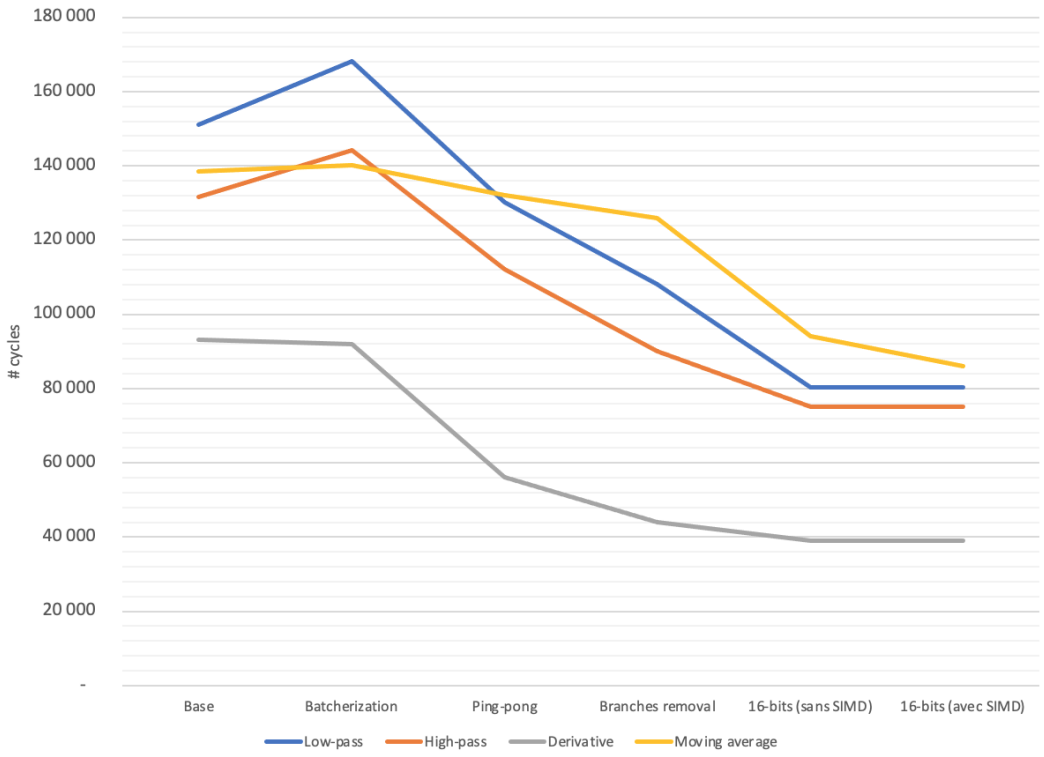

Figure 5.2: Evolution of performance for each filter - batch size: 1000

5.2 Parallelisation: assessment

Optimizing the algorithm was the main goal of this work and parellelisation had been identified has the most potent way to improve performance on a algorithm that is already known to be the state of the art of ULP arrhythmia detection. Working towards this goal has brought to light a number of related specificities that are worth examining in more detail.

5.2.1 Benefits & drawbacks

Parallel programming as such has clear performance benefits, as the literature review identified in Chapter 3.
Setting up the parallelisation was not immediate. It was necessary to go through intermediate steps to allow the algorithm to be a compatible environment for the implementation. These various intermediate steps have themselves been beneficial in improving the performance of the algorithm.

- The batcherization reduced the number of wake-up interrupts, which are

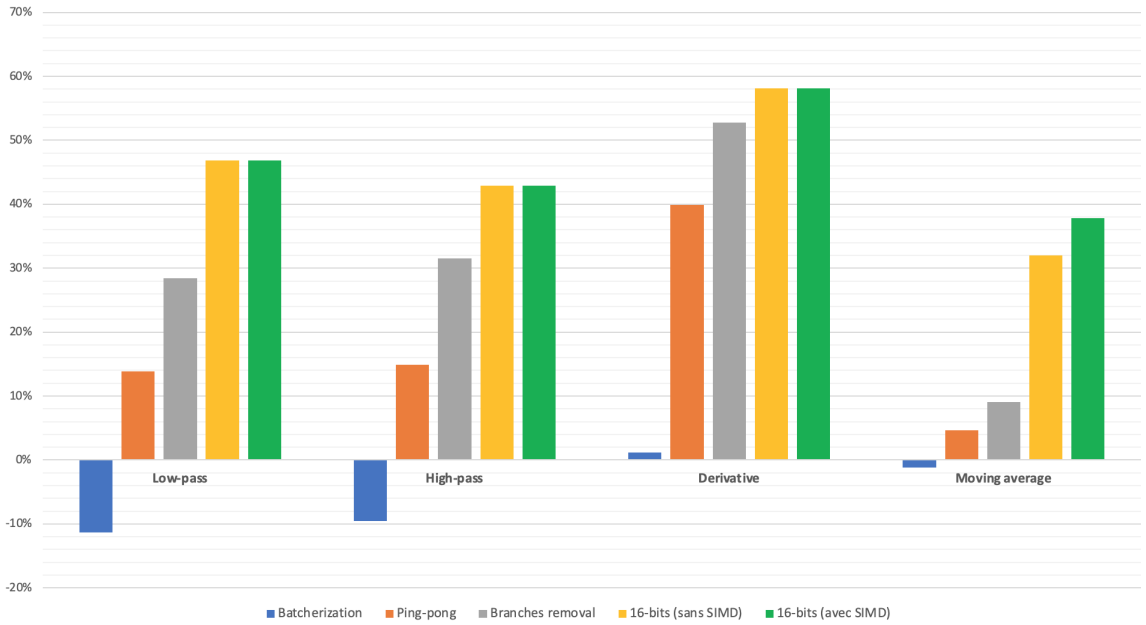

Figure 5.3: Performance gain/loss for each filter compared to the base algorithm - batch size: 1000

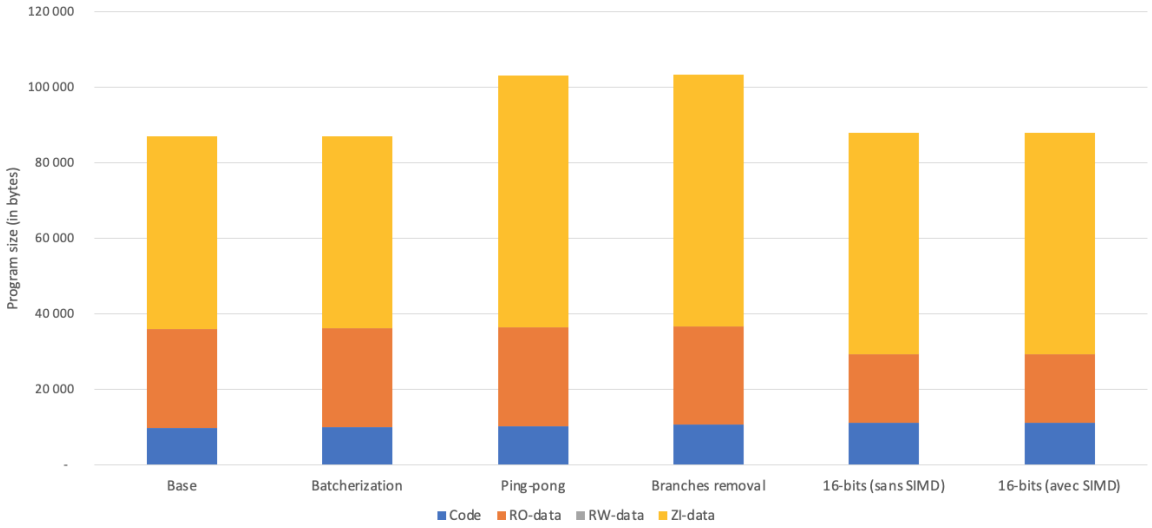

Figure 5.4: Evolution of memory usage - batch size: 1000

responsible for a significant part of the SleepRider's power consumption.

- The move to 16-bit has reduced the number of cycles by making it easier for the compiler to store the data.

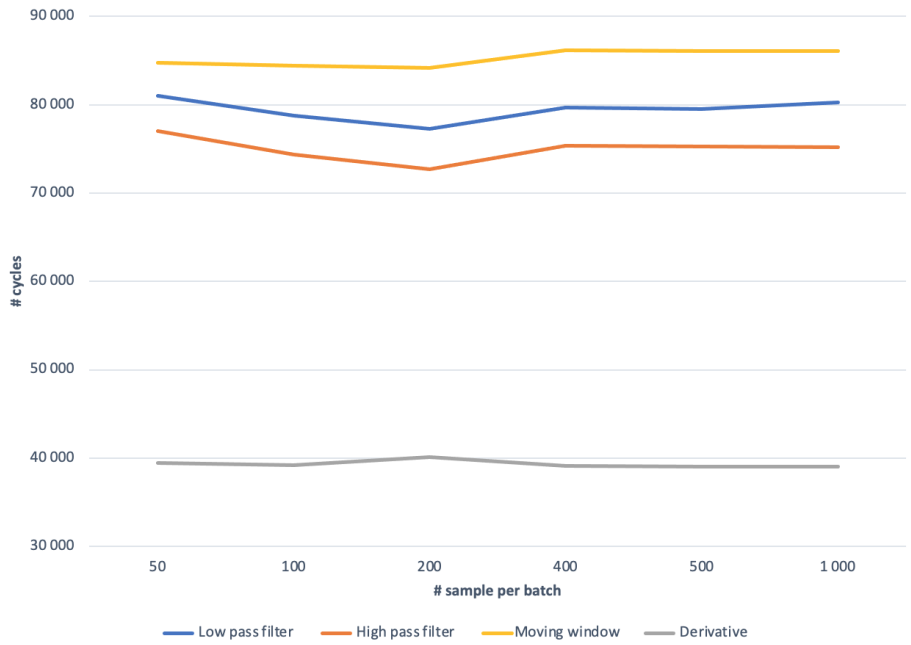

Figure 5.5: Performance comparison for different batch size - 16-bit (with SIMD) version

- The study of the available SIMD instructions has made it possible to discover useful intrinsics other than for vectorisation.
- Going from 32- to 16-bit reduced the memory usage.

Stepping back from the whole implementation process, it can be said that the most important benefit is actually the whole process. Indeed, having in mind all the considerations necessary for the algorithm to be parallelizable allows to realize many performance bottlenecks. As the study in Chapter 3 showed, vectorisation is a process that takes place at several levels. The results show that harmonising the pre-processing code to make it fully parallelizable leads to significant gains: up to 58.1% fewer cycles (in the case of the derivative) and 45.47% fewer cycles for the whole pre-processing.

5.2.2 Limitations

In the end, even if the results obtained by vectorising are light, their analysis in chapter 4 allows us to realise the potential behind this method. The first thing to notice is the limitations of word size. The SIMD instructions of the Cortex-M4 are limited to operations on 16-bit and 8-bit words, which is not adapted

to the application of the SleepRider and therefore limits the scope of the optimisations. Some calculations could have been effectively vectorized without this constraint. We can then see that there was a shortfall in the Load and Store. The compiler did not take into account the fact that the data was aligned in memory and could have been recovered in a single instruction (a Multiple Load for example).

The application would therefore have benefited from an architecture with vector registers (and larger) and native vector instructions, not requiring the preparatory steps that wasted a few cycles. This is why the Cortex-M33 and its Helium vector extension seemed to be the ideal solution for an improvement and would be the next best step in the path to parallelisation.
The main reason why the simulation could not be performed on the Cortex-M33 was the μVision IDE. This environment offers many useful tools for profiling, debugging and simulation for cores that are compatible, which is the case with the Cortex-M4. This is not the case for the CM33, which required an extension to uVision called Fixed Virtual Platform (FVP) in order to replace the physical target hardware. [69] However, working with FVP proved to be difficult, even with assistance with ARM Customer service. Therefore, the Helium intrinsics could not be simulated.

Another disadvantage of the method adopted in this thesis is that the match size is limited by certain constraints. Firstly, it is imperative that the number of iterations in a batch must be an even number for the performance to be optimal. Although the impact on performance is small, this made it more difficult to study the results until the problem was identified. It can be seen that overall, the batch size has very little impact on performance. The results slightly favour a batch of 200 samples.

5.3 Improvements

5.3.1 Further optimisation

This master thesis focused on optimising the pre-processing step due to time limitations. However, the whole algorithm, under thorough analysis, could have benefited from the SIMD intrinsics.

5.3.2 Cycles assessment

Neither μVision simulator or ARM Fast Models simulation is "cycle accurate". Both μVision simulator and ARM Fast Models simulation are only "instruction accurate", meaning that it can only simulate the instructions of the application

which are executed by the ARM CPU core comparing with the real hardware implementation. The pipeline effect and wait-states to access different memory areas are not considered in the simulation. [70] In order to have a cycle accurate model, another model product from ARM should be used: Cycle models. [71]

5.3.3 Memory constraints

As the focus of this work was put on performance, memory constraints have not been taken into account. Memory has been upscaled in order to ensure that no problem would occur (like the heap and the stack colliding). However, the memory settings remained the same all the time, to maintain consistency all over the work. Further study is required in order to take memory requirement into account.

5.3.4 Power consumption

A key element of this thesis is that the application being studied must operate under ULP constraints. In order to have a complete picture of the energy consumption of the application after making changes to the algorithm, energy profiling would have been an important element. This could serve as a starting point for future work.

Conclusion

Heart rhythm disorders and related conditions are becoming increasingly common, due to the ageing population. An arrhythmia is a heart that beats too slowly, too fast or in an uncontrolled manner. Many forms of arrhythmia do not cause any health problems; however, they can cause a variety of troublesome symptoms, such as dizziness or chest pain. Other, more dangerous forms of arrhythmia affect the blood supply and therefore require medical attention. If left untreated, they can lead to stroke, heart attack, heart failure or sudden death. [2, 1, 21]
ECG is the main diagnostic method used to detect arrhythmia and determine their cause. It provides a graphic representation of the electric impulse that produces each beat. The main focus of chapter 1 is to give a medical context to the reader. [19]
Wearable biomedical systems are solution that allow data to be taken on a regular basis and allow physicians to continuously monitor patients over longer periods of time.

This work first analysed the functioning of cardiac arrhythmia detection algorithms, taking stock of existing technologies. Then, the device on which this thesis is based, the SleepRider, was studied as well as the detection and classification application that runs on it. Chapter 2 allows us to become familiar with such applications and to understand how they are implemented. This study identifies that performance is pivotal and opens up the question about what can be done to optimise the throughput.

The answer to this question is the subject of Chapter 3, which is an in-depth analysis of the concept of parallelisation and the possible benefits that its application can have on an algorithm. In particular, vectorisation has been identified as the solution to the research question. Therefore, its operation is studied, allowing to understand how the different components of a system for embedded application (hardware, compiler and user) must be taken into account when taking advantage of vectorisation. The processors targeted for this application (Cortex-M4 and Cortex-M33), both for compatibility with the SleepRider and for their characteristics, are

analysed so that we can know exactly what technical tools are available. In this case, it is the SIMD instructions for the CM4 and the Helium extension for the CM33. [11, 12]

As the main contribution of this thesis, the implementation of a vectorisable algorithm, via its successive steps, is presented in chapter 4. Vectorizing requires the implementation of an enabling environment in the algorithm. As the code is adapted to make it compatible with SIMD instructions, the analysis of the results shows how having all the considerations in mind (from hardware to compilation) makes it possible to make choices that harmonise the algorithm and make it perform better, even before the application of SIMD intrinsics. [61]

Finally, Chapter 5, by compiling the results, provides an overview of what has been done. A discussion of the results shows that the gain from vectorisation is not only due to the application of the intrinsics, but also to all the thinking behind it, which allows the algorithm to be fully designed to be parallelizable. The implementation achieved good results, with cycle output decrease of 46.89% for the LPF, 42.93% for the HPF, 58.09% for the derivative, 37.86% for the moving average window and 45.47% globally.
The weaknesses of the approach, which does not address important aspects such as energy consumption or memory constraints, are highlighted. On top of that, the limitations of the IDE did not allow to go to the end of the approach. However, they did not prevent us from drawing lessons on which future work can be based on to deepen the study, whether for a cardiac arrhythmia detection and classification application or for any ULP application that could benefit from the principles of parallelisation.

Bibliography

[1] MD James Beckerman. *Arrhythmia*. 2021. URL: https://www.webmd.com/heart-disease/atrial-fibrillation/heart-disease-abnormal-heart-rhythm.

[2] Blood National Heart and Lung Institute. *What Is an Arrhythmia?* 2022. URL: https://www.nhlbi.nih.gov/health/arrhythmias.

[3] L. Brent Mitchell. *Vue d'ensemble des troubles du rythme cardiaque*. URL: https://www.msdmanuals.com/fr/accueil/troubles-cardiaques-et-vasculaires/troubles-du-rythme-cardiaque/vue-d-ensemble-des-troubles-du-rythme-cardiaque.

[4] PhD John E. Hall. *Guyton and Hall Textbook of Medical Physiology*. 13th Edition. W B Saunders Co Ltd, 2013. ISBN: 9781455770052.

[5] Raúl Alonso Álvarez, Arturo José Méndez Penín, and Xosé Antón Vila Sobrino. "A comparison of three QRS detection algorithms over a public database". In: *Procedia Technology* 9 (2013), pp. 1159–1165.

[6] Rémi Dekimpe et al. "SleepRider: a 5.5 μW/MHz Cortex-M4 MCU in 28nm FD-SOI with ULP SRAM, Biomedical AFE and Fully-Integrated Power, Clock and Back-Bias Management". In: *2021 Symposium on VLSI Circuits*. IEEE. 2021, pp. 1–2.

[7] G.B. Moody and R.G. Mark. "The impact of the MIT-BIH Arrhythmia Database". In: *IEEE Engineering in Medicine and Biology Magazine* 20.3 (2001), pp. 45–50. DOI: 10.1109/51.932724.

[8] Jiapu Pan and Willis J. Tompkins. "A Real-Time QRS Detection Algorithm". In: *IEEE Transactions on Biomedical Engineering* BME-32.3 (1985), pp. 230–236. DOI: 10.1109/TBME.1985.325532.

[9] Patrick S. Hamilton. *Open Source ECG Analysis Software Documentation*. URL: http://www.eplimited.com/osea13.pdf.

[10] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. USA: Benjamin-Cummings Publishing Co., Inc., 1989. ISBN: 0805301771.

[11] ARM Limited. *Cortex-M4*. URL: https://developer.arm.com/Processors/Cortex-M4.

[12] ARM Limited. *Cortex-M3*. URL: https://developer.arm.com/Processors/Cortex-M33.

[13] OpenStax College. *Anatomy & physiology*. OpenStax, 2013. ISBN: 1938168135.

[14] Texas Heart. *Heart Anatomy*. URL: https://www.texasheart.org/heart-health/heart-information-center/topics/heart-anatomy/.

[15] Rosaire Verna. "The Only EKG Book You'll Ever Need". In: *American Family Physician* 61.7 (2000), p. 2279.

[16] Biology dictionary. *Cardiac Cycle*. URL: https://biologydictionary.net/cardiac-cycle/.

[17] National Cancer Institute. *Physiology of the Heart*. URL: https://training.seer.cancer.gov/anatomy/cardiovascular/heart/physiology.html.

[18] Chhabra L. Kashou AH Basit H. "Physiology, Sinoatrial Node". In: *PubMed* (2021).

[19] Heart.org. *Arrhythmias and Congenital Defects*. URL: https://www.heart.org/en/health-topics/congenital-heart-defects/the-impact-of-congenital-heart-defects/arrhythmias-and-congenital-defects.

[20] Mitchell J. R. & Wang J. J. "Expanding application of the Wiggers diagram to teach cardiovascular physiology". In: *Advances in physiology education* 38 (2014), pp. 170–175. DOI: https://doi.org/10.1152/advan.00123.2013.

[21] Blood National Heart and Lung Institute. *Heart Tests*. 2022. URL: https://www.nhlbi.nih.gov/health/heart-tests.

[22] Benoit Macq & Greets Kerckhofs John Lee Frank Peeter. "LGBIO2050: Medical Imaging". In.

[23] British Heart Fundation. *Drug cabinet Anti-arrhythmics*. URL: https://www.bhf.org.uk/informationsupport/heart-matters-magazine/medical/drug-cabinet/anti-arrhythmics.

[24] Phillipe Lefèvre & Renaud Ronsse. "LGBIO1114: Organes artificiels et réhabilitation". In: chap. 7.

[25] Paulus Kirchhof et al. "Guidelines for the management of atrial fibrillation developed in collaboration with EACTS". In: *European Heart Journal* 37.38 (Aug. 2016), pp. 2893–2962. ISSN: 0195-668X. DOI: 10.1093/eurheartj/ehw210. eprint: https://academic.oup.com/eurheartj/article-pdf/37/38/2893/23787249/ehw210.pdf. URL: https://doi.org/10.1093/eurheartj/ehw210.

[26] Physionet. *The QT Databse*. URL: http://www.physionet.org/physiobank/database/qtdb/doc/node3.html.

[27] American Heart Association. *AHA Databse*. URL: https://www.heart.org/HEARTORG/.

[28] Ary L Goldberger et al. "PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals". In: *circulation* 101.23 (2000), e215–e220.

[29] ANSI AAMI and AAMI EC57. "(R) 2008-Testing and reporting performance results of cardiac rhythm and ST segment measurement algorithms". In: *American National Standards Institute, Arlington, VA, USA* (2008).

[30] Eduardo José da S. Luz et al. "ECG-based heartbeat classification for arrhythmia detection: A survey". In: *Computer Methods and Programs in Biomedicine* 127 (2016), pp. 144–164. ISSN: 0169-2607. DOI: https://doi.org/10.1016/j.cmpb.2015.12.008. URL: https://www.sciencedirect.com/science/article/pii/S0169260715003314.

[31] PA Lynn. "Recursive digital filters for biological signals". In: *Medical & biological engineering* 9.1 (1971), pp. 37–43.

[32] Lizhe Tan and Jean Jiang. "Chapter 9 - Adaptive Filters and Applications". In: *Digital Signal Processing (Third Edition)*. Ed. by Lizhe Tan and Jean Jiang. Third Edition. Academic Press, 2019, pp. 421–474. ISBN: 978-0-12-815071-9. DOI: https://doi.org/10.1016/B978-0-12-815071-9.00009-9. URL: https://www.sciencedirect.com/science/article/pii/B9780128150719000099.

[33] Shawhin Talebi. *The Wavelet Transform*. URL: https://towardsdatascience.com/the-wavelet-transform-e9cfa85d7b34.

[34] Reza Sameni et al. "A nonlinear Bayesian filtering framework for ECG denoising". In: *IEEE Transactions on Biomedical Engineering* 54.12 (2007), pp. 2172–2185.

[35] Omid Sayadi and Mohammad B Shamsollahi. "Multiadaptive bionic wavelet transform: Application to ECG denoising and baseline wandering reduction". In: *EURASIP Journal on Advances in Signal Processing* 2007 (2007), pp. 1–11.

[36] Patrick S. Hamilton and Willis J. Tompkins. "Quantitative Investigation of QRS Detection Rules Using the MIT/BIH Arrhythmia Database". In: *IEEE Transactions on Biomedical Engineering* BME-33.12 (1986), pp. 1157–1165. DOI: 10.1109/TBME.1986.325695.

[37] Cuiwei Li, Chongxun Zheng, and Changfeng Tai. "Detection of ECG characteristic points using wavelet transforms". In: *IEEE Transactions on biomedical Engineering* 42.1 (1995), pp. 21–28.

[38] Riccardo Poli, Stefano Cagnoni, and Guido Valli. "Genetic design of optimum linear and nonlinear QRS detectors". In: *IEEE Transactions on Biomedical Engineering* 42.11 (1995), pp. 1137–1141.

[39] Juan Pablo Martı´nez et al. "A wavelet-based ECG delineator: evaluation on standard databases". In: *IEEE Transactions on biomedical engineering* 51.4 (2004), pp. 570–581.

[40] Mohammed Bahoura, M Hassani, and M Hubin. "DSP implementation of wavelet transform for real time ECG wave forms detection and heart rate analysis". In: *Computer methods and programs in biomedicine* 52.1 (1997), pp. 35–44.

[41] Gari D Clifford, Francisco Azuaje, Patrick McSharry, et al. *Advanced methods and tools for ECG data analysis*. Vol. 10. Artech house Boston, 2006.

[42] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 1999. Chap. 5.6.

[43] Rohith Gandhi. *Support Vector Machine — Introduction to Machine Learning Algorithms*. URL: `https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47`.

[44] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[45] Philip De Chazal, Maria O'Dwyer, and Richard B Reilly. "Automatic classification of heartbeats using ECG morphology and heartbeat interval features". In: *IEEE transactions on biomedical engineering* 51.7 (2004), pp. 1196–1206.

[46] Amalia Villa et al. "Are we training our heartbeat classification algorithms properly?" In: *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE. 2019, pp. 6363–6366.

[47] Ming Chen et al. "Unsupervised domain adaptation for ECG arrhythmia classification". In: *2020 42nd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*. IEEE. 2020, pp. 304–307.

[48] V Mondéjar-Guerra et al. "Heartbeat classification fusing temporal and morphological information of ECGs via ensemble of classifiers". In: *Biomedical Signal Processing and Control* 47 (2019), pp. 41–48.

[49] Li Guo, Gavin Sim, and Bogdan Matuszewski. “Inter-patient ECG classification with convolutional and recurrent neural networks”. In: *Biocybernetics and Biomedical Engineering* 39.3 (2019), pp. 868–879.

[50] Jiaquan Wu et al. “A neural network-based ECG classification processor with exploitation of heartbeat similarity”. In: *IEEE Access* 7 (2019), pp. 172774–172782.

[51] Portland Pattern Repository. *Circular buffer*. URL: `http://wiki.c2.com/?CircularBuffer`.

[52] Willis J Tompkins. “Biomedical digital signal processing”. In: *Editorial Prentice Hall* (1993).

[53] Gene M Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the AFIPS conference proceedings, vol. 30 (Atlantic City, NJ, Apr. 18–20), AFIPS press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California”. In: *IEEE Solid-State Circuits Society Newsletter* 12.3 (2007), pp. 19–20.

[54] L John. “Gustafson. Reevaluating Amdahl’s law”. In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[55] Aydun Buluc & Jim Demmel. “Applications of Parallel Computers”. In: U.C. Berkeley.

[56] Michael J Flynn. “Very high-speed computing systems”. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909.

[57] ARM Limited. *Arm Compiler for Embedded*. URL: `https://developer.arm.com/Tools%5C%20and%5C%20Software/Arm%5C%20Compiler%5C%20for%5C%20Embedded`.

[58] ARM Limited. *Arm Compiler for Embedded User Guide*. URL: `https://developer.arm.com/documentation/100748/0618/Using-Common-Compiler-Options/Selecting-optimization-options?lang=en`.

[59] *Requirements for Vectorizable Loops*. Feb. 2012. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/requirements-for-vectorizable-loops.html`.

[60] Richard M Stallman et al. “Using the gnu compiler collection”. In: *Free Software Foundation* 4.02 (2003).

[61] ARM Limited. *CMSIS-CORE*. URL: `http://www.disca.upv.es/aperles/arm_cortex_m3/curset/CMSIS/Documentation/Core/html/group__intrinsic___s_i_m_d__gr.html`.

[62] ARM Limited. *ARM HELIUM*. URL: https://www.arm.com/technologies/helium.

[63] Sebastian Hack & Christoph Weidenbach. "Profiling". In: Saarland University. Chap. Advanced C Programming.

[64] Adam Osborne. *Introductions to Microcomputers: Volume One, Basic Concepts*. McGraw-Hill Osborne Media, 1980.

[65] Daniel Vaquerizo Hernández et al. "Continuously Energy Consumption Measure Approach Using a DMA Double-Buffering Technique". In: (2020).

[66] ARM Limited. *Technical Reference Manual*.

[67] David Harris and Sarah L Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[68] Michael Abrash. *Michael Abrash's Graphics Programming Black Book, with CD: The Complete Works of Graphics Master, Michael Abrash*. Coriolis group books, 1997.

[69] ARM Limited. *Fixed Virtual Platform*. URL: https://developer.arm.com/Tools%5C%20and%5C%20Software/Fixed%5C%20Virtual%5C%20Platforms.

[70] ARM Limited. *Timing accuracy of Fast Models*. URL: https://developer.arm.com/documentation/100964/1117/Introduction-to-Fast-Models/Fast-Models-accuracy/Timing-accuracy-of-Fast-Models?lang=en.

[71] ARM Limited. *ntroduction to Cycle Model reference platforms*. URL: https://developer.arm.com/documentation/101497/1105/Introduction-to-Cycle-Model-reference-platforms/Introduction-to-Cycle-Model-reference-platforms?lang=en.

Appendix A

Profiling data

	Mean	Calls	Total	Percentage
declarations	5	1	5	0.0%
ecg_init()	2575	1	2575	0.1%
for loop preparation	4	4000	16000	0.4%
for - case A	584,4	3983	2327[U+202F]665	51.5%
for - case B	1212	1	1212	0.0%
for - case C	131543	16	2104688	46.6%
for - others	67732	1	67732	1.5%
ecg_close()	164	1	164	0.0%

Table A.1: Global functions profile

	Mean	Percentage (case A)	Total	Percentage (total)
QRS_filter	246	42.1%	979818	21.7%
QRS_det	201,4	34.5%	802176	17.7%

Table A.2: Case A - majority functions profile

	Mean	Percentage (case C)	Total	Percentage (total)
buffer_get_features	33685,43	25.6%	538966,86	11.9%
svp_predict	94606,71	71.9%	1513707,43	33.5%
QRS_filter	520,57	0.4%	8329,14	0.2%
QRS_det	1465,43	1.1%	23446,86	0.5%

Table A.3: Case C - majority functions profile

Appendix B

Performance analyzer data

modules	**Time** (ms)	**Cycles**	**Percentage**	**# functions**
startup	0,00175	21	0,0%	5
main	4,001	48012	1,1%	1
svm	125,991	1511892	33,4%	6
signal_buffer	10,459	125508	2,8%	5
qrsnorm	3,035	36420	0,8%	3
qrsfilt	88,712	1064544	23,5%	6
qrsdet	60,852	730224	16,1%	5
feature_extract	5,760	69120	1,5%	11
ecg	22,785	273420	6,0%	3
dwt	35,944	431328	9,5%	5
beat_buffer	12,663	151956	3,4%	7
beat	6,859	82308	1,8%	15
total	377,063	4524753	100,0%	72
total target	334,284	4011408	85,3%	40
total select	302,166	3625992	80,1%	10

Table B.1: Performance analyzer O1 profile - global data

svm	**Time** (ms)	**Cycles**	% (local)	% (global)
svm_predict	40,876	490512	32,4%	10,8%
exp_custom	84,566	1014792	67,1%	22,4%

qrsfilt	**Time** (ms)	**Cycles**	% (global)	% (global)
lpfilt	22,679	272148	25,6%	6,0%
hpfilt	19,011	228132	21,4%	5,0%
mvwint	17,343	208116	19,5%	4,6%

qrsdet	**Time** (ms)	**Cycles**	% (local)	% (global)
QRSDet	43,460	521520	71,4%	11,5%
Peak	15,804	189648	26,0%	4,2%

dwt	**Time** (ms)	**Cycles**	% (local)	% (global)
downsamplingConvolution_l	17,823	213876	49,6%	4,7%
downsamplingConvolution_h	17,823	213876	49,6%	4,7%

ecg	**Time** (ms)	**Cycles**	% (local)	% (global)
ECG_wrapper	22,781	273372	100,0%	6,0%

Table B.2: Performance analyzer O1 profile - detailed data

modules	**O1**	**O2**	**O3**	**Ofast**	**std dev**	**norm sd**
startup	21	21	21	21	-	-
main	48012	48012	48012	48012	-	-
svm	1511892	823128	672948	672948	400745	0,53
signal_buffer	125508	125508	109428	109428	9284	0,01
qrsnorm	36420	36420	36420	36420	-	-
qrsfilt	1064544	998268	794220	794220	139587	0,19
qrsdet	730224	599880	494688	494688	111831	0,15
feature_extract	69120	60876	59292	59292	4710	0,01
ecg	273420	273000	273384	273384	199	0,00
dwt	431328	217548	217596	217596	106874	0,14
beat_buffer	151956	151896	147264	147264	2692	0,00
beat	82308	75180	75228	75228	3548	0,00
TOTAL	4524753	3409737	2928501	2928501	752910	1,00

Table B.3: Performance analyzer optimisation level comparison

Appendix C

Optimisation data

cycles	**Base**	**50**	**100**	**200**	**250**	**400**	**500**	**800**	**1000**	**2000**
Low pass filter	0	152160	165996	165000	164796	164496	164400	168252	168204	168096
LP - Buffer	151092	0	0	0	0	0	0	0	0	0
High pass filter	0	132960	130560	129276	129024	128640	128508	144300	144240	144120
HP - Buffer	131688	0	0	0	0	0	0	0	0	0
Derivative	0	93204	92640	92316	92256	96168	96132	92100	92076	92040
Derivative - Buffer	93168	0	0	0	0	0	0	0	0	0
Integrator + AV	0	139200	137796	136896	136704	140436	140352	140220	140172	140088
Integrator - Buffer	138504	0	0	0	0	0	0	0	0	0

Table C.1: Batcherization data

size (lpf)	half 2 (us)	half 1 (us)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	4910	6407	58920	76884	135804
100	4540	6370	54480	76440	130920
160	4703	6102	56436	73224	129660
200	4520	6352	54240	76224	130464
400	4510	6343	54120	76116	130236
500	4508	6341	54096	76092	130188
800	5406	5070	64872	60840	125712
1000	4504	6337	54048	76044	130092

size (hpf)	half 2 (us)	half 1 (us)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	4053	5550	48636	66600	115236
100	3695	5525	44340	66300	110640
160	3832	5295	45984	63540	109524
200	3681	5513	44172	66156	110328
400	3840	5506	46080	66072	112152
500	3839	5505	46068	66060	112128
800	4604	4402	55248	52824	108072
1000	3836	5502	46032	66024	112056

size (deriv)	half 2 (us)	half 1 (us)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	2180	2843	26160	34116	60276
100	1842	2838	22104	34056	56160
160	1912	2723	22944	32676	55620
200	1837	2836	22044	34032	56076
400	1835	2835	22020	34020	56040
500	1835	2834	22020	34008	56028
800	2201	2267	26412	27204	53616
1000	1834	2834	22008	34008	56016

size (ma)	half 2 (us)	half 1 (us)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	5380	6043	64560	72516	137076
100	5025	6022	60300	72264	132564
160	5216	5773	62592	69276	131868
200	5012	6011	60144	72132	132276
400	5007	6005	60084	72060	132144
500	5005	6004	60060	72048	132108
800	6004	4802	72048	57624	129672
1000	5003	6002	60036	72024	132060

Table C.2: Ping-pong buffer - halves data

size (lpf)	half 2 (us)	half 1 (cycles)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	5773	4910	69276	58920	128196
100	5720	4540	68640	54480	123120
200	5693	4520	68316	54240	122556
400	5680	4510	68160	54120	122280
500	5677	4508	68124	54096	122220
1000	5672	4504	68064	54048	122112

size (hpf)	half 2 (us)	half 1 (cycles)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	4920	4053	59040	48636	107676
100	4877	3695	58524	44340	102864
200	4855	3681	58260	44172	102432
400	4844	3840	58128	46080	104208
500	4842	3839	58104	46068	104172
1000	4838	3836	58056	46032	104088

size (deriv)	half 2 (us)	half 1 (cycles)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	2373	2180	28476	26160	54636
100	2353	1842	28236	22104	50340
200	2343	1837	28116	22044	50160
400	2338	1835	28056	22020	50076
500	2338	1835	28056	22020	50076
1000	2502	1834	30024	22008	52032

size (ma)	half 2 (us)	half 1 (cycles)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	5727	5380	68724	64560	133284
100	5697	5025	68364	60300	128664
200	5682	5012	68184	60144	128328
400	5674	5007	68088	60084	128172
500	5673	5005	68076	60060	128136
1000	5670	5003	68040	60036	128076

Table C.3: Bit masking buffer - halves data

size (lpf)	half 2 (us)	half 1 (us)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	4653	4910	55836	58920	114756
100	4577	4540	54924	54480	109404
200	4527	4520	54324	54240	108564
400	4513	4510	54156	54120	108276
500	4511	4508	54132	54096	108228
1000	4509	4504	54108	54048	108156

size (hpf)	half 2 (us)	half 1 (us)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	3897	4053	46764	48636	95400
100	3782	3695	45384	44340	89724
200	3687	3681	44244	44172	88416
400	3676	3840	44112	46080	90192
500	3674	3839	44088	46068	90156
1000	3676	3836	44112	46032	90144

size (deriv)	half 2 (us)	half 1 (us)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	1860	2180	22320	26160	48480
100	1847	1842	22164	22104	44268
200	1840	1837	22080	22044	44124
400	1838	1835	22056	22020	44076
500	1837	1835	22044	22020	44064
1000	1835	1834	22020	22008	44028

size (ma)	half 2 (us)	half 1 (us)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	5417	5380	65004	64560	129564
100	5458	5025	65496	60300	125796
200	5471	5012	65652	60144	125796
400	5486	5007	65832	60084	125916
500	5489	5005	65868	60060	125928
1000	5495	5003	65940	60036	125976

Table C.4: For loop separation buffer - halves data

size (lpf)	half 2 (us)	half 1 (us)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	3300	3450	39600	41400	81000
100	3233	3333	38796	39996	78792
200	3200	3237	38400	38844	77244
400	3436	3202	41232	38424	79656
500	3432	3195	41184	38340	79524
1000	3424	3263	41088	39156	80244

size (hpf)	half 2 (us)	half 1 (us)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	3287	3133	39444	37596	77040
100	3053	3143	36636	37716	74352
200	3027	3033	36324	36396	72720
400	3265	3017	39180	36204	75384
500	3262	3013	39144	36156	75300
1000	3256	3007	39072	36084	75156

size (deriv)	half 2 (us)	half 1 (us)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	1613	1673	19356	20076	39432
100	1598	1670	19176	20040	39216
200	1674	1668	20088	20016	40104
400	1590	1667	19080	20004	39084
500	1588	1667	19056	20004	39060
1000	1586	1668	19032	20016	39048

size (ma)	half 2 (us)	half 1 (us)	half 2 (cycles)	half 1 (cycles)	sum cycles
50	3807	3867	45684	46404	92088
100	3778	3897	45336	46764	92100
200	3764	3903	45168	46836	92004
400	3928	3910	47136	46920	94056
500	3926	3911	47112	46932	94044
1000	3921	3914	47052	46968	94020

Table C.5: SIMD - halves data

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**