university of groningen / faculty of science and engineering

UNIVERISTY OF GRONINGEN

MASTER'S THESIS

A parallel approach for the *Shape, Illumination and Reflectance from Shading* algorithm

A thesis submitted in fulfillment of the requirements for the degree of MSc. Computing Science

in the

Software Engineering and Distributed Systems
Faculty of Science and Engineering

Author: Ștefan-Cosmin CREȚU

Supervisor: Prof. Dr. Alexandru C. TELEA

July 24, 2018

Declaration of Authorship

I, Ștefan-Cosmin CREȚU, declare that this thesis titled, "A parallel approach for the *Shape, Illumination and Reflectance from Shading* algorithm" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UNIVERISTY OF GRONINGEN

Abstract

Faculty of Science and Engineering

MSc. Computing Science

A parallel approach for the *Shape, Illumination and Reflectance from Shading* algorithm

Shape from shading (SfS) is a generic term for a family of algorithms that aims to reconstruct 3D geometric information from one or multiple views (static 2D images) of a 3D scene. A recent method, called Shape, Illumination and Reflectance from Shading (SIRFS) proposes a high-quality solution to the SfS problem. However, SIRFS is computationally expensive, which makes its trial-and-error usage for parameter setting experimentation prohibitive.

In this project, we examine the end-to-end problem of parallelizing SIRFS. For this, we first reverse-engineer the current MATLAB implementation and propose a porting plan to C/C++. Next, we examine the performance bottlenecks of such a C/C++ solution, and propose parallelization solutions to it both on the CPU and GPU. We examine the pro's and con's of these solutions in detail. We conclude by discussing the obtained insights, stressing on the techniques and designs which have proven successful for the acceleration of SIRFS, but also outlining still existing hard bottlenecks that were discovered but not solved.

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Program Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
IDE	Integrated Development Environment
POSIX	Portable Operating System Interface
SFS	Shape from Shading
SIMD	Singe Instruction Multiple Data
SIRFS	Shape, Illumination and Reflectance from Shading
STL	Standard Template Library

Chapter 1

Introduction

Shape from shading is a computationally complex problem that aims at recovering 3D information from 2D images of 3D shapes. Recently, a high-quality algorithm was proposed to this end, called Shape, Illumination, and Reflectance from Shading (SIRFS). However, one major problem of this algorithm is that it is computationally very inefficient, and its MATLAB implementation cannot be easily optimized. Separately, replicating the algorithm's functionality directly from its accompanying documentation is not easily possible, given its complexity and the scarcity of this documentation.

In this thesis it is addressed the problem of reverse-engineering the MATLAB SIRFS implementation, and then it is proposed a software refactoring and parallelization strategy aiming at completely replacing the original algorithm with a functionally identical one which has high computational performance.

That said, in this chapter is described the SIRFS algorithm, followed by the statement of the identified problems of its current MATLAB implementation. Therewith, it are emphasized the motivations for reverse-engineering this implementation, for its code refactoring and parallelization.

1.1 SIRFS

Shape from shading is a generic term for a family of algorithms that aims to reconstruct 3D geometric information from static 2D images. Such algorithms are useful for a wide spectrum of applications, such as inverse modeling, computer games, 3D shape reconstruction, and 3D shape recognition.

Shape from shading is a complex inverse problem: if we model the computer graphics pipeline, that consists of a viewpoint, projection and lighting parameters, as a function f, which when applied on a 3D object x produces a 2D rendered image y, then shape from shading essentially aims to find the 3D object x given the image y.

The function f is illustrated in the figure below together with its arguments: the shape of the 3D object (s), the viewpoint on it (v), the illumination/lightning parameters involved in the 3D scene (i) - which are light's color, intensity, position, and orientation - , and the reflectance of the 3D object's material (r). The result of this function is a 2D rendered static image, which can represent an input of the *Shape from shading* algorithm. By only using the

information in the input 2D image, the algorithm aims at recovering the parameters related to the object's shape (s), but not the other ones.

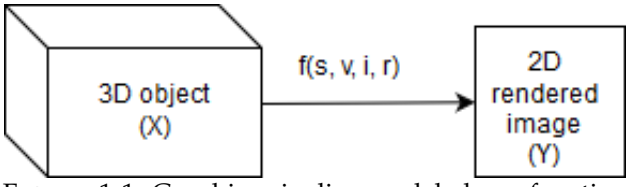

FIGURE 1.1: Graphics pipeline modeled as a function

As mentioned above, the function f does not depend only on the geometry of the object y, but also on viewing parameters, illumination parameters, and reflectance parameters. Thus, a more complex and recent method aims to solve the Shape from Shading problem by incorporating viewpoint, illumination, and reflectance into the model of the viewing function f. This method is called SIRFS and is briefly introduced in this section.

The SIRFS algorithm is currently one of the most advanced techniques in the area. It is able to treat a wide set of images y in near-automatic fashion, and achieves plausible reconstructions of the underlying shape, illumination and reflectance factors. In other terms, the algorithm estimates the original values for shape, light parameters (light direction, light strength and light color), reflectance (material diffusivity in each reconstructed point of the 3D shape) and viewing parameters (viewpoint, viewing direction, perspective projection parameters) using the information contained in a 2D image. [1]

1.2 Problem statement

However effective, SIRFS is not very efficient, as its current implementation aims to solve a complex, high-dimensional, optimization problem in finding the inverse of the function f outlined above. This makes the method impractical, as one has to often fine-tune various parameters, run the method, assess the result, and repeat the cycle until a good result is obtained. Hence, parallelization of SIRFS is an interesting goal.

A recent research internship [2] has covered the first step of the above goal. In this work, the current SIRFS MATLAB implementation was reverse-engineered and a porting strategy to C/C++ was devised. The strategy was validated by porting a subset of the SIRFS implementation and verifying that the ported code produces the same results as the original MATLAB implementation. The main outcomes of this research are introduced in Chapter 2.

The goal of this project is to detect the most critical time-consuming components of SIRFS, select from these the ones which lend themselves to parallelization, and parallelize these using NVIDIA's CUDA platform. Furthermore, several optimization and refactoring procedures with respect to the translated code are also introduced. These methods are used with the scope to enhance the translated code's efficiency where parallelization cannot help.

Overall, the aim of the entire project is to answer the question: *How can we achieve a significant performance increase for SIRFS by using GPU parallelization?*. Sure, this is a general question

and a short and comprehensive answer to it is hard to be given. Next to that, it is obvious that the general note of the above question denotes a rather broad scope of this thesis. However, the purpose of this research can be fined-tuned and objectified by providing answers to the following more concrete questions, as follows:

- Where are the major computational bottlenecks of the current C/C++ SIRFS implementation?
- How to migrate from the current implementation of SIRFS to one which is amenable to GPU parallelization?
- Which are the suitable migration patterns to support the GPU parallelization?
- Which is the expected performance gain?

Having said that, the end goal of this work - that is obtaining a highly efficient SIRFS implementation by using parallelization - may sound, at first sight, trivial. However, doing this involves a set of interrelating complexities and challenges, which in turn relate to several areas of computer science, that are outlined below.

First, one could argue that the end goal could be reached by directly re-implementing SIRFS in a parallelized version, using a suitably-chosen programming language. Doing this is however far from trivial: The current documentation of SIRFS [1] mainly focuses on the scientific novelty and proposed (continuous) numerical models that underlie the optimization problem that captures Shape from Shading. No detailed information is given on how various sub-steps of the entire pipeline are tackled. Moreover, a cursory study of the available MATLAB implementation shows that there are numerous design decisions that are important and can affect the outcome of the algorithm in various ways, including both computational efficiency but also accuracy and even convergence. As such, one has to start working on our end goal considering the current MATLAB implementation.

However, as outlined above, this implementation is not documented in detail. Moreover, the entire computational pipeline it covers is quite complex. As such, a first goal we have to cover is to *reverse engineer* the functionality of this implementation. This in turn asks for using suitable *program comprehension* techniques to analyze the available MATLAB code. While several program comprehension tools exist and are frequently used in software maintenance, few of them are directly able to handle MATLAB code. As such, a specific plan and custom tool sets and techniques have to be devised to accomplish this step.

Furthermore, directly parallelizing MATLAB code is not a viable option. Although MATLAB provides computational primitives which are internally accelerated, not *all* of the time-consuming operations in SIRFS are of this kind. As such, and to achieve maximal freedom for parallelization, the MATLAB implementation of SIRFS has to be ported to a different programming language. In this case it was opted for C/C++. Designing a *porting plan* which allows incremental porting, testing, and validation is a challenge that follows from these constraints.

Last but not least, the gains obtained by parallelizing SIRFS have to be kept in balance with the effort required for analyzing, porting, and testing the code. Here, the key question is how to identify the most parallelization-effective parts of the algorithm, i.e., the parts that (a) can be easily parallelized and (b) deliver the highest computational gains. In turn, sub-question (a) required understanding the inter-dependencies of the SIRFS code components, which is a task for program comprehension.

Summarizing, the technical main challenges for the work in this thesis regard a mix of activities involving program comprehension, reverse engineering, dependency engineering, code refactoring and optimization, parallelization, and algorithmic validation. No such single challenge is overly complex in itself. However, their combination makes the project a non-trivial example of software engineering and program comprehension.

That said, a step-by-step approach was used to achieve the goals highlighted by the questions listed above, organization reflected in this document's structure, that is as follows: Chapter 2 briefly introduces the results of the research project which captures the reverse-engineering and code porting of SIRFS, as they represent the cornerstone of this project. In Chapter 3 and Chapter 4 are introduced the optimization and code refactoring procedures that do not involve parallelization. Following, the parallelization strategy is discussed in Chapter 5, covering theoretical aspects regarding CPU multi-threading which are used in several code implementation experiments whose results are highlighted at the end of the chapter. Thereafter, the GPU parallelization aspects, with focus on CUDA API, and their usage within several implementation experiments with respect to the SIRFS C/C++ code are presented in Chapter 6. Then, the conclusions on the entire project are presented in Chapter 7, including the future developments on the analysis and procedures discussed in this document.

Except for the Chapter 2, which presents the main findings and concepts of the research project [2], the next chapters broadly have a similar structure, as each one starts with the presentation of several theoretical aspects that were considered for the methods in discussion, together with the integration method for 3rd party libraries, if the case, followed by a description of the implementation and ending with the discussion of the validating results and the impact on the C/C++ code's runtime efficiency.

On top of that, throughout the document, the term efficiency is widely used and is defined as follows: an increasing of the execution speed of the C/C++ SIRFS code. This execution speed increasing implicitly means a drop in the execution time, so these definitions can be seen as equivalent. Likewise, the term performance is used to define the same concepts. Next to that, it has to be mentioned that the efficiency assessments were done by measuring the execution times of certain code sections. It is widely known that these times differ from machine to machine, as they have different hardware, thus different behavior. Even on the same machine, the execution times are influenced by the machines current state, that is given by its workload at a given moment. In order to define this context, Appendix A illustrates the details used for the performance assessments.

Chapter 2

Background

As mentioned in the previous chapter, the analysis presented in this document thesis is based on the methods and results introduced in the research project [2]. Having said that, these methods and results are shortly described in this chapter, as they are referred to further in this document.

In detail, the initial work presented in [2] focuses on reverse engineering the static (structure and control-flow) architecture of SIRFS, and proposes step-by-step porting of the main SIRFS concepts (data structures and functional components) to C++. In order to achieve this, the architecture was based on two key-drivers, as follows: **correctness**, defined as the ability of the C++ translated code to give the same output as its MATLAB corresponding code block, and **modularity**, which represents the encapsulation into methods and functions of every behavior implemented by the MATLAB code, in order to allow for an incremental porting, testing and debugging.

In the following, it is briefly explained how the above-defined key drivers drove the architecture of the project described in [2], namely during the reverse-engineering and code translation processes, with an emphasis on the main outcomes of those procedures.

2.1 Preliminary analysis

The analysis presented in [2] is composed of several steps, which are briefly presented in this section.

First, an **overview** of the Shape from Shading family of algorithms is provided, with an emphasis on SIRFS, discussion that is summarized in 1. Then, it is presented the code structure of the SIRFS MATLAB implementation, which is needed in the reverse-engineering process. This process is performed with the scope of finding a suitable code translation strategy, which is in turn applied during the code translation process.

The **reverse engineering** of the SIRFS MATLAB implementation particularly entailed the identification and illustration of the algorithm's call graph, which contains the functions called for processing gray scaled input images. The importance of obtaining the algorithm's call graph comes from the fact that it acts as a cornerstone in devising the code translation strategy. Furthermore, not only the call graph is illustrated and discussed, but the methodology used for discovering the control flow is also presented. Specifically, the call graph presents the functions that are related by a calling relationship. In this relation, a function

is referred to as *caller* if it calls the other function involved in the same relation, which is named *callee*.

The first results of the reverse engineering were included in the Appendix A1 of the document [2], in a tabular form that distinguishes amongst the callees, by dividing them into two categories: SIRFS functions and MATLAB functions. For the MATLAB callees it was specified the list of arguments, information is needed as, in some cases, it influences the way the considered function executes. Next, in order to emphasize the calling sequence, to each function is assigned an ID, also included in the table.

ID	File + function name	Caller	Callees	MATLAB callees
1	SIRFS.m SIRFS	the user	• medianFilterMat_mask • getBorderNormals • doSolve It includes the following files: • PARAMETERS.m • CONSTANTS.m	• eval(expr) • load(filename) • disp • repmat(ṽalid, [1,1,size(im,3)]) • im(repmat(ṽalid, [1,1,size(im,3)])) • log(im)

TABLE 2.1: Snapshot of the table that presents the call graph of SIRFS MATLAB implementation

Ordering of call graph functions: As the first results of the methodology did not present the functions in a total ordered manner, the reverse engineering process outcomes are further refined into a total ordering of the call graph functions. By this, it is meant that the control flow graph of the program is visited so that it produces an ordering of its nodes (functions) so that these can be ported, in this order, with minimal effort, and also allowing at any moment a fully functional program to exist, for testing purposes.

That said, the total ordering was of high importance as it dictates the order of translation and, thereby, it fully shapes the code porting strategy. Nonetheless, the reason for having an unordered initial set of functions lies in the fact that there is no direct relation between any pair of functions. For example, two functions can have the same caller, but they do not call each other, so there it cannot be assessed which one should be translated first, unless the above-mentioned methodology is used. The discussion covers this issue and proposes a strategy that makes a slightly deeper analysis of the call graph which has as result the total ordered set of functions.

Thereafter, an **analysis of functions' signatures** in the obtained call graph was made with the scope to identify the main data types used in the MATLAB code, which also added to

the development of the translation strategy. Specifically, since MATLAB is not an object-oriented language, determining the key data types that are used by the various steps (functions) of the SIRFS pipeline has to be done by seeing which are the data types that these functions use to communicate with each other. Equivalents for these data types were created during the porting, in the form of C structures or C++ classes.

Furthermore, the analysis also illustrates, at implementation level, the way any two functions in the call graph interact, which allowed for a high level understanding of the functions' behavior. The results of this analysis were included in the Appendix A2 of the document [2], in a tabular form. The table includes the full function's signature, describes the inputs and the outputs, and also provides a short description of its behavior. A snapshot of this table is shown below in order to illustrate the previously-mentioned results, however, the full descriptions are not provided - hence the (..) symbol - as they can be found in [2].

ID	File + function	Description
1	SIRFS.m output = SIRFS(input_image, input_mask, input_height, eval_string)	Inputs: • input_image = 2D double matrix (..) • input_mask = a 2D Boolean matrix mask (..) • input_height = empty array (..) • eval_string = empty string (..) Output: • a data structure, representing the result of function do_Solve Description: This function represents the entry point in the code. (..)

TABLE 2.2: Snapshot of the table presenting SIRFS function's signatures

Code translation strategy: Once the total ordered call graph was identified and the relationships between the functions were analyzed and understood, a code translation strategy was proposed. Broadly, it involves the division of the code into four major blocks, named A, B, C and D, which follows a separation of concerns inquired from the previously made analysis. Since these blocks still proved to have a quite considerable size, in terms of the amount of code contained within, they were further divided in sub-blocks, leading to a more fine grained structuring.

The result of the above-mentioned procedure was illustrated in Table 3.4 in [2], which presents a matching between each block with the corresponding sub-blocks and functions. In this table, the functions are referred to using the IDs given in Table 3.1, presented in Appendix A1 of [2]. Afterwards, the call graph illustration was redone to match this new structure. Since in this document it is often referred to a given block or sub blocks, the diagram is illustrated in figure 2.1, for an improved readability. The figure shows all sub blocks that form the SIRFS MATLAB implementation. The letters represent the name of the block, whereas the number near the letter represents the ID of that sub block within the considered block.

The alphabetical and numbers' order provide the order for the code translation of each sub block.

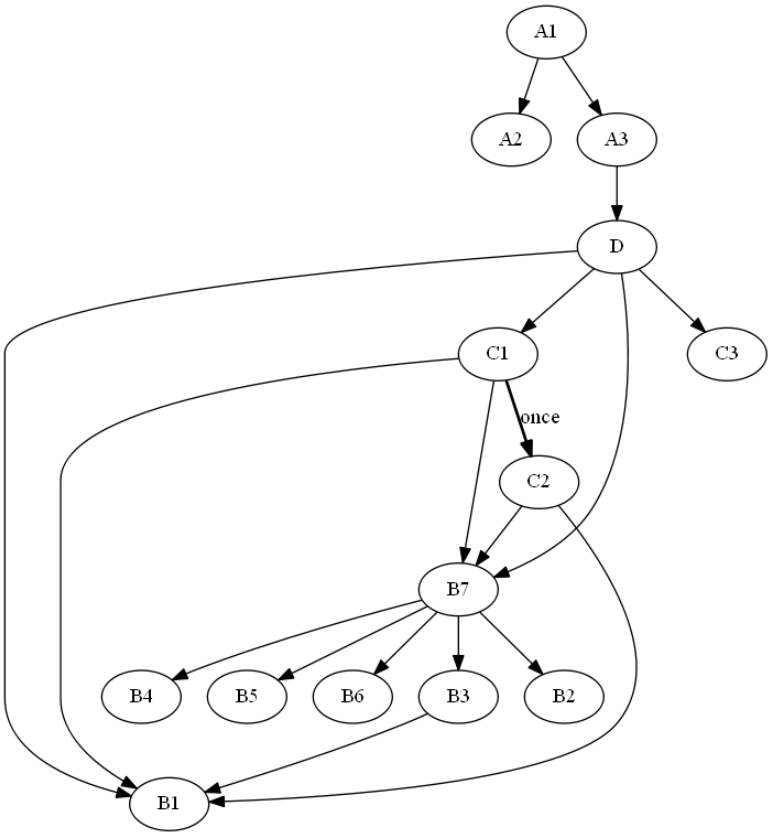

FIGURE 2.1: Code sub-blocks control flow

Afterwards, the **MATLAB project's file structure analysis** was done, with the structure being shown and explained. It is also illustrated in the figure 2.2, as it is referred to in this document, as well. This way, there could be found the relations between the functional blocks subject to translation, illustrated in figure 2.1, and their physical counterparts, that are files and folders. Subsequently, this helps in understanding if the per-block porting effort is localized to certain parts of the code base or spread over its entirety. In turn, this helps in assessing the porting effort (in program comprehension, understanding code spread over an entire code base is well-known to be more costly than understanding the same amount of code which is located in a well-defined subset thereof, e.g., a folder).

Once all the above steps were completed, it was proceeded to code translation. Briefly, the implementation contains the translations of sub blocks A1 and A2, with the corresponding files being included in a separate folder, which is named accordingly. Next to that, the *templates* folder contains the implementation of template classes corresponding to 2D and 3D matrices, which are the data types often used by the SIRFS algorithm. The *helpers* folder contains declarations of classes, enumerations and functions that are used as support for the code porting process, making it easier and better organized. They do not have any correspondent in the MATLAB code. Next, the folder *matlab* contains functions that implement several vector-based operations which are not defined in the STL and functions the

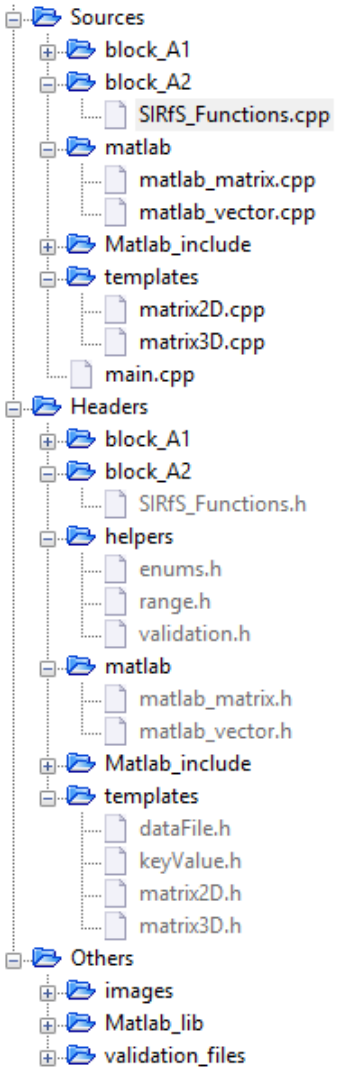

FIGURE 2.2: Project file structure

implement typical operations for a given type of matrix, such as conversion from matrix of integers into a matrix of doubles.

All these folders are referred to in the forthcoming chapters as new features and enhancements are added to the code they encapsulate. The folders which appear in 2.2 but are not yet described here were subjects of deletion, due to adding of new features and enhancements. However, more information about this can be found in the next chapters.

Data types mapping: Once the structure of the ported code was depicted, the research project discusses the matching between the mainly used data types, which is quite straightforward. Also, it describes the method used for reading the input image. Thereafter, the implementation details of sub blocks A1 and A2 are presented.

An important aspect discussed in [2] is the **porting validation**. The research report proposes a thorough validation method, which involves outputting to text files the values computed by the C++ code, then load them in the MATLAB space for comparing them to the values computed by the SIRFS MATLAB implementation. An alternative is also suggested, which

is actually an inverse procedure of the previously described one and that is used by this project as well. That said, in the file *validation.h* are defined some test functions, one for each major used data set: 2d matrix, 3D matrix and STL vector. Each such function takes as a parameter a file which should contain the values computed by the SIRFS MATLAB code and then compares to the values held in the corresponding data structure from the C++ space.

Performance assessment: Finally, the research report describes how the above-mentioned validation methods were applied to the translated sub blocks, in correlation to their implementation details. Finally, the project concludes with a performance assessment, which is summarized in a table. Since in this document is aimed at enhancing the values illustrated in the mentioned table, it is illustrated below, as it is often referred to it. It shows the execution times per code section. Here, a code section is a finer-grained decomposition of a code-block, which can mean a 1-to-1 matching with a function, in some cases.

Code section	Execution time in C++ (seconds)	Execution time in MATLAB (seconds)
Parameters initialization	0.001	0.009
Load priors	12.657	0.224
Initialize "data" class members	0.021	0.002
Apply median filter	1.035	0.323
Build border normals	0.624	0.052

TABLE 2.3: Execution times per code section of the translated code

2.2 Execution strategy

In the previous section, the theoretical and the technical aspects discussed in the document [2] are briefly described, as they represent the cornerstone of this thesis. Nevertheless, in order to have a clearer definition of the scope of the analysis presented in this document, several considerations were made during the development of this project, which are introduced in this section.

First, it is worthy to mention that the development of the project [2] lasted for approximately nine months, whereas the progress of this thesis is expected to last for seven months. These firstly made considerations call up for some decisions regarding the development strategy of this thesis, decisions which aim to optimally downsize the activities that need to be done with respect to it, in order to properly set its boundaries.

In order to do as such, two development strategies have been identified, based on the progress of the project [2]. The first one represents a **breadth-first** development, which entails the continuation of the code porting of the SIRFS algorithm, from MATLAB to C++, with the scope to finish it. Thus, given the time constraints specified above, it was concluded that it would be almost impossible to finish the full translation and to still have a

reasonable amount of time left to dedicate for the GPGPU parallelization implementation and documentation.

On top of this, this breadth-first strategy would come with further disadvantages: by focusing on the translation, several C++ coding efficiency-related aspects and approaches would be omitted, as the scope would be to have a fully ported C++ code that is correct, but not necessarily efficient. Consequently, this approach would hinder the performance of the GPU-parallelized C++ SIRFS code, with the final execution times being far from expectations.

Concretely, despite the fact that the MATLAB code was split into blocks and sub-blocks that would allow a step-by-step translation and porting validation [2], by focusing on a full translation would not allow for finding solutions to all bottlenecks identified during the intermediate testing and validation step. Here, these bottlenecks might be proper to a certain block, or sub-block, and would call for a solution that cannot be reusable within the other blocks. Clearly this would make the efficiency-improvement research and development more complicated in the end, as it would broaden the range of possible solutions to be addressed. Thus, their implementation and testing would be almost impossible to do, given the above-mentioned time constraints.

On the other hand, an advantage of the breadth-first strategy would be that the most time consuming and meticulous step would be done. Hence, it would allow for a complete identification of the bottlenecks and for the development of a parallelization strategy that could involve bigger blocks of code, as one could fine tune the pieces of code on which parallelization would be applied, as pleases. All in all, it was clear that this approach has a rather quantitative bias and would not allow for a proper analysis of the efficiency-hampering aspects of the C++ SIRFS code.

Therefore, the adopted strategy is a **depth-first** one. It involves working with the translated code described in [2] and studying several aspects of its implementation, with the scope to identify all possible bottlenecks that hamper the code's execution efficiency. Then, it presents and tests different solutions with the aim to analyze their impact on the C++ code and to assess whether they can solve the previously identified issues, or at least to bring certain improvements with respect to them. Broadly, these solutions entail the using of some 3rd party libraries, of some good practice coding methods, of some tools provided by certain C++ standards and libraries, such as smart pointers and STL vectors, and, finally, of the CPU and GPU parallelization approaches.

Moreover, the depth-first strategy comes with several advantages. Firstly, it allows for focusing on efficiency-related issues of the C++ SIRFS code withing the given time boundaries. In addition, unlike the breadth-first strategy, it allows for researching on methods and approaches with the scope to address the runtime efficiency-issues particular to a given code block or sub-block. Subsequently, if during the next porting steps those bottlenecks will prove to be more general, a solution to them would already be implemented.

Furthermore, by studying several approaches that could help in achieving better execution times, or in having a better code structure while not hindering the execution performance,

a complete porting and efficiency-improvement strategy is devised. This strategy can be applied, afterwards, to the rest of the SIRFS code, that has not been translated yet. By having this clear picture of what it has to be done, then it is just a matter of time for finalizing the full porting, which would not be just a raw code translation, but one with focus on the previously-mentioned efficiency-related approaches.

Surely, the depth-first brings along some drawbacks. Obviously, its main downside is that it does not come with advantages of the breadth-first strategy, discussed above. Furthermore, it will be difficult to estimate to which extent the final results of this strategy, including the conclusions on them, can be generalized and applied for the entire SIRFS code, or for a similar algorithm, as they clearly have an implementation bias being highly dependent on the code. Nevertheless, these outcomes can still be considered as starting points for the translation and the optimization of the remainder of the SIRFS code, or for a similar algorithm, with the performance assessments done within this project acting as indicators for the performance of similar approaches implemented in the previously-mentioned different contexts.

Chapter 3

Incremental code optimization strategy

As mentioned in chapter 2, the translated code described in [2] was not meant to be optimal from all points of view, but it was aimed for it to work correctly, by providing same results as the MATLAB corresponding code blocks. Thus, a proper and correct translation which also focused on solving the execution efficiency problems, at the same time, was impossible to do. Thereby, the two tasks, namely the porting and the code's optimization, had to be done in a sequential way.

This sequence of operations reflects a well-known principle in software architecture and maintenance, namely *separation of concerns*: By separating porting from optimization, there can be used specific instruments, and handled fewer constraints, during the two steps. Specifically, during porting, syntactic and semantic mapping of MATLAB concepts to C++ ones, as well as enforcing the correctness of the translated program, are covered, with no concerns for computational efficiency. Separately, during optimization, starting from a single C++-only code base to work on, which is known to perform correctly, the activities can be focused purely on optimization and parallelization.

That said, the sequence is depicted below. It can be seen as a pipeline with two filters, with the first one receiving as input the raw MATLAB code and outputting a rather non-efficient C++ code which, in turn, becomes the input of the second filter that optimizes it. The output of this pipeline is a C++ piece of code that is more optimal than the input one and which is expected to perform better than the input MATLAB one.

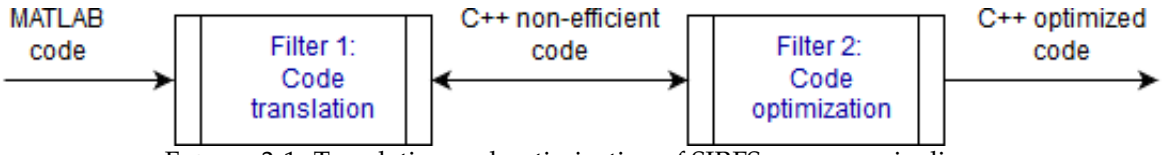

FIGURE 3.1: Translation and optimization of SIRFS seen as a pipeline

That said, the remainder of this document focuses only on describing the second filter, called *Code optimization*, as it researches for methods and approaches to define its implementation.

3.1 Bottlenecks discussion

Nevertheless, in order to define the implementations of the optimization filter, besides taking the C++ non-efficient code as input, there were also considered the bottlenecks implied in the table 2.3.

As mentioned in [2], the SIRFS algorithm receives as input a PNG image (which is referred to as Y image in chapter 1) whose reading process uses the MATLAB engine. Though its execution times are not considered throughout the analysis, it actually introduces time delays in execution, so it was considered as a bottleneck. Clearly, this bottleneck is typical only to this code section and the solution to it cannot be reused for the remaining blocks of code.

Next to that, by analyzing the figures in the table 2.3, it can be easily seen that the *Load priors* code section entails the most major bottleneck. Likewise in the previous case, the priors are loaded only in this code block, so the solution will not be reused for the other blocks of code, as well.

However, it can be noticed that the execution times of the last three code sections are still higher than those of their MATLAB correspondents. Since these code sections involve mostly matrix-based operations, it becomes obvious that the main bottlenecks come from those, namely from the implementation of the matrix template classes. On top of that, other types of bottlenecks lie in the increased modularity of the C++ code.

Nonetheless, as mentioned in the beginning of this document thesis, a GPU parallelization of the code is envisaged as main efficiency operation for the C++ SIRFS code, but having identified the different types of bottlenecks mentioned above, it becomes clear that the GPU optimization approach would not suffice. On top of that, it is obvious it cannot be applied for solving the issues with the first two bottlenecks. In addition, there were noticed parts of the code sections mentioned in the table 2.3 that are not amenable to GPU parallelization, but to a CPU one. In other words, a solution exclusively based on the GPU parallelization would not solve all identified execution efficiency problems.

Therefore, given these different types of bottlenecks, the optimization of the C++ SIRFS code requires a more complex solution, which has to be implemented in a step-by-step manner. These multiple steps entail an incremental optimization of the input of *Filter 2* illustrated in figure 3.1 above. Thus, this filter can be further decomposed into a series of filters, that form an internal pipeline, as illustrated in the figure 3.2 below.

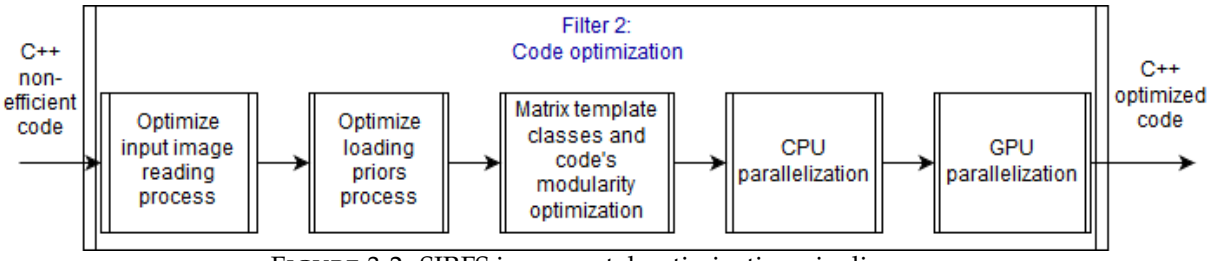

FIGURE 3.2: SIRFS incremental optimization pipeline

Here, the first two filters aim to address the efficiency issues related to the first two bottlenecks outlined above, whereas the remaining three have a broader scope and entailing solutions that can be reusable for the SIRFS code blocks which have not been yet translated. Each filter in this pipeline outputs a code that is more optimal than the code it gets as input.

On the other hand, as it is mentioned in the document [2], the translation of the SIRFS MATLAB code was done using a Windows platform, because of the need to permanently check the MATLAB code implementation and execution, which would not be possible on Linux Octave, due to compatibility issues. That said, as it can be seen in the figure 3.2 above, the optimization of the C++ code will involve a CPU parallelization. If implemented entirely on Windows, the would make use of adapted Linux solutions for Windows, concretely of APIs extending the UNIX *pthread* library, such as *pthreads Win32* [3]. These API calls are expected to be slower than the original Linux-based ones, which is not in line with the scope of this project. Thereby, a hybrid development platform for the optimization solution is envisaged, in order to mitigate these issues.

Specifically, on Windows OS there is support for multithreading, but it requires Win32 API calls if C runtime libraries are used [4], as well as using Visual Studio for development [5], which also introduces some overheads. Therefore, the development of the solutions for the CPU and GPU parallelization will be done using a Ubuntu platform, solution which entails the porting of the C++ SIRFS code from Windows to Ubuntu, including the required refactoring procedures.

That said, in this chapter are outlined two major refactoring procedures that were applied to the C++ SIRFS code with respect to the first two bottlenecks mentioned above. Firstly, the *Optimize input image reading process* filter is described by introducing a new method for reading the input image, which makes use of a third party library, called *libpng*. Next, is presented the solution for the *Optimize loading priors process* filter, that is a more optimal solution for loading the priors, which also makes use of a third party library, called *matio*.

Since the last three filters provide solutions reusable for the untranslated SIRFS code blocks, each one is analyzed and described in a separate chapter, following this one, preserving the order in the figure 3.2. Concretely, chapter 4 discusses the implementation for the *Matrix template classes and code's modularity optimization*, then in chapter 5 is aimed at finding a suitable CPU parallelization approach that optimizes the SIRFS total execution time. Lastly, in chapter 6 discusses methods to parallelize the SIRFS algorithm using the GPU.

3.2 Input image reading

The translated code presented in [2] uses a methodology for reading the input image that involves calling the MATLAB engine from the C++ native code. As the table 2.3 illustrates, this procedure proved to be the most time consuming and, therefore, it is calling for an alternative.

A solution to this lies in using *libpng* for reading the input PNG test images. Since all the test images used by the SIRFS MATLAB implementation have PNG format, this library seems to be the right choice. Thus a new procedure for reading the input image was built atop of it, leading to a total decoupling between the C++ and the MATLAB implementations of SIRFS algorithm, which is highly desirable, as it would add to the C++ code's efficiency and portability.

This procedure is described in this section, starting with a discussion on the integration of the *libpng* library with the existing code, continuing with some theoretical aspects and the implementation details, and ending with a discussion on the new procedure's validation and impact on the performance of the translated code.

The discussion of all the aforementioned issues is done in great detail, thereby introducing concerns regarding low-level implementation issues, C/C++ language specific issues, and portability issues. At first sight, outlining these details may seem to be uninteresting for the overall end goal we have. However, we argue differently: Understanding (and solving) such issues is crucial for the end-to-end success of the porting and optimization process. Moreover, such issues are relevant, and appear, also during the porting and optimization of other components of the SIRFS pipeline. As such, it is presented here in detail for the *libpng* component as an example of the type of work that needs to be done for completing the end-to-end goal of this project.

3.2.1 Integrate *libpng* with the IDE

In order to work with *libpng*, it is needed the installation of *libpng* and *zlib*, as the former includes and uses the latter. For Windows platforms, the setups for both libraries are available at [6] and [7], respectively, under "Complete package, except sources".

Once these libraries are installed, they need to be integrated with the IDE. That said, for Windows platforms, in case of using the *MinGW* compiler, set the linker option such as it links to *libpng.dll.a*, as it cannot directly link to *libpng.dll* file. Furthermore, set compiler option to *-m32* (32 bit target) as the library is compatible only with 32-bit code. Thereafter, include the path of *libpng/include* directory in the *Search directory* paths for the compiler. Similarly, include the paths for *libpng/lib* and *libpng/bin* folders to the linker's *Search directory*.

The final integration step is to include the only needed header, namely png.h, in a *extern "C"* block. The header defines the *libpng* data types used by *libpng*, as well as some useful macros, but also declares all functions' prototypes needed for reading a PNG file. Including the header file within a *extern "C"* block has to be done as the translated code is compiled with -c++11 option, whereas the *libpng* library is implemented in C. With respect to this, more details are provided in the next section of this chapter, which discusses the theoretical aspects behind *extern "C"* and *extern* keywords. On the other hand, *zlib.h* does not need to be included as it is not required directly, it is included by png.h.

Another discussion can be derived from the fact that one has to link against *.dll.a* file. That said, though a *GCC*-based compiler, such as *MinGW*, can link directly to a *.dll* file, linking

to a *.dll.a* file instead of .dll "aids its job", according to *CygWin*, the developers of *MinGW* compiler [8].

3.2.2 *Extern "C"* and *extern* discussion

The code included in the *extern "C"* block has C linkage and not C++ one, like the rest of the (C++) code. This is useful as the library functions' names do not need to get mangled by the compiler. The C++ compiler mangles those by adding to their names information about the parameters, in order to support function overloading (same name, different parameters) and, thus, differentiating between them. This does not happen in C, as C compilers do not mangle functions names and, thereby, any C implemented function included in a C++ compiled code should not get its signature mangled.

In other words, if the headers would not be included in the extern "C" block, the C++ compiler would change the functions' names by adding information about their parameters (the parameters types, for example) causing linking errors (*undefined reference to...*) as for the function with the mangled name is not found the suitable implementation (which does not have a mangled name) to link with.

According to C++03 standard, each C++ compiler should provide C linkage (extern "C") option. This option occurs only in the namespace and is a linkage specification. It is omitted for class members as overloading there is allowed, therefore the mangling is allowed. If 2 functions are identical but have different language linking (C vs C++) then they are different. Also, the linkage from other languages to C++ objects or from C++ to objects defined in other languages is defined by implementation and depends on the language. The linkage can be done only if the object representations (layouts) of the 2 languages are similar enough.[9, 10]

Extern is a linkage modifier that applies to variables and functions in C, extending their visibility. Declaration of a variable or function means that variable or function exists in the program, but no memory is allocated for it. Definition allocates memory for them. A variable/function can be defined only once, but declared multiple times (within different scopes). In case of functions, the *extern* modifier followed by function's declaration, tells the function is defined somewhere else, and the compiler knows how to link (non-statically) with the definition (from other file or module), as there is only one definition. In case of variables, extern helps in declaring the variable without defining it (no memory allocation), as the variable should be defined in another file/module. Declaring a variable with extern and also initializing it (i.e. extern int data = 0) leads to that variable's definition as well. [11]

3.2.3 Implementation of the input image reading procedure

The input image reading procedure depends on two third party libraries, namely *libpng* and *zlib*. These libraries can be updated with time, for various reasons like bug fixing or adding new features. In their turn, these updates might have a direct impact on the procedure implemented for reading a PNG image for the translated version of SIRFS, presented in this document. The newest versions of the above-mentioned libraries can be found at [6] and

[7] respectively. Thus, if one would want to have the C++ SIRFS code working with the latest versions of these libraries, then he has to follow the steps described in section 3.2.1. Therefore, a function called *libs_version_info()* was implemented to print the currently used versions for *libpng* and *zlib*, aiming at helping the future developers of the C++ SIRFS code to easily come to a decision whether such an update would be useful or not.

The PNG image reading is encapsulated in one function, called *read_png_file()*, that takes no parameters and returns a *Matrix2D* object, that holds double values. The reading procedure converts the input image to grayscale, thereby it holds only one value per pixel. That said, the entire code presented in this document only follows the call graph corresponding to gray images processing, as mentioned in [2]. The previously-named function is declared in the *read_input_image_libpng.h* file and implemented in *read_input_image_libpng.cpp*. These files are included in *blockA1* folder. Next to that, *read_png_file()* calls *libs_version_info()*, as it is the only function in the whole code that uses *libpng*.

The *read_png_file()* function firstly reads the PNG file name, allowing the user to insert the path in the console. Then, it calls *libs_version_info()*. Following up, the PNG file is opened and in case of failure an error message is printed and the programs exits. In case of success, the program continues and verifies the signature of the file by reading the first 8 bytes and storing the result in a *png_byte array*. The array is checked using *png_check_sig* with the scope to determine whether it as a PNG file or not. In case of failure, an error message is shown and the program exits.

Once the file is opened and it is known that it has a valid PNG format, two *libpng* structures that will store all the required data for reading the image are set; they are informally named *read* and *info*. Thereby, by calling the functions *png_create_read_struct* and *png_create_info_struct*, respectively, memory space is allocated for these structures, with each function outputting the address of the allocated space to a pointer. If the first call fails, the program prints an error message and exits. Similarly it happens if the second call fails, but before exiting, it deallocates the pointer that corresponds to read structure (which was successfully allocated before), by calling *png_destroy_read_struct* function. The function which declares the read structure, namely *png_create_read_struct*, takes 4 arguments, with the first one being a *libpng* macro (*PNG_LIBPNG_VER_STRING*) that indicates the library's version, and the remaining ones that can be set to NULL, as they are not needed. In a nutshell, they represent pointers to a user defined structure and to two functions for error and warning handling, respectively [12]. The *png_create_info_struct* function takes as input only the pointer to the read structure.

In a brief, the read structure is used by *libpng* internally to keep track of the PNG image state at any given moment, whereas the info structure indicates the state of the image after all user transformations are performed. Besides these two structures an additional one can be used, but it is not required in this case. Shortly, it stores the PNG chunk data that follows the image data and is informally referred to as *end*. Also, the *png_destroy_read_struct* function

takes as arguments three addresses as it can simultaneously deallocate the pointers to all three structures. [13]

Thereafter, the setting of the above-mentioned structures starts with the specification of a generic code error handling approach. This approach avoids using error codes from each component used by *libpng*. Instead, it calls the *setjmp* function that takes as argument the *jmp* buffer from the *png* structure, which is retrieved by calling *png_jmpbuf* function. If the attempt to set the *jmp* buffer fails, an error message is printed, followed be deallocation of read and info structures and the program exit. The *jmp* buffer is set in such a way that it holds a snapshot of the program, that is the state of the stack and registers. [13]

Afterwards, the pointer to the PNG file is stored in the read structure using the function *png_init_io*, which takes as inputs the pointer to the structure and the file pointer. Next to this, the same structure is set not to read the first 8 bytes of the files, since they have been previously checked for file's signature verification. This is done by calling *png_set_sig_bytes* function, which takes the pointer to the read structures as argument, as well as the number of bytes to be skipped.

Once the file pointer was set at the right position, the PNG chunks are read by calling the *png_read_info function*, which takes as inputs the pointers to read and info structures. Note that the previously named function only reads the chunks that precede the image data. The data retrieved here refers to image's width and height, color type, number of channels and bit depth. This data is then stored in some variables, named accordingly, by calling appropriate functions (such as *png_get_color_type*, for example), each one taking as inputs the pointers to read and info structures.

Thereafter, several transformations are applied, in relation to input image's color type, mostly. That said, if the input image is colored (or paletted), the *png_set_palette_to_rgb* function is called to expand it to RGB [14]. If the input image is gray scaled and its bit depth is less than 8, the *png_set_expand_gray_1_2_4_to_8* function is called to expand it to 8 bits [15]. None of these transformations expands to alpha channel [15]. Thus, regardless the input image is gray scaled or paletted, if the *tRNS* chunk contains any transparency-related information, a full alpha channel is added by calling the *png_set_tRNS_to_alpha* function [15]. Lastly, it is checked if the input image uses a 16-bit sampling per channel, and if so, it is converted to 8-bit sampling, for both gray and colored images, by calling the *png_set_strip_16* function. All previously mentioned functions take as only input the pointer to read structure.

If the input image is gray scaled, it might not be known if it was taken as such or modified using a certain algorithm. Since the gray scaling method used in the MATLAB *SIRfS* implementation computes the unweighted average of the R, G and B components, the C++ code does the same, for consistency matters. Therefore, the last applied transformation envisages only gray images (including those with the transparency channel enabled) and converts them to color image by calling the *png_set_gray_to_rgb* function, which takes as input the

pointer to read structure. Finally, the results of the above-described transformations are updated in the read and info structures, by calling the *png_read_update_info* function, which takes as inputs the pointers to them.

At this point, it is known that the input image is colored, has a 8-bit sampling and no transparency enabled, so the program is ready to load the image data. For this purpose, it firstly dynamically allocates a buffer to hold the image data, using a pointer to *png_bytep* (which is already a pointer to byte data type). The allocation takes into account the number of channels, as for a RGB image the number of column is three times bigger. Thereafter, the image data is loaded into this buffer by calling *png_read_image*, function that takes as inputs the read structure and the buffer. Once the image data is loaded, the *png_read_end* is called to tell *libpng* to skip the remaining chunks that follow the image data stream [15].

Next, the loaded image data is converted to gray scale, with each pixel's value being an unweighted sum of its R, G, B values (each one matters equally). As it is done in MATLAB, the values for R, G, B are also scaled to [0;1] by division to 255. The data is stored in a Matrix2D object that holds data of double type. Here, an own developed approach is preferred to the *libpng* one, which involves calling of *png_set_rgb_to_gray*, as the latter proved to output incorrect results.

The procedure ends with cleaning up the allocated memory for read and info structures. Moreover, it deallocates the buffer used to lead the image data and closes the pointer to PNG file.

The impact of this new feature on the C++ code file structure lies in the fact the *Matlab_include* and *Matlab_lib* folders were deleted as they are not needed anymore since the MATLAB engine is not called at all. A new folder, called *libpng* was added and it contains all files needed to link to and work with *libpng*.

3.2.4 Verification and performance assessments

The result is verified against the data output by MATLAB in a text file, after reading the same image. This verification step can be done optionally, by enabling the *TEST_INPUT_IMAGE* macro defined in *validation.h* file, under the *helpers* folder. If it is required to verify the correctness of the input image reading procedure, it outputs 1 if the results are correct and 0 otherwise, together with a message that informs the input image procedure verification is done. The data computed by the MATLAB code is found in *input_image.txt* file, which was previously used to read the input image by calling the MATLAB engine.

When the verification is performed, the execution time of the entire procedure is around 0.26 seconds, but this is not to be taken into account when the performance enhancements are designed, since it is normal to have an overhead when comparing results. Moreover, the execution times for the MATLAB code do not involve any verification procedures, hence the lack of consideration with respect to the verification parts, in general, when the performance of the code is assessed.

All in all, the verification successfully passed so this method was preserved as an implementation for the first filter in the SIRFS incremental optimization pipeline illustrated in the figure 3.2. Though the execution times of this step are not considered when making performance assessments, it is worthy to mention that they are around 0.005 seconds, without counting the verification part. Thus, the solution proved not to add big overheads and significantly contributed to downsizing the execution times of the whole C++ SIRFS implementation, with a performance gain of 99.98%, when compared to the initial solution.

The next section presents a solution for the second filter in the SIRFS incremental optimization pipeline, as it aims at finding a solution to efficiently read a MAT file consisting of several nested structures that hold matrix and vector data.

3.3 Alternative solution for loading priors

The next enhancement added to the C++ SIRFS implementation is a new and faster method for loading the priors. As it can be seen in table 2.3, the loading priors section proved to be the most time consuming, adding a considerable processing overhead as it is 60 times slower than the corresponding MATLAB procedure.

The data for priors is stored in a *.mat* file, which contains several nested structures. These structures' fields are mostly 2D matrices and vectors whose values need to be loaded. The current solution requires using text files that hold data for each structure field separately. Thereafter, each file is opened and read in the C++ code. Opening each file and reading the values corresponding to 1D, 2D, 3D arrays proved to be a bottleneck, fact which demands for a better solution.

The solution is built atop of *matio* library. For this project the 1.5.11 version was used as it was the most recent at the time of implementation. As it is described in its own documentation, provided at [16], "matio library (libmatio) is the primary interface for creating/opening MAT files, and writing/reading variables". In the following it is explained how to integrate the library with the development environment, the implementation details involving it are provided and its impact on the code performance is discussed. Also, the result of the validation process is described.

3.3.1 Integrate *matio* with the IDE

The *matio* library depends on HDF5 and *zlib* [7], but the latter is already used by *libpng*, thus, its usage here does not represent an issue. However, for the former it is required to create an account and then download the archive from the website specified at [17].

For Windows platforms, the proper paths for HDF5 installation directory should be provided to the Visual Studio project that aims at building the *libmatio.dll*. The archive for this Visual studio project can be downloaded from [18]. Once these steps are done, some compilation errors should be fixed for the HDF5 and *matio* source files. These errors are related

to failure in including certain header files, some of them being native to UNIX based operating systems (such as *unistd.h*) and, therefore, not compatible with Windows platforms. However, for those there can be found some alternatives for Windows platforms, such as equivalent libraries or even own defined headers, when the missing header only contains few declarations of constants, types or macros. Afterwards, the Visual Studio project should be set on *Release* mode. Then, the solution should compile successfully and output the aforementioned dynamic library.

Thereafter, in case of using the *MinGW* compiler, set the linker option such as it links to *libmatio.dll*. Also, keep compiler option to *-m32* (32 bit target) as the project should still be able to link with *libpng*. Thereafter, include the path of *matio/src* folder in the *Search directory* paths for the compiler. Similarly, include the path for *matio/visual_ studio/Release* folder to the linker's *Search directory*. In the previously mentioned paths, *matio* represents the folder where the *matio* library was built and where the Visual Studio project output the dynamic library. Likewise *libpng*, the library should be included inside a *extern "C"* block, as it is built exclusively in C and its functions should not get mangled.

3.3.2 Implementation

First of all, it was reused the C++ model consisting of classes that map the structures in the *.mat* file. The mapping is straightforward as for each vector field, a STL vector was used, whereas for each 2D or 3D matrix a corresponding Matrix2D or Matrix3D object was created. The matrices are implemented as template classes, under the *templates* folder. The model was used in the previous method which opened a text file for each data field of each structure to load the data. Thereby, the new solution still involves loading data into the fields of these classes. The model is implemented under the *blockA1* folder and includes the *prior.h* file and all files under the *prior* folder.

The *matio* library provides interfaces that allows to get certain data about a given structure contained in the MAT file, such as its level in the hierarchy (supposing there are nested structures, which is the case here) and the pointer to the structure from the MAT file, called *matvar_t*, that encapsulates all the data for a MATLAB structure. Since the *prior.mat* file used in this project contains nested structures, these can be seen as an hierarchy of nodes, where each node represents an instance of a structure.

Therefore, it was implemented a class that models the metadata of a MAT file. It can be seen as a wrapper over the *matvar_t* structure, which contains the needed metadata and data. The implementation of this class allows for a better structuring of the procedure, as it groups together several *matio* function calls in two main methods called by *SIRfS*. Moreover, this allows for a better understanding of the MAT files structuring and enables the future developers of *SIRfS* to reuse this code without a deeper understanding on how the *matio* library works.

This implementation can be found in *prior_struct_node.h* and *prior_struct_node.cpp* files. It uses the class *StructNode* which models the hierarchy described above. The class contains

five fields, as follows: the pointer to the MATLAB structure held in the MAT file and an integer that indicates its level in the hierarchy, both representing metadata about the node in cause. The next three fields represent metadata about the node's children and they are: an integer that stores the number of children nodes, a *char* pointer of constant pointers to hold the names of the children nodes and an array of pointers to the children of that node. Next to this, the class provides methods to set and read these fields, alongside the constructors, destructor and some operators overriding.

That said, the first important method is named *getChildrenNodes* and it does not take any inputs. Briefly, its behavior is the following: when a node object calls it, it firstly allocates a STL vector of pointers to *StructNode* which represents the pointers to its children nodes, which is returned by the method. Then, for each such children node, it creates an object and sets the pointer to it by indexing the the array of the pointers of the node in discussion. Also, it sets the children node's index by incrementing the parent's index by one. In other words, the newly created node's metadata is set.

The second method is called *setNodeFieldsData* and it also does not take any inputs and does not return anything. It wraps the *matio* functions that get the number of fields for a given node, the names of these fields and the pointer to their MATLAB structure. It is called by *getChildrenNodes* once each object is created in order to set the fields that contain the metadata regarding its children. That said, when *getChildrenNodes* returns the vector of pointers to *StructNode*, each object has all its fields set.

Thereafter, the C++ class model that maps the MATLAB structure implements, for each class, a method called *initialize*ClassName*Data*. The method gets as input a reference to a *StructNode* object which contains the metadata for the class in cause. Then, the method calls *getChildrenNodes* for the input object and stores the values in an array. For each *StructNode* object in the array is checked to which field it corresponds (by checking the name) and then its values are assigned to the corresponding field in that class, by getting the pointer to its *matvar_t* structure and dereferencing the *data* field. The *StructureNode* defines a wrapper for getting the previously named pointer, called *getStructureP*.

Here, if the children node does not represent an instance of another structure, its data can be read directly, otherwise the procedure continues as the node represents another structure. Since it is known for the beginning how the *prior* structure looks like, the method for loading its parameters was implemented for this particular case, in order to avoid eventual overheads caused by a more general implementation.

A special case is loading the data for the first class, that is *prior*, as the pointer to its MATLAB structure is returned when calling the *Mat_Open* function to open the MAT file. Then, this pointer is passed to its initialization method, which gets data for its children nodes and then follows the above-described logic.

3.3.3 Verification and performance assessments

The procedure is verified against the data output by MATLAB in the text files previously used for loading the data on priors. This verification step can be done optionally, by enabling the *TEST_PRIOR_HEIGHT*, *TEST_PRIOR_LIGHT* and *TEST_PRIOR_REFLECTANCE* macros defined in *validation.h* file. The macros enable the testing for the *priors* structure fields, separately, allowing for a better separation of concerns when testing.

When the verification is executed, it outputs 1 if the results are correct and 0 otherwise, together with a message that informs the input image procedure verification is done. The verification is performed for each data field of the considered structure. All in all, the verification passed successfully.

When the verification step is performed, the execution time of the entire procedure is around 20.5 seconds. However, important here is the execution time without the verification step included, as it is not measured in the MATLAB code, and which is almost 1.5 seconds. This means a 88% performance gain measured at the *Loading priors* code section level. Though it is 6 times faster than the initial implementation, it still proves to be time consuming, as it takes almost half of the execution time of the entire code, which is around 3.1 seconds.

This computation overhead is mostly caused by a native *matio* function call, named *Mat_VarReadNext* that retrieves the pointer to the prior structure. Its execution takes almost 1.47 seconds out of the total 1.5 seconds. Since it is a function call from a 3rd party library, there is not much that can be done regarding its efficiency improvement, so it is seen as a compromise between the C++ implementation and its execution time. Nonetheless, this solution is adopted as an implementation for the second filter in the SIRFS incremental optimization pipeline.

Nevertheless, the remaining 0.3 seconds represent the computation time taken by reading the data from each field in the structure hierarchy into the class hierarchy that maps it, starting from the pointer retrieved by the previously-named function. When compared to the similar MATLAB procedure which takes approximately 0.23 seconds, it can be said that its implementation is acceptable from the execution efficiency perspective. Nonetheless, the 0.07 seconds gap could be explained by the necessity of checking the name of each node in order to know to which class and which field of this class to assign the data, as it involves a sequential check of the field names from each vector of *StructNode* objects. However, there might still be room for improving this procedure's efficiency through parallelization, by using CPU multithreading.

That said, the total execution time of the C++ SIRFS code is still far from expectations, as it is 6-7 times slower than the MATLAB implementation. Therefore, it required further improvements, refactoring and enhancements. The next ones are presented in the next chapter, composing a solution for the third filter in the SIRFS incremental optimization pipeline, which aims at further optimizing the C++ SIRFS code, by addressing the overheads given by the matrix template classes implementation and by the increased modularity of the whole code.

Chapter 4

Refactoring and optimization of the ported code

In this chapter several refactoring and optimization procedures are presented. They are applied to the translated code as it is described in the document [2], including the changes presented in chapter 3 of this document. Thus, this chapter aims to define an optimal implementation of the third filter in the SIRFS incremental optimization pipeline illustrated in figure 3.2.

Specifically, section 4.1 discusses the efficiency of the matrix template classes implementation, namely the containers' memory allocation and indexing, as well as the methods' implementation. Further on, sections 4.2 and 4.3 explore the possibility of implementing the containers of the matrix template classes using STL vectors and unique (smart) pointers, respectively. If they do not add to the code's efficiency enhancement, it is assessed whether they do not hamper the runtime efficiency while adding a better code structuring brought along by their template classes implementations. Lastly, in section 4.4 is discussed the modular architecture of the SIRFS C++ implementation.

The need for researching and applying these procedures comes from the fact that parallelization cannot always improve each bottleneck in the code. This happens when the data types used for these critical sections are not amenable to parallelization, as they do not involve operations on big data sets such as matrices are arrays. On the other hand, there can exist operations on such data types, but their implementation complexity would make their eventual CPU or GPU parallelization less efficient than certain non-parallel solutions. That said, the aim of this chapter is to research such solutions and to illustrate their impact on the C++ SIRFS code. Certainly, if any of these methods proves to enhance the execution speed of a code section, it would positively impact the execution speed of the entire code.

4.1 Matrix template classes' efficiency discussion

As already discussed before, the 2D matrices represent the most used data sets in the SIRFS implementation. Also, several 3D matrices are used. For these data sets, two C++ template classes were implemented in order to model them. These classes have as field data integers for storing the matrices' dimensions and a container of a general type to store the matrices'

data. Furthermore, they embed several methods that implement certain behaviors, such as matrix addition and convolution.

That said, the manner of storing, indexing and processing the matrix data has a direct impact on the C++ SIRFS code compilation and execution performance. Therefore, with respect to these aspects, the initial implementation of these classes is discussed, by firstly analyzing the efficiency of the container storage and indexing, followed by the analysis of some methods.

4.1.1 Containers' efficiency improvement

The initial implementation of the 2D and 3D matrix template classes used an approach that dynamically allocates the containers' memory space separately, for each dimension, in order to limit the maximum size of the allocated blocks, thus preventing potential cases where the dynamic allocator would fail. Explicitly, for 2D matrices an array of pointers is firstly allocated to match the number of rows, and afterwards, for each pointer in this array it was allocated the memory space to match the number of columns. A similar approach is used for 3D matrices. Though this allows for an easier indexing and understanding of the matrices, from the user's perspective, it actually can be changed to a better implementation, with respect to execution performance.

Before introducing this better approach, firstly is presented the efficiency issue of the above-described implementation. For each dynamically allocated array it is allocated a contiguous space in memory, which allows for an easier indexing of consecutive elements, regardless the function used for this purpose, that can be *new* for C++ [19] or *calloc* and *malloc* in C [20][21]. In the initial implementation, in the case of 2D matrices for example, this contiguous space existed only within each row, with the data from different rows being allocated at different memory addresses. Thereby, iterating through a matrix implied jumping from a memory address to another whenever a new row was read or written. Furthermore, this approach involved multiple memory allocations, an operations that is computationally expensive.

Therefore, the new approach involves storing the matrix data in a contiguous dynamically allocated array. It stores a 2D matrix as one row after another, whereas for a 3D matrix this implementation was adapted for the third dimension. Concretely, if we see a 3D matrix as a collection of 2D matrices placed one behind each other (like a cube) firstly the rows of the front matrix are stored, then the rows of the one behind it and so on. This modification did not lead to the change of the interfaces provided by the classes, but just on the implementation of some methods. This was possible as there were implemented methods that linearize the 2D and 3D indexes. Thus, addressing such a matrix still happens in a natural way, by providing the 2D or 3D indexes to a given method, whereas the method indexes the container in a more efficient way.

This implementation improved the execution time of the entire translated code by approximately 1.5 seconds, contributing in achieving the total of 3.1 seconds mentioned in the previous section. In other terms, the total execution time dropped by 10%. The testing of

this approach was performed altogether with the procedure described in section 3.3, when the data loaded from the MAT file into matrices was tested against the MATLAB data. The execution times per code section are summarized in the table 4.1, below. Please note that using the matio library for loading the priors also contributed in achieving the presented values.

Code section	Execution time in C++ (seconds)	Execution time in MATLAB (seconds)
Parameters initialization	0.001	0.009
Load priors	1.49-1.51	0.224
Initialize "data" class members	0.004-0.006	0.002
Apply median filter	0.92 - 0.93	0.323
Build border normals	0.609-0.621	0.052
Total execution time	3.1	0.61

TABLE 4.1: Execution times per code section using single raw pointers for matrix containers

4.1.2 Methods' efficiency improvement

As previously mentioned, the 2D matrix is the most used data structure within the SIRFS code. As a consequence, the template class that implements it embeds a series of methods which define various behaviors required by the SIRFS algorithm. Besides the well known constructors, destructor, data accessors and methods that overload some operators (such as addition and assignment) there are methods for computing the convolution (between an input matrix and another matrix, called kernel) or for applying a mask (which is a matrix) on a given matrix, amongst others.

Initially, all methods were implemented in such a way that the input matrix was always the caller, with the result of the method's computation being stored in a matrix whose reference is received as a parameter. The reason for this lies in the fact that for some methods it is always known the type of the output matrix, but not for the input. For example, when the indexes of certain values are to be found based on a given logic and then stored in a different matrix, the latter will always be of integer type (as the indexes are integers). Therefore, having such a result matrix as the caller for a method of a template class would not be possible, as the C++ standard would consider it has a generic type, which also have to be the same with the types of the input matrices. So, for such cases, the reference to such a result matrix needs to be sent as input, and the input matrix (which can have a generic type) is the caller.

That said, for preserving consistency in between all the methods, the same strategy was applied for all methods, even if the result and the input matrix can have the same type, and therefore, can be interchanged in the method's implementation. Nonetheless, this approach is somehow counter intuitive, as the way one sees a method is like a black box that receives

some inputs (that are the parameters or arguments) and gives an output (which could be the method's caller). Thus, where it was the case, the method's implementation was changed in such a way that the result matrix is the caller, and the matrices required for internal computation are arguments, as illustrated below, in figure 4.1.

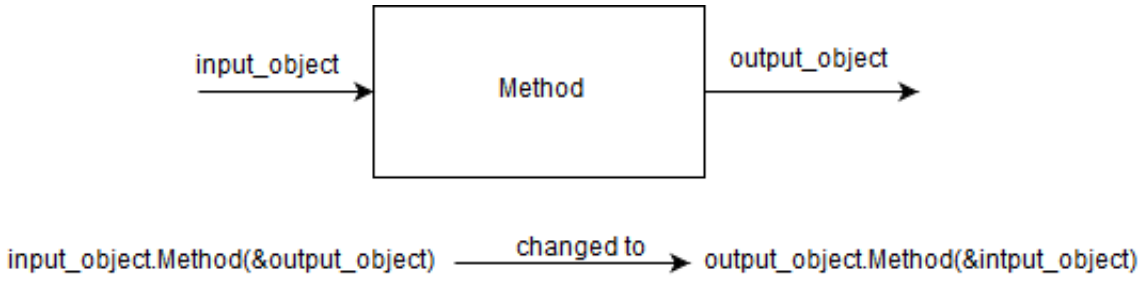

FIGURE 4.1: C++ template classes methods' caller and callee changes

This approach had an impact on the execution time of SIRFS as it dropped by almost 0.2 seconds. The changes were observed in *Build border normals* section, as the changed methods are called within it. Implicitly, the total execution time decreased by the same amount, achieving 2.8-2.9 seconds, which means a 6.45% performance gain, when compared to the total time shown in table 4.1.

This approach was applied to the code version presented in section 4.1.1. The table below presents the execution times per code section of the discussed approach.

Code section	Execution time in C++ (seconds)	Execution time in MATLAB (seconds)
Parameters initialization	0.001	0.009
Load priors	1.472-1.51	0.224
Initialize "data" class members	0.004-0.006	0.002
Apply median filter	0.95 - 0.964	0.323
Build border normals	0.387-0.396	0.052
Total execution time	2.8 - 2.9	0.61

TABLE 4.2: Execution times per code section when 2D matrix template class methods' signatures were changed

When compared to values from table 4.1, it can be noticed that the execution time for *Apply median filter* section increased by 0.03 - 0.04 seconds, but the 0.2 seconds drop for the *Build border normals* helped in achieving the best execution time so far. Therefore, this approach is adopted as a solution for the C++ SIRFS implementation.

4.2 STL vectors as alternative to raw pointers for matrix containers

In this section is aimed at finding if the implementation of the matrix template classes' container using STL vector can have a positive impact on the SIRFS code's execution performance. Certainly, the template class implementing the vector in the STL provides several

methods that can make coding easier, but an easier development does not necessarily imply a better execution performance. Therefore, in the following, it shall be considered a positive impact on the code's efficiency if the usage of STL vectors helps in achieving execution times that are at most equal to those presented in table 4.2.

4.2.1 Theoretical aspects

The STL library provides a standard implementation for vectors in the form of a template class. This class contains several members and methods that help in manipulating the vector's data container. An important aspect regarding the container's data manipulation is represented by iteration through it [22]. The STL provides several approaches for performing this operation, some of them being briefly introduced below, together with the corresponding syntax:

1. The first method uses an (unsigned) integer to index each element in the vector's container. It is a widely used approach in C, and it can be reused in C++:

   ```
   for(unsigned int index = 0; index < vector.size(); index++)
            vector[index] += 1;
   ```

2. The second method is called range based indexing and is available since C++11. It is easier to use and is more readable, as it directly addresses the value in the vector [23]. When it is only needed to read the values from the vector, the syntax look like the following:

   ```
   for(double n:vector)
            cout<<n<<endl;
   ```

   In the example code above, it can be noticed that the so called iterator holds the value of the vector at a given index, therefore it must have the type of the data contained in the vector. If the type is not known at compile time, there can be specified *auto* as type, which is available from C++11 and is used the deduce the data type when the given variable is initialized [24]. On the other hand, when the data needs to be written or updated, an ampersand must be used in the for loop as there is needed a reference to each value in the vector in order to make the changes persistent, as it is presented below:

   ```
   for(auto &n:vector)
            n += 1;
   ```

3. The third method makes use of the STL vector class member *size_type*, which is similar to *size_t* [22]. It can be used in the following manner:

   ```
   for(typename vector<double>::size_type index = 0;
       index != vector.size(); index++)
           vector[index] += 0.1;
   ```

4. The fourth method involves using the STL vector class member *iterator*. An iterator is a pointer to a given element in the vector. It can be used to iterate forward or backwards using appropriate methods in the STL for retrieving the pointers to the start and end of the vector [22]. Below is illustrated an example of reverse iteration that prints the vector's values to the console:

```
for(typename vector<double>::reverse_iterator rit = vector.rbegin();
    rit != vector.rend(); rit++)
     cout<<*rit<<" ";
```

4.2.2 Implementation discussion

The reason for bringing into discussion the above vector indexing methods lies in finding if they impact the execution time of the C++ SIRFS implementation. Also, since the last three methods can be applied only if the matrix template classes use a STL vector as container, their efficiency is discussed only in section 4.2.

Firstly, a discussion is required as it can be noticed that indexing methods two and four, presented in sub section 4.2.1, cannot be applied when simultaneous looping over different containers is required. This represents an issue as most methods implement such a behavior, when elements from a matrix need to be assigned or copied to another one, for example. Surely, nested loops can be used as a workaround, where the case, but they would increase the complexity, thus the execution time, which is far from the goal of this analysis. Thus, only the indexing methods one and three, presented in sub section 4.2.1, were implemented and tested for performance assessments.

Thus, both 2D and 3D matrix template classes implementations were updated such their container was changed to a STL vector and the method's implementations were updated to call the STL vector class methods, where the case. Since the majority of the 2D and 3D classes' methods involve looping over the container, the iterating procedures one and three, presented in sub section 4.2.1, were tested alternatively and several observations were done afterwards, with the results being highlighted in the section below.

The implementation guards the third indexing method with the macro *INDEXINGMETHOD_3*, with the first method being considered as default. This allows for an easy switch in between the two, when testing their performance. The macro is defined in *matrix2D.h* file.

4.2.3 Performance assessments

The performance of the two considered indexing methods are included in the tables below, followed by the conclusions made on the illustrated figures.

Code section	Execution time in C++ (seconds)	Execution time in MATLAB (seconds)
Parameters initialization	0.001	0.009
Load priors	1.481-1.488	0.224
Initialize "data" class members	0.016 - 0.017	0.002
Apply median filter	1.13 -1.135	0.323
Build border normals	0.758 - 0.762	0.052
Total execution time	3.43 - 3.45	0.61

TABLE 4.3: Execution times per code section using STL vectors with indexing method 1

The total execution time for this method was around 3.43 - 3.45 seconds. Event though the method three had not been tested yet, an important observation can be already made: using STL vector for container's implementation slows down the SIRFS execution. Nevertheless, the experiment was repeated for the indexing method 3, with the results illustrated in the following table:

Code section	Execution time in C++ (seconds)	Execution time in MATLAB (seconds)
Parameters initialization	0.001	0.009
Load priors	1.482 - 1.498	0.224
Initialize "data" class members	0.016 - 0.017	0.002
Apply median filter	1.132 - 1.135	0.323
Build border normals	0.758 - 0.765	0.052
Total execution time	3.43 - 3.45	0.61

TABLE 4.4: Execution times per code section using STL vectors with indexing method 3

With a total execution time around 3.43 - 3.45 seconds, this indexing method proved not to have a significant difference when compared to the previous one. All in all, the results shown in tables 4.3 and 4.4 prove that STL vectors do not help in improving the efficiency of the C++ SIRFS implementation, as they lead to a performance drop of 20%. Therefore, this implementation is not preserved in the C++ SIRFS implementation. However, it comes in a separate folder (branch) with the final code, so it can allow for further research in the future, if desired.

That said, it can be noticed a difference of almost 0.5 - 0.6 seconds between this approach and the one discussed in section 4.1.2.The reason for having a slower execution when STL vectors were used as containers mostly lies in the calling of methods for performing certain operations, even for simple ones (such as initializing the vector's values).

Though is usually preferred when working with STL vectors, using *push_back()* method proved to be slower than using indexing and the assignment operator when copying data

from a container to another, for example. This happens because the *push_back()* method also resizes the vector by allocating further memory when adding a new element in the vector, whereas it is not the case for the assignment operator. Moreover, there exist some tips regarding this aspect [25], which state that using the assignment operator for copying data to vector is faster than using the STL *push_back()* method. Also, there is evidence [26] that the fastest mode to access the elements of a STL vector is to convert it to a single raw pointer and index it in the C-style manner, using the first method listed in sub section 4.2.1.

Nevertheless, there was a difference in between the 2D and the 3D matrix template classes implementation impact on the execution time. The latter contains less methods than the former, which are also called few times and, thereby, its internal implementation has a lower impact on the code's execution efficiency. With respect to this, the 3D matrix template class implementation was changed to STL implementation and the 2D matrix template class preserved the implementation described in section 4.1, for testing purposes, and it slowed down the execution by 0.01-0.02 seconds when compared to the values in table 4.2. This aspect strengthens the idea that using STL vectors is not suitable to improve the execution times of SIRFS C++ implementation. However, this approach contributes to a better organization of the code and improves the implementation easiness by having a set of methods already implemented.

4.2.4 Verification

Since all SIRFS operations involve working with matrices, the changes in the matrix template classes need to be tested for the entire code. Thus, firstly the matrix reading method and the priors loading were verified by enabling the testing macros, as specified in sections 3.2.4 and 3.3.3, respectively.

Once these tests were passed, the block A2 was tested. As mentioned in [2], it involves the verification of two big functions: *medianFilterMatmask* and *getBorderNormals*, with the third function, *conv2mat*, being automatically tested with the first one. Unlike the testing procedure applied in [2], which makes use of verification *method 2*, here was preferred the the approach described in the same document as an alternative to this procedure. This involves using the C++ code as a testing environment and makes use of the MATLAB data which has to be stored, in an appropriate format that complies with the ported code, in a text file. The advantage of this procedure is that it allows for a testing process fully decoupled of MATLAB (as the needed text files are created together with this project and can be reused afterwards).

That said, the two matrices computed by *medianFilterMatmask* in MATLAB were preprocessed and written into text files using the script *test_medianFilterMatMask2.m* that is found under *blockA2* folder, together with the files that contain the values, called *ZM.txt* and *AM.txt*. Please note that, for a better separation of concerns, these files were not included under *validation_files* as there reside only the files and scripts used by the verification method 1, as described in [2].

Thus, these files are read in the C++ space and tested against local computed values, using the *test_matrix2D* function, likewise is described in sections 3.2.4 and 3.3.3. In this case, the local computed matrices contain six columns, with the first two and the fourth and fifth storing 0-based indexes of non null values in sparse matrices (see [2]), hence the needed preprocessing in the MATLAB. This involves incrementing the indexes by 1 in the C++ space, only before testing, and decrementing them by 1 after testing, as the 0-based indexing has to be preserved for further computation.

Furthermore, an extra processing step was added in C++ *medianFilterMatmask* which transforms the original output matrix that has two columns of KeyValues objects (that is 6 values per row) into a matrix that stores these values individually, avoiding the usage of KeyValue objects outside the function, which makes the use of this function more straightforward. This also made the verification easier, but also would help with the further computation, as the use of KeyValue objects is kept within the *medianFilterMatmask* and *conv2mat* functions. This extra step lead to the increasing of the execution time by approximately 0.01-0.02 seconds.

Thereafter, since the testing proved to be successful, the verification of *conv2mat* was not required, so the final testing step was executed, namely for the *getBorderNormals* function's results, which are three matrices and one vector. Firstly, their MATLAB values were written in text files, using the script called *test_getBorderNormals.m* that can be found in *blockA2* folder, together with the previously-mentioned text files. As proceeded before, these files are read in the C++ space and tested against local computed values, using the *test_matrix2D* and *test_vectors* functions. For testing the values for *Position* matrix and for *Idx* vector, it was necessary a similar approach to the one executed for *medianFilterMatmask* verification, as these containers also stores indexes.

The tests shown some minor issues which required some adjustments which, in the end, helped in achieving a successful testing. Thereafter, the analysis still focused on matrix template classes' containers, but involved working with smart pointers, as which are presented in the following section.

4.3 Smart pointers as an alternative to raw pointers

Likewise the previous section, this section aims at studying whether a C++ mechanism, namely smart pointers, can help in improving the code's execution performance while providing a better approach in working with pointers.

4.3.1 Theoretical aspects

Smart pointers represent template classes that manage the dynamically allocated objects in C++ (using the *new* operator) and that can be mostly used like a raw pointer. Here, it is worthy to mention that they cannot be used to point to objects allocated on stack [27]. That said, as the template classes that implement them are roughly wrappers over raw pointers,

the following questions arise: can smart pointers help in improving the efficiency of the C++ SIRFS code? Or can they help in having a better coding style while not hampering the performance of the C++ code's execution? Thus, by analyzing some aspects regarding their implementation, we can get an idea for answering to the above questions.

The smart pointers can be involved in code when an object is dynamically allocated using the *new* operator. This definition (or the pointer returned by it) can be passed as argument to the smart pointer's constructor, which internally creates a *manager object* for the allocated object, which is referred to as *managed object*. The manager object contains meta data of the managed object, including a pointer to the managed object, which is accessed when calling the overloaded *operator->* from the smart pointer template class. Moreover, the management includes the automatically deletion of the dynamically allocated object, by calling the smart pointer's template class destructor. This is an important aspect in C++, as the object ownership is rather seen as the responsibility of deleting that object. The smart pointers are automatically deleted when they go out of scope, by calling their destructor. [27]

That said, involving this middle management leads to having a pointer to the actual pointer that was dynamically allocated, which means that there are further deferences which might entail small overheads. In [28] is shown that, at least in the case of *unique* pointers, at assembly level there is an extra operation. The study makes a comparison with the usage of raw pointers and suggest that difference between the two scenarios "are going to be quite low, perhaps as low as 10%".

As the concept of *unique* pointers was mentioned, it is worthy to say that there are three types of smart pointers: shared pointers, weak pointers and unique pointers. They are presented below:

1. **Shared pointer**: It allows the shared ownership over the same managed object. The manager keeps track of a counter that is incremented every time a shared pointer to that managed object is defined. The definition of the first shared pointer to a given object includes the definition of the pointer to the managed object (using the *new* operator). On the other hand, the next shared pointers to the same managed object are defined using the first shared pointer, and involve the calling of the assignment operator or copy constructor, which increment the counter. All these shared pointers will actually point to the same manager object. The counter is decremented when a shared pointer is deleted or when it is set to point to another object. [27]

   If a new shared pointer is declared to point to the same dynamically allocated object, then a new manager is created (which is also dynamically allocated) so it leads to more memory consumption and can affect execution time.

2. **Weak pointer**: It keeps a reference to the manager object, not to the managed one, therefore it acts like an "observer" of the dynamically allocated object and does not influence its lifetime. A weak pointer can be constructed from a shared pointer or from another weak pointer. Furthermore, from a weak pointer it can be obtained the shared pointer to the managed object (if the last shared pointer was previously deleted) by

calling the *lock* method. The function returns an empty shared pointer if the managed object does not exist, otherwise it returns the shared pointer to that object. Nevertheless, a weak pointer cannot be dereferenced and used like a raw pointer, only the shared pointer can do as such.[27]

The manager object also keeps a count of the weak pointers referencing to it, as it has to deal with the managed object's deletion. That said, when the last shared pointer to the managed object is deleted, but there is at least weak pointer to it, only the managed object is deleted (and the pointer to it is set to 0), but the manager object is kept. It is deleted when both counters are 0. The decrementing is handled by the destructor of the shared or weak pointer, when the pointer is deleted. [27]

3. **Unique pointers**: Similarly to shared pointers, they point to a dynamically allocated object and when the pointer goes out of scope, the objects it points to is deleted automatically. Nonetheless, the difference comes from the fact that it entails unique ownership, which means that one unique pointer can point to a dynamically allocated object. As a consequence, its constructor and destructor do not have to increment or decrement any counter, just allocate or deallocate memory. Therefore, it does not involve any overheads and can be used with the same efficiency as a raw pointer as, implementing-wise, it does not contain a manager object, but just the pointer which is dynamically allocated. [27]

   Since it uniquely points (and identifies) a dynamically allocated object, copy constructor and assignment operator cannot be used, as they would violate the principles of unique ownership. However, the ownership transfer is possible, by using *std::move*, which takes as argument the unique pointer which owns the object and assigns its result to a new unique pointer. After its call, the new unique pointer is the owner of the object, with the old one pointing to nothing. [27]

In addition, the study [27] addresses the memory efficiency problem regarding smart pointers. That is, by looking closely to the syntax for defining a shared pointer, there can be noticed two memory allocations, one for the managed objects and one for the shared pointer itself (including its manager object):

```
shared_ptr<Type> p(new Type);
```

The memory allocation is a slow process [27] and having two instead of one (like in the case of raw pointers) will affect the code's execution efficiency. For mitigating this issue, C++11 makes use of a template function, called *make_shared*, that does the previously-named allocations in one big memory allocation. This function returns a shared pointer and can take as inputs the parameters that should be send to the constructor of the managed object. Similarly, for unique pointers can be used *make_unique*, which is available with C++14.

```
shared_ptr<Type> p(make_shared<Type>());
```

Another problem is related to getting a shared pointer for this object, as doing it using the syntax shown above leads to creation of new manager object for the same managed

object, every time the this object is needed. To mitigate this issue, C++11 makes use of the class *enable_shared_from_this* which has a weak pointer member and a method, named *shared_from_this*, that returns the shared pointer constructed from that weak pointer. Therefore, the user defined class should inherit from the above-named class, by that the weak pointer will be initialized when the first shared pointer to the managed object is defined (by the shared pointer constructor, due to inheritance). Also, the shared pointer can be retrieved by calling the previously-specified method, in the following manner:

```
shared_ptr<Type> sp_this = shared_from_this();
```

A downside comes from the fact that *shared_from_this* cannot be called in constructors, as the weak pointer has to be set after the constructor of the user defined class finished its execution, by the shared pointer's constructor.[27]

4.3.2 Implementation

The previously-presented theoretical analysis on smart pointers brings into the light that weak pointers cannot be used to point to dynamically allocated objects, but to the manager object created when a shared pointer is defined. Thereby, it is clear that when it comes to working with smart pointers, one has to come to the decision whether shared pointers or unique pointers have to be used. Surely, in order to make such a decision it has to be taken into account the scope of the project or analysis that is undergone, or, more concretely, which quality attributes are envisaged by that project or analysis.

That said, since this thesis aims at improving the execution performance of the translated C++ SIRFS code, it was clear from the beginning that unique pointers should be used, as, from the theoretical perspective, they do not introduce overheads. Next to that, the shared pointer's extra operation in allocating memory for the management object is discussed above as having negative effect on runtime performance, therefore making the decision clearer. Nevertheless, since the project is developed under the C++ standard, it cannot leverage the usage of *make_unique*, which is available with C++14, for a faster memory allocation, but it is an indicator that this implementation can be improved in the future, by applying newer C++ standards.

Once these aspects were clear, the implementation focused entirely on 2D and 3D matrix template classes. The firstly made change was in container's declaration, as it requires to pass to the unique pointer's template constructor the data type of the pointer. Since the container has to behave like a vector, as described in section 4.1.1, this type has to be a vector to the generic type of the matrix template classes, as shown below:

```
std::unique_ptr<Type []> container;
```

The syntax above means that *container* is a unique pointer to a dynamically allocated vector of type *Type*. Thereafter, the destructor was changed, as it does not require any implementation because the destructor for the container is called automatically, as explained before.

Next, the constructors' implementations were updated, as well as the assignment operator's overloading, in order to allocate memory using an appropriate syntax, presented below:

```
this->container = unique_ptr<Type[]>(new Type[this->rows*this->cols]);
```

Here, it can be noticed the double allocation that is made: one for the dynamically allocated vector and one for the unique pointer. Thus, this approach is expected to introduce some overheads, but it is hard to assess at the moment of the implementation whether they will be considerable or negligible. Another aspect to be noticed is that a unique pointer is created inside the method and then it is assigned to the container declared as a field member. That is because in the moment when the assignment takes place, the field member *container* becomes the only owner of the dynamically allocated memory, behavior implemented by the overloaded assignment operator of the unique pointer template class.

That said, throughout this implementation, a special attention is given to the ownership transfer, as it represents the main attribute of unique pointers and which entails special design and implementation considerations, whenever an operation is performed on a matrix object. Concretely, when a method is called to execute an operation on a matrix received as an argument, the ownership of that matrix unique pointer has to be transferred, using the *std::move* function. For example, when the convolution is applied, a matrix and a kernel matrix are sent as inputs to the method computing their convolution, using the previously-named function. Such a call can look like the following:

```
C.conv2DValid(std::move(absj), std::move(absi));
```

This has to happen because during the execution of the given method, it takes the ownership of the input objects, as they contain an unique pointer as a data member, and no other function or method can hold the ownership on their containers simultaneously. Consequently, the context in which the given method is called (that is the caller) loses ownership on those objects. Thereby, it is clear that the entire C++ SIRFS code was updated in such a way that all methods and functions involving matrix data manipulation receive their 2D and 3D matrix inputs using *std::move*, including those used for verification.

Next to that, the copy constructor and the assignment operator overloaded method were implemented as move constructor and move assignment operator, respectively, as their input matrix objects need to be sent using the function mentioned before. Furthermore, the other methods' signatures were updated to receive as input a reference to the rvalue returned by the *std::move*, which calls the move constructor that returns a temporary address to the object sent as argument [29]. The reference to an rvalue is denoted with a double ampersand [30].

Lastly, the getter method for the matrix container was also changed, firstly to update the type of the returned data and secondly to call *std::move*, as each time the container's data is gotten, the ownership needs to be transferred.

4.3.3 Verification and performance assessments

The reason for studying the effects of unique pointers on the C++ code involve assessing whether they really can be used with the same efficiency as raw pointers, while allowing for a better memory management through the automatic calling of the destructor.

Code section	Execution time in C++ (seconds)	Execution time in MATLAB (seconds)
Parameters initialization	0.001	0.009
Load priors	1.524 - 1.541	0.224
Initialize "data" class members	0.023 - 0.025	0.002
Apply median filter	2.18 - 2.231	0.323
Build border normals	1.031 - 1.053	0.052
Total execution time	4.823 - 4.866	0.61

TABLE 4.5: Execution times per code section using unique pointers for containers' implementation

Clearly, the results show this solution does not serve for the purpose of this thesis, as it causes a performance drop of 70%. Moreover, it can be noticed that its performance is worse when compared to STL vectors approach. Therefore, this implementation was not preserved as a solution for the C++ SIRFS translated code, but it comes in a separate folder (branch), likewise the STL vectors implementation, in order to provide the basis for eventual further analysis on it. The code that implements the unique pointers solution is guarded by a macro called *U_PTR_CONTAINER*, which is defined in the file *matrix2D.h*. By uncommenting it, the code is reverted to the solution described in the section 4.1.2.

The verification of this implementation was pretty straightforward as it followed all steps described in section 4.2.4. Therefore, all the macros used for the validation procedure were enabled and minor fixes or adjustments were required for some identified issues, but in the end all tests passed successfully and the analysis continued with the code's modularity, presented in the next section.

4.4 Modularity trade-off

One of the key drivers of the project describe din [2] is modularity. Nevertheless, a balance between the number of called functions and methods is desirable when performance improvements are envisaged. This is because the increased modularity adds an execution overhead, as each logic or behavior, no matter how small is it, is encapsulated in a function or method in the initial C++ SIRFS version described in [2]. The overhead comes from the fact that with each function call, several further steps are executed by the compiler, such as pushing the arguments of the callee on the program's stack, increment the stack pointer, jump to the beginning of the new code (the callee's code), then execute it and return from

it. Based on different calling conventions and with respect to two major OS platforms, these steps are described in [31] for x86 platforms.

Therefore, after analyzing the template classes it was concluded that the number of their methods is reasonably fair. Furthermore, if for some it would be preferred to use the implementation instead of calling them (implementation externalization) in order to decrease the overall number of called methods, not only it would make the code harder to understand and to debug, but it would impact the code reusability as well.

Thus, this analysis had to be focused on functions, not on methods. As explained in [2] the only implemented functions can be found in the *matlab* folder, excluding the functions used for validation. The implementations of the former were analyzed with the scope of finding whether the externalization of their behavior would not negatively impact the easiness in understanding and debugging the code. Since the SIRFS code is not entirely translated, it is hard to assess the impact of this externalization on the code reusability, therefore the functions whose implementations are externalized are still kept with the code, but are not called anymore.

That said, in file *matlab_matrix* are implemented several functions that capture some interactions between different types of matrices, such as building a matrix from a vector of matrices of *KeysValue* type, apply a mask on a *KeysValue* matrix and convert a matrix of a given type to a matrix of another type.

The first decision was to replace the functions that perform type conversion between matrices with their implementation, by substituting them were they are called. The impacts on code's performance were minor so it was proceeded further with the functions that involved the processing of *KeysValue* matrices. These functions are only called by *medianFilterMatmask* and it was noticed that its execution time slightly increased once the changes mentioned in section 4.2.4 were applied. The reason for choosing these functions is that they were called only once, which is an indicator that here is a trade-off point between modularity and reusability, on one side, and performance, on the other side, with focus on the latter.

The same way was proceeded with some functions from folder, namely *createVectorMask* and *convertVectorOfVectorsToMatrix*. Since the former did not influence the execution times at all, its function call was preserved, but the latter's implementation externalization shown slightly improvements so its replacement was kept. For all these replaced functions, their calling within the code have just been commented out, so it easy to search for them and revert to the more modular code version, if desired.

The externalization of the above functions' implementations proved to have a positive impact on the code's execution times, which are illustrated in the table below:

Code section	Execution time in C++ (seconds)	Execution time in MATLAB (seconds)
Parameters initialization	0.001	0.009
Load priors	1.451 - 1.487	0.224
Initialize "data" class members	0.005	0.002
Apply median filter	0.914 - 0.92	0.323
Build border normals	0.385 - 0.393	0.052
Total execution time	2.79 - 2.825	0.61

TABLE 4.6: Execution times per code section with less code modularity

The values for the total execution time illustrate that the initial C++ SIRFS implementation could have a better balance of the number of called functions and methods. Thus, this approach is preserved for the final solution of the C++ SIRFS code, as it comes with a 1.4% performance gain.

Surely, the results presented in this section should not lead to the conclusion that methods and functions should not be used. Without those, it would be almost impossible to translate the MATLAB SIRFS implementation, but it shows that there should be a balance of the number of implemented functions and methods. Furthermore, it can be an indicator that, in some cases, grouping together the implementations of several functions might be desirable. Please note that during the translation process it was preferred an approach that allowed for a better separation of concerns, which helps with code development, debugging and reusability.

Chapter 5

CPU parallelization

In the previous chapter several optimization approaches are discussed with focus on their impact on the SIRFS C++ code's runtime performance. Nevertheless, none of these approaches involves parallelization.

Parallelization is a technique that allows a program to execute multiple tasks simultaneously, if the hardware allows for. Since it is highly dependent on the hardware, it can be done on CPU and GPU, as well, using appropriate APIs that allow to leverage the parallel architecture of the CPU and GPU, respectively. That said, whereas the GPU parallelization of SIRFS is discussed in chapter 6, in this chapter is presented only the SIRFS CPU parallelization approach.

Specifically, the CPU parallelization can be done in two different ways: by using processes, or threads. For the SIRFS CPU parallelization it was chosen the latter approach, which is implemented using the POSIX thread, also called *pthread*, library. The reason for choosing *pthread* library lies in the fact that they are a well known and widely used approach for CPU parallelization.

That said, the chapter starts with a presentation of threads using POSIX standard for Linux platforms. A theoretical basis of the concepts used for the implementation is given, alongside a reasoning of choosing threads over processes. Then, several coarse-grained parallelization approaches are implemented and tested aiming to find an optimal solution for the fourth filter in the pipeline shown in figure 3.2. The chapter ends by presenting the performance assessments on the presented experiments and by concluding on them.

5.1 Theoretical aspects

In his book called *Linux Programming Interface* [32], the author Michael Kerrisk, presents a series of Linux API features. Amongst those, a discussion is made on POSIX threads, also referred to as *pthread*, likewise the library that implements them. In this section, several aspects outlined in the previously-named book are illustrated as they represent the cornerstone of the CPU parallelization implementation experiments envisaged in this chapter.

In order to be able to work with POSIX threads, they are defined and presented together with the POSIX functions need for the implementation of the experiments outlined in section 5.2. Concretely, the threads represent a mechanism which allows for performing multiple tasks

concurrently. Unlike processes, which represent another mechanism that allows for multi-tasking computation implementation, the threads coexist at the level of the same process, that is the main process.

All threads share certain attributes in between them, including with the main program, which is seen as a single thread. Amongst these attributes it is worthy to mention the file descriptors, the global memory, that is represented by the heap memory segment, initialized and uninitialized data stack segments, which are illustrated in figure 5.1. These aspects contribute to a more efficient creation of each new thread, when compared to a process creation, as the latter involves duplicating all the above-mentioned segments for each newly created process (mostly through copy-on-write). Furthermore, by sharing heap and global memory the data transfers between threads are easier and faster, than in between processes. [32]

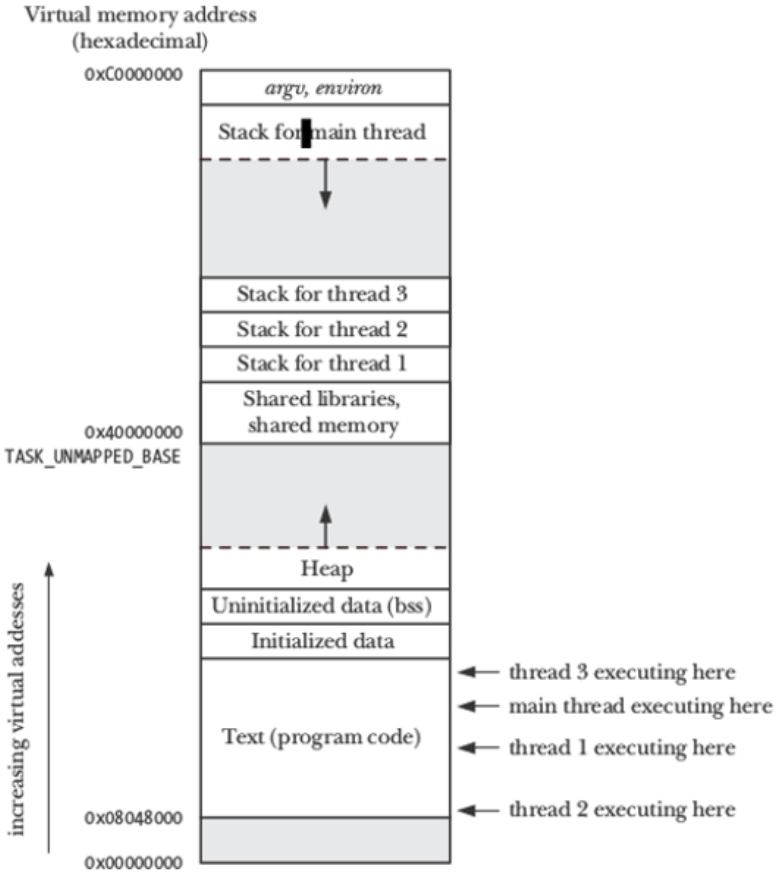

FIGURE 5.1: Program stack of a process with three child threads [32]

Besides the stack segment, there are more attributes typical to each individual thread, such as the signal mask, thread-specific data and the thread ID. A thread can access its own ID, whose type is *pthread_t*, by calling the function*pthread_self*, which takes no arguments, whereas the parent thread gets it as a pointer, by calling the function *pthread_create*. The signature of the *pthread_create* is presented and explained below. [32]

```
int pthread_create(pthread_t *thread_id , const pthread_attr_t *attr ,
void *(* start )(void *), void * arg );
```

The function returns 0 on success or a positive value which indicates the ID of an error status, in case of failure. The first argument (*thread_id*) is a pointer where the ID of the newly created thread is stored, followed by a pointer (*attr*) where the attributes required for creation are stored. For default creation, this pointer can be specified as *NULL*, which is a widely-spread approach. Please note that the POSIX standard complies with ANSI C one and not necessarily with C++11 (or newer) one, thus *nullptr* value is not a valid. Next, the third argument, called *start*, is a pointer to a function that returns a pointer to *void* and takes as argument a pointer to *void*. This function is executed by the newly created thread and it is considered the starting point in its execution. The last argument is a pointer to *void* representing the argument for the *start* function. [32]

However, in the book [32] is mentioned that the data type *pthread_t* is not represented the same on all architectures, so when comparing the IDs of two threads, the *pthread_equal* function should be used, which returns 0 if the two IDs sent as arguments are different, and a positive value otherwise.

Once created, a thread starts processing the *start* function, in a separate stack segment. Therefore, if the *start* function outputs a value, or more, they have to be returned by the thread. One way to do so is to use the *return* statement as, by default, the *start* function returns a pointer to *NULL*, which can be replaced with the desired return value. However, there is another approach to do as such, which makes use of the function:

```
pthread_exit(void * retval)
```

The only difference in between these two approaches lies in the fact that the latter can be used in one of the functions called by the the *start* function to end the thread's execution. Both methods are used to indicate the end of the execution for a given thread. [32]

The value output from one child thread can be retrieved in the parent thread by calling the function

```
int pthread_join(pthread_t thread , void ** retval )
```

The function returns 0 on success or a positive value to indicate an error status, in case of failure. It takes as input the ID of the thread that is to be joined and a double pointer (to void) where the child thread's return value is stored, in order to make it accessible in the parent thread. Moreover, by joining a thread is meant that the parent thread waits for the child thread to finish its execution. If it is not proceeded as such, there is a chance for the main program to end its execution before the child thread finish its one, which is not a desired behavior, as it would output wrong results. On top of that, joining with a thread that was previously joined can lead to wrong results, as a new thread with the same ID might have been created meanwhile, ending with a scenario when it is waited for a result that is actually computed by another implementation. [32]

All in all, threads allow for an easy data sharing in between them, as they share common global memory and heap memory. Nevertheless, when more threads need to modify data stored in one of these sections, they must be prevented from simultaneously accessing that

critical section, otherwise their computation's final result will be wrong. Moreover, there can be cases when it is useful for a thread to be informed by another one when the latter changed the status of a global variable. Thus, a synchronized access to those critical sections is desired and, therefore, two synchronizations mechanisms, namely mutexes and condition variables, are outlined in the following two subsections, as they are used by the experiments explained in section 5.2.

5.1.1 Thread synchronization: Mutexes

Mutexes allow a thread to atomically access a shared resource. In other terms, when a thread executes the critical section guarded by a mutex, any other thread cannot access it, either for reading or writing. This is done through *lock* and *unlock* operations, which define the two possible states of a mutex. That said, before modifying a shared variable, a thread must lock (or acquire) the mutex, thus becoming the owner of that mutex and being the only thread that is able to unlock it. The unlock (or release) operation should be performed once the access to the critical section is finished. [32]

In order to be accessible to all threads, the mutex is declared globally, having as data type *pthread_mutex_t*. A mutex must be initialized before it is used, one manner of doing as such being the static allocation, using a macro defined in the *pthread* library, presented beneath.

```
pthread_mutex_t mutex_variable = PTHREAD_MUTEX_INITIALIZER;
```

Nonetheless, the author Michael Kerrisk specifies in [32] that when a mutex is initialized using the above method, all operations should be performed on it, not to a copy of it, as any copy of the mutex might lead to undesired results. On the other hand, the dynamic initialization is performed by calling the function shown below.

```
int pthread_mutex_init(pthread_mutex_t * mutex ,
const pthread_mutexattr_t* attr );
```

The function returns 0 on success or a positive integer in case of failure. It takes as inputs the address of the mutex declared (but not defined) globally and the address of a structure holding certain attributes that can be considered for the mutex definition. Typically, the latter is set to *NULL* for a default initialization. Again, the author Michael Kerrisk specifies in [32] that reinitialization of a mutex using the dynamic approach leads to undefined behavior.

Moreover, when dynamically initialize a mutex, it has to be explicitly destroyed when it is not need anymore, by calling the *pthread_mutex_destroy* function, which takes as input the address of the mutex, as illustrated underneath. It is recommended to destroy a mutex when it is unlocked, otherwise a thread that acquired it will try to release a non-existent mutex.

```
int pthread_mutex_destroy(pthread_mutex_t * mutex );
```

After initialization, a mutex is unlocked and it is ready to be acquired and then released, once the access to the critical zone is done. In order to do so the following two functions are used, the first one for acquiring access on the mutex and the second one for releasing it.

```
int pthread_mutex_lock(pthread_mutex_t * mutex );
int pthread_mutex_unlock(pthread_mutex_t * mutex );
```

The functions return 0 on success or a positive integer in case of failure. Their argument is the address of the mutex defined using one of the methods described above. The lock request of a thread on a mutex which is already acquired by another thread, typically is blocked until the other thread releases it, or throws an error. Once the other thread releases it, the lock is performed successfully in the calling thread. However, if there are more threads waiting to acquire the mutex, it cannot be said which one will get it, as the behavior is non-deterministic. Moreover, when a thread tries to unlock a mutex that is not locked or that is locked in another thread, the request fails with an error.[32]

However, a special attention is given to deadlocks. The deadlock is a situation when two or more threads lock the same set of mutexes in such a way that leads to a situation when each thread waits for another one to release a given mutex in order to progress. As a workaround, Michael Kerrisk proposes a hierarchy of threads, which means that all threads will try to lock the set of mutexes in the same order. [32]

In the figure below is shown a deadlock situation when two threads, T1 and T2, try to lock mutex_A and mutex_B. If thread T2 tries to acquire mutex_B immediately after thread T1 acquired mutex_A, but before T1 requested to lock mutex_B, thread T1 will wait for T2 to unlock mutex_B. Thread T2 will progress with the next instruction and will request to lock mutex_A, but it will wait for T1 to unlock it, which already waits for mutex_B to be released by T2. So, each thread will wait for each other continuously, without progressing further.

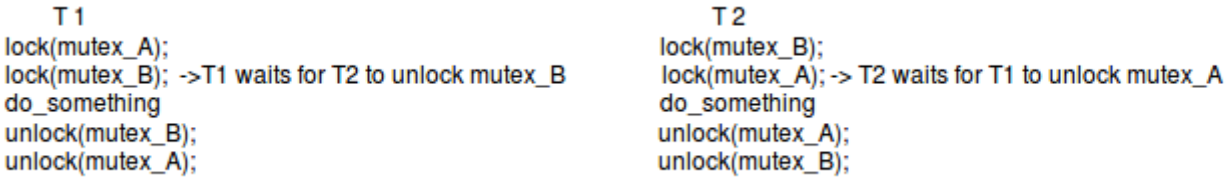

FIGURE 5.2: Deadlock situation with two threads accessing two mutexes

In order to illustrate the solution proposed by Michael Kerrisk in [32]. The above-described scenario is changed accordingly and illustrated below. Here, it is supposed that thread T1 is the first to acquire mutex_A. Once it locked it, T2 will try to lock the same mutex, but since it is locked, T2 will wait for it to be unlocked and does not proceed further. Then, T1 locks mutex_B, as it is not locked by T2 and afterwards gains access to the critical section. T2 will start its execution once T1 released mutex_A.

All in all, a thread cannot lock the same mutex twice or more and it cannot unlock a mutex that is not locked by itself or that is already unlocked. Nonetheless, these situations might occur when implementing threads, for various reasons, such as having a big data base of source code files that makes hard to follow the code flow. Thus, the behavior of the lock and unlock functions in the above-mentioned scenarios is influenced by the type of mutex. In

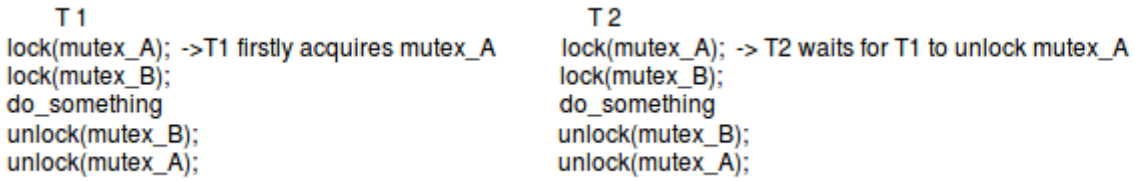

FIGURE 5.3: Solution for deadlock situation with two threads accessing two mutexes

the book [32], four distinct types of mutexes are presented, which are also listed and briefly described below:

1. PTHREAD_MUTEX_NORMAL: a deadlock occurs if a thread locks a mutex twice; an undefined behavior appears when the thread unlocks a mutex not owned by itself or already unlocked.
2. PTHREAD_MUTEX_ERRORCHECK: its is slower than the previous one as it performs error checks for each of the listed scenarios, with the lock and unlock functions returning an error code. It is useful for debugging purposes.
3. PTHREAD_MUTEX_RECURSIVE: a lock count is updated, per mutex. Initially set to 0, it is incremented every time a thread locks the mutex in cause, and decremented when a release operation on the mutex occurs. However, the mutex is made available for other threads to acquire when its counter is 0. When an unlocked mutex is unlocked, the call fails.
4. PTHREAD_MUTEX_DEFAULT: it is the default type of mutex, which is automatically created as such if statical initialization is used, or if the attributes pointer is *NULL*, in the case of dynamical initialization. On Linux, it behaves like the normal mutex which "allows maximum flexibility for efficient implementation of mutexes" [32].

For setting the type, it is used an instance of the *pthread_mutexattr_t* structure and the function shown underneath, as an example for setting a recursive mutex.

```
pthread_mutexattr_settype(&mtxAttr, PTHREAD_MUTEX_RECURSIVE);
```

Afterwards, the structure instance, called *mtxAttr* in the example above, is sent as second argument for the *pthread_mutex_init* function.

5.1.2 Thread synchronization: Condition variables

Condition variables are used to signal the change of a variable shared by multiple threads. In other terms, a condition variable allows one thread to inform the other threads that the state of the shared data has changed, with the other threads waiting for this notification.

A condition variable is used together with a mutex, as the latter must ensure mutual exclusion when the shared variable is accessed by the threads, while the condition variable

informs the other threads when a change in the shared variable's state happened. Furthermore, the condition variable is just a "mechanism for communicating information about the application's state" [32], and it does not carry any information with it. Thus, if there is not any thread waiting at the condition variable when the signal is sent, it is lost.

Like mutexes, condition variables need to be initialized before used. For this, a global data of type *pthread_cond_t* must be declared. Then, it is initialized statically or dynamically. As in the case of mutexes, the statical initialization involves using a macro, that is PTHREAD_COND_INITIALIZER. An example in this sense is provided below. Similarly to the mutexes' case, the author Michael Kerrisk specifies in [32] that the operations applied to copies of condition variables declared statically lead to undefined behavior.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

The dynamical initialization is realized by calling the function *pthread_cond_init*, which returns 0 on success and a positive integer in case of failure. The function's signature, illustrated below, entails two arguments: the first one being the address of the globally declared *pthread_cond_t* data and the address of a structure of type *pthread_condattr_t* which specifies certain creation attributes. However, for default creation it can be set as *NULL*. Moreover, initializing an already initialized condition variable leads to undefined behavior. [32]

```
int pthread_cond_init(pthread_cond_t * cond , const pthread_condattr_t * attr );
```

As in the case of mutexes, a dynamical creation of a condition variable entails an explicit destruction of it, using the function beneath, which returns 0 on success and a positive integer in case of failure, receiving as argument the globally declared condition variable. In addition, the author suggests that is not safe to destroy a condition variable as long as at least one thread is waiting for it. [32]

```
int pthread_cond_destroy(pthread_cond_t * cond );
```

Whereas the mutexes involve two operations, lock and unlock, the condition variables also entail two operations: signal and wait. The first one notifies other threads regarding a change that occurred in a shared variable, with the second operation involving waiting/blocking until a signal is received. In order to signal a condition variable, two distinct functions can be used; their signatures are presented underneath. As it can be seen, both are applied to the globally declared condition variable. Also, both return 0 on success or a positive integer in case of failure.[32]

```
int pthread_cond_signal(pthread_cond_t * cond );
int pthread_cond_broadcast(pthread_cond_t * cond );
```

Basically, the difference between the signal and the broadcast function is that the first one unblocks at least one thread, if multiple threads are waiting for a signal, whereas the second one ensures that all waiting threads are unblocked. Next to that, the signal function is

more efficient, in terms of runtime performance. Here, the author explains that the broadcast function should be used when the threads waiting for a signal perform different tasks, therefore having different testing conditions associated with their condition variable. [32]

The waiting procedure entails calling the function beneath. It receives as arguments the condition variable and the mutex which ensures mutual exclusion on the shared variable.

```
int pthread_cond_wait(pthread_cond_t * cond , pthread_mutex_t * mutex);
```

The mutex is needed as the *pthread_cond_wait* function unlocks and then locks it internally, in order to block the calling thread while waiting for receiving the signal from another thread that performs a change in the shared variable's state. Thus, the calling thread does not lock on the condition variable, allowing the other thread to modify its state. Here, it is worthy to mention that the internal lock and blocking on the condition variable are performed atomically. In addition, the function is typically called within a loop, whose condition, called predicate, is applied to the shared variable. This verification is guarded by mutex lock and unlock. The need to use the loop comes from the fact that there is no guarantee that the condition of the shared variable has changed after the first call of the wait function (as another thread might be woken up first, for example), thus its status should be checked repeatedly, till the condition is met.[32]

For an easier understanding, a schema of the above procedure is shown underneath, based on an example from the book [32].

```
//global declarations
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_var = PTHREAD_COND_INITIALIZER;
//the shared variable modified by multiple threads
int shared_var = 0;

//code section residing in the thread that waits for being notified

//lock the mutex so shared_var can be checked safely
pthread_mutex_lock(&mutex);

// as long as shared_var does not have the desire value, this thread will wait
while shared_var does not hold the desired value   //the predicate
pthread_cond_wait(&cond_var, &mutex);

perform some operations on shared_var

// the shared_var has the desired value, unlock the mutex
pthread_mutex_unlock(&mutex);
```

5.2 Implementation

In the previous section several advantages of the threads over processes were outlined. All in all, the advantages converge to the same idea: using threads is faster than using processes. Therefore, giving the scope of this thesis, the multi-threaded implementation of the C++ SIRFS code is preferred to a multi-tasked one. That said, in the following, the multi-threaded approach is also referred to as CPU parallelization.

Before introducing the implementation details, it is worthy to mention that it only envisaged a coarse-grained parallelization, as the goal was to include a finer grained GPU parallelization within it. Therefore, this architecture would involve multiple levels of parallelism, which is an approach that aims to address solutions for the different types of bottlenecks outlined in section 3.1. Concretely, the coarse-grained CPU parallelization approach introduced in this section involves the parallel execution of the code sections: Load priors, Apply median filter and Build border normals. Thus, these code sections are embedded within the start function of a CPU thread.

The advantage of this approach lies in the fact that it looks for a solution to parallelize the SIRFS code which is not amenable to GPU parallelization, mostly due to the hardware limitations highlighted in appendix A. Hence, the idea of having multiple levels of parallelism, namely the GPU parallelism (that is finer grained) within the CPU one. Nevertheless, this approach comes with the following drawback: that the amount of the CPU coarse-grained parallelization is limited by the high-level structure of the program, that is the number of the code sections that can run independently, in this case. Next to that, the approach assumes that the finer grained parallelism existing within each CPU parallelized code section, is amenable to GPU parallelization.

Whereas, for the first drawback, there is not much to be done, in case of the second one, an alternative lies in the fact that a finer-grained CPU parallelism can be used if the GPU one does not work, regardless of the fact that the coarse-grained CPU parallelism proves to enhance the SIRFS runtime performance or not. This finer-grained CPU parallelism is data driven, as it focuses an dividing the data sets and, implicitly, the loops involved in their computations, into blocks that have a smaller size than the original dimension of the problem in cause. Therefore, each such block is processed by a thread, that performs the same logic for all considered blocks, following the SIMD pattern. Sure, this approach is also influenced by the number of available CPU cores and its main disadvantage is represented by the big effort required for re-coding. Nevertheless, given the time constraints the CPU fine-grained parallelization doe snot make the object of this thesis.

That said, the CPU parallelization involved two implementations, each one matching with one experiment, that are outlined in the subsections below.

5.2.1 3-threaded coarse grained parallelization

The first experiment firstly envisaged the functions *medianFilterMatMask* and *getBorderNormals*, as the computation of one is independent from the other one's. The outputs of these functions, which are stored in the *data* class, are respectively represented by a bidimensional sparse matrix and a class instance, called *Border*, containing three bidimensional matrices and one vector.

As explained in section 5.1, the output of a thread is retrieved by calling the function *pthread_join*, and is represented by a double pointer to *void*. So, in the main thread it is needed to cast each output to the desired format, and then assigning them in the target class, by calling the appropriate set method with the newly casted data as argument.

Furthermore, the functions' signatures were changed in order to comply to the standard signature of a start function as requested by the *pthread* API, which means that both *medianFilterMatMask* and *getBorderNormals* return a pointer to *void* and receive as argument a pointer to *void*. Internally, each function casts the input to the appropriate format, likewise it is proceeded in the main thread with the outputs of these threads.

However, here is a difference from the above-described case, when it comes to *medianFilterMatMask* function. As it takes two arguments, it is needed a supplementary mechanism for passing the required inputs, as the POSIX standard allows for passing exactly one argument to the thread function. This mechanism involves creating a C structure whose fields are exactly the arguments that must be sent to the thread and initialize those fields with the required data before passing them to the *pthread_create* function, thus the structure acts as a wrapper. In this case, the wrapper structure is called *medianfilter_thread_args* and is declared in *threads_data.h* header, under the *helpers* folder, so it can be seen in the main file (where output casting is performed) and in the *SIRfS_functions.cpp* (where the thread functions are implemented).

When passed to the thread creation function, the structure instance is casted to *void* type, hence the need of internal cast to the structure type, as mentioned above. Thus, by internal cast to an appropriate format in the case of *medianFilterMatMask*, it is meant a cast to *medianfilter_thread_args* structure type. This approach is a standard mechanism applied when working with POSIX threads that need to receive more than one argument data required for their internal processing.

As a consequence of using the POSIX API, the data instances whose values are computed by the previously-mentioned functions, are not passed as arguments anymore (as their initial implementation involved passing a pointer to these data, which is declared in the main function) but defined internally as pointers and then returned once their data is processed. By this, it is not required to embed the objects to hold the functions' output into the argument list of the thread functions.

5.2.2 4-threaded coarse grained parallelization

The second implementation experiment involved creating a thread for loading priors routine. The idea of the experiment was to preserve the three threads from the first implementation and add one for loading the priors. Since priors loading is the most time consuming processing step and it only loads data which is not required by the other threads, it was an indicator that running it in a separate thread could be feasible.

Since the loading priors routine is started by a non-static method of the *Prior* class (defined in *prior.h*, under the *block_A1* folder), it also required a mechanism for passing the arguments to the start function, relatively similar to the one described above. Therefore, a wrapper structure called *prior_class_wrapper* was created in the *prior.h* file. As one of its fields is an instance of the *Prior* class, it could not be created in the *threads_data.h* header, as the latter is included by *prior.h*.

However, creation of the wrapper structure *prior_class_wrapper* would not suffice in order to call the loading priors method. Since it is a method and the *pthread_create* function has a C implementation, passing the method as the third argument (start function, namely) to the latter would lead to error, because of the *this* parameter that comes with the method and which cannot be processed by *pthread_create* function. Thus, the method itself is wrapped in a C-style function, called *prior_thread_func*, also defined in the *prior.h* file.

Likewise any *pthread_create* start function, it takes as argument a *void* pointer and returns a *void* pointer. Internally, it dynamically allocates an instance of the *prior_class_wrapper*, as unique pointer, and initializes it by calling the structure's constructor. The constructor's parameter is the *void** argument casted to *prior_class_wrapper* type. Then, the function opens the MAT file, in order to retrieve the pointer to the root node of the internally stored structure, which is then passed as argument to the method that starts the priors loading. The method is called by dereferencing the *Prior* object from the *prior_class_wrapper* unique pointer. Finally, the instance of *prior_class_wrapper* is returned, so the initialized values for priors can be accessed in the main thread.

Lastly, a wrapper structure was created for the *getBorderNormals* thread's start function's arguments. The need for it comes from the fact that for all the created threads it is necessary to pass as argument the thread id, which is an integer different from the *pthread_t* id. Thus, the *getBorderNormals* thread gets a second argument. The wrapper structure is called *getBorderNormals_thread_args* and is declared in the *threads_data.h* header. Furthermore, this header is included in *SIRfS_Function.h* and *prior.h*, where the start functions for all threads are defined.

Moreover, for each of the previously described wrapper structures, a constructor was defined, as the C++ standard allows for that, because it regards a structure as a more generic type of a class, having all its members and methods as *public*, by default. This added to an easier initialization of the structures' instances.

The above-mentioned integer thread id is needed for keeping track of the thread's states and is used internally by the threads' start functions. The possible states are included in *thread_state* enumeration, declared in *enums.h*, which are: ALIVE, FINISHED and JOINED. Then, a structure called *thread* was created in *threads_data.h* header, having as data fields the thread state (of type *thread_state*, the *pthread_t* id and the thread's name (*char* pointer), necessary for the thread identification when it finished and needs to be joined. A pointer to this structure is declared, but not defined in the *threads_data.h* header, as it is later used as a vector of threads.

Additionally, the *threads_data.h* header contains the declarations of the mutex, condition variable and of two global counters: one that acts as a shared variable and counts the number of threads that finished their execution and another that counts the number of joined threads. All these four global data are defined in the main file. Also, the pointer to *thread* structure is allocated in the main file and hold data for all four threads. Since it was declared as *extern*, it is visible to the files where the start functions are implemented, like in the case of the four global data. Both the mutex and the condition variable are initialized using the macros described in the previous section. Thus, the mutex is a default one.

That said, the threads are created in the main file. Inside a loop, for each thread is set the state to ALIVE in the vector of threads, then, based on the loop index, a separate thread is created for each case, as there are three different start functions and all our threads have different arguments. Thereafter, for each of the considered case, an appropriate name is set in the vector of threads, with the *pthread_t* id field from the same structure being sent as argument to *pthread_create*, alongside the needed arguments, using the appropriate wrapper structures described above. Also, a dedicated clock is started for each case to measure the execution time.

Please note, when initializing the wrapper structures instances, for passing the thread id (which is the loop index) it is needed to dynamically allocate memory and assign its value to this new variable, which is then sent to the structure's constructor. Otherwise, the threads will get wrong indexes as they will read from the address of the loop's index later than it was assigned, thus getting a higher value.

Once the threads are created, the start functions begin their execution. Before returning, each thread locks the mutex in order to access to global vector of threads and change to state field to FINISHED. Also, each thread increments the global counter for the finished threads, then unlocks the mutex and sends a signal on the condition variable. Then, the threads are joined in the order they finished (preserving the order of signal receiving on the condition variable), procedure which represents the second feature of this second experiment.

Thus, in the main file was implemented a waiting procedure based on the schema illustrated in the previous section. Within a loop that checks whether the global counter for joined threads has reached the maximum number and guarded by lock and unlock of the mutex, is checked if the state of the global counter for finished thread has changed. As explained in the previous section, it is done within a while loop in order to make sure that the wait

function has received the signal sent by any of the threads. Once a thread signaled, it is verified in a for loop which thread was it, in order to retrieve its results in an appropriate way. That said, there is checked if the thread's status is FINISHED then is checked its name. The name is useful here as the id used to index the global threads vector might change, as another loop is used and the id received as argument by the start functions is not returned. For each of the four cases, the dedicated clock is stopped and the execution time is shown.

5.3 Verification and performance assessments

As mentioned in chapter 3, the incremental porting strategy entails the adaptation of the C++ SIRFS code for Linux - Ubuntu operating systems in order to leverage the POSIX *pthread* library efficiency for multi-threaded programs.

Thus, once the translation was performed, several compile and runtime errors were firstly fixed. As they were quite straightforward and easy to solve for a beginner-mid experience C++ programmer, they are not brought into discussion. Thereafter, an analysis of the obtained code was performed. Firstly, its validity was checked using the methods introduced in section 4.2.4. Once the verification results proved to be correct, the code's performance was assessed, with the results outlined in the table below.

Code section	Execution time in C++ on Linux (seconds)	Execution time in C++ on Windows (seconds)	Execution time in MATLAB (seconds)
Parameters initialization	0.001	0.001	0.009
Load priors	0.83-0.9	1.451-1487	0.224
Initialize "data" class members	0.0044 - 0.0046	0.005	0.002
Apply median filter	0.602 - 0.617	0.914 - 0.92	0.323
Build border normals	0.293 - 0.296	0.385 - 0.393	0.052
Total execution time	1.764-1.846	2.79 - 2.825	0.61

TABLE 5.1: Execution times per code section on Linux platforms

It can be observed a difference of approximately 1 second in execution time between the same code running on the two OS platforms. An explanation for this figures lies in the fact that the Linux system uses its native libraries and compiler to run the program, whereas Windows uses an adaptation of those, such as MINGW.

The direct use of system calls in the case of Linux avoid calling of wrapper functions that intermediate the call of the desired behavior, therefore the speed improvement in performing several operations such as memory allocation and file creation. In order to support this idea, this study [33] presents several performance benchmarks on the major OS platforms, comparing their efficiency when performing certain operations, such as the ones mentioned

before, plus process and thread creation and launching. The results show that the considered Linux platforms always performed better than the Windows ones. That said, all these results support the decisions highlighted by the incremental porting strategy described in chapter 3.

Furthermore, it can be observed that the execution time for the priors loading code section almost halved, with the execution times for median filter processing decreasing by almost 35%, whereas the execution times for the other code section had quite small decreases. This happened as the first two code sections involve many dynamic memory allocations when compared to the others, as they entail working with big matrices, even those that model the MATLAB sparse matrices, fact that is a direct consequence of the above-discussed performance in performing certain operations on Linux and Windows platforms.

On the other hand, similar execution time decreases should not be expected for the build border normals code section, as its implementation involves the calling of a series of smaller functions, one for each separate logic. Thus, it is an indicator that this section's internal fine-grained modularity prevents a significant boost of its runtime performance and, subsequently, there is more room for a modularity trade-off with respect to its implementation.

The first multi-threaded implementation did not involve any synchronization, as the threads were joined by the main thread after they were launched, in order to retrieve their results. In other words, the calling order of the above-named functions was preserved, but it involved separate threads for executing them. As it can be seen in the table below, there were not any performance gains as the total execution times were about 1.75 - 1.83 seconds, that entail a performance drop of 1%. This fact lead to the following question: can it be done better by using a more efficient joining method?

Code section	Execution time in C++ (seconds)	Execution time in MATLAB (seconds)
Parameters initialization	0.001	0.009
Load priors	0.83 - 0.9	0.224
Initialize "data" class members	0.0043 - 0.0045	0.002
Apply median filter	0.607 - 0.620	0.323
Build border normals	0.301 - 0.335	0.052
Total execution time	1.773 - 1.867	0.61

TABLE 5.2: Execution times per code section on Linux using 3 pthreads and no joining method

As presented in the previous section, a second experiment was conducted in order to test the runtime performance when four threads are run, including the performance assessments of the joining procedure presented in section 5.1.2. The results are presented in the table underneath and show that this approach performed worse than the previous one, with total execution times around 1.95-2.25 seconds, that make it slower by 0.2-0.5 seconds that the first implementation, which represents a 16.5% performance drop.

Code section	Execution time in C++ (seconds)	Execution time in MATLAB (seconds)
Parameters initialization	0.001	0.009
Load priors	2.1 - 2.3	0.224
Initialize "data" class members	0.0043 - 0.0046	0.002
Apply median filter	1.62 - 1.96	0.323
Build border normals	1.3 - 1.9	0.052
Total execution time	1.95-2.25	0.61

TABLE 5.3: Execution times per code section on Linux using 4 pthreads and a joining method

It can be noticed that the total execution time is the sum of the execution times for the second and the third code sections illustrated in the table. Also, though the times for each individual thread are higher than in the table 5.2, since the threads run in parallel it is not a big problem, as long as the maximum time for a thread (that almost give the entire execution time) is around 1.75 seconds, that is the execution time for the first implementation.

However, an explanation for the above execution times can lie in the fact that while joining one thread and retrieving its result (which involves type casting and assigning it to a class), another thread just finished and needs to be joined, but it has to wait for its predecessor to be joined. Nevertheless, the measurements have shown that those waiting times summed up to 0.1 seconds, so the hypothesis is negated.

On top of that, other two experiments were conducted. For the first one, the joining method was eliminated and the threads were created and joined sequentially, thus avoiding mutex operations and signaling on condition variable. In other words, to the first implementation described in the previous section was added an extra thread for loading the priors, keeping the idea of having four concurrent threads, except for the main one. However, the execution times were almost similar to those shown in table 5.3.

Then, for the second experiment, it was preserved only one thread, namely the one responsible for loading priors, whereas the median filter and border normals were computed sequentially, in the main thread, as they had been before the CPU multi threaded implementations were started. The reason behind this experiment was to determine whether the application performs better with less threads, as a conclusion from all previous implementations could be that the SIRfS code performed better with three threads instead of four.

Another reason comes from the fact that the priors loading execution time is almost the same with the execution time required for processing median filters and border normals together, so processing them in parallel seemed to be a good idea. In addition, for this experiment were tried two versions: one that excludes the joining scheme and one that includes it, so it can be seen if the latter would really come as performance enhancement. Nevertheless, the execution times were around 1.8 - 1.9 seconds, which makes this approach slightly faster than the previous two experiments, but slower than the initial multi threaded approach.

Thus, the explanation for having higher execution times in multi threaded approaches can come from the fact that for each thread it is required a context switch both when it starts and when it ends its execution, mainly for allocating stack memory and for transferring data arguments and the processing results. Hence, the conclusion is that using CPU coarse-grained parallelization, namely by applying the implementations outlined in the previous chapter and their variations described above, do not help in improving the runtime performance of SIRfS, so this approach is not accepted as a solution in the SIRfS C++ code.

Chapter 6

GPU parallelization

In the previous chapter were presented several CPU multi threaded approaches for the *SIRFS* C++ implementation, alongside an identification of code segments amenable to CPU parallelization and a theoretical description of POSIX threads, which includes a reasoning on why choosing threads over processes for the CPU parallelization implementation, amongst others. The results shown that any of the presented CPU coarse-grained multi-threaded implementations did not help in enhancing the code's runtime performance.

That said, the last approach considered for enhancing the *SIRFS* C/C++ implementation's runtime performance is the GPU parallelization. It is built atop of the CUDA API which leverages the parallel architecture of NVIDIA graphics card, which is suitable for processing not only visualization-based algorithms but also complex general purpose algorithms. Therefore, this approach is used as a solution for a fine grained parallelism and aims to find a suitable implementation for the fifth filter illustrated in the figure 2.2.

Therefore, the rest of the chapter is organized as follows: the first section presents the CUDA programing model and several hardware considerations and limitations. Next, the second section explains the CUDA API functions that were used for the implementation outlined in this chapter, which is followed by the description of these implementations. Lastly, the performances of the implemented solutions are assessed.

6.1 Programming model and hardware considerations

In order to be able to write programs that run on GPU, first, its hardware architecture must be understood, with the scope to grasp how certain API functions leverage the parallel architecture of the GPU. Thus, in this section are presented the main hardware considerations that need to be taken into account when using the CUDA API.

First, one has to understand what is meant by "parallel architecture" of the GPU. It is known that nowadays the CPUs are multi-cored, having between 2 and 18 cores [34], architecture that allows to parallelize a program by mapping a thread per each core, thus allowing more tasks to execute in parallel. Sure, by using hyper threading [35], there can exist 2 threads per core, in the sense that when one thread stalls the other one is immediately executed on the same core, with as less time penalty as possible.

Nevertheless, when working with big data or sparse matrices, like in the case of SIRFS, parallelizing their computation on CPU using at most 36 threads (if one avails a 18 core

CPU with hyper threading enabled), that is the maximum number of threads that the CPUs state-of-the-art allows for [34], still does not represent the finest parallelism that would suit the dimensions of these problems.

Concretely, considering the CPU details described in appendix A, it can be noticed that the CPU used for the development of this project allows for having maximum 8 concurrent threads, with hyper threading enabled. This means that a possible solution for any big matrix or vector that is envisaged for CPU parallelization involves the division into 8 blocks (groups of elements in the given matrix or vector) that are executed in parallel. If the parallelization strategy involves working at element level, it would entail a certain sequentiality, as at most 8 elements being processed at a given time (if hyper threading is considered to introduce negligible time penalties, otherwise there would be 4 elements processed at a given time), with the remaining threads waiting to be processed. Still, this approach would not represent the finest suitable parallelism for computing big data sets.

However, by looking closer at the hardware architecture of a GPU, it can be noticed that it "devotes more transistors for data processing" [36] than the CPU, as illustrated in figure 6.1, beneath. Specifically, considering the hardware of the GPU used for the development of this project, whose main specifications are presented in appendix A, it can be observed that it has 768 cores, hence the increased number of transistors dedicated for processing. Clearly, this architecture allows for a finer-grained parallelism than on CPU, as one can divide a given problem into multiple blocks, each one processed by a thread, with each thread being mapped to one core.

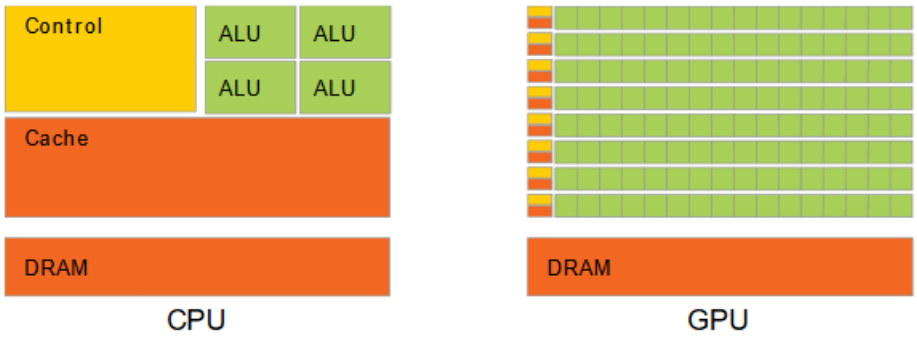

FIGURE 6.1: Architecture comparison of CPU and GPU [36]

Nonetheless, an even finer-grained approach could be possible, as each element of the given matrix or vector can be processed within one thread. Like in the same scenario considered for CPU parallelization, outlined above, if the dimension of the problem exceeds the number of physical cores, not all elements could be processed at once, thus involving a certain sequentiality in processing. However, the number of the elements processed at a given time is higher that in the CPU parallelization scenario, therefore the sequence would be smaller. Therefore, this clearly illustrates that the GPU processing of a big matrix or vector is faster when compared to a CPU multi-threaded approach, hence the motivation behind the idea of parallelizing the SIRFS algorithm on GPU.

As mentioned above, different levels of parallelism, in terms of graining, are possible on GPU, likewise it can be done on CPU, as mentioned in chapter 5. Nonetheless, on GPU the parallelism is finer-grained than on CPU due to the previously highlighted hardware properties. Furthermore, unlike the CPU parallelism, the GPU one uses only threads, as they coexist at the level of the same process that runs on CPU, with the GPU being seen as a co-processor. Each such thread is mapped to one GPU core. Next, depending on the desired graining, the total number of threads can be equal to the total number of elements in the target matrix or vector, or less than this number, as more elements can be grouped in data blocks, with each data block assigned to one thread, but this represents a rarely used approach. Thus, each such thread executes the same instruction on different input data, following the SIMD pattern, requiring less flow control and overcoming the memory latency issue "with calculations instead of big data caches" [36].

In order to process the target matrix or vector in parallel using the GPU, the programmer has to specify the desired number of threads. As already mentioned, this is usually equal to the number of elements in the considered data set and is referred to as "grid", in the CUDA programming model. Furthermore, the threads' grid is further divided into blocks, each such block consisting of multiple threads. That said, the programmer must specifically provide the number of threads per block, and based on this number and on the dimension of the problem, the number of blocks per grid can be inferred.

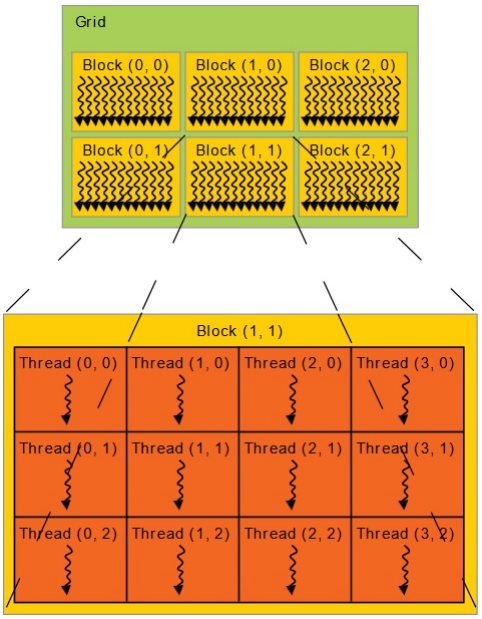

FIGURE 6.2: CUDA threads hierarchy [36]

In order to match the dimension of the problem, the blocks and, implicitly, the grids, can have one, two or three dimensions, thus each block has an unique ID within the grid, with a thread having an unique ID within the block. Therefore, the global ID of a thread within a grid can be inferred by using the thread's ID in the block and its block's ID within the grid. In the figure 6.2, shown above, is illustrated an example of a bidimensional grid, consisting of bidimensional blocks, together with their afferent IDs, in order to grasp the thread hierarchy

entailed by the CUDA programming model.

Here, it is worthy to mention that the maximum number of thread per block is 1024 [36] and usually a number grater than or equal to 32 of threads per block is preferred, as the GPU's scheduler considers groups of 32 threads for processing at a given time, thus achieving an efficient thread scheduling. This groups are called warps [36].

Furthermore, in figure 6.1 it can be noticed that the GPU's cores are organized in so called "lines", with each line having own (local) memory, illustrated with small orange and yellow rectangles. These "lines" are called streaming multiprocessors (SMX) and represent groups of cores that have access to the local memory that is smaller, but faster than the GPU global memory. The threads within a block are scheduled on such an SMX and they can share data amongst them using the mentioned faster memory, called shared memory [36].

Thus, the shared memory only allows the threads within the same block to exchange data, but does not allow data transfers between the threads belonging to different blocks. In order to share data between all threads, the global memory is used, which is available for all the threads within the grid. Next to that, each thread can access local memory that is proper to each core, and which is represented by registers. These different types of memory compose the memory hierarchy proper to the CUDA programming model, that is illustrated in the figure 6.3 below.

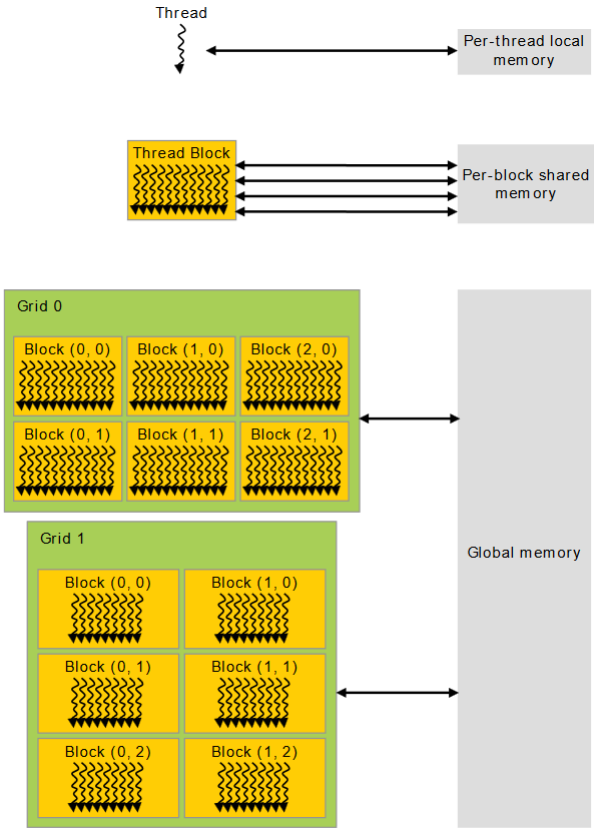

FIGURE 6.3: CUDA memory architecture [36]

Furthermore, GPUs have different hardware, as newer ones have bigger memory and more cores. Besides these, the newer cards entails more features. In order to determine which

features are available on a given graphics card, they come with a version number, called compute capability. This is number used by applications, at runtime, to determine the available features and instructions of the used GPU [36].

As illustrated in appendix A, the compute capability of the used GPU is 3.0. This brings along several hardware limitations, the most important being that is it does not offer support for dynamic parallelism. Dynamic parallelism is a feature that allows the GPU to "create work" directly on GPU, leading to a nested parallelism architecture. Concretely, the threads within a grid can represent grid of threads that involve a finer grained parallelism when compared to the first grid's one's. Thus, the idea of using the finer-grained GPU parallelism withing the coarse-grained CPU parallelism was meant to act, to a certain extent, as a workaround for the lack of dynamic parallelism.

6.2 CUDA

CUDA (Compute Unified Device Architecture) is an API accessible to anyone who wants to program on the GPU cards produced by NVIDIA. It is mainly documented in [36] and consists of different functions that allow the programmer to leverage the parallel architecture of a GPU, depending on its compute capability, as explained before.

When using the CUDA API, the program that runs on the GPU is written in special files that have *.cu* extension. Within this files are written dedicated functions that run on GPU, called *Kernels*. A kernel is a function containing instructions executed by each thread in the grid, thus achieving SIMD architecture.

Moreover, the CUDA API offers different declaration specifiers, used for data and functions, which are briefly described below, based on details provided in [37]:

- **__global__**: It is used to declare kernels, showing that the function will execute on GPU.
- **__host__**: data is declared on CPU, which is referred to as host in the CUDA programming model. In case of functions, shows that they execute only on CPU.
- **__device__**: data is stored in GPU memory, which is referred to as device in the CUDA programming model. In case of functions, shows that they execute only on GPU. It cannot be used together with **__global__**, but can be used together with **__host__**, meaning that the function is callable both from CPU and GPU.
- **__constant__**: used to declare data on GPU's constant memory, which is global, but used only for data that is not modified during the GPU processing.
- **__shared__**: used for declare data in shared memory and which has the lifetime of a thread block, being accessible only to threads within the considered thread block

Nevertheless, the kernels are not directly callable from a .c or .cpp file and their call needs to be embedded in a function, called launcher or wrapper. This function can act simply like

a wrapper or can contain the operations necessary for a kernel to execute. These operations compose the context creation for the kernel processing, as they specify where the kernel can find the input data, by dynamically allocating memory at addresses accessible to the given kernel. Furthermore, the context creation involves copying data from the considered data sets to these memory addresses, and once the kernel finished the computation, to copy it back to the data sets used by the program. Finally, the memory addresses used by kernel are freed. All these operations happen in a given order and they entail using of specific API functions, briefly described below, based on the information in [36].

1. **Device/host pointers declaration**. This step involves declaring pointers of an appropriate data type that matches the data types used by the program. If these data types are complex, such as own defined classes or structures, it is preferred to define those both on CPU and GPU.

2. **Device/host memory allocation**. A widely used approach when programming with CUDA is to dynamically allocate memory on GPU's memory, using the pointers previously defined. This is done using *cudaMalloc* function and represents linear memory allocated in the global memory of the GPU. The function takes as input the pointer where the starting address of allocated memory is stored.

   However, the performance of this step can be increased by using page-locked memory on host. Sure, using too many locked pages is not desirable as there would not be many pages left for the operating system to use for swapping, thus affecting its performance [38]. Page-locked host memory is allocated using *cudaHostAlloc* and besides the input pointer it takes an extra input, which is a flag, that defines the behavior of the function. The ones used in this project are:

   - CudaHostAllocDefault: default page-locked allocation on host
   - CudaHostAllocWriteCombined: frees the L1 and L2 CPU cache allowing for faster memory transfers, when the kernel automatically reads data from it (the data transfer is managed by the driver, not by programmer). [38]
   - CudaHostAllocMapped: maps host memory into device memory, thus the page-locked memory having two addresses, one on CPU and one on GPU, with the last one retrieved using the function *cudaHostGetDevicePointer()*. The data transfers are performed by the kernel and the atomic CUDA operations are not atomic from the CPU perspective. [38]

     Nevertheless, since it is a newly allocated memory, it might be required to copy data from the data sets (considered for computation) into this page-locked memory. In other terms, it involves copying data from one address in CPU RAM to another address in CPU RAM. This mis the case when using CudaHostAllocMapped, as well. A workaround to this could be the direct use of page-locked memory allocated with *cudaHostAlloc* for the data sets that are considered for GPU parallelization.

3. **Data transfer from host to device**. If the memory of the data sets used for GPU computation is allocated on GPU, this memory needs to be initialized, so it holds the values of the corresponding data sets used by the CPU computation. In other terms, the values required by the GPU computation must be transferred from CPU RAM to GPU RAM. This is done by calling *cudaMemcpy* function. It takes 4 inputs, as follows: the address of the destination memory (the address in GPU global memory, in this case), the address of the source memory (the address in CPU RAM, in this case), the size of the copied memory (the size of matrix or vector) and the flag *cudaMemcpyHostToDevice*, which tells the function to copy data from host to device.

4. **Setting threads' grid and block sizes**. In order to specify the sizes of the threads' grid and blocks, a structure called *dim3* can be used. It allows to specify the sizes for all 3 dimensions, but it works with one or two dimensions, as shown in the example below, that allocates a bi dimensional block of threads.

   ```
   dim3 block_dimension(16, 16);
   ```

5. **Launching kernels**. Once the number of threads per block, and blocks per grid are specified, the kernel can be executed using the syntax shown beneath.

   ```
   my_kernel<<<numBlocks, threadsPerBlock>>>(input_data);
   ```

6. **Data transfer from device to host**. If the memory of the data sets used for GPU computation is allocated on GPU, once the computation is done, the results will reside only in GPU's memory, but not in the CPU RAM, thus it is needed to copy the results from device to host. This is done by calling *cudaMemcpy* function. It takes 4 inputs, as follows: the address of the destination memory (the address in CPU RAM, in this case), the address of the source memory (the address in GPU RAM, in this case), the size of the copied memory (the size of matrix or vector) and the flag *cudaMemcpyDeviceToHost*, which tells the function to copy data from device to host.

7. **Free memory**. Regardless the memory is allocated on host or on device using the previously-described CUDA functions, once the computation is done, the memory must be freed. If page-locked host memory is used, the function *cudaFreeHost()* must be called for freeing it. Otherwise, if device memory is used, for freeing it is called *cudaFree()*. Both functions take as input the pointer holding the address of the memory to be freed.

6.3 Implementation

The GPU implementation entails multiple steps, each of them involving implementations for different code segments within a given code section. Here, by code section is are referred the ones mentioned in table 2.3, with focus on the last two, namely "Apply median filter" and "Build border normals" as they are the ones amenable to GPU parallelization. A code segment is a portion of code within a given code section. With each implementation it is discussed whether it contributes to improving the runtime performance of SIRFS C++ code,

and if it hampers the execution performance, then it is aimed at finding the reasons for that. That said, these steps are outlined in the subsections beneath.

Furthermore, as already mentioned in section 6.1, by using this GPU fine grained parallelism within the coarse grained one, outlined in chapter **??**, it was meant to have different levels of parallelism, with each level aiming at further optimizing the SIRFS code. Nonetheless, as specified in chapter 5, the coarse grained parallelism proved not to be an optimal solution with respect to the goal of this project.

Therefore, within this section is aimed at trying different implementations entailing slightly different grained GPU-based parallelism, when compared with each other, but not as coarse as the ones outlined in the previous chapter. The purpose of this approach is to capture the graining level that is the most optimal regarding the C/C++ SIRFS code.

6.3.1 GPU parallelization of 2D matrix template class methods

In section 4.1 is discussed the impact on the runtime speed that the implementations of 2D and 3D matrix template classes do have. Also, several improvements were presented with respect to the matrices' containers and methods implementations, which have a positive impact on the entire code's execution performance.

Since in this chapter is aimed at finding the functions, methods and code segments amenable to GPU parallelization, the first one that is brought into discussion is the implementation of the 2D matrix template class, that is the most used data set by the SIRFS algorithm, as already mentioned in section 4.1. Thus, the focus is on the class methods which are widely used throughout the code, as a performance upgrade of any method will positively reflect into the entire code's execution speed.

Before performing any implementation for this step, the methods amenable to GPU parallelization were identified. Since the majority of the methods perform operations with a reduced complexity, such as adding two matrices or computing the logarithm for each element of the matrix and returning the result matrix, only a few were considered as suitable for GPU parallelization. The reason for this is quite simple: the methods with reduced complexity do not entail multiple nested loops or do not perform multiple successive calculations on the same input matrix data in order to require a fine grained parallelism using CUDA.

Therefore, only 3 methods were considered as amenable to GPU parallelization, that are: *conv2DFull*, *conv2DValid* and *conv2DSame*. These methods perform the convolution product between the input filter and the input matrix. However, their implementations differ, as each one matches the different behaviors that the MATLAB *conv2* function has when called with different input parameters (full, valid and same), which are described in [39].

Thus, the first design decision was to preserve the CPU implementation for each of the methods mentioned above and add a CUDA based one within the same method. The coexistence of both implementations in the same method implies the calling of one or another

through a boolean that is passed as a third argument (so, an extra parameter was added to the methods' signatures), whose value needs to be *true* or 1 for running the CUDA-based implementation, 0 (false) otherwise.

Thereafter, it was created the context creation required to process the CUDA kernel. As mentioned in section 6.2, by context creation is meant the declarations and the memory allocations, on device or host, for the data required by the CUDA kernel computation, both for inputs and outputs. Moreover, the context creation entails the data transfers of inputs and outputs between the CPU RAM and GPU RAM, as well as freeing the dynamically allocated memory, either on host or device.

For the considered 3 methods it was preferred a standard memory allocation using *cudaMalloc* to the ones that use page-locked host memory. As mentioned in section 6.2, the reason behind this decision lies in the fact that using to many locked pages can lead to a significant slow down of the operating system due to the shortage of memory pages available for swapping [38]. Furthermore, it is hard to predict, at this time, how much of the future-to-be parallelized SIRFS C/C++ code will require page-locked host memory and by using it within the 2D matrix template class implementation, which is widely used, can hamper the design decision process for its developments in relation with the operating system's performance.

In other words, it is suggested that page-locked memory should be used for the implementation of the functions proper to SIRFS algorithm, for which it can be known that are not called too often and, therefore, not leading to a significant and hard to predict decrease of the available page-locked host memory. On top of that, since the code section used during the development of this project represents the first 18% of the SIRFS computation, an even higher-demanding computation resides in the remaining 82%, for which the usage of page-locked memory on host might add a performance boost during the GPU parallelization process.

Next, the final step before calling the kernel is represented by the specification of the threads' grid and blocks dimensions. Here were preferred two dimensional thread block and grids, as they easily match the dimensions of the convolution problem. For the block size, it was chosen a standard dimension of 16-by-16 threads, which is a common choice according to CUDA programming toolkit [40].

For computing the grid size, the block size is used alongside a small mathematical trick, illustrated below, which ensures that sufficient threads' blocks are allocated in order to match the size of the problem. The trick is needed as in most of the cases the dimension of the input data does not have a value that is an exponent of 2. Thus, as threads are scheduled in warps of 32, the number of the allocated threads is always a multiple of 32. Therefore, the trick involves allocating more threads than required by the input data size and to verify inside the kernel if the index of the thread is within the input data size.

```
grid_size_X = (input_size_X + block_size_X - 1)/block_size_X
```

Once everything is set, the CUDA kernels can be launched for execution. Nonetheless, the CUDA kernels are not called directly, as it is impossible from a .h, .c or .cpp file, but only from .cu files, whose extensions are identified by the NVIDIA compiler and the syntax is recognized. Instead, as mentioned in section 6.2, functions called wrappers or kernel launchers are used to call the kernels, which are declared in the file where they are called or in a header file included by the former, but defined in the .cu file.

For the 2D matrix template class was created a file where the kernel launchers and the kernels are defined. The file is called *cuda_Matrix2D.cu* and can be found under the *templates* folder. Within this file the 3 kernels are defined with the following names: *conv2D_full*, *conv2D_valid* and *conv2D_same*, respectively. Their internal implementations closely follow their CPU implementations as the parallelization is done at the level of the loops that iterate through the input matrix. Thus, the matrix is indexed using the global indexes computed using the CUDA structures that allow for retrieving the thread index, the block index and the block dimensions. These indexes are checked with respect to the sizes of the convolution output matrix, in order to avoid out of bounds indexing.

Since the launchers are called by the methods of a template class, they are defined as template functions, likewise the kernels. Nevertheless, in order to call the launchers for a given data type, they must be specialized [41]: each template launcher is instantiated with the desired data types as the linker cannot link the template definition with the launcher's call for a given data type, as they are defined in different files. Each launcher is specialized for the major data types: int, float and double. This solution helps in writing less code, as the same behavior is used for any needed data type. Here, the role of the launcher is simply to call the kernel and to pass the necessary arguments to it, as the context creation takes place inside the methods. It also calls *cudaDevieSynchronize()* for making sure that all CUDA operations have finished before it call *cudaGetLastError()* to return the error thrown by the latest CUDA call.

6.3.2 Conversion of STL vectors into raw pointers

Before identifying the code blocks amenable to GPU parallelization, within the *Apply median filter* and *Build border normals* code sections, their internal implementations needed to be analyzed. Thus, the initial implementation used STL vectors as a solution to store array values. The former were mainly preferred as they allowed for a faster code development.

However, in order to make an easier integration with the future-to-be-implemented CUDA code, the vectors whose size can be determined at declaration time, were converted into raw pointers. This can be seen as GPU parallelization preprocessing step, as passing the raw pointer to *cudaMemcpy* is easier and more straightforward, with respect to the function's implementation presented here [42]. Sure, STL vectors can still be used with *cudaMemcpy*, as the address of the first element in vector can be passed to the function in the following manner.

```
cudaMemcpy(device_vector, &vector.front(), sizeof(Type) * input.size(),
```

```
cudaMemcpyHostToDevice);
```

Additionally, as mentioned in section 4.2, where the impact on STL vectors on matrix template classes performance is discussed, the STL vectors proved to have a negative impact on runtime performance, when compared to raw pointers.

Thus, the motivation for changing the STL vectors into raw painters was twofold. Once done, it was proceeded next, in order to identify the code blocks amenable to GPU parallelization within the above-named code sections.

6.3.3 GPU parallelization approach for the *Apply median filter* code section

After applying the improvements described in the previous two subsections, the *Apply median filter* code section's execution time dropped below the execution time of the afferent MATLAB code block, as shown in section 6.4.2. Thus, the objective of improving the runtime performance of this code section was already achieved.

Nevertheless, the study continued aiming at determining whether a CUDA-based implementation would contribute to a further boost of this code section's execution speed. That said, it was looked for a one of the most computational intensive code segments. By analyzing the code, it can be noticed that inside the *medianFilterMatMask* function is called another SIRFS function, named *conv2mat*. The call occurs within a loop and is followed by more sequential operations. Since the internal implementation of the latter is much bigger when compared to the other functions called by *medianFilterMatMask*, its improvement seems an interesting goal, at this point.

Having said that, the analysis continued with the *conv2mat* function, and it was envisaged the discovery of the most computational expensive code segment. Again, the calculations which take place within loops were analyzed and the only 3-level nested loop was preferred to the other single and double nested loops. Here, the first two loops proved to have reduced sizes (below 15 iterations in total), whereas the third one iterates through all values of the input image's mask.

Since the compute capability of the graphic card used during the experiments is 3.0, it does not support dynamic parallelism, thus it was impossible to create nested kernels to fully match the 3-level nesting of these loops. Thus, giving the size of the inner loop, its parallelization was considered as computational demanding, therefore amenable to GPU parallelization.

Thus, all the operations within the above-mentioned loop were included in a CUDA kernel. The kernel is called *conv2mat_kernel* and is implemented in the *blockA2_cudacode.cu* file, which is placed in the *block_A2* folder. Since it is a linear loop, for the threads' block and grid sizes were preferred linear dimensions, as a block size was set to 64, allowing for a finer parallelism, due to creation of multiple thread blocks.

The grid size was computed based on the formula shown in subsection 6.3.1. Similarly to the case presented in subsection 6.3.1, here are allocated more threads than required by the

input data size and the linear index used to access the input data values is checked with respect to the input data size, so not to encounter an out of bounds indexing.

Moreover, shared memory was used to store the intermediate vector results. When using shared memory on CUDA, the size of the vector or matrix has to be known at compile time and must be a constant. Normally, a macro is defined and used. The value assigned to the macro value is the dimension of a block of threads, as a shared memory is accessible at the level of such a thread's block. Sine the access to shared memory is faster than to global memory, it is expected to add to runtime performance of the kernel.

The kernel's launcher (named *launch_conv2mat_kernel*) not only calls the previously described CUDA kernel, but it also handles the required CUDA memory allocations and deallocations, as well as the data transfers between the GPU RAM and CPU RAM. Here, it was preferred a standard memory allocation using *cudaMalloc* given the context of the kernel's execution that involves nested loops and the execution of its caller, namely the *conv2mat* function, within another loop. Thereby, it was avoided the usage of too much page-locked memory, leaving more available for the next experiments.

The kernel launcher is declared in a header file called *kernels_wrappers.h* that can be found in the *block_A2* folder. The launcher is then defined in the *blockA2_cudacode.cu* file and called in the *SIRFS_Functions.cpp*, where the *Apply median filter* code section resides, thus the header file being included by both previously-named code files. In addition, the launcher takes as inputs references and pointers to objects and data structures, respectively, used in the CPU code, in order to avoid copying data when passing the matrix and vector parameters [43]. Thus, the launcher directly reads the data stored in the input data structures when copying it into the device memory.

In *SIRFS_Functions.cpp*, both CPU and CUDA-based implementations were kept, as the latter can be used by enabling the macro called *CUDA_KERNELS*, which is defined in SIRFS_Functions.h. This allows for switching in between the implementations and, therefore, for assessing their performance by comparing them.

Given the time and the hardware resources constraints, it is considered that the above-described implementation would suffice in order to grasp whether a CUDA implementation at the level of this code section would help in achieving the goal of this project. On top of this, taking into consideration that this code section is already faster than its MATLAB equivalent, the remaining time resources were fully allocated for finding improvements with respect to *Build border normals* code section, that is still slower than its MATLAB counterpart, as it can be seen in subsection 6.4.2.

That said, in the next subsection are presented the implementations that aim at speeding up the execution of the *Build border normals* code section.

6.3.4 GPU parallelization approach for the *Build border normals* code section

Once the optimizations described in subsections 6.3.1 and 6.3.2 were implemented, it was noticed that the *Build border normals* execution time is still higher than the afferent MATLAB code segment's runtime, as shown in section 6.4.2. Thus, the main goal at this point is to find whether and how a CUDA-based implementation can help in enhancing the runtime performance of this code section and, implicitly, of the entire C/C++ SIRFS code.

That said, the next step aimed at identifying at least one code segment within this code section that is amenable to GPU parallelization. By analyzing the code, it was observed that computation of the normals represents the most computational expensive code segment, as it involves the most successive computation steps which, on top of that, take place within a loop.

Thus, the first attempt was to port the entire loop into a CUDA kernel, with each thread executing one iteration of the CPU loop. Sure, this approach is not the finest-grained possible parallelization and it proved to be impossible to implement due to memory limitation of the GPU. Concretely, the considered operations required the device memory allocation of multiple matrices to store the intermediate results which proved to exceed the available amount of GPU memory. Nonetheless, this approach was tried based on the following statement: **"Applications should strive to minimize data transfer between the host and the device. One way to accomplish this is to move more code from the host to the device, even if that means running kernels with low parallelism computations. Intermediate data structures may be created in device memory, operated on by the device, and destroyed without ever being mapped by the host or copied to host memory."**, found in NVIDIA's CUDA toolkit documentation [36].

Given the statement outlined above, the attempt of implementing "kernels with low parallelism computations" [36] continued, but it divided the computations within the considered loop, into smaller pieces, so they can fit within a kernel, in the sense that their processing does not exceed GPU memory. Concretely, the approach first involves the implementation of a CUDA kernel that computes the first steps of the loop that can be handled with the given amount of memory. Thereafter, the selected processing steps for parallelization were included in a kernel called *getBorderNormals_computePatch*, which resides in the *blockA2_cudacode.cu* file. Then, the kernel's results are stored in arrays (array of matrices and an array of integers, namely), as the remaining computation steps that are still processed on CPU need to iteratively access this data, within the previously-mentioned loop.

Having said that, the dimension of the thread's grid was set to be equal to the loop's size, and the kernel is called once, outside the loop. As for the tested cases, it was noticed that the loops size does not exceed more than 1000 iterations, a finer grained parallelism was preferred by creating many thread blocks, as the size of such a thread block was set to 64. If a 256 size had been used, it would not have led to a full occupancy of the graphics card, as only 4 thread blocks would have been created. Furthermore, shared memory could not have been used, as each thread is responsible for computing one output matrix, so each thread

needs to use global memory to store the results, as the amount of shared memory is way more limited.

Likewise the case described in the previous subsection, the kernel is executed within a launcher, that is declared in the header *kernels_wrappers.h* that can be found in the *block_A2* folder, and that is defined in the *blockA2_cudacode.cu* file. Also, the header performs all the required operations for creating the context necessary for the kernel's execution, that are device memory allocations and deallocations, and host memory initialization. Since the kernel is called only once, it was considered that using page-locked memory is suitable for this case, in order to speed up the memory transfers between host and device. Here were tried two scenarios: using *cudaHostAllocDefault* flag, for default allocation, and *cudaHostAllocWriteCombined*, described in section 6.2. Since no noticeable performance differences were observed, the first approach was preserved.

Then, a second kernel was implemented. It contains the next processing steps from the above-mentioned loop, that could be processed given the device memory constraints. As it was clear that the entire loop cannot be parallelized at once, the aim of this implementation was to determine if dividing its internal processing into multiple kernels would be more efficient when compared to the current CPU version.

The kernel is called *borderNormals_compute_temp_a* and is defined in the *blockA2_cudacode.cu* file. As its computation requires memory allocation outside the kernel, that is within the launcher, it was not the case to use shared memory, as no internal memory allocations are performed for storing the intermediate results. Next, like in the case of the previously described kernel's implementation, the size of the thread blocks was chosen to be as 64, to allow for creation of more thread blocks, thus leading to a finer grained parallelism. However, in this case the blocks and the grid were set as bidimensional, in order to match the scope of the output data, whereas in the previous situation one dimensional threads' blocks and grid were suitable for its scope.

As in previous case, the kernel is executed within a launcher that is declared in the header *kernels_wrappers.h* and that is defined in the *blockA2_cudacode.cu* file. The launcher is named *launch_borderNormals_compute_temp_a*. Similarly, the launcher performs all device memory allocations and deallocations, as well as data transfers between device and host. Since this CUDA kernel is called within a loop, page-locked memory was not used due to reasons outlined in section 6.2 and in subsection 6.3.3.

The third and the last implemented kernel envisaged a subset of the sequential operations that precede the first kernel's execution. Not all operations were suitable for GPU parallelization due to a separation of concerns amongst them, as they process different data, of different dimensions. Thus, the selected subset of operations involves processing of matrices that are big enough (more than 1000 elements) in order to justify their GPU parallelization. They contain several basic matrix operations, such as addition, mask creation and comparisons, which take place in sequence. They are not called within a loop, nor is the function within they are computed.

That said, the aim of this implementation was to find if a CUDA-based implementation would enhance the code's performance when it is not called in a loop, or in another function, or it does not envisage the parallelization of a loop, as it was the case with the previous two kernels described in this subsection and with the implementation presented in subsection 6.3.3.

The kernel is called *borderNormals_compute_maskedP* and is defined in the *blockA2_cudacode.cu* file. For all data required by its computation, namely input, outputs and intermediary results, it was chosen an allocation based on page-locked memory, using *cudaHostAlloc*. On top of that, the flag *cudaHostAllocMapped* was used for leveraging the advantages of mapped memory, namely limiting the amount of data transfers between host and device, as the kernel uses the memory allocated on host that is mapped in the device virtual address space, as explained in section 6.2. Sure, for input and output it was still necessary to allocate memory, and since these memory needed initialization, *cudaMemcpy* was used to transfer data to it, from the C++ objects.

These computation steps are part of the context creation of the discussed kernel and are included in the launcher called *launch_borderNormals_compute_maskedP*. Moreover, since internally each thread needs to update a counter by using atomic operation, for storing it it was used device memory, as the atomic operations on mapped memory "are not atomic from the point of view of the host or other devices" [38]. Also, for initializing the counter, a kernel whose threads grid and block dimensions are set to 1 was called before the above-named kernel.

Likewise it was proceeded with the previous two kernels, the thread block dimension of this kernel was set to 64, in order to allow for a finer-grained parallelism. On top of that, similarly to the previous two kernels, the grid dimension was computed using the procedure shown in subsection 6.3.1. Next, no shared memory was used, as all intermediary results are stored in the mapped memory.

6.4 Verification and performance assessments

The previous section describes different implementation steps that entail GPU parallelization of SIRFS algorithm, using the CUDA API. The reason for having these multiple steps lies in their impact on the execution speed of the code. Thus, in this section is discussed the impact each of the above presented steps has on the C/C++ SIRFS code. The performance of each implemented step is discussed in a separate subsection, following a similar structure as it is shown in the section 6.3, and is highlighted by giving approximate percentages.

Also, the validity of each step was checked using the methods introduced in section 4.2.4. Nevertheless, some implementations required further specific testing, that is presented within the afferent subsection.

Moreover, for each step, the calling context for each kernel is explained, as it has a direct impact on the code's runtime speed. They way the context impacts the performance is given

by the number of calls it makes to a given kernel (in a loop or in a sequence, but more than once). Concretely, each time a kernel is called, through its launcher, memory data allocations and deallocations occur, as well as data transfers, each of these operations adding to the total execution time. Thus, the importance of describing the calling context comes from the fact that it gives an explanation for the newly obtained execution times.

The values for the execution times registered during the experiments are included in tables. However, when comparing them with previous results, percentages are used, in order to give a more concrete idea about the magnitude of the impact of a given implementation.

Moreover, unlike the tables used in the previous performance assessments outlined in preceding chapters, the tables used here are changed. Firstly, for the first two subsections, they only include the execution times for *Apply median filter* and *Build border normals* code sections, as well as the total execution times. That it because the implementations evaluated within the two subsections only impact the previously-mentioned code sections. For the remaining two subsections, tables include the execution times afferent to the considered code section and have different categories, as they illustrate the execution time of a given code segment on CPU, the amount of time the kernel launcher computing the respective code segment on GPU takes to execute, and the execution time of the kernel itself, without counting memory allocations, deallocations and data transfers.

6.4.1 Performance assessments on 2D matrix template class methods' parallelization

As described in subsection 6.3.1, the first GPU implementation focused on the bidimensional matrices template class. Here, only 3 methods proved amenable to such a parallelization, namely: *conv2DFull*, *conv2DValid* and *conv2DSame*. However, the first one is not called by the current C/C++ SIRFS implementation, thus, this analysis focuses only on the remaining two methods.

Before analyzing their calling contexts, both methods were called using only their CUDA implementations. Their performance impact is reflected only in *Apply median filter* and *Build border normals* code sections, were they are actually called for execution. When compared to the reference values presented in table 5.1, the results shown that the total execution time increased by 1.7%. Moreover, it was noticed an increase of 12.1% of the execution time of *Apply median filter* code section, and a decrease of 3% of the execution time of *Build border normals* code section. The time values per these code sections are illustrated in table 6.1, underneath.

Code section	Execution time in C++ (seconds)	Execution time in MATLAB (seconds)
Apply median filter	0.676 - 0.692	0.323
Build border normals	0.283 - 0.289	0.052
Total execution time	1.813 - 1.863	0.61

TABLE 6.1: Execution times per code section with CUDA implementations for convolution methods

Therefore, the values outlined above show that the experiment introduced an overall overhead. Thus, it calls for a better solution. As already mentioned, the context in which each CUDA kernel is called has an impact on the runtime performance, thus the calling contexts for *conv2DValid* and *conv2DSame* are explained in the following, in order to identify in which context the overhead is introduced.

That said, in the function called *medianFilterMatMask* the *conv2DValid* function is called twice, in two different loops. Here, the experiment shown that the first call introduces an overhead of 13.5%, whereas the second one speeds up the processing by 1.4%. These values can be seen as a consequence of the loops' sizes of the two contexts, in the first one proving that the data transfers between host and device, as well as device memory allocations, are computationally expensive. Thus, for the first call it was preserved the CPU implementation, whereas for the second one it was used the CUDA implementation.

Given the fact that for *Build border normals* code section the impact was positive, it can be said that the calling context is favorable for executing the afferent CUDA kernel. Thus, no further discussion is required and the GPU implementation is preferred to the CPU one for method *conv2DSame*, in this context. Having said that, all these results are shown in the table beneath, where it can be noticed an improvement of 3% of the total execution time, with a noticeable 6.88% speed up for the *medianFilterMatMask* code section. Therefore, for the next experiments, these values will be considered as references.

Code section	Execution time in C++ (seconds)	Execution time in MATLAB (seconds)
Apply median filter	0.558 - 0.578	0.323
Build border normals	0.283 - 0.289	0.052
Total execution time	1.70 - 1.75	0.61

TABLE 6.2: Execution times per code section with mixed CUDA and CPU convolution methods

Finally, as an extra validation step it was chosen to verify the output matrix of each method that uses CUDA implementation, by comparing it with the matrix processed by the CPU implementation. Thereafter, the methods introduced in section 4.2.4 were used.

6.4.2 Performance assessments on converting STL vectors into raw pointers

In subsection 6.3.2 is mentioned that changing STL vectors into raw pointers not only makes the integration with CUDA API much easier, but also can help in boosting the code's runtime speed, according to the experiments outlined in section 4.2.

The results have shown that the hypothesis is correct, as these changes improved the total execution time by 22%. The major impact was noticed in *Apply median filter* code section, whose execution time dropped by 60%. In addition, an improvement of 9% was noticed

with respect to the *Build border normals* code section's execution speed. The nominal values per code section are illustrated in the following table.

Code section	Execution time in C++ (seconds)	Execution time in MATLAB (seconds)
Apply median filter	0.215 - 0.236	0.323
Build border normals	0.257 - 0.262	0.052
Total execution time	1.331 - 1.370	0.61

TABLE 6.3: Execution times per code section after replacing STL vectors with raw pointers in block A2

Thus, the value shown above represent the reference for the upcoming experiments execution times.

6.4.3 Performance assessments on *Apply median filter* code section parallelization

In subsection 6.3.3 is described the implementation of a CUDA-based approach with respect to the *Apply median filter* code section. Next, in this subsection, the previously-mentioned implementation's performance impact is assessed, in the table underneath.

The table captures the total execution time of the considered code section, then the execution time of the code segment (that resides within the code section) to be parallelized is illustrated. The third column shows the execution time of the kernel, whereas the fourth column shows the execution time of the launcher, that is the execution time of the memory allocations and data transfers plus the kernel's execution time. Last column shows the processing time on MATLAB of the *Apply median filter* code section.

Code section	Execution time in C++ (seconds)	Processing time on CPU	CUDA kernel processing time	Launcher's processing time	Execution time in MATLAB (seconds)
Apply median filter	0.223 - 0.239	378e-6 - 528e-6	7e-6 - 31e-6	776e-6 - 109e-5	0.323
Total execution time	1.34-1.431				0.61

TABLE 6.4: Execution times of *Apply median filter* using a CUDA-based implementation

When compared to the values in table 6.3, it can be observed that the total execution time for the *Apply median filter* slightly increased by 1.7%. Next, it can be noticed that the kernel's execution time is more than 10 times faster than its CPU counterpart, result which supports the idea that GPU-based processing can speed up the computation of the algorithm. Nevertheless, as mentioned in section 6.2, the kernel cannot be executed without a context, that is created within the launcher, in this case. That said, another observation that can be made

is that the launcher's execution time is twice slower than the CPU implementation of the considered code segment. By putting these ideas together, it can be concluded that device memory allocations and transfers are way too computational expensive in order to make the current CUDA implementation efficient with respect to the goal of this project.

Since memory allocations and transfers cannot be avoided, a solution might be expanding the kernel with more operations, in such a way that its internal implementation would include more operations, as mentioned in [36]. Then, the new CPU counterpart will also contain more processing steps, and implicitly will have an increased processing time, so the new CUDA-based implementation might be more efficient that the current one, but not necessarily faster than the new CPU counterpart. That is because the the following aspect should be taken into account: with increasing number of computation steps, there increases the number of intermediate results, that are matrices and vectors, which require device memory allocations and deallocations, which are computational expensive.

In order to grasp how expensive is device memory allocation it was measured the runtime of the kernel and memory transfers within the launcher. The results shown that it takes 30-35% out of the total execution time of the launcher, with the remaining 65-70% being the time spent on device memory allocations and deallocations.

Supposing the above aspects would be accepted as a compromise, a more concrete implementation view of the previously-described workaround (that is adding more operations to the kernel) shows that the kernel's expansion has to continue with the double loop that currently calls the kernel, which basically must be parallelized on GPU. If done as such, each thread would need to sequentially process the big array that is currently processed by the existing kernel, as the nested parallelism is impossible due to the lack of dynamic parallelism, as already mentioned in section 6.1. As already specified, this implementation would not reduce the number of memory allocations and data transfers, thus no noticeable performance gains should be expected. Moreover, the double loop is relatively small, having less than 10 iterations, thus it is not prone to GPU parallelization.

Having said that, the study can continue with the parallelization of the next loops in *medianFilterMatMask* and in the *conv2DValid* functions, leading to an architecture similar to a pipeline of kernels within the previously-named functions. However, the findings outlined in the table above indicate that no performance enhancements should be expected, leading to the conclusion that no more improvements can be done for this code section, given the hardware constraints, mainly the lack of dynamic parallelism and not enough GPU memory.

6.4.4 Performance assessments on *Build border normals* code section parallelization

In subsection 6.3.4 is described the implementation of a CUDA-based approach with respect to the *Build border normals* code section. Next, in this subsection, the previously-mentioned implementation's performance impact is assessed, in the tables underneath.

Similarly, to previous subsection, the table captures the total execution time of the considered code section, then the execution time of the code segment (that resides within the code section) to be parallelized is illustrated. The third column shows the execution time of the kernel, whereas the fourth column shows the execution time of the launcher, that is the execution time of the memory allocations and data transfers plus the kernel's execution time. Last column shows the processing time on MATLAB of the *Build border normals* code section.

That said, the table below illustrates the runtime performance with respect to the firstly implemented kernel, within this code section. When compared to the values in table 6.3, it can be noticed that the total execution time for the *Build border normals* code section slightly increased by 1.9%. The next made observation is regarding the kernel's execution time which is more than 100 times faster than its CPU counterpart. However, as already mentioned in section 6.3, the kernel cannot be executed without a context, that is created within the launcher, likewise the case presented in the previous subsection. Therefore, measuring the launcher's execution time is a key aspect and it can be seen that is 3.3 times slower than the CPU implementation of the considered code segment. On top of this, the kernel's execution time represents just 0.3% of the launcher's time, showing the magnitude of the impact that the kernel's context creation has on the execution performance.

Code section	Execution time in C++ (seconds)	Processing time on CPU	CUDA kernel processing time	Launcher's processing time	Execution time in MATLAB (seconds)
Build border normals	0.262 - 0.269	15.5e-4	14e-6	52e-4	0.052
Total execution time	1.342-1.37				0.61

TABLE 6.5: Execution times using one CUDA kernel in *Build border normals* code section inner loop and page locked memory

Again, these results came as a consequence of device memory allocations, deallocations and data transfers between host and device, which proved to be too computational expensive in order to make the parallelized code segment more efficient than the CPU implementation. Furthermore, it was considered once more to parallelize more operations within the kernel, thus increasing its complexity (implicitly, the CPU afferent code segment to be parallelized also entailing higher complexity), and like in the case explained in subsection 6.4.3, it was noticed that the intermediate results, stored in matrices and vectors, would require more device memory allocations.

More important, adding more operations to this kernel proved to be impossible due to high memory demand of the considered computation steps. Concretely, when more computation steps were added, not all threads could reach the last steps, leading to memory overwriting and incorrect results. That said, the implementation outlined in subsection 6.3.4 is the only possibility in this case, giving the hardware constraints, mainly limitation of available GPU memory.

Thus, part of the inner loop's computation was parallelized in a separate CUDA kernel, whose performance assessments are shown in the table below. For this experiment, the first CUDA kernel implemented within the *Build border normals* code section was still used. That is because the newly implemented CUDA kernel depends on the first one, as it uses the data output by the latter. Therefore, the table below presents the combined impact that the two CUDA kernels have on the execution time of the *Build border normals* code section.

Code section	Execution time in C++ (seconds)	Processing time on CPU	CUDA kernel processing time	Launcher's processing time	Execution time in MATLAB (seconds)
Build border normals	0.684 - 0.700	0.1368	0.0036	0.6252	0.052
Total execution time	1.78 - 1.8				0.61

TABLE 6.6: Execution times per code section using two CUDA kernels in *Build border normals* inner loop

The values outlined above are compared to the ones shown in table 6.5, in order to assess whether the combined impact of the two kernels is better than the performance of the first kernel. That said, it can be noticed a significant growth of the code section's execution time by 2.6 times. Given the fact that the negative impact on performance of the first kernel of this code section is 1.9%, it can be easily inferred that the considerable overhead was added by the new implementation. Concretely, calling the second kernel's launcher within thew loop becomes very inefficient. The reason lies in the repetitive memory allocation, deallocations and data transfers that were already discussed above and in the subsection 6.4.3 as being computationally expensive.

Next, it can be observed that the launcher's execution time is 4.6 times slower that the CPU implementation of the corresponding code segment. Sure, the kernel proved to be 38 times faster than the same CPU code, but as specified before, this comparison is exaggerated as the measurements of the kernel's context creation are not considered. Nonetheless, the comparison just aims to prove that the kernel's computation itself is way faster than the CPU one and, more importantly, that it represents a small percentage (0.6% in this case) of the launcher's total execution time.

Since both implementations proved to hamper the runtime performance of the SIRFS C/C++ code, they are not preserved as solutions for the GPU parallelization of the algorithm. That said, next is assessed the performance of the last implemented kernel. As explained in subsection 6.3.4, its implementation envisaged part of the sequential code the precedes the loop whose parallelization performance is described above.

As its launcher is not called within a loop, like in the case of the second kernel of this code section's GPU parallelization, nor in a function called within this code sections, as it is the case in subsection 6.4.3, the impact of CUDA memory transfer and allocations is expected to

be lower when compared to these situations. Next, the execution times for this experiment are illustrated in table underneath.

Code section	Execution time in C++ (seconds)	Processing time on CPU	CUDA kernel processing time	Launcher's processing time	Execution time in MATLAB (seconds)
Build border normals	0.263 - 0.266	6.5e-5	1.6e-5	0.00304	0.052
Total execution time	1.34-1.431				0.61

TABLE 6.7: Execution times per code section using one CUDA kernel in *Build border normals* with mapped memory

The first observation to be made is regarding the fact that the kernel's execution time represents 0.52% of the launcher's execution time. Then, the latter is 4.7 times slower than the afferent CPU code segment. Again, the conclusion to be drawn from here is that the time required by device memory allocations and deallocations, as well as by host-device data transfers introduces a considerable overhead. Moreover, the total execution time of the entire code section increased by 1.9% when comparing to the values in table 6.3, as it was the case with the first kernel discussed in this subsection.

On top of that, it has to be noted that the amount of memory transfers in this case has been decreased, as mapped memory was used. Moreover, in order to map memory to CUDA virtual address space, page-locked memory was used, but still it proved to hinder the runtime performance of the code.

Whereas in the case of the first two kernels implemented with respect to this code section it was impossible their expansion in order to include more operations (thus, increasing the complexity of the parallelized code) due to memory limitations, in this case the expansion was not possible due to other reasons. The first one is that expanding with the next operations with mean to combine the the first and the third kernels together as they execute one after each other, in a sequence, but as already explained, it cannot be done as there is not enough device memory to handle all required intermediate results. The other possibility would have been to include the operations that precede the call of the third kernel. Nevertheless, their complexity is reduced, as the computations are performed on relatively small data sets (less than 200 elements), thus their GPU parallelization would not pay off.

Having said that, this subsection presented three performance assessment of three different CUDA kernels implemented in three different contexts. The reason for implementing these kernels in different contexts was to capture whether and how a CUDA-based implementation can optimize the *Build border normals* code section. Nonetheless, all these experiments proved that with the current hardware constraints, mentioned in section 6.1, the current implemented CUDA-based solutions do not help in enhancing then execution speed of the *Build border normals* code section.

Chapter 7

Conclusions

Shape from Shading is a term that defines a family of algorithms that aim at reconstructing the 3D geometric information of a single object captured in a 2D statical image. One of the most advanced technique in this area is called SIRFS and not only involves recovering of the 3D geometrical data, but also information regarding the other factors that influenced the 2D scene of the input image: viewpoint, light parameters, reflectance. The SIRFS algorithm is implemented in MATLAB and documented in [1]. Nevertheless, the documentation does not provide implementation details, as it focuses on the scientific novelty brought in the area.

However, in order to achieve plausible reconstructions of the original 3D shape of the object captured in the input image, one has to fine tune several parameters. Next to that, the computation is based on an optimization function, that minimizes the cost of recovering the above-enumerated parameters in several iterations. Moreover, most of the performed operations sequentially process bidimensional matrices. All these aspects make the current implementation slow, thus, this study aimed at finding whether a parallelization using a GPU-based approach, as well as a C/C++ implementation, would help in improving the overall runtime performance of the algorithm.

Given the scope of the project, it can be seen as consisting of two parts: first one involving code translation from MATLAB to C/C++ and the second one aiming at optimizing the code produced during the first part. Having said that, a recent research internship [2] covered the first part. Specifically, the MATLAB code was analyzed and reverse engineered in order to discover the following: the code's call graph, the way the functions in the obtained call graph interact (by analyzing their signatures) and the major used data types and how to map them to C/C++. Then, a subset of the code (18%) was translated and tested against its MATLAB counterpart in order to validate its results. This was done based on a verification method also developed in the project [2].

The approach described in [2] can be reused for the reverse-engineering of any code that allows a given IDE to place breakpoints and that allows for placing print functions (proper to the native language of the code), in order to discover its call graph. Once this is done, the analysis of the target code can continue with the steps briefly explained above. Thus, with little variations, the code translation strategy devised in [2] can be reused for translating other algorithms into another programming language, regardless the considered algorithms are similar to SIRFS or not.

Thereafter, the study presented in this thesis document started by highlighting the development strategy that should be followed, given the project's goal, mentioned above. Thus, a breadth-first approach could have been followed in order to finish the entire translation of the SIRFS MATLAB code, or a depth-first approach which entailed the optimizing of the translated code presented in [2]. Given the time constraints and the scope of the entire project, the depth-first approach was used.

After choosing the development strategy, it was clear that the scope is code optimization, In order to achieve it, the major bottlenecks in the C++ SIRFS implementation were found. By analyzing them it was concluded that GPU parallelization would not suffice to achieve this goal, thus a more complex solution was envisaged. This solution is called "Incremental code optimization" and consists of 5 steps, each one being seen as a filter. Thus, the solution presented in this thesis is based on a pipeline of filters architecture. For each filter, this project aimed to find a solution, in terms of code implementation, in such a way that the filter can solve the problems introduced by certain bottlenecks (a subset of the total identified bottlenecks).

For the first filter's implementation, it is presented a new method for loading the PNG input images, using *libpng* library. The solution proved to be 99.98% faster than the one used in [2] so it was considered an optimal implementation in this case. This solution can be reused in every C/C++ program that deals with reading PNG images. Furthermore, this filter had a secondary goal: fully decoupling the C/C++ SIRFS code from the MATLAB one. The new solution contributes to that, but a newer validation method was devised in order to fully achieve this scope.

Thereafter, an implementation based on *matio* library was presented as a possible solution for the second filter. Since it was devised especially for the SIRFS case, it cannot be reused in other cases in the form it is presented here. Nevertheless, the idea of creating a tree structure that maps the structures included in the MAT file, as well as a metadata structure, that is used to traverse the tree to read the data stored in these structures' fields, can be reused whenever one deals with reading, into a C/C++ program, data from nested structures stored in MAT files. Though the solution did not bring along the required optimization factor (98%), it was accepted as a implementation for the second filter as it introduced an 88% enhancement of this procedure.

Since the first two filters aim at solving issues introduced by bottlenecks that are not found in the rest of the SIRFS code, their solutions cannot be further used for the remaining SIRFS code, as there is no context left to use them.

The solution for the third filter addressed multiple problems. Firstly it focused on the implementation of the containers of the matrix template classes. It was concluded that using a raw pointer with linear indexation is better than using double or triple pointers, as it proved to be 10% faster. Also, alternatives based on STL vectors and unique smart pointers were tried, but they proved to be inefficient, as they lead to performance drops of 20% and 70% respectively. Afterwards, it was changed the way these classes' methods are called, in the

sense that the output object of their computation became the caller of the method. This approach helped in achieving a 6.45% performance gain.

Lastly, the third filter's solution addresses the overheads introduced by the modularity of the initial C/C++ SIRFS implementation. Thus, it was concluded that replacing some function's call with their implementation would help in improving the runtime performance. With a 1.4% performance gain, this hypothesis was validated. Given its generality, the solution introduced by the third filter can be reused for the remaining SIRFS code that needs to be translated, as well as in other cases involving optimization of a C/C++ code.

Thereafter, a coarse-grained CPU parallelism approach was suggested as a solution for the fourth filter. The idea behind it was to have multiple levels of parallelism, both an CPU and GPU, with the GPU-based one nested within the CPU-based one, in order to simulate the dynamic parallelism, which is not supported by the GPU used for this project. Nonetheless, this approach proved to hinder the execution performance, having tried 2 different experiments that introduced overheads of 1% and 16.5%, respectively. Furthermore, in order to implement these experiments, the code was ported from Windows to Linux, fact which helped in enhancing the total execution time by 36%.

Due to time constraints, a finer-grained CPU parallelism was not tested, thus, the fourth filter does not entail any solution. On top of that, it is hard to assess whether this coarse-grained CPU parallelism would help in optimizing other algorithms, but their impacts on C/C++ SIRFS code indicate that the approach is not suitable for the remainder of the SIRFS code. Thus, the code porting to Linux is the only achievement brought by the experiments done with the scope to determine the solution for this filter.

For the last filter, more experiments were tried in order to devise its implementation. First, it was concluded that the 2D matrix template class methods that compute the convolution can be GPU parallelized, but not the others. Initially, they hampered the runtime performance, but it was noticed that in one case, the CPU implementation was faster than the GPU one, unlike the remaining two, fact which lead to a 4.5% performance gain. Next, the analysis focused on two code sections and, before trying different GPU-based implementations for them, their implementations were changed in such a way that multiple STL vectors were converted to raw pointers, thus leading to a 21.5% performance gain.

As a consequence of this, the execution time of the first code section, out of the considered two, dropped below the one of its MATLAB counterpart, thus achieving the goal of this thesis, but only within this code section. Nevertheless, a rather coarse-grained GPU-based parallelization approach was tried in order to determine whether its execution time can be further improved, but the implementation proved to hinder the runtime performance by 1.7%. Since the second considered code section was still slower than its MATLAB counterpart, the analysis further focused on it. Here, three GPU-based parallelization approaches were tried, each one entailing different levels of graining. Nevertheless, all proved to hamper the runtime speed, by 1.9%, 165% and 1.9% respectively.

Based on these performance factors, it was concluded that the code segments considered for GPU parallelization do not involve a high complexity in computation, as their GPU-based implementations added overheads due to CUDA memory allocations. So, it was clear that, not only in the case of SIRFS, but in general, a CUDA-based implementation pays off only when the code entails enough complexity, such as nested loops (each one having a reasonable number of iterations), or complex arithmetic operations.

Therefore, the solution for the last filter is composed of the parallelization of the convolution methods and the conversion of STL vectors to raw pointers. However, more complex GPU-based solutions could not have been tried, due to hardware constraints, mainly GPU memory limitation and lack of dynamic parallelism. Sure, these experiments can be redone if the hardware constraints are eliminated, namely if a newer GPU is used, or even a GPU farm, case which is expected to enhance the runtime speed even with the approaches outlined in chapter 6.

The impacts of all accepted solutions for the previously-described filters are summarized in figure 7.1, beneath, where it can be observed a constant decrease in execution time. Concretely, the total performance gain was 90.35%, which represents a considerable improvement. Nonetheless, the total execution time is still slower than in MATLAB. This, leads to the following question, that can be answered to only when the project will be finalized: *can this overhead be negligible for the final ported code?*. In other terms, will the improvements of the next-to-be-translated code help in optimizing the entire SIRFS implementation in such a way that the actual overhead will not represent a problem with respect to the overall goal of this project?

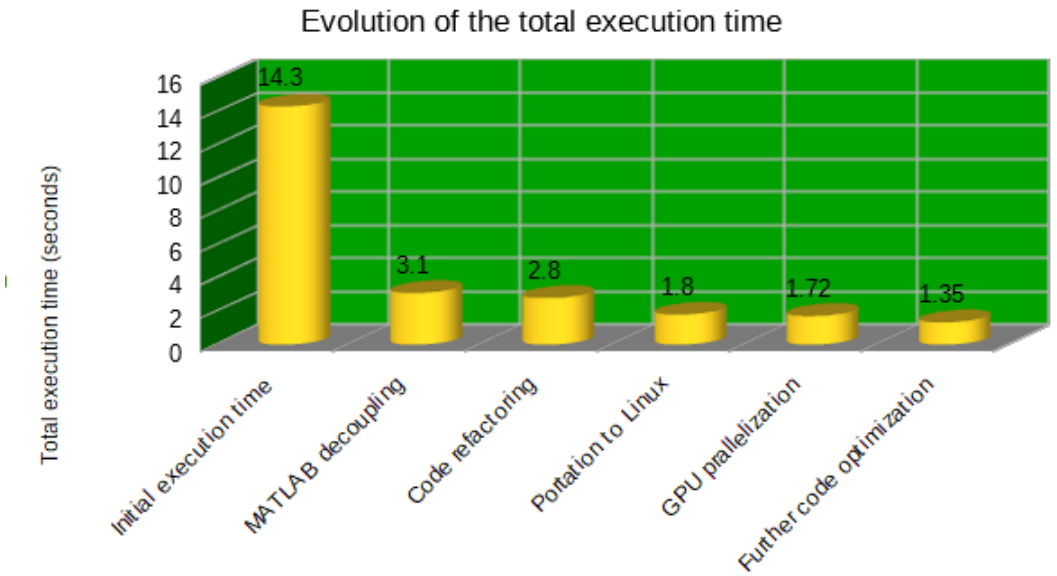

FIGURE 7.1: Evolution of the total execution time with every accepted solution

Chapter 8

Future developments and recommendations

As mentioned in chapters 3 and 7, the development of this project followed a depth-first strategy, with focus on optimization of the translated code presented in [2]. This strategy refines the solution in the form of a pipeline consisting of five filters. Due to time and hardware constraints, for some of them, there could not be tried multiple possible experiments that can help in shaping a better solution for them, with respect to the goal of this project. That said, these possible experiments are presented in this chapter, as future developments.

Firstly, with respect to the *Load priors* filter (the second filter) it was noticed that it introduces the biggest overhead. No alternative solution was found to the one presented in section 3.3, but throughout the lifetime of this project it should be considered to research for an alternative, especially if the answer to the question that ends chapter 7 proves to be negative. In other terms, this bottleneck should not represent the main object of the future developments of the project, but rather an issue to be taken into account once the full code porting and optimization will be done, thus an answer to the previously-mentioned question will be known.

Secondly, based on the results in section 4.4, it is recommended to keep a balance between the used functions and the behaviors that still need to be translated from MATLAB to C++. Furthermore, based on the results outlined in sections 4.1.1, 4.2 and 6.4.2, the raw pointers should be preferred to STL vectors, when working with arrays, for the next code porting steps. On top of that, the next-to-be-added methods should follow the strategy outlined in section 4.1.2.

However, before starting to port to C/C++ the remaining SIRFS code, several experiments must be tried, in order to fully shape the depth-first approach. That said, first, a finer-grained CPU-based parallelism should tried for the existing C/C++ SIRFS code. This can be seen as a parallelism that occurs at the loops level, within the functions considered for the coarse-grained CPU parallelism, described in chapter 5. This should be tried especially for the loops with reduced number of iterations, that are not amenable to GPU parallelization. Based on the results on this new experiment, it can be determined if the CPU parallelism can serve for the purpose of this thesis and, implicitly, if the presence of the fourth filter illustrated in figure 3.2 is justified or not.

Then, if one can access a GPU farm, or can use multiple GPUs in a server, the experiments presented in sections 6.3.3 and 6.3.4 can be redone, thus determining whether these implementations can improve the runtime speed when running on a much faster hardware. This would also solve the issues caused by lack of GPU memory. If all GPUs in the farm have a computed capability higher or equal than 3.5, this would solve the lack of dynamic parallelism issue.

If this is not possible, but a newer GPU card can be used, that supports dynamic parallelism and has a higher amount of memory, there can be devised different solutions for the code segments considered within the previously-named sections of this document. Concretely, for those that cannot be expanded because of lack of dynamic parallelism, a dynamic parallelism-based approach should be tried. Similarly, for those that cannot be expanded because of lack of GPU memory. If these experiments prove to enhance the execution performance, then a complete solution for the fifth filter would be shaped. Otherwise, for the GPU parallelization of the next-to-be-translated SIRF code should be considered only the code segments that entail enough complexity, such as nested loops (each one having a reasonable number of iterations), or complex arithmetic operations, as already specified in chapter 7.

Once the experiments for the *CPU parallelization* and *GPU parallelization* (namely, the fourth and the fifth) will be tried, based on the conclusions on them there will be obtained a complete image on the implementation of these filters. At that moment, the depth-first approach will be finished and the project can continue with the breadth-first approach, following the recommendations outlined in this chapter and during the experiments that will be conducted for the previously-named two filters.

Appendix A

Development context details

In this Appendix is presented the data about the hardware and the software used for the development of this project, in the table below.

CPU model	Intel Core i7 4700HQ - Haswell
Number of CPU cores	4 [44]
Number of CPU threads	8 [44]
CPU frequency	2.4 GHz base frequency; 3.4 GHz turbo frequency [44]
Number of CPU memory channels	2 [44]
CPU memory maximum bandwidth	25.6 GB/sec [44]
Number of GPU cores	768 [45]
GPU amount of memory and memory type	2GB GDDR5 [45]
GPU memory bandwidth	64 GB/sec [45]
GPU clock	2 GHz [45]
Compute capability	3.0
Operating system(s)	Windows 10 64-bit and Ubuntu 14.04 LTS
RAM type, amount and frequency	DDR3L, 8 GB, 1600 MHz
Permanent memory type, amount and frequency	HDD, 750 GB, 7200 rpm

TABLE A.1: Hardware and software context of C++ SIRFS development

Furthermore, whenever the performance times were measured, on the Windows platform, the context was similar: there was one Firefox tab open, with this Latex document, and the Code::Blocks IDE with the running code. Additionally, the required NVIDIA processes were running, as well as the anti-virus program.

Bibliography

[1] J. Malik J. T. Barron. "Shape, Illumination, and Reflectance from Shading". In: *IEEE Transactions on pattern analysis and machine intelligence* (2015).

[2] Ș. C. Crețu. *Parallelization-ready shape, illumination, and reflectance from shading*. MSc research internship report, JBI institute, University of Groningen. 2018.

[3] *Pthreads win32 - Windows multithreading API*. [Online. Accessed on 29.04.2018]. URL: http://www.sourceware.org/pthreads-win32/.

[4] *Multithreading with C and Win32*. [Online. Accessed on 29.04.2018]. URL: https://msdn.microsoft.com/en-us/library/y6h8hye8.aspx.

[5] *Win32 support for multithreading on Windows*. [Online University of Washington course resource. Accessed on 29.04.2018]. URL: http://courses.washington.edu/css443/Spring2003/thread_examples/MultiThreadingWithWin32.pdf.

[6] *Source files and Windows installer for libpng*. [Online. Accessed on 9.02.2018]. URL: http://gnuwin32.sourceforge.net/packages/libpng.htm.

[7] *Source files and Windows installer for zlib*. [Online. Accessed on 9.02.2018]. URL: http://gnuwin32.sourceforge.net/packages/zlib.htm.

[8] *Linking to a .dll*. [Online. Accessed on 10.02.2018]. URL: https://www.cygwin.com/ml/cygwin/2008-11/msg00037.html.

[9] *Include C libraries in C++ code using "extern "C"*. [Online. Accessed on 5.02.2018]. URL: https://www.geeksforgeeks.org/extern-c-in-c/.

[10] *Use "extern "C" to include C-implemented library in C++ code*. [Online. Accessed on 5.02.2018]. URL: https://stackoverflow.com/questions/1041866/in-c-source-what-is-the-effect-of-extern.

[11] *Understand the meaning of "extern" keyword in C*. [Online. Accessed on 5.02.2018]. URL: https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/.

[12] *Description of png_create_read_struct signature*. [Online. Accessed on 11.03.2018]. URL: http://refspecs.linuxbase.org/LSB_4.1.0/LSB-Desktop-generic/LSB-Desktop-generic/libpng12.png.create.read.struct.1.html.

[13] *Read an image using libpng*. [Online. Accessed on 10.02.2018]. URL: http://www.libpng.org/pub/png/book/chapter13.html.

[14] *Description of png_set_palette_to_rgb signature*. [Online. Accessed on 11.03.2018]. URL: http://refspecs.linuxbase.org/LSB_3.2.0/LSB-Desktop-generic/LSB-Desktop-generic/libpng12.png.set.palette.to.rgb.1.html.

[15] *Libpng image transformations*. [Online. Accessed on 11.03.2018]. URL: http://www.libpng.org/pub/png/libpng-manual.txt.

[16] Christopher C. Hulbert. *Matio User Manual for version 1.5.11*. 2017. URL: https://sourceforge.net/projects/matio/files/matio/1.5.11/matio_user_guide.pdf/download.

[17] *Source for downloading HDF5, used by matio library when reading .mat file into C/C++*. [Online. Accessed on 10.02.2018]. URL: https://www.hdfgroup.org/downloads/hdf5/.

[18] *Download Matio 1.5.11 archive*. [Online. Accessed on 14.03.2018]. URL: https://sourceforge.net/projects/matio/files/matio/1.5.11/.

[19] *New operator in C++*. [Online. Accessed on 28.03.2018]. URL: http://www.cplusplus.com/reference/new/operator%20new[]/.

[20] *Malloc operator in C*. [Online. Accessed on 28.03.2018]. URL: http://en.cppreference.com/w/c/memory/malloc.

[21] *Calloc operator in C*. [Online. Accessed on 28.03.2018]. URL: http://en.cppreference.com/w/c/memory/calloc.

[22] *STL vector documentation*. [Online. Accessed on 31.03.2018]. URL: http://www.cplusplus.com/reference/vector/vector/.

[23] *Range based iteration with STL vectors*. [Online. Accessed on 31.03.2018]. URL: http://en.cppreference.com/w/cpp/language/range-for.

[24] *Auto keyword in C++*. [Online. Accessed on 31.03.2018]. URL: http://www.learncpp.com/cpp-tutorial/4-8-the-auto-keyword/r.

[25] *STL vector performance enhancement tips*. [Online. Accessed on 31.03.2018]. URL: http://www.acodersjourney.com/2016/11/6-tips-supercharge-cpp-11-vector-performance/.

[26] *Fastest way to index a vector*. [Online. Accessed on 31.03.2018]. URL: http://fastcpp.blogspot.nl/2011/03/fast-iteration-over-stl-vector-elements.html.

[27] David Kieras. *Using C++'s smart pointers*. EECS Department, University of Michigan. 2016.

[28] *Unique pointers*. [Online. Accessed on 5.04.2018]. URL: http://www.drdobbs.com/cpp/c11-uniqueptr/240002708.

[29] *std::move description*. [Online. Accessed on 24.04.2018]. URL: http://www.cplusplus.com/reference/utility/move/.

[30] *rvalues and std::move*. [Online. Accessed on 24.04.2018]. URL: https://www.artima.com/cppsource/rvalue.html.

[31] *Calling conventions on the x86 platform*. [Online. Accessed on 24.04.2018]. URL: http://www.angelcode.com/dev/callconv/callconv.html.

[32] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, Inc., San Francisco. 2010.

[33] *Performance benchmarks on different OS platforms*. [Online. Accessed on 22.05.2018]. URL: http://www.bitsnbites.eu/benchmarking-os-primitives/.

[34] *Most performant CPUs 2018*. [Online. Accessed on 20.07.2018]. URL: https://www.hardware-revolution.com/best-cpu-processor-apu-july-2018/#1800x.

[35] *Hyper threading*. [Online. Accessed on 20.07.2018]. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[36] *CUDA programing model*. [Online. Accessed on 30.06.2018]. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model.

[37] *CUDA declaration specifiers*. [Online. Accessed on 22.07.2018]. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-language-extensions.

[38] *CUDA page-locked host memory*. [Online. Accessed on 20.06.2018]. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#page-locked-host-memory.

[39] *Description of MATLAB conv2 function*. [Online. Accessed on 28.06.2018]. URL: https://nl.mathworks.com/help/matlab/ref/conv2.html.

[40] *CUDA thread hierarchy and threads' grid dimensions*. [Online. Accessed on 20.06.2018]. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#thread-hierarchy.

[41] *C++ template functions' full specialization*. [Online. Accessed on 18.06.2018]. URL: https://en.cppreference.com/w/cpp/language/template_specialization.

[42] *CUDA API for memory management*. [Online. Accessed on 29.06.2018]. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html.

[43] *Pass arguments by reference to functions in C++*. [Online. Accessed on 29.06.2018]. URL: http://www.learncpp.com/cpp-tutorial/73-passing-arguments-by-reference/.

[44] *Specifications of the CPU hardware*. [Online. Accessed on 30.04.2018]. URL: https://ark.intel.com/products/75116/Intel-Core-i7-4700HQ-Processor-6M-Cache-up-to-3_40-GHz.

[45] *Specifications of the GPU hardware*. [Online. Accessed on 30.04.2018]. URL: https://www.geforce.com/hardware/notebook-gpus/geforce-gtx-765m/specifications.