MASTER

Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring

Spiridonescu, V.P.

Award date:
2015

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring

*Embedded Systems MSc Thesis*
Vlad Spiridonescu

Supervisors:
Prof. Dr. Ir. Gerard de Haan, EE, TU Eindhoven
Dr. Ihor Kirenko, Philips Research

Eindhoven, June 2016
Abstract

Several non-contact based respiration monitoring algorithms have been developed at Philips. This thesis presents the porting and optimization of such algorithms, work that has been done in the Patient Care & Measurements department at Philips Research in Eindhoven. The ProCor and AutoROI algorithm implementations have been ported on an embedded platform. Moreover, different parallelization strategies have been proposed and an attempt has been made to use other SoC components than the CPU in order to balance the utilization. A large part of the application has been ported on the Graphical Processing Unit. The overall execution time has been improved by over 3 times for an input of 768x576 pixels and a division of the frame in 1728 rectangles, making it real-time with respect to the needed framerate for respiration monitoring (7 fps). Moreover, the responsible CPU core works now at 40% compared to 100%. New methods to capture frames from the camera have been implemented that provide possibility for 8 times higher framerate and, also, direct control of camera settings through the application. The ability to stream captured images through network has also been achieved.
Preface

Going through the last stages of my master program and especially the master thesis, I look backwards with complacency to the time when I started the internship at Philips Research. I am deeply grateful to Dr. Ihor Kirenko, my daily supervisor, for this opportunity. I would like to thank him for his patience and professionalism that encouraged me continuously in performing and also enjoying my time at Philips. I owe to Mukul Rocque many hours spent in explanations and feedback regarding different aspects of my work. He is a great person to work with. I would also want to thank Professor Gerard de Haan and Professor Sander Stuijk for providing the necessary support and guidance in working for this project. Special thanks to my colleagues from Philips Research that made these months a great period to not be forgotten. Furthermore, I appreciate all the support from my friends who always wanted to help me in stressful situations. Finally, I would like to thank my family for giving me their love, and showing me besides all, what it truly means to strive and eventually win great battles. They remain my greatest inspiration above all.

Thank you,
Vlad Spiridonescu
Contents

1 Introduction 1
   1.1 Patient care and respiration monitoring ............................ 1
       1.1.1 Non-contact based respiration monitoring .................. 2
   1.2 Problem statement .................................................. 3
   1.3 Challenges and objectives .......................................... 3
       1.3.1 Objectives .................................................. 3
       1.3.2 Challenges .................................................. 4
   1.4 Report ........................................................................ 4

2 Camera-based respiration monitoring algorithms 6
   2.1 Respiration monitoring algorithms .................................... 6
   2.2 Respirate - ProCor algorithm ......................................... 6
   2.3 Respirate - AutoROI algorithm ....................................... 8

3 Software development on the embedded platform 12
   3.1 The embedded platform ................................................. 12
       3.1.1 The hardware specifications .................................. 12
       3.1.2 The available software for the board ....................... 13
   3.2 Setting the platform for development ................................ 14
       3.2.1 Setting up device software ................................... 14
       3.2.2 Tools and libraries to use in development on Phytec board .. 15
       3.2.3 Use of compiler flags .......................................... 16
       3.2.4 GPU programming .............................................. 18
   3.3 Framework for video processing applications on Phytec board .... 18
       3.3.1 Frame grabbing ............................................... 19
       3.3.2 Color space conversion ...................................... 23
       3.3.3 Network streaming on Phytec board ......................... 24
       3.3.4 Quality benchmarking of video processing applications .... 26

iv Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Respiration monitoring software on an embedded platform</td>
<td>27</td>
</tr>
<tr>
<td>4.1 Reference applications</td>
<td>27</td>
</tr>
<tr>
<td>4.2 Real-time video processing on embedded platforms</td>
<td>28</td>
</tr>
<tr>
<td>4.3 Porting of respiration monitoring applications on the embedded platform</td>
<td>28</td>
</tr>
<tr>
<td>4.3.1 Respirate-ProCor implementation</td>
<td>29</td>
</tr>
<tr>
<td>4.3.2 Porting Respirate-ProCor on the embedded platform</td>
<td>29</td>
</tr>
<tr>
<td>4.3.3 Respirate-AutoROI implementation</td>
<td>37</td>
</tr>
<tr>
<td>4.3.4 Porting Respirate-AutoROI on the embedded platform</td>
<td>37</td>
</tr>
<tr>
<td>4.3.5 Timing analysis</td>
<td>38</td>
</tr>
<tr>
<td>4.3.6 Calculating only one profile</td>
<td>39</td>
</tr>
<tr>
<td>4.3.7 Data Parallelism with OpenMP</td>
<td>40</td>
</tr>
<tr>
<td>4.4 GPU implementation of Respirate-AutoROI</td>
<td>42</td>
</tr>
<tr>
<td>4.4.1 Motivation</td>
<td>42</td>
</tr>
<tr>
<td>4.4.2 GPU naive porting</td>
<td>42</td>
</tr>
<tr>
<td>4.4.3 Timing analysis</td>
<td>45</td>
</tr>
<tr>
<td>4.4.4 Optimization of the GPU-based implementation</td>
<td>45</td>
</tr>
<tr>
<td>4.5 Final optimal implementation</td>
<td>49</td>
</tr>
<tr>
<td>5 Conclusions</td>
<td>50</td>
</tr>
<tr>
<td>Bibliography</td>
<td>52</td>
</tr>
<tr>
<td>Appendix</td>
<td>53</td>
</tr>
<tr>
<td>A Board Bring-up</td>
<td>54</td>
</tr>
<tr>
<td>A.0.1 Installed applications</td>
<td>56</td>
</tr>
<tr>
<td>B ABI board details</td>
<td>58</td>
</tr>
<tr>
<td>C GPU programming</td>
<td>60</td>
</tr>
</tbody>
</table>

Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring
# List of Figures

1.1 ECG respiration monitoring ........................................... 2  
2.1 ProCor algorithm .......................................................... 7  
2.2 ProCor algorithm .......................................................... 8  
2.3 AutoROI algorithm detailed ............................................ 9  
2.4 AutoROI principle ......................................................... 11  
3.1 Provided hardware platform ............................................ 13  
3.2 Board initial exploration ................................................ 15  
3.3 phyCAM-P camera ........................................................ 19  
3.4 Camera software access ................................................. 20  
3.5 Example of Gstreamer pipeline ....................................... 20  
3.6 Gstreamer pipeline used for this board .............................. 21  
3.7 Capturing using the v4l2 API ......................................... 21  
3.8 Maximum camera framerate for VGA resolution ................... 22  
3.9 CPU usage for VGA resolution ......................................... 22  
3.10 Gstreamer pipeline for uncompressed AVI reading ................ 23  
3.11 Needed color space conversion flow ................................ 23  
3.12 Color space conversion execution times ........................... 24  
3.13 Streaming of camera-grabed frames over TCP ................... 25  
3.14 Client-side connection pipeline to visualize the streams ..... 25  
3.15 Streaming an OpenCV Mat with overlay text over TCP .......... 26  
4.1 Respirate-ProCor implementation .................................... 30  
4.2 ProCor implementation .................................................. 32  
4.3 Motion vector calculation .............................................. 32  
4.4 Porting procedure for ProCor and AutoROI ....................... 33  
4.5 Use of compiler flags ................................................... 34  
4.6 Execution modes .......................................................... 35  
4.7 ProCor execution time on Phytec board ............................ 36  
4.8 AutoROI implementation on Phytec board ......................... 38  
4.9 RespirateAutoROI running on Phytec .............................. 39  
4.10 RespirateAutoROI with only one profile calculated ............ 40
LIST OF FIGURES

4.11 RespirateAutoROI with OpenMP optimization 41
4.12 Comparison between parallel and serial execution in terms of core usage 42
4.13 GPU kernels on Respirate-AutoROI 43
4.14 Profile calculation kernel - input and output parameters 43
4.15 DFT kernel 44
4.16 Spectrum multiplication 44
4.17 Inverse DFT 45
4.18 Shift calculation 45
4.19 Timing analysis for the GPU kernels 46
4.20 Running time for different parts of the application 46
4.21 The two DFT versions 47
4.22 Improved time for the kernels on the GPU 47
4.23 Timing analysis for application components 48
4.24 Execution times for the CPU and GPU-accelerated implementations 48
4.25 Core usage for the CPU and GPU-accelerated implementations 48
4.26 Processing pipeline for the optimal solution implementation 49
4.27 Comparison between execution times of the optimized solutions 49

A.1 Toolchain use 55
B.1 Software running stack 59
C.1 Development paths 62

Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring
Chapter 1

Introduction

1.1 Patient care and respiration monitoring

One of the most critical activities in a medical setting involving observing and caring of a patient, is respiratory monitoring.

The main goal of a respiratory monitoring system is to observe respiratory distress signs that may denote first symptoms of a possible pathology such as apnea. There are several signs that a patient may show during monitoring of respiration: use of accessory muscles, changes in skin color, loss of consciousness or an inability to oxygenate the blood, a case that require mechanical ventilation.

The physical signs of respiratory distress may present as a patient appearing short of breath, having an increased work of breathing, use of their accessory muscles, and changes in skin color, general pallor, or partial or complete loss of consciousness.

When the initial efforts of respiratory monitoring show evidence of a patient’s inability to adequately oxygenate their blood, the patient may require mechanical ventilation. [15] The current standard of practice to measure respiration rate is for a nurse to manually count the number of times the person’s chest rises in one minute. Along with breathing behaviour, many phenomenons can be observed, for instance, respiratory sounds, respiratory airflow and respiratory chest movement. Based on measuring one of the above parameters, many respiration monitoring methods have been proposed that fall under certain categories.

Invasive procedures are the ones that penetrate or break the skin or enters a body cavity. Examples involve perforation, incision, surgery. Non-invasive methods are the current standard when it comes to respiration monitoring, including ECG-Derived Respiration (EDR). The ECG(electrocardiogram) is recorded from the surface of the chest which is influenced by the motion of the electrodes with respect to the heart. Through changes in the electrical impedance of the thoracic cavity, the expansion and contraction of the chest accompanies respiration results in motion of chest electrodes. Commercial ECG monitors are usually integrated with respiration monitoring.

Looking at the electrodes from fig 1.1 for this method, contact-based respiration monitoring requires a direct contact with the subject’s body, which can irritate the skin, and
cause skin to grow itchy or become sore, especially if the device is worn for a long period of time. Moreover, it is obtrusive in daily life, considering extra efforts required to wear on and off the device, and may also influence the accuracy of the measurement because of the subject’s awareness of the sensor.

1.1.1 Non-contact based respiration monitoring

As an unobtrusive way to monitor the subject, non-contact respiration monitoring attracts more and more attentions recently. Compared with contact-based respiration monitoring, non-contact methods have advantages in([29]):

- Unobtrusiveness. Subjects are absent from distress caused by a contact sensor to avoid interferences of the measurement.

- Comfort. Sensors attached may irritate the subject’s skin, especially for long time surveillance.

- Convenience. It is convenient for subjects without consideration for sensors and even connected wires.

Non-contact methods are based on measurement of chest and abdomen movement caused by respiration using different types of cameras that track these movements. A basic digital camera such as a webcam can further provide portability and low cost. Philips Research developed several respiration measurement algorithms, referred as ProCor [6],[9] and AutoROI [13],[16] based on optical detection of chest movement. The algorithms were developed and implemented initially on a general purpose desktop computer in order to prove the concept.
1.2 Problem statement

When it comes to using these respiration monitoring methods in the medical environment, the focus needs to be placed on the products that will have them implemented.

The purpose of the application brings the obligation to enclose it into an embedded system for several reasons:

- must not depend on an external infrastructure, like cloud computing environment or a server which can, through failures, jeopardize the monitoring of patients
- assigned to do only a strict set of tasks (monitoring in this scenario)
- efficiency in use of resources (energy, computation resources)
- fault tolerance (have ways of coping with problems during operation)
- stability of the system during continuous operation
- small size
- responsiveness when accessed by a nurse or a hospital worker

A general purpose desktop computer will not conform to all of these requirements hence there is a need to address the system from a software-hardware architectural point of view such that the final product can be successfully used in a medical domain having attained its goals.

The research done at Philips aims at exploring on using an embedded system for respiration monitoring. The system will need to run also other software applications which exceed the scope of this report and project.

1.3 Challenges and objectives

The project is focused on porting and optimizing algorithms for respiration monitoring algorithms on an embedded platform.

1.3.1 Objectives

The objectives of the project are:

- understanding the algorithms
- acquire practical knowledge of the specified embedded system platform
- developing software for using the camera and for streaming in generic video processing applications

Optimization and real-time implementation of video processing algorithms for
camera-based respiration monitoring
CHAPTER 1. INTRODUCTION

- porting of the algorithm on the platform
- optimization of the implementation using an accelerator (GPU)

The use of main processor (CPU) needs to be minimized and, hence, the system needs to turn into a heterogeneous computing platform where other available computing subsystems have to be used. The possibility of using various hardware blocks like graphical processing unit, video processing unit, image processing unit, has to be explored.

Moreover, the board needs to provide the capabilities of installing different software applications and libraries, use of devices like camera (for capturing), disk drive (read large video files), TFT screen (debugging and proof of concept/demo). This is called software board bring-up and presumes that the board hardware circuit works accordingly to the specifications.

1.3.2 Challenges

In such a project, some challenges need to be faced in order to manage successfully to provide the needed embedded system:

- ways to access peripherals like camera, file system or display needs to be explored and developed
- an implementation that should provide real-time execution of the algorithm ensuring functional correctness
- provide ways of hardware acceleration that give the same functional correctness but within a shorter execution time and to release CPU resources

1.4 Report

The report is organized in 6 chapters.

Chapter 1 is an introduction to respiration monitoring together with main focus of the project, objectives and challenges.

Chapter 2 explains the respiration algorithms that are used in the project.

Chapter 3 presents the embedded platform on which the algorithms are to be ported together with details on the interfaces with the peripherals and a basis for further video processing development on the board.

In Chapter 4, the porting procedure is shown with exploration on different parallelization strategies and implementations together with the use of the graphical processor unit. The conclusion presents a chosen optimal implementation by means of software, hardware
and mapping of the former to the latter.

Chapter 5 presents final conclusions on the project and findings regarding possible ways to improve more on the implementation side as future work.

Lastly, one can find in the appendix details regarding the use of the hardware board, system set-up and use of different software, all having been explored during the project timeframe.
Chapter 2

Camera-based respiration monitoring algorithms

2.1 Respiration monitoring algorithms

During this chapter the two respiration monitoring algorithms developed at Philips Research will be discussed: Respirate-ProCor and Respirate-AutoROI. The latter one is a generalized form of the first one. The emphasis in this chapter is being put on the flow of the algorithm and not on the implementation.

2.2 Respirate - ProCor algorithm

ProCor ([6], [9]) has been developed at Philips Research in order to satisfy the requirements of an unobtrusive non-contact respiration monitoring method. The algorithm detects small chest motions of a subject, taking into consideration that this motion is created by respiration. The procedure requires selection of a region of interest as being the chest of the subject considering that it is the place of the body that moves correspondingly to the inspiration and expiration. Certain processing steps are being made onto the selected Region-of-interest(ROI) of the camera frames in order to obtain a valid respiration signal. In fig 2.1 one can see a diagram with the algorithm.

As preprocessing, the frames are received from the camera in RGB format, are converted to YUV from where only the Y(Luma) value is considered in the algorithm. The 1D(one-dimensional) profile of the ROI is obtained through a projection-like transformation onto a vertical axis([6]). The main reason for this is because the strongest respiration motion is perceived by the camera on the vertical axis, hence, it is sufficient to calculate the profile on this axis along each row of the ROI. For the projection, a sum of standard deviation and pixel mean is used as can be seen in formula

\[ y_i = \bar{x}_{i,s} + \sigma_{i,s} \]  \hspace{1cm} (2.1)
where $\bar{x}_{i,*}$ is the mean of pixel values on line $i$ and $\sigma_{i,*}$ is the standard deviation of pixel values on line $i$.

Further on, a high-pass filter is considered to enhance edges and make it more insensitive to changes in global illumination. Afterwards, a low pass filter using Hann windowing is applied to suppress the noise. The algorithm finds the correlation between the current calculated profile and previous one. For that reason, the profiles are stored in a buffer in order to be extracted as a previous profiles.

To calculate the cross-correlation, the two projections are transformed by a Discrete Fourier Transform in frequency spectrum which are multiplied (current profile with the...
conjugate of the previous). The correlation is obtained by applying the inverse Fourier Transform onto the result. The motion vector results from the position of the maximum correlation value. The raw respiration signal is obtained by integration of the motion vector in time domain. After signal normalization peak detection determines the position in the signal of the limit between subject inspiration and expiration, which, further on, is used to determine the respiration rate.

![ProCor algorithm](image)

**Figure 2.2: ProCor algorithm**

Respirate-ProCor has the drawback that it needs as an input the ROI position on the image. This can be done in different ways: a user selects it through a UI that sends the coordinates to the algorithm implementation, a projection onto the subject etc. This drawback is eliminated in another, more improved developed algorithm called AutoROI (Automatic Region of Interest).

### 2.3 Respirate - AutoROI algorithm

Conventional methods to determine such regions for breathing would determine the chest or belly of a subject based on face detection. However, this methodology might fail in situations where the face is partially or fully occluded or hidden in cases such as when covered by a blanket. This is a typical case in hospitals where the monitoring of respiration is highly critical.
Figure 2.3: AutoROI algorithm detailed
Furthermore, methods requiring shape analysis for the detection of the chest or belly would be limited by the position of the subject relative to the camera or to the clothing worn.

For example, Respirate-Procor needs as an input the ROI position in the image. This implies manual selection of ROI or use of computationally expensive algorithms for the selection.

Respirate-AutoROI eliminates the need of such prerequisites and improves the robustness of camera-based respiration monitoring by automatically selecting an appropriate ROI ([13]).

Now, the subject of interest can be located freely in the image because the algorithm detects the ROI position information from the acquired frames.

This is done through the approach of detecting the regions in a video from where the best respiratory signals can be extracted and combined to obtain a high SNR respiratory signal (2.3).

The image is divided into a set of blocks equally sized, the number of the blocks being set beforehand. The algorithm runs on each frame on a video or camera input and has 3 main parts:

- ProCor for each block
- obtain ROI
- calculate AutoROI value and extract respiration signal

In the profile and shift calculation stage the principles and algorithm sub-blocks from Respirate-ProCor are kept, with the slight difference that now to the vertical profile, the horizontal profile is also added for situations when the subject is in a position which proves respiration to be more of a horizontal change than vertical (how it was seen in Respirate-ProCor). The shift values for both directions are accumulated in a form of a single one.

Obtaining the Region-of-Interest means the selection of blocks that contain respiration motion. The motion vectors(shift values) are buffered in order to be used as input for a SNR(Signal-to-Noise ratio) calculation algorithm that will correspond to each block. Spectral analysis is used to make distinction between candidate blocks(blocks that contain respiration) and non-respiration ones.

The candidate block determination step is performed once every frame and can yield a number of true-negatives and false-positives. To allow for a more robust ROI measurement, all the blocks are weighted over time. The weight, called persistence value, is proportional to the number of times a block is selected as a candidate. This way of calculating the actual ROI shift value improves the robustness of ROI measurement since it provides bigger weights for shift values of blocks that have respiration than shift values of blocks that do not have respiration([13]).

The respiration signal is obtained the same way like in Respirate-Procor: integration of the motion vector in time together with signal normalization and peak detection to calculate the respiration rate. In fig 2.4 the overview of Respirate-AutoROI is presented.
The important thing is that each rectangle from the frame has a buffer with the last signal samples in that area thus the ones that are inside ROI will have a signal similar to the respiration whereas the ones outside will look more noisy.

![Figure 2.4: AutoROI principle](image)

Figure 2.4: AutoROI principle
Chapter 3
Software development on the embedded platform

A video processing system represents a device that encompasses several components: sources (capture device, storage), computing environments and outputs (display, storage, network), that performs various techniques on video files or video streams in order to get relevant information, improve the videos or even control a larger system.

The requirements of designing such a system in terms of hardware and software involve a vast amount of data that needs to be processed on platforms with limitations in terms of achievable performance, power consumption and computational resources provided that the computation needs to be done in limited time.

3.1 The embedded platform

The hardware platform that is used for this application is an embedded board with a supplier-specific Linux-based operating system. A photo of the board is provided in the figure 3.1 (for further details, see [10] ).

The main board that carries the processor and memory is a SOM(System on Module) with the name phyFLEX i.MX6 from the vendor Phytec. It is based on the Freescale i.MX6 processor. It is connected through a mapper board to the baseboard that contains all the ports and connector with a touchscreen TFT display and a camera. The camera connection is made through a parallel interface which transmits up to 10 bit color or grey images(limited to 8 bit by the Linux driver).

3.1.1 The hardware specifications

The technical characteristics of the board is shown in table 3.1:

The board has 2 USB 2.0 and a SD slot which can be used to transfer/read large files, for example video files. A 7 inch touchscreen LCD is present which makes it easy
CHAPTER 3. SOFTWARE DEVELOPMENT ON THE EMBEDDED PLATFORM

Figure 3.1: Provided hardware platform

for prototyping and testing without forwarding a screen session to the computer used for development.

The camera supports various resolutions from VGA (640x480) to 2592x1944. It has by default manual control for exposure.

Connection of the board with host has been done through two connections: network cable and serial cable.

3.1.2 The available software for the board

The board arrived with an operating system and tools which can be used to rebuild the system and to develop other applications for it.

On the device side, the operating system is Linux-based, stripped down to necessary device drivers and services. It also included a series of installed applications that are used to run the demo scripts. The Linux kernel version is 3.0.35 and the bootloader is Barebox([5]).

On the host side, the board has BSP(Board-support package) configuration files that can be later used to rebuild the Linux root images with additional packages.

There are also available ARM toolchains that contain the compilers and other necessary tools critical in the building process and later on development process.
CHAPTER 3. SOFTWARE DEVELOPMENT ON THE EMBEDDED PLATFORM

<table>
<thead>
<tr>
<th>CPU name</th>
<th>Freescale i.MX6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>32-bit ARMv7-A</td>
</tr>
<tr>
<td>Core implementation</td>
<td>Cortex-A9</td>
</tr>
<tr>
<td>Nr. of cores</td>
<td>4</td>
</tr>
<tr>
<td>Frequency</td>
<td>1 GHz</td>
</tr>
<tr>
<td>Instruction and data cache</td>
<td>32KB</td>
</tr>
<tr>
<td>L2 cache</td>
<td>1MB</td>
</tr>
<tr>
<td>3D GPU</td>
<td>Vivante GC2000</td>
</tr>
<tr>
<td>3D GPU frequency</td>
<td>500 MHz</td>
</tr>
<tr>
<td>3D GPU nr. of cores</td>
<td>4</td>
</tr>
<tr>
<td>3D GPU API support</td>
<td>OpenGL ES 3.0, OpenCL EP 1.1</td>
</tr>
<tr>
<td>Composition 2D GPU</td>
<td>Vivante GC320</td>
</tr>
<tr>
<td>Vector graphics GPU</td>
<td>Vivante GC355</td>
</tr>
<tr>
<td>VPU(Video-processing unit)</td>
<td>HW video decoder/encoder</td>
</tr>
<tr>
<td>RAM memory</td>
<td>1GB DDR3-SDRAM</td>
</tr>
<tr>
<td>NAND flash memory</td>
<td>1GB</td>
</tr>
<tr>
<td>Camera sensor</td>
<td>12 bit BayerRGB</td>
</tr>
<tr>
<td>Camera interface</td>
<td>10 bit Parallel 33-pin</td>
</tr>
<tr>
<td>USB support</td>
<td>2.0</td>
</tr>
<tr>
<td>Host connections</td>
<td>Ethernet, Seria</td>
</tr>
</tbody>
</table>

Table 3.1: Hardware platform specifications

3.2 Setting the platform for development

In the beginning there was a need for initial exploring of the platform in terms of hardware components and already installed software. Also, there is a need to provide a foundation of tools and specifications as a framework for use in subsequent video processing algorithms on the board.

In order to develop/port an application on an embedded platform one has to analyze all the software layers between the user-interface usage(highest level) to processor runtime details during the application execution. During the initial workings with the board, the class of video processing applications has been decided as an input for package selection to install on the board. Having considered another class of applications, like user interfaces, the set of packages would have needed to be changed accordingly. This is a difference when it comes to software installation on an embedded platform versus a desktop general-purpose computer.

3.2.1 Setting up device software

After considering that the application(s) that need to run on the board are in the video processing class, a series of packages have been verified and installed(if necessary) in order

14 Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring
to support a correct run and various possibilities to use the camera, display, network streaming, utility tools and optimization:

- gstreamer([25]) together with different plugins including: vpuenc(encoder with VPU-Video Processing Unit hardware support), tcpserversink, tcpclientsrc, ffmpegcolorspace, rtph264pay(RTP streaming)
- OpenMP for parallelization support ([27])
- QT including QT server examples ([28])

The board image has been built using software ABI(Application Binary Interface) floating-point support initially since it was by default from the Phytec support DVD(for details on ABI, consult appendix B).

### 3.2.2 Tools and libraries to use in development on Phytec board

A series of tools, programs and libraries have been used in the project:

- OpenCV is a library of programming functions mainly aimed at real-time computer vision ([19])

Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring
• OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors([26]).

• GNU toolchain is a collection of programming tools produced by the GNU Project: cross-compiler arm-cortexa9-linux-gnueabi-gcc, profiler gprof, debugger through network connection - gdb server; ([18])

• Gstreamer is a pipeline-based multimedia framework written in the C programming language([25])

• V4L2 is a video capture and output device API and driver framework for the Linux kernel, supporting many USB webcams, TV tuners, and other devices. Video4Linux2 is closely integrated with the Linux kernel.([21])

• Matlab is a multi-paradigm numerical computing environment which allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages

• Linux tools: top, dmesg - is a command on most Linux- and Unix-based operating systems that prints the message buffer of the kernel, microcom - a is a minimalistic terminal program for accessing devices via a serial connection, ssh - OpenSSH SSH client (remote login program), scp - secure copy (remote file copy program)

• Vivante tools: vCLcompiler - off-line OpenCL GPU kernel source compiler, vEmulator - Vivante GC2000 GPU emulator for desktop GPUs

• Phytec board BSP - Board Support Package provided for the Phytec board

• Eclipse - Integrated Development Environment that runs on Linux and Windows also

• VMWare Player - program to run virtual machines

• OpenMP - is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems ([27])

• POSIX threads - Pthreads, is a POSIX standard for threads and handling concurrency in C/C++ applications ([20])

### 3.2.3 Use of compiler flags

Various compiler flags have been explored and tested that provide optimization, profiling or functionality. Here is a list of the most used ones:

- `-O2`, `-O3` - two levels of code optimization
• -falign-functions=1 - align the start of functions
• -fopenmp - enables OpenMP support
• -falign-jumps=1 - align branch targets to a power-of-two boundary
• -falign-loops=1 - align loops to a power-of-two boundary
• -falign-labels=1 - align all branch targets to a power-of-two boundary
• -mfloat-abi=softfp/hardfp - specifies which floating-point ABI to use.
• -mcpu=cortex-a9 - this specifies the name of the target ARM processor.
• -mtune=cortex-a9 - this option specifies the name of the target ARM processor for which GCC should tune the performance of the code.
• -mfpu=neon/vfpv3 - this specifies what floating-point hardware (or hardware emulation) is available on the target.
• -msoft-float - specifies the format of floating-point values
• -march=armv7-a - this specifies the name of the target ARM architecture.
• -ftree-vectorize - enables vectorization of code
• -std=gnu99 - disables GNU extensions that conflict with C99 code standards
• -funsafe-math-optimizations - allow mathematical optimizations that may be give incorrect results for certain IEEE inputs.

There are also some GPU OpenCL compiler flags used:

• -cl-unsafe-math-optimizations - allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid, (b) may violate IEEE 754 standard and (c) may violate the OpenCL numerical compliance requirements
• -cl-fast-relaxed-math - allows optimizations for floating-point arithmetic that may violate the IEEE
• -cl-mad-enable - allows a * b + c to be replaced by a mad. The mad computes a * b + c with reduced accuracy.

OpenMP

OpenMP (Open Multi-Processing) provides an API for data parallelization of applications onto computational units with multiple cores in shared memory environment.

It has been used in the project in order to optimize the applications by using all 4 cores available on the Freescale i.MX6 CPU. In order to be used, already build OpenMP shared library has been copied in the root filesystem of the board.
3.2.4 GPU programming

There are a couple of points that lead to the use of GPU in an (embedded) system when it comes to video processing applications:

- the independent execution blocks that form the application and can be run in parallel
- the large amount of data which is used in the program and which by division can be accessed in parallel by multiple threads
- the need to free CPU resources (cores) and in some cases system memory (RAM)

The GPU on the Phytec board is Vivante GC2000. It supports OpenCL 1.1 Embedded Profile. Also, the RAM memory is shared between the CPU and GPU. This leads to some limitations in terms of development but also some advantages compared to using a graphics card on a desktop computer, which, usually has its own VRAM (Video-RAM) memory. One advantage is the fast memory transfer between the two, since it is on the same physical storage place.

The OpenCL GPU programming model involves: ([1],[8],[4],[11],[14])

- setting the host (CPU-based computer in most of the cases) and device (GPU)
- setting the parallelization model to be used (data parallel or task parallel)
- deciding on which memories to be used

A set of programs that need to run on the device are written specifically for it in a language similar to C. The Vivante GC2000 GPU has 4 cores that can run each 256 threads in parallel depending on the type of data/instructions that needs to be in execution. There is a memory hierarchy model: the GPU has access to maximum 96MB of global memory, each core has 4KB of local memory and registers.

3.3 Framework for video processing applications on Phytec board

One of the main goals of the project was to specify a framework in the sense of newly developed software, already present software and other rules that can be later one used to develop on the board. The framework needs to address various video processing techniques and algorithms that may be ported on the embedded platform. This involves a way to capture frames from a camera or to read them from a file, a way to change the frame-rate of capture and processing chain, the exposure together with the algorithm used and possibility to visualize the output on the display, in a file or through network streaming.

The explored software blocks are:

- frame grabbing from camera
• frame grabbing from video
• color space conversion
• writing to file
• profiling/debugging of applications
• multi-threading
• output verification
• parameter modification

3.3.1 Frame grabbing

In most video processing applications the input data is obtained from a camera or a video present on the embedded system storage device, in the Phytec board being the NAND flash memory.

Input from camera

The camera provided with the board is phyCAM-P (3.3); a 5M 12 bit color sensor camera that connects through a parallel interface to the IPU(Image Processing Unit) of the Freescale CPU.

Various wrappers that access the camera have been used during the project and they will be presented in terms of advantages and disadvantages.
CHAPTER 3. SOFTWARE DEVELOPMENT ON THE EMBEDDED PLATFORM

OpenCV wrapper

OpenCV is a library with functions for computer vision that simplify and offer a more abstract view on different video/image processing methods. Its use is critical in prototyping algorithms for research and development in this area. To capture frames using OpenCV, one must instantiate the `VideoCapture` giving as a parameter the Linux camera device.

Gstreamer wrapper

Gstreamer is a multimedia framework which provides blocks to create various media-processing applications from audio playback to video playback, editing or streaming. It uses as a design the pipeline pattern.

Each element of the pipeline is provided by a Gstreamer plug-in. Elements can be grouped into bins, which can be further aggregated, thus forming a hierarchical graph. This is an example of a filter graph.

Elements communicate by means of pads. A source pad on one element can be connected to a sink pad on another. When the pipeline is in the playing state, data buffers flow from the source pad to the sink pad. Pads negotiate the kind of data that will be sent using capabilities (controls for the pads that rely on plugin specifications). ([25])

A typical Gstreamer pipeline would look like in fig 3.5, where the source is v4l2 interface with the camera driver and the sink is the standard X-based videosink(from Linux X Window System).

![Figure 3.5: Example of Gstreamer pipeline](image)

The pipeline for the phyCam and Phytec board would be the one in fig 3.6. It uses also a Freescale plugin `i2c` that receives as an input a file which contains the register settings for the camera related to exposure, test patterns, etc. After the camera is set, the frames are grabbed and processed by the pipeline until the element `appsink`
In order to use this pipeline, the same OpenCV class VideoCapture needs to be instantiated having as a parameter the string representing the pipeline.

**V4L2-based capturing**

The OpenCV solution is impossible to use for this problem since the framerate is too low due to the wrapper and the camera parameters like resolution or exposure cannot be changed.

The Gstreamer wrapper does a better job than the OpenCV one but the pipeline is complicated to be controlled once it started. Another issue is the fact that the frame-rate is not constant every time and this may bring issues in the processing done by the algorithm.

<table>
<thead>
<tr>
<th>Solution</th>
<th>OpenCV wrapper</th>
<th>Gstreamer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fps at VGA resolution</td>
<td>3.5-4fps</td>
<td>29-32fps</td>
</tr>
<tr>
<td>CPU use</td>
<td>1 core 100%</td>
<td>1 core 100%</td>
</tr>
<tr>
<td>Constant framerate</td>
<td>Yes</td>
<td>Near - 9/10 frames</td>
</tr>
<tr>
<td>Camera control</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3.2: Camera grabbing solutions comparison

Another solution has been implemented by catching the camera raw output directly. The method is expressed in fig 3.7.

![Figure 3.7: Capturing using the v4l2 API](image)

Of course there is also the device closing step and de-initialization of the camera. The device opening is like a normal Linux file opening statement. The capture querying determines the ability of the camera, then the registers are set through the external Freescale i2c tool. After that, the image format is being set consisting of resolution and color space. Ultimately the capturing is being done in buffers of raw BayerRGB information.
CHAPTER 3. SOFTWARE DEVELOPMENT ON THE EMBEDDED PLATFORM

This capturing option brings great deal of freedom for the developer since it allows one to interact in this way directly with the hardware, hence, assuring a better control when it comes to setting fixed frame-rate or fast change of camera parameters. It also allows one to use the Posix threads standard for setting the capture on another thread independent on the video processing that would be done in the follow-up.

In fig 3.8 and fig 3.9 the maximum framerate achieved through each solution together with the CPU usage (provided by the top tool) respectively are shown.

If the v4l2 interface option is being used, then the application needs also a color space conversion from BayerRGB to RGB, since this was implemented before through a Gstreamer pipeline element.

![Figure 3.8](image1.png)  
**Figure 3.8:** Maximum camera framerate for VGA resolution

![Figure 3.9](image2.png)  
**Figure 3.9:** CPU usage for VGA resolution

Uncompressed AVI video file input

When it comes to use of video file as an input, several formats have been tested. In case of uncompressed AVI file as input, the OpenCV video capture method doesn’t work
appropriately since it provides the video frames in non-constant period of time, hence another Gstreamer pipeline has been used, namely:

![Gstreamer pipeline for uncompressed AVI reading](image)

The pipeline uses the filesrc source element that outputs buffers to the avidemux plugin which ultimately provides each frame to the application. In this case, Gstreamer provides a constant framerate due to the nature of the input: a video is a set of frames at constant frame-rate already compared to camera input.

### 3.3.2 Color space conversion

In video processing applications on general purpose computers not too much emphasis is being put on the color space conversion step since it’s not a very important processing block in an algorithm. Also, the USB cameras have more complete image processing units and drivers that provide this utility already implemented in hardware, invisible to the developer.

On the Phytec board, the applications need optimized algorithms of color space conversion. The provided camera doesn’t have very supportive drivers and the IPU unit integrated in the Freescale processor does not have color conversion support having Bayer RGB as input.

Since the algorithms that will be ported on the Phytec board work only with luminance grey value of the pixels, there is a need for Bayer RGB to RGB and then to YUV conversion (3.11).

![Needed color space conversion flow](image)

The Bayer RGB to RGB conversion is being made by the bayer2RGB element in the Gstreamer pipeline. The RGB to YUV conversion has been reviewed with different implementations:

- use of the OpenCV function RGB to YUV

Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring
CHAPTER 3. SOFTWARE DEVELOPMENT ON THE EMBEDDED PLATFORM

- use of OpenCV function RGB to Y
- re-implementation
- optimized implementation of conversion

In fig 3.12 the different implementation times are shown for a VGA image. The optimized implementation uses bit-level operations for better execution time and calculates only the Y value.

![Figure 3.12: Color space conversion execution times](image)

3.3.3 Network streaming on Phytec board

An important requirement for the project has been to provide ways of streaming images from the camera or from a running program to the network in order to give remote monitoring of the subjects.

The chosen way to achieve this is by using Gstreamer pipelines but with network-specific plugins that deal with all the layers that prepare an image to be sent on the network). There are two possible ways to stream information over the network through Gstreamer:

- stream the frames grabbed from the camera directly to a client through a port
- stream packets of data that can be anything from camera images to debugging output from the program

Streaming camera frames

In the case of the first scenario, the pipeline is shown in fig 3.13.
CHAPTER 3. SOFTWARE DEVELOPMENT ON THE EMBEDDED PLATFORM

Figure 3.13: Streaming of camera-grabed frames over TCP

The streaming pipeline sends the frames grabbed by the camera in JPEG compression format through network, namely as a TCP server with an address and a port to which clients need to connect to visualize the streams. The element \textit{vpuenc} of the pipeline represents the JPEG encoder with hardware support through the VPU unit (Video-Processing Unit).

The client would have to connect with a pipeline like in fig 3.15.

Figure 3.14: Client-side connection pipeline to visualize the streams

Streaming other information

To stream other information than the camera-grabbed frames, an application has been developed as proof of concept for using Gstreamer programming framework and stream through it different data.

The application uses a special Gstreamer element that is used to insert data into the pipeline, namely \textit{appsrc}. The architecture is shown in fig

Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring
3.3.4 Quality benchmarking of video processing applications

It is considered that the main goals of the project involve porting and optimizing a set of applications in order for them to run on another platform. This means that all applications have already an implementation available and can be tested in their original environment. Hence the output can be compared with the new output obtained from the new ported programs.

For this the correlation between the obtained signal of the original implementation and the new signal has been measured. This depends on the input framerate, exposure, and other variables that need to be the same in both testing situations. A video is used in order to compare more subjectively the implementations.

In terms of the real-time requirements of the application, the to-be-ported algorithm determines a minimum frame-rate for which it will execute correctly. In this project, the respiration monitoring application needs to run in real-time at minimum of 7 fps (frames-per-second) in order to extract the respiration signal correctly.
Chapter 4

Respiration monitoring software on an embedded platform

In this chapter there will be presented the work done on porting and optimization of the respiration algorithms from Chapter 2 for the specific hardware platform described in Chapter 3. For further details please consult appendix on the embedded platform hardware and software.

Firstly, some points will be explained regarding the original implementation with timing analysis on desktop system of the application.

Afterwards, the porting of the implementation onto the embedded platform will be explained and considered as a naive implementation for further optimization. On this naive implementation, detailed timing analysis is being shown together with parallelization of the code for the specific hardware.

The GPU section presents the use of the graphical processor programming model OpenCL in Respirate-AutoROI and the improvement that it brings with regard to execution time.

Later on, the integration between the use of CPU and GPU is discussed and an optimal implementation is being presented for the algorithm onto the Phytec board.

4.1 Reference applications

Reference applications based on the two algorithms were provided. They were developed in C/C++ with Visual Studio IDE to run on Windows environment with x86/64 general-purpose processor. No attention was given for parallelism of the implementation and neither any hardware acceleration.

There were used as algorithm plugins in the form of DLL libraries which were imported into another application for execution, testing and debugging developed within Philips.
4.2 Real-time video processing on embedded platforms

What does real-time mean? In the software engineering paradigm, a real-time system is one whose logical correctness is based both on the correctness of the outputs and on their timeliness, more specifically, the time at which the output is obtained needs to be known beforehand and is a requirement in building the system. There are three types of real-time systems ([7]):

- **Hard**: A real-time task is said to be hard if producing the results after its deadline may cause catastrophic consequences on the system under control.

- **Firm**: A real-time task is said to be firm if producing the results after its deadline is useless for the system, but does not cause any damage.

- **Soft**: A real-time task is said to be soft if producing the results after its deadline has still some utility for the system, although causing a performance degradation.

For this project, a real-time system definition within the signal processing paradigm would be more precise since signal processing applications are a subclass of software applications: completing a certain number of operations in a specified interval of time set by the period over which the data arrived in the system. If some calculations need to be performed on input from a camera, the result needs to be obtained in the time between grabbing of two consecutive frames. In signal processing it is also very important to analyse the time taken by data transfers([12]).

Another point of view to be considered in signal processing applications, especially video processing, is determined by the physical properties of the ones that interact with the system: humans. For example, in order for a person to perceive a continuous motion on a screen, the image needs to be updated at each 40 ms (25 frames per second).

4.3 Porting of respiration monitoring applications on the embedded platform

In order to successfully port an application onto a new(embedded) platform, certain things need to be taken into consideration from the start:

- performance in terms of application execution time should not drop significantly

- small binary code resulted(this depends on the platform)

- paying attention to libraries used

- functional correction of output for each code block in the application needs to be verified
CHAPTER 4. RESPIRATION MONITORING SOFTWARE ON AN EMBEDDED PLATFORM

4.3.1 Respirate-ProCor implementation

Refer to 2.1 for theoretical details of the algorithm. The implementation relies on four main parts:

- ROI (Region-of-Interest) selection
- ProCor algorithm
  - Profile (vertical projection) calculation
  - Shift (motion vector) calculation
- Respiration signal extraction
- Respiration rate calculation

4.3.2 Porting Respirate-ProCor on the embedded platform

Since the original application was developed in C/C++, the cross-compiler toolchain explained in Chapter 3 was used to compile the code for the ARM platform. The steps in the porting of application are:

- separate all OS/hardware dependence functions from the project (algorithm) logic
- find the non-standard libraries that are used in the implementation and install/test them on the new platform
- consider the algorithm blocks and partition the code into logical independent implementation blocks
- run the application blocks on the new platform one by one, considering the computation result
- consider optimizing the implementation for the specific platform after a naive porting is working correctly i.e. gives the same output
- consider real-time execution and benchmark the application for different parameter values

Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring
CHAPTER 4. RESPIRATION MONITORING SOFTWARE ON AN EMBEDDED PLATFORM

Figure 4.1: Respirate-ProCor implementation

In fig 4.1 one can see a UML sequence diagram of the ported algorithm. Since the implementation is in C, there is no Object Oriented Programming hence the classes are declared structure variables. Here is an explanation of the elements in the figure:

- **User** represents an actor, in this case the user that needs to set the Region-of-Interest
- **ROI GUI** is the GUI window where the user selects the ROI
- **main** is the entry point of the program
- **Camera** is of OpenCV camera VideoCapture type in this example and it makes use of what has been said in 3.3.1, in this example the Gstreamer wrapper being used
- **RespirateWrapper** is a structure type with wrapping methods for the respiration monitoring algorithm

30 Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring
• **Respirate** is the structure for the respiration monitoring algorithm

• **ProCor** is the structure for the motion vector calculation

• **RespiratieSignal** contains the extracted respiration signal information

• **RespiratePeaks** contains the peak information of the respiration signal

• **ProCor GUI** represents the data structure for the results’ display on the screen as OpenCV windows.

• the blue frames contain the application that has been mostly changed due to reference incompatibilities with the Board Support package on the board

• the red **ProCor** frame represents the actual algorithm for respiration monitoring

• the application returns to the **User** through the ProCor GUI the respiration rate and plots the respiration signal.

In fig 4.2 the motion vectors calculation is being shown. The profile buffer is implemented in order to memorize a fixed set of past vertical profiles for the selected ROI.

The motion vector(shift value) is calculated like in fig 4.3.

The procedure used to port the application is shown in fig 4.4.

The figure shows a simplistic overview on the different steps taken into successfully getting the same results on the Phytec board. The main goal is to achieve a real-time execution of the algorithm.

The implementation makes use of the following OpenCV functions:

• filtering function

• Discrete Fourier Transform calculation function

• frequency domain spectrum multiplication

• search for maximum value in an array

• normalization of an array

Gladly, the version installed on the Phytec board supports all of them.

The framework and system information from chapter 3 has been used in order to use the camera accordingly. Another change that had to be made was related to a global timer function that provides the timestamps at which the respiration signal samples are stored. Since the methods to initialize and use a timer in Windows are different than in Linux, some methods had to be changed.
Analysis of output functional correctness

A challenge was faced in the first steps of the porting phase when dealing with the image gray input values for the algorithm. The output values from the color space conversion
were different compared to the ones in the reference implementation. As stated in section 3.3.2 of the report, this was the main reason for choosing to re-implement the conversion.

Though the respiration rate calculation would not get affected by this issue, extracting other relevant respiration data out of the signal would need to take this into consideration. The implementation of such video processing algorithms is usually modular, that is, the framework (as the one in the reference implementation and the one implemented during this project) is independent of the algorithm implementation and such reconsideration of reimplementing one resulted from analysis of the other does not confine to good software development ethics.

In the end of the porting phase the output of the each block and subsequently whole application has been achieved to be the same as in the reference implementation.

**Use of compiler optimization flags**

In section 3.2.3 various compiler flags have been shown that were tested on the Phytec board. All the compiler flags have been used in Respirate-ProCor for basic optimization. Between -O2 and -O3, the latter has been used; between -mfpu=neon/vfpv3 the former has been used to force the compiler to use the NEON SIMD acceleration unit on the ARM-Cortex A9 cores. The running results are given in fig 4.7.
CHAPTER 4. RESPIRATION MONITORING SOFTWARE ON AN EMBEDDED PLATFORM

Figure 4.5: Use of compiler flags

Hardware floating point support

In section 3.2.3 it has been discussed about a version of the BSP having hardware floating point support, that is, the code is compiler using specific instructions to be executed by the VFP(Vector Floating Point) unit on the ARM processor.

This can bring major execution time optimization(in theory). Running the Respirate-ProCor on this different Linux configuration gives very interesting results.

<table>
<thead>
<tr>
<th>Execution mode</th>
<th>Output correctness</th>
<th>Time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard FP - Naive</td>
<td>Correct</td>
<td>30</td>
</tr>
<tr>
<td>Hard FP - With optimization on</td>
<td>Not correct</td>
<td>18</td>
</tr>
<tr>
<td>Soft FP - Naive</td>
<td>Correct</td>
<td>35</td>
</tr>
<tr>
<td>Soft FP - With optimization on + NEON support</td>
<td>Correct</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 4.1: Table Various execution modes for Respirate-ProCor on Phytec
CHAPTER 4. RESPIRATION MONITORING SOFTWARE ON AN EMBEDDED PLATFORM

Figure 4.6: Execution modes

The execution time on hardware floating point support fluctuates (the table from above contains an average value). With compiler optimizations on, the resulted respiration signal is not the same as the reference one although the respiration rate correct. This, in fact, is known as the 323 "Bug" of GCC compiler when it comes to compile for VFP floating point unit on different architectures (it is not really a bug, but more of an impossibility of the GCC compiler to optimize the code at the same time with generating instructions for the VFP unit and to give the same output). The profile calculation in Respirate-ProCor makes use of \( \cos \) as cosine and \( \text{pow} \) as power in its computation which brings inconsistencies in calculation result on the floating point unit when optimization is on ([22]).

The similar execution times between the two execution modes (with hardfp and with softfp) results also from the balancing between the advantages and disadvantages of both: the fact that the NEON and VFP units share most of their registers, the VFP unit is serial (the ARM-v7a instruction-set architecture deprecates the vector mode usage) and has floating-point support while the NEON is SIMD vectorial but deals with integers.

Because of this result (4.6 and 4.1), the hardware floating point version of the BSP is not chosen as a viable solution for future software implementation as an optimized version. The rest of the algorithms running on the board will be executed using the softfp and neon compiler flags together with -O3 optimization which brings the optimal balance between performance, functional correctness and stability (the execution time is the same for each frame, no fluctuation).

Timing analysis with parameters variations

An analysis has been made on ported ProCor on Phytec board in order to see execution time of each block in the processing chain depending of the size of ROI. As one can see, in figure 4.7, all the processing elements execution time scale with the amount of data that needs to be considered (ROI size).
CHAPTER 4. RESPIRATION MONITORING SOFTWARE ON AN EMBEDDED PLATFORM

Figure 4.7: ProCor execution time on Phytec board

The vertical projection, high-pass filtering, Hann windowing, DFT, Spectrum multiplication and Inverse DFT depend a lot on the size of the input data, whereas the other blocks, shift calculation, signal normalization, peak detection, respiration rate calculation have a fixed execution time since these blocks use as inputs fixed-size data.

Moreover some algorithm blocks depend more on the vertical size of ROI and less on the horizontal size. For example, the DFT block will depend only on the vertical size since that is the length of the vertical profile that has been calculated.

Real-time processing in Respirate-ProCor

As can be seen in fig 4.7, the use of compiler optimization flags makes the Respirate-ProCor Phytec implementation to work in real-time at 30 fps (frames per second) for a ROI having the full-frame size, since it runs at around 20 ms per frame.
4.3.3 Respireate-AutoROI implementation

Refer to fig 2.3 for theoretical details on the algorithm.

The blocks that implement camera frame grabbing, color space conversion, signal normalization, peak detection and respiration rate calculation are the same like in Respireate-ProCor hence no emphasis will be put on them. The only initial algorithm difference is the fact that in the beginning the rectangle coordinates for the image are calculated beforehand in order to be fed to the algorithm later on.

Another difference comes in the profile calculation phase when in fact the rectangles are projected each on both axis(not only vertical) hence there are two profiles that are calculated which are accumulated later on.

4.3.4 Porting Respireate-AutoROI on the embedded platform

In fig 4.8 is shown the implementation of the AutoROI respiration monitoring algorithm. Here is some information regarding the diagram and the algorithm execution:

- There are two loops, the black framed one that contains execution information for each frame and the blue frame one that contains execution for each rectangle
- The profiles are calculated in CalculateProfiles message 2, accumulated and stored in the ProfileBuffer
- The motion vector for the block is calculated
- The motion vector for the block is stored in the BlockSigBuffer as a raw signal sample
- The SNR(Signal-to-noise ratio) is calculated for the block and it determines if the block represents valid respiration
- The persistence of the value(number of frames when the block has been valid) is calculated
- the ROI is chosen
- Motion vector for the chosen ROI is calculated with persistence values and motion vectors of blocks that form it
- The ROI motion vector is stored in the RespirationSignal raw signal buffer
- The signal is normalized
- The peaks of the signal are extracted
- The respiration rate is calculated
4.3.5 Timing analysis

In fig 4.9, Respirate AutoROI is executed with different number of blocks on a 768x576 resolution video. An interesting thing that can be seen from the chart is that once the image
is split into more blocks, the compilers detects more independent data and it optimizes more. With the block number increase, the difference in execution time between the naive implementation and the optimized one increases.

Also, the execution time does not scale only according to the number of blocks. Once there are more blocks, the calculated profiles are smaller, hence, the algorithm sequence that runs for each block takes less time.

4.3.6 Calculating only one profile

One thing that has been changed in the algorithm is to not calculate the horizontal profile since it doesn’t have much effect on the respiration signal extraction. In any position the subject is, his chest will move on the horizontal axis, even if he is in bed(same as in Respirate-ProCor).

A new timing analysis has been done on the modified algorithm, which is shown in fig 4.10.

This clearly brings improvement in performance in terms of execution time. The algorithm can now run with 32x24 blocks at 10 frames per second real-time.

Figure 4.9: RespirateAutoROI running on Phytec

Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring
4.3.7 Data Parallelism with OpenMP

A big step in optimization the algorithm is to use the CPU multi-core resources and try to divide the execution on the all 4 cores. The motivation for this is the independence of operations and data used on each block in AutoROI. In the best case scenario, this would bring an execution time 4 times less hence achieving real-time run for bigger number of blocks at higher frame rate.

There are various ways to parallelize an application. For this task OpenMP has been used, a high level API for specifying parallelism inside programs for multi-core processors.

OpenMP requires operating system support in the sense of an installed shared library, a compiler flag at compilation time, a header included in the beginning of the file and a statement added where the code is possible to be parallelized. Usually, the loops specified by for keyword are the aimed ones. Consequently, the data used inside the loops needs to be independent for each iteration.

A new timing analysis done after using OpenMP is shown in fig 4.11.

The new execution time is not that satisfactory as it was stipulated in the paragraph before. One reason for this is the fact that OpenMP creates multiple threads on the cores that are available but does not copy the memory objects for each thread specifically. Each
thread will now access the same shared memory but this is limited when it comes to the bandwidth between the ARM cores and the RAM memory. As implementation examples for this, the input image is accessed in parallel by all cores at once and the profile buffer is written by all cores at the same time.

In the serial version, Core 1 is doing the AutoROI processing while Core 2 is used for capturing the frames from the camera and color space conversion.

In the parallel version, Core 3 is used for capturing the frames while cores 1, 2 and 4 are used to execute the AutoROI algorithm.

The board is limited now by memory bandwidth and not processor frequency. In fig 4.12 the core usage is being shown when OpenMP is used versus the serial execution.

This brings the need of more optimization.
4.4 GPU implementation of Respirate-AutoROI

4.4.1 Motivation

The optimization of Respirate-AutoROI using OpenMP gave possibility to implement parallelism on the Freescale i.MX6 processor with some improvement in execution time and core usage balance.

One of the requirements for this project is to free the CPU cores from doing work hence the need of using hardware acceleration available in the SoC(System-on-Chip). Keeping this in mind, the independency of data from the rectangles and the fact that the rectangles have the same size, it has been decided that a GPU-based implementation is a good solution.

4.4.2 GPU naive porting

OpenCL API has been used for the implementation having the version 1.1 Embedded Profile.

Refer to section 3.2.4 for how the GPU is being initialized on Phytec board.
Profile calculation and motion vector (shift) calculation have been ported on the GPU in the form of on-device executed kernels. Fig 4.13 shows the kernels that have been developed and the execution order.

![Figure 4.13: GPU kernels on Respirate-AutoROI](image)

In the coming subsections there will be details about the implementation of each kernel.

**Profile calculation kernel**

The parameters for the kernel are specified in fig 4.14.

![Figure 4.14: Profile calculation kernel - input and output parameters](image)

The implementation is similar to the profile calculation in CPU version of the implementation.

**DFT kernels**

The next step in the algorithm is the calculation of the DFT (Discrete Fourier Transform). As a naive port the DFT has been calculated by using the DFT matrix form of the transformation (4.15). The output array \( Y \) is obtained by multiplying a constant matrix \( Tw \) with the input array \( X \).

\[
Y = Tw \times X
\]

\( Tw \) represents the DFT matrix calculated from the twiddle factors (complex roots of unity - [2]). For development speed, these factors have been provided from Matlab calculation.
CHAPTER 4. RESPIRATION MONITORING SOFTWARE ON AN EMBEDDED PLATFORM

Figure 4.15: DFT kernel

Spectrum multiplication kernel

The spectrum multiplication receives as inputs the DFTs of the previous and current profiles and provides the correlation as output (4.16).

Figure 4.16: Spectrum multiplication

Inverse DFT kernel

For the Inverse DFT, the same twiddle factors from the DFT have been used but with a small change in kernel implementation (4.17).

Shift calculation kernel

The motion vector value (shift value) has been calculated in the last GPU kernel by taking as an input the resulted correlation array between the previous and current profiles and determining the value based on the maximum value of the array and its position (4.18).
4.4.3 Timing analysis

Timing analysis on the naive implementation is being done for the video case with 768x576 input resolution image and a division in 48x36 rectangles of the frame.

It proves that there is much need for improvement as it can be seen in fig 4.19, where the GPU execution time for each kernel is shown. Moreover in fig 4.20 it can be seen that the GPU execution time takes the most in comparison with the accumulated time taken by memory transfers of objects between the CPU memory and GPU memory. This is the case considering that the DFT implementation is not a very efficient FFT (Fast Fourier Transform) and the global memory is the same for the CPU and GPU (shared memory).

4.4.4 Optimization of the GPU-based implementation

The optimization has been focused on re-implementing the Discrete Fourier Transform and diminishing memory transfer time where was the case.
DFT re-implementation

Since in 4.19 it can be seen that the bottleneck is the DFT kernel execution time, a new algorithm had to be reimplemented. For this, the implementation in [3] has been used. It is the Cooley-Tukey FFT algorithm([24]).

The implementation is GPU specific by having the following characteristics:

- use of SIMD hardware by replacing reference data types with vector data types like float2
- use of GPU registers inside the kernel and not the global memory or local memory when it comes to computation of the output array

Moreover, the twiddle factors do not need to be computed in another environment(Matlab) and copied to the GPU memory before running the kernel, hence there is less memory transfer.

If the anterior DFT kernel can have multiple sizes for the input array, in this current implementation the only possibility is the size of 16, hence the number of blocks for 768x576
video input is 48x36 (due to project time constraints the generalized implementation could not be finalized). In fig 4.21 it can be seen the improvement in execution time for this specific case.

Eliminating a kernel

Another optimization that has been done is to eliminate one DFT kernel (for transforming the previous profile) by keeping a DFT-profile buffer on the HOST-CPU memory where the calculated DFT of the current profile is stored.

In fig 4.23 one can see the new execution times for the kernels on the GPU and the memory transfer time. The memory transfer time is higher since now there is a need to copy the DFT result from the DFT kernel to the host buffer.
Figure 4.23: Timing analysis for application components

**New GPU vs CPU**

As can be seen in fig 4.24, the execution time on a 768x576 input image and 48x36 rectangles has been improved. Also, the core usage has been decreased since the GPU is doing part of the processing (fig 4.25).

Figure 4.24: Execution times for the CPU and GPU-accelerated implementations

Figure 4.25: Core usage for the CPU and GPU-accelerated implementations
4.5 Final optimal implementation

A final comparison is being made on all the optimized implementations in order to select an optimal solution. Referring to fig 4.27, one can see that the best solution until now is the GPU-accelerated one which releases a CPU core from processing and improves the execution time.

The OpenMP parallelized version has a similar performance to the GPU-accelerated version but uses more than 1 CPU core (100% of one core and 30% of another). In fig 4.26, the processing pipeline (without many details) is shown, where the Graphical processing unit calculates the motion vectors for each block providing the result for the Region-of-interest selection and respiration signal extraction.

The optimal GPU-based implementation runs at 7 fps in real-time according to the requirements specified in the beginning (3.3.4).

Figure 4.26: Processing pipeline for the optimal solution implementation

Figure 4.27: Comparison between execution times of the optimized solutions
Chapter 5

Conclusions

This master thesis presents the work in development of video processing applications on an ARM-based embedded platform. First, the platform has been explored and set to be used in this class of applications by assessing the camera capturing (using the Gstreamer framework or the V4L2 Linux Media infrastructure) and the network streaming possibilities (using Gstreamer).

Afterwards, two algorithms have been studied and their reference implementation has been ported on the embedded platform considering a specific procedure. Consequently, the compiler flags and the parallelization on the CPU did not provide the necessary execution times in order to allow real-time processing at minimum allowed frame-rate.

The need of freeing CPU resources and the parallel nature of the algorithm provided the motivation to optimize the application by porting different blocks on the GPU using OpenCL EP API. This allowed a better utilization of the SoC computing resources and shorter execution time by 30%. At every step, the functional correctness of the results has been validated.

The result of the project represents a starting point for further development on the board especially on the path of video processing.

Future work

Many things can be made to improve the current CPU-GPU implementation of the Respirate-AutoROI algorithm.

Firstly, the device-code kernels can be improved to make better use of the SIMD vector instructions, especially using float4 and float16 operations since benchmarks have been made and shown that these two provide the best timing. At the moment, only two kernels use vector instructions float2 which provide faster execution time than normal float but it is still not the best solution ([17]).

Moreover, the DFT kernels need to be extended in order to calculate the result for arrays with size different than 16.

The GPU implementation can be further optimised by paying attention to the memory access from the point of view of pattern accesses to the space where the objects are stored:
global memory, local memory or registers.

This brings together the possibility to try to access directly the host-CPU RAM by using host pointers. It would allow the elimination of memory transfers which would be after all useless since the memory is shared.

Furthermore, the GPU execution parameters like number of work-items and local-items can be changed and research upon different configurations. This, of course, should be done alternatively with other optimizations or changes in the code since there isn’t know how the Vivante compiler runtime schedules the thread execution.

Another future step is to integrate all solutions, network streaming based on VPU (video processing unit), fast camera access and GPU-based computation that could allow the platform CPU to be used with other applications that do not provide same parallelism.

A much better solution would provide the ground for the GPU to get the frames from the camera directly and then do the needed processing on the GPU.

Once the GPU solution is better managed, the remaining computation on the CPU can be analysed in order to see if NEON assembly instructions can be used for optimization purposes.

Through these guidelines, the application can be further optimized in order to let others to run in a heterogeneous embedded computing environment where the resources are managed safely and the hardware platform proves its power.
Bibliography


Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring


Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring.
Appendix A

Board Bring-up

OSELAS toolchain is the cross-compiler that is used to build the packages and, later on, the programs that run on the board. It is configured for a fixed version of kernel, glibc, binutils, gcc and processor architecture. Cross-compilation means the action of compiling(building) software on a computation system for another computation system in order to speed up the development time, in the case of more needed resources (RAM memory, drive space), and/or when the software needs to be accessed by multiple nodes through network access.

For this board, the current toolchain configuration(version 2011.11.1) needs the following prerequisites of the host computer:

- kernel - 2.6.39 sanitized
- glibc - 2.14.1
- binutils - 2.21.1a
- gcc - 4.6.2


This toolchain can be updated with a newer version provided that the hardware drivers have support for the software components (being the link in the software chain that is attached to hardware, which cannot be changed).

Refer to image A.1. The OSELAS toolchain needs to be build again in case one needs to update it. This process takes time since it’s about the compiling of a compiler as a complex process.

Details regarding the prerequisites: Kernel refers to the kernel present in the host system, main reason why this cannot be done in a Windows environment(that easy) - the system needs to be Linux itself. Glibc is the main C library of Linux http://www.gnu.org/software/libc/. It basically is the implementation of C programming language for Linux and it sticks to the compiler since the OS is written in this language all-together. For this reason the compiler(cross-compiler in our case) needs it. Binutils is the collection of
binary tools used in the building process of a program in Linux environment: as, ld (GNU assembler and linker). GCC is the compiler on host computer that is used to compile the cross-compiler. As one can see, gcc command is different than arm-cortexa9-linux-gnueabi-gcc one. This shows that is the compiler for x86-x64 processor architecture - desktop processors - versus the arm version for Cortex-A9 cores.

On the DVD that came with the board, the toolchain was already compiled for speeding development.

Once it is built, it will be used later as a main resource in PTXdist image build and development of applications for the platform.

The toolchain resides in folder /opt/OSELAS.Toolchain-2011.11.1/ having the compiled binaries in /opt/OSELAS.Toolchain-2011.11.1/arm-cortexa9-linux-gnueabi/gcc-4.6.2-glibc-2.14.1-binutils-2.21.1a-kernel-2.6.39-sanitized/bin/ on the host computer (in the DVD system). Here are the main tools in the toolset:

- arm-cortexa9-linux-gnueabi-as is the assembler
- arm-cortexa9-linux-gnueabi-gcc is the C compiler for linux,
- arm-cortexa9-linux-gnueabi-g++ is the C++ compiler for linux
- arm-cortexa9-linux-gnueabi-ld is the linker
- arm-cortexa9-linux-gnueabi-gdb is the GNU debugger
- arm-cortexa9-linux-gnueabi-gprof is the profiler

There are also other tools present for development purposes but these are the most important ones used in the process.

PTXdist is a program used to config and build the final BSPs for the board. It provides a possibility to save various configs, graphically chose packages and set things. It
also detects dependencies between packages so that if one package needs another one it will install the latter one automatically beforehand. If one intends to understand better how PTXdist works, here is a manual http://www.pengutronix.com/oselas/bsp/pengutronix/download/OSELAS.BSP-Pengutronix-Generic-arm-Quickstart.pdf

There is also a good presentation related to this: http://elinux.org/images/6/6c/Schwebel-Customizing_with_PTXdist.pdf

One may find these two docs also in the SVN folder PTXdist. On the DVD system, PTXdist is installed in folder /usr/local/lib/ptxdist-2012.03.0/ and the BSP projects are maintained in /opt/PHYTEC_BSPs/.

In the folder /opt/PHYTEC_BSPs/BSP-Phytec-phyFLEX-i.MX6-PD13.2.3/ one may find the PHYflex-i.MX6 project with the already built BSP for the board. The folder platform-phyFLEX-i.MX6 is the current project one which contains the packages, sources, libraries, board root folder and the images created. Each BSP has two sets of settings and packages to be configured for the board:

- the kernel to be configured with ptxdist kernelconfig
- the root filesystem to be configured with ptxdist menuconfig

The root filesystem represents will represent the image that is ported on the board NAND flash memory of 1GB. It is limited to a size(unknown now but probably under 1GB). Refer to BSP quickstart for further details on building the OSELAS toolchain and configuring PTXdist.

The kernel configuration is hardware constrained and includes the add of certain drivers, settings for the i.MX6 processor and peripherals.

Both the kernel and root configurations need to be run from the folder of the project. After setting things, the command ptxdist go is the one that looks for package dependencies, downloads sources from the Internet, builds the packages for the board. Refer to file Octavian_phytec.docx for details. Refer to section A.0.1 for the applications on the board section. Also, add the package avi of Gstreamer (look in each set of plugins and check it on) for reading uncompressed avi files.

Since the operating system is based on Linux, there is an easy access to find sources, packages, already build applications and information on the Internet.

### A.0.1 Installed applications

There are many sorts of applications that can run on the board. Each has certain software prerequisites that need to be installed on the board:

- applications with a GUI - need QT, OpenGL
- multimedia applications: use a camera, do signal processing, plays video/audio - need Gstreamer, OpenCV,
• applications that do computations with GPU/CPU - need OpenCL, CPU hardware floating point support, OpenMP

The main application that was used in the project is one that combined all 3 types with different possibilities to achieve an optimal solution. In order for an application to use the display, it needs to

• write directly to the display memory mapped framebuffer
• use QT that has access to the framebuffer to easily draw/write objects on the display
• use OpenGL that also has access to the framebuffer through the DirectFB package that needs to be installed

OpenCV is using QT as a backend to print objects to the display. Refer to the QT document that I sent for details. Basically, QT works as a client-server environment where an application can be a server or a client. From this point of view OpenCV applications are considered clients hence they need a server to run in the background. The server applications are called QWS applications and are run with -qws parameter to the command. One can use the demo that comes with the board as a server application when developing OpenCV programs but can also use the examples in the ptxdist menuconfig configuration in multimedia applications/QT/QTexamples/qws/.
Appendix B

ABI board details

In computer software, an application binary interface (ABI) is the interface between two program modules, one of which is often a library or operating system, at the level of machine code.

ABIs cover details such as (see [23]):

- the sizes, layout, and alignment of data types
- the calling convention, which controls how functions' arguments are passed and return values retrieved; for example, whether all parameters are passed on the stack or some are passed in registers, which registers are used for which function parameters, and whether the first function parameter passed on the stack is pushed first or last onto the stack
- how an application should make system calls to the operating system and, if the ABI specifies direct system calls rather than procedure calls to system call stubs, the system call numbers
- in the case of a complete operating system ABI, the binary format of object files, program libraries etc.
Optimization and real-time implementation of video processing algorithms for camera-based respiration monitoring
Appendix C

GPU programming

In order to use the GPU some things need to be taken into consideration:

- The technology that is used in order to program the GPU, example: OpenCL;
- The version of the technology that is available in terms of hardware, example: 1.1;
- The profile of the technology that is implemented, example: Embedded Profile
- Use of C or C++
- Implementation of a standard GPU application
- Way to test the application
- Debugging the application

1. The technology represents the API supported by the GPU hardware acceleration unit. In Phytec case, it can be:

- OpenGL
- OpenCl

In this project case, OpenCL has been used because of improved support for parallelization and code transfer from host to device. See http://stackoverflow.com/questions/7907510/opengl-vs-opencl-which-to-choose-and-why for further details. OpenGL needs a change of structure that respects the OpenGL framework: shaders, textures etc.

The version used is 1.1 since this is hardware supported. At this moment 1.1 and 1.2 are the most used versions of OpenCL. Version 2.0 is barely used as is the newest. To see differences between 1.1 and 1.2 see http://streamcomputing.eu/blog/2011-11-19/difference-between-openc1-1-2-and-1-1/ . To see the 1.1 specification see https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf .
There are two profiles possible in general: Full Profile and Embedded Profile. The board implements the Embedded Profile. For differences between the two, see https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, chapter 10, page 354.

Implementation: There is a pattern for the host code that accesses the GPU and programs it. For details, see presentation of project:

- Get Platforms available - get available platforms -  clGetPlatformIDs(platformCount, platforms, NULL);
- Get Devices of platforms available clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, 1, &device_id, &deviceCount);
- Create context on devices context=clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
- Create command queues in context and device command_queue = clCreateCommandQueue(&context, &device_id, NULL, &ret);
- Create program from source - see AutoROI code in gputest.c-®GPU_Init for example clCreateProgramWithSource(&context, 1, (const char **)source_str, (const size_t *)&source_size, &ret);
- Build program clBuildProgram(program, 1, &device_id, options, NULL, NULL);
- Create kernel from built program kernel = clCreateKernel(program, "kernel", &ret);
- Create buffers in GPU memory clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(char), NULL, &ret);
- Copy objects in the allocated buffers on the GPU memory for input and output
- Run kernels on those objects
- Copy back output results in host memory
- Analyze results

To test an application, several possibilities are available:

a. Run the application directly on the board, write results in a buffer; then copy the buffer back in the CPU memory and verify;

b. Use a combination of an off-line compiler with a board emulator on the host; basically; it’s the same like a) but without using a hardware board;

c. Use other OpenCL devices that support the same API version+profile but keep in mind the hardware difference in terms of memory available and core frequency etc.

For the beginning until one is accustomed with OpenCL programming, it is not advisable to deal with b) and c) yet.
APPENDIX C. GPU PROGRAMMING

Version b) is interesting. It actually gives two options: 1. use only the OpenCL offline compiler in the GPU kernel development and then try them directly on the board; or; use also the emulator; in this case being the need to port the application from Linux (on the board) to the PC where the emulator runs. This second option is similar to c), only that through the emulator provides a constraint for the specifications of the graphical card that is present on the Host/PC where one wants to develop OpenCL kernels.

![Figure C.1: Development paths](image)

Here there are some issues that were found out while writing the kernels that need to run on the GPU.

- the instruction size is limited, room for only 512 GPU assembly instructions hence the kernel cannot be too long. In terms of errors one may get the "AXI Bus error!!" in `dmesg` output, from where a board reset is needed

- there is no double support on this GPU hence you either need to change all the code in float or to do type casting before transferring data to GPU for execution and after transferring data back to read on the CPU side.

- there are quite strict rules with array declaring inside the kernels; proceed to read [https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf](https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf) and find the rules for the Embedded Profile when it comes to declaring arrays inside kernels, like: float *h; or float h[20];
• the exact number of threads that run in parallel when the program executes is determined by the size of the kernel and the two variables global\_work\_size and local\_work\_size.

• always when doing operations that involve integers and floats together, don’t forget casting: float $a,b=20; a=b/20$ is not correct; $a=b/20.0$ is correct;

• having an array inside the kernel that can change its size depending on a parameter;