NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

BRUNO DANIEL AFONSO DE BRITO
BSc in computer science

# TOMOGRAPHIC IMAGE PROCESSING USING JULIA AND GPU

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

DEPARTMENT OF
COMPUTER SCIENCE

# TOMOGRAPHIC IMAGE PROCESSING USING JULIA AND GPU

**BRUNO DANIEL AFONSO DE BRITO**

BSc in computer science

**Adviser:** Pedro Abílio Duarte de Medeiros
*Associate Professor, NOVA SST*

**Examination Committee**

**Chair:** João Pires
*Associate Professor, NOVA SST*

**Rapporteur:** André Mora
*Assistant Professor, NOVA SST*

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

**Tomographic image processing using Julia and GPU**

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.5) [1].

# Abstract

Image processing is an essential work component for material science researchers, and there's a constant quest for novel ways to explore it. One of the areas this can be done is in computerized tomography (CT) processing. CT images are an efficient and commonly used method to characterize materials. The resulting images can be combined and processed by a GPU. Work on CT image processing has been done previously, however the appearance of novel programming languages allows for further improvement. Julia programming language has the advantage of being both fast and user-friendly and poses an interesting resource for the image processing area.

In this dissertation, the development and evaluation of an application written in Julia capable of processing CT images is described. Additionally, an analysis of the potential this programming language has on the image processing field was performed. It was possible to conclude that Julia contributes as an useful tool for material science researchers, and that the solution developed can aid developing their work.

**Keywords:** Julia, Computerized tomography, GPU, Image processing

# Resumo

O processamento de imagens constitui um elemento base no trabalho desenvolvido por investigadores de engenharia de materiais. Isto resulta numa procura contínua de novas maneiras de explorar este tema. Uma das áreas de processamento de imagem em que se pode fazê-lo é no processamento de tomografia computadorizada (TC). As imagens de TC são meios eficazes e comumente utilizados para caracterizar materiais. As imagens resultantes podem ser posteriormente combinadas e processadas por uma GPU. No passado já foram desenvolvidos projetos no âmbito de processamento de imagens TC, contudo o aparecimento de linguagens de programação mais recentes abrem espaço para novos testes e desenvolvimentos. A linguagem de programação Julia, em particular, apresenta a vantagem de ser simultaneamente rápida e de ter uma sintaxe de fácil compreensão, pelo que pode constituir um recurso útil para a área de processamento de imagem.

Nesta dissertação é descrito, o desenvolvimento e avaliação de um programa de computador escrito em Julia capaz de processar imagens de TC. Adicionalmente, é feita uma análise do potencial desta linguagem de programação no campo de processamento de imagem. Foi possível concluir que Julia contibui como uma ferramenta útil para investigadores de de engenharia de materiais, e que a solução desenvolvida pode auxiliar a desenvolver o seu trabalho no futuro.

**Palavras-chave:** Julia, Tomografia computadorizada, GPU, Image processing

# Contents

# List of Figures

# Acronyms

|  |  |
|---|---|
| **2D** | Two Dimensional vii, 4, 5, 11, 18, 20 |
| **3D** | Three Dimensional vii, 5, 9, 11 |
| **ACCL** | Accelerated Connected Component Labeling 19–21, 26, 34, 35 |
| **APU** | Accelerated Processing Unit 16 |
| **CCL** | Connected Component Labeling 19, 25 |
| **CPU** | Central Processing Unit 10–12, 14 |
| **CT** | Computerized Tomography vii, 2, 4–10, 13, 17, 19, 20, 29, 30, 33, 36 |
| **GPU** | Graphics Processing Unit 2, 11, 13–16, 36, 38 |
| **JIT** | Just In Time 12 |
| **LLVM** | Low Level Virtual Machine 12, 13 |
| **NRRD** | Nearly Raw Raster Data 5, 6, 14, 38 |
| **REPL** | Read Evaluate Print Loop 12, 13 |
| **RGB** | Red Green Blue 6–8 |
| **SIMD** | Single Instruction Multiple Data vii, 10, 11, 13, 14 |

# Chapter 1

# Introduction

## 1.1 Context

Material science researchers use tomographic images to analyse samples of composite materials, which combine a base material with reinforcements of distinct nature. The main goal of their analysis is to evaluate the results obtained by a given method of producing the material. The quality of the fabrication process is assessed by obtaining a geometric characterization of the reinforcement population.

A tomographic image is stored in memory as a 3D matrix where each voxel is represented by an integer value, corresponding to a gray level. Before the reinforcement characterization, several image processing operations must be performed in order to get a black and white image, where white represents the base material and black the reinforcements. After getting a binary image, a labeling process assigns a unique identifier to each reinforcement, which corresponds to a set of connected voxels [2].

Due do the size of the data and the complexity of some processing steps, the pre-characterization and characterization steps require great computational power, suggesting the use of parallel processing. Due do the data organization, the use of GPUs [3] allows a significant reduction of the execution time.

In a recent work [4], the above image processing algorithms were implemented using Python, CUDA, and Numba. Numba is a compiler that translates a subset of Python into fast machine code. The use of a Numba compiler improved Python's performance, compared to when Python. However, execution times obtained were still not satisfactory when compared with the corresponding C/CUDA versions.

## 1.2 Thesis goal

The main goal of this work is to evaluate alternative implementations of the algorithms needed for the tomographic image analysis, more specifically the use of the Julia programming language [5].

The goal of this project is to process CT images in a way that allows the isolation of each material particle, so that material science researchers are able to classify the features and characteristics of the different elements that compose them. Ultimately this will allow to aid researchers in developing their work in the material science domain. The project will be done by resorting to GPU and Julia programming language.

An additional goal of this work is to test how Julia performs in the image processing area. This evaluation could provide interesting findings since a project like this has never been developed before in this university.

## 1.3 Methodology

To achieve the goals proposed, first the materials tomographic image will be converted from a gray-scale image to a black and white image. This duality of colors will allow separating the base material, in white, from the study material, in black, with higher precision. This step is further explained in section 2.2.1.2 Afterward, the material will be subjected to a connected-component labeling algorithm that will separate the different material particles, represented by continuous voxel units.

## 1.4 Contributions

After concluding the implementation of the work, material science researchers will have an additional tool to aid in their work. Additionally, this project could provide performance statistics for Julia in the image processing field. This parameter could be relevant considering the novelty of this programming language.

## 1.5 Structure

This dissertation will describe what the project proposed to develop consists of, as well as the steps and methods utilized to do so. The text follows a logical organization in six main chapters, further divided into subsections.

The first chapter introduce the project and what is to be accomplished with its development. It also exemplifies the project's utility and how this knowledge could be incorporated upon further research in the material science domain.

In the second chapter, the basic concepts will be elaborated on to build the foundation of knowledge necessary to understand how the project will be developed.

Chapter three will go into the variety of program implementation options available for our program, discussing both the advantages and disadvantages each one has. Additionally, the chosen approach will be properly justified in this chapter.

In chapter four the implementation of the solution chosen in chapter three will be demonstrated. This will be done through the use of code snippets and text descriptions, to provide a complete overview of the code developed.

Chapter five will cover the results obtained through the implemented solution. This will include time metrics and image comparison between before and after processing, that will be further discussed and analysed here.

Finally, chapter six will conclude this dissertation by providing a broad overview of the work developed and the final results obtained. Additionally, future advancement alternatives will be discussed.

# Chapter 2

# Basic Concepts

In the present chapter the relevant concepts for this dissertation will be introduced. It'll begin with an introduction on what tomographic images are, how they are generated, stored and visualized. Next, the basis of image processing will be introduced, including the image transformation, cleaning and segmentation. Afterward, the concept of program parallelism and its relevance for the present work is going to be expanded on. Finally, Julia, the programming language of choice for this work will be introduced and explained.

## 2.1 Tomographic Images

Computerized Tomography (CT) is a useful technique to characterize a diverse set of materials, such as ceramics, metal foams, metal alloys, and bio-materials. It generates a three-dimensional reconstruction of the material, that can be further processed and analyzed. The technical basis behind the generation of CT images, and how they are formatted and visualized will be discussed in this subsection.[2]

### 2.1.1 Image generation

A computerized tomography, or CT for short, is an imaging procedure where beams of x-rays are aimed at a person or object. These X-rays will interact with the matter and attenuate. Their degree of attenuation will depend on the materials absorptivity. This means that different materials will generate different X-ray intensities. Multiple x-ray projections are taken by rotating the x-ray source, detector or the material sample. These image projections of various orientations can be joined and transformed from a 2D specimen to a three dimensional object. [6]

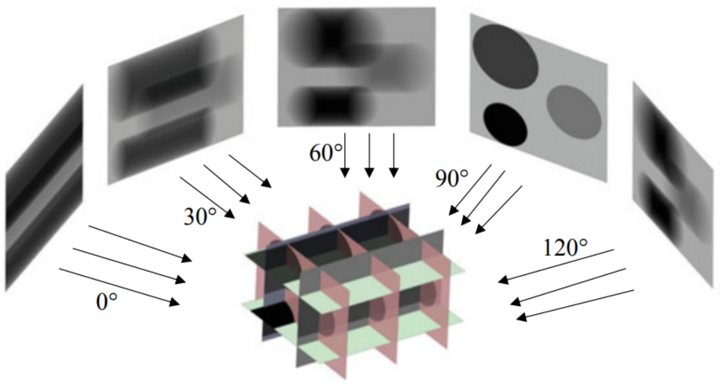


Figure 2.1: 3D reconstruction from 2D cross-sectional images. [6]

When referring to a CT volume, the term voxels is used alternatively to pixels. A voxel is the volumetric representation of the pixel. The geometric detail in tomographic imaging is determined by voxel size. A decrease in voxel size corresponds to a higher image quality. [6]

### 2.1.2 Composition

The tomographic images that will be analysed are composed by two components: the base material and target material. The base material, that occupies the largest area, needs to be separated from the target material that we intend to analyze to allow for better visualization of the material in study.

In some cases image noise might be present and need to be removed resorting to image processing techniques, which will be described later on. This will allow to provide a better visualization of the target material in study.

### 2.1.3 Format

Nearly Raw Raster Data (NRRD) file format is commonly used for image processing and it will be used to store CT images. A NRRD file is composed by two main parts: the header and the data. A header example is shown in the figure below. After the header there is a blank line to help identify where the header ends and where the data begins. Note that data is not shown in the figure below. The purpose of the header is to define how the data after the blank line is organized.[7]

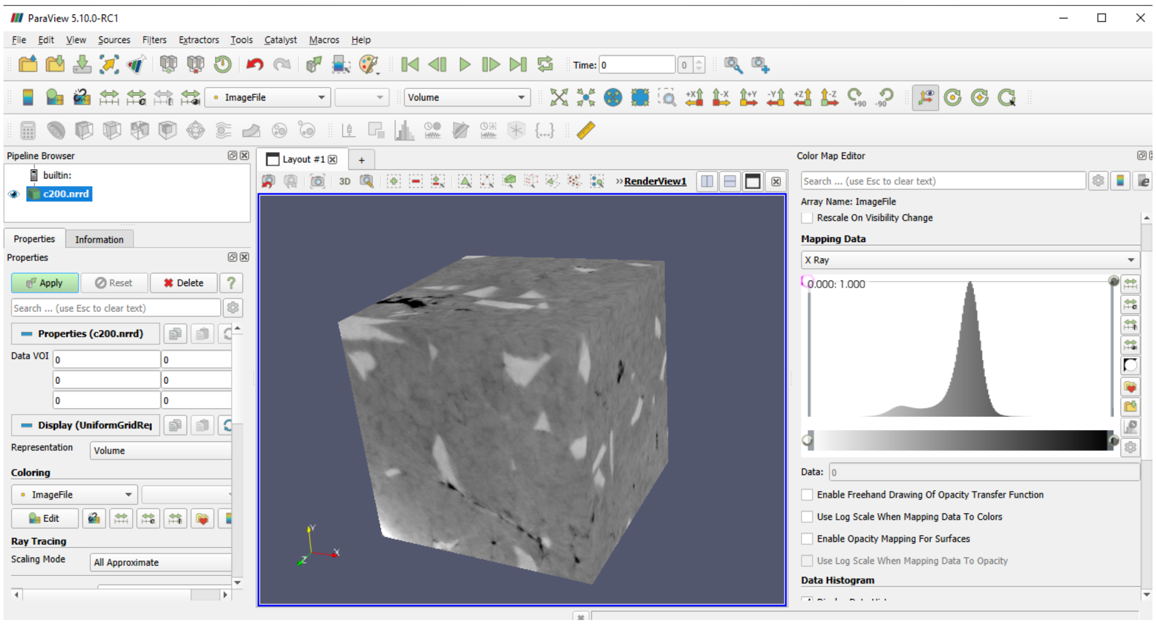
```
NRRD0001
type: unsigned char
dimension: 3
sizes: 200 200 200
spacings: 0.002 0.002 0.002
encoding: raw
```

Figure 2.2: NRRD File Header Example

The specifications vary from file to file. Not all have to be present except for some mandatory fields, such as format version, type, dimension, sizes, and encoding. The "format version", represented by NRRD000X in the example above, follows an alphanumeric code where the last number, represented by X, identifies which version of NRRD file format is being used. The "type"field corresponds to the type of each element of data. The "dimensions"specification indicates the number of dimensions the data has. For example, if the data contains three dimensions, it will represent a three-dimensional object. The "sizes"field informs us about the number of elements that each data dimension has. The "spacings"field represents the size of each voxel in each dimension. The encoding parameter gives us information regarding how the data is written. [7]

### 2.1.4 Visualization

There are many tools to visualize CT images with NRRD format. Paraview was the chosen tool to use during this project for being an intuitive free software. Since it will only be used to verify the results, a more sophisticated and powerful tool is not needed. [8]



Properties (c200.nrrd)
Data VOI 0 0 / 0 0 / 0 0
Display (UniformGridRep)
Representation: Volume
Coloring: ImageFile
Ray Tracing
Scaling Mode: All Approximate
Array Name: ImageFile
Rescale On Visibility Change
Mapping Data: X Ray
0.000: 1.000
Data: 0
Enable Freehand Drawing Of Opacity Transfer Function
Use Log Scale When Mapping Data To Colors
Enable Opacity Mapping For Surfaces
Use Log Scale When Mapping Data To Opacity
Data Histogram

Figure 2.3: Paraview software interface.

## 2.2 Image Processing

After receiving the CT image, changes to the base image must be made to generate a new and better version. This statement comes from the need to achieve better visualization and classification results.

The RGB color model corresponds to an additive color system, where red, green, and blue are combined in a variety of ways to form secondary colors. When equal amounts of red, green and blue are added, this will always result in various shades of gray. The shade of gray will depend on the intensity of color components. Zero intensity for each component will represent the darkest color, black. In opposition, full intensity of all color components will represent white. This color model allows to display colored images on electronic systems, such as computers. When generating histograms or 3d material models this color system will be used.

### 2.2.1 Image Binarization

The first transformation that will be applied will be to enhance the separation between the base material and the material in study. To achieve this the strategy will start by transforming the image to black and white, being the black portion the composite material and the white one the base material[2].

#### 2.2.1.1 Histogram Generation

The first step will be to generate an histogram based on voxel RGB values with the objective of identifying the two highest peaks.[2]

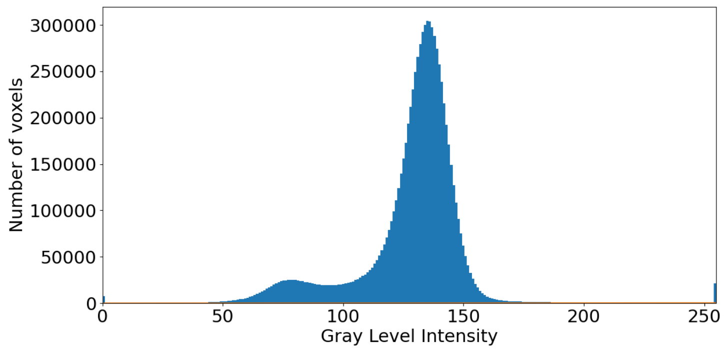


Figure 2.4: Histogram of Voxel Gray Level Intensity Distribution

The most sizeable peak represents the color of the base material, since it is the dominant material in the CT image, as seen in 2.1.2. The second highest peak represents the color of the material that we want to focus on and separate from the base material. At this point the two material's main color are identified and can be used to determine for each voxel, if it belongs to the base or composite material.[2]

#### 2.2.1.2 Image Cleaning

Image cleaning constitutes an assortment of image processing techniques, whose use is to remove image noise and thus enhance the starting image. Image enhancement refers to the improvement of the detectability of relevant image details, by either a man or a machine. This will be based on the important notion that an image contains signals that are unwanted and that we wish to suppress, alongside signals or structures that we want to extract and preserve.

Several methods can be employed to meet this end, such as the removal of small components and grey level mapping, the latter of which will be described below. [9]

In gray level mapping, the pixels value will be changed by transforming them through a function. This function will map input grey values into new output values. This technique can be utilized to obtain a black and white image. Voxels with an RGB value near the value of the highest peak will be assigned the value 255, meaning they will turn white. The same procedure will be applied to the second highest peak, but instead of 255, they will be set to 0 so they can turn black. By doing this, the majority of the points of interest will change color.[2, 9]

This will create an intermediate zone between both histogram peaks that wont be changed, this is called the grey zone. To deal with the grey zone the color assignment to each voxel will be based on its neighbouring voxels. The software will analyze all the voxels surrounding our target voxel. If the majority of the voxels surrounding it are black, the target voxel will be black. If the majority of them are white, the target voxel will be white. Like in a cube, each voxel has 26 other voxels surrounding it: one in each corner, totalling 8, one on each surface, totalling 6 and one at each edge, totalling 12. A schematic representation of this is shown below.

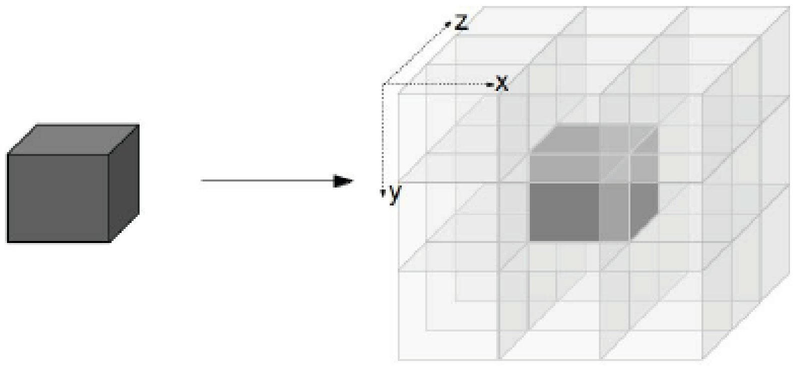


Figure 2.5: Voxel neighbourhood.[10]

Deciding the color of the target voxel based on its neighbours poses an issue: There is an even number of voxels surrounding our target voxel, making it possible for a tie to happen between the number of white and black voxels. This is overcome by randomly assigning a color in cases where there is a tie.[2]

In image 2.6 we can see the result of image cleaning a gray scale CT image to black and white.

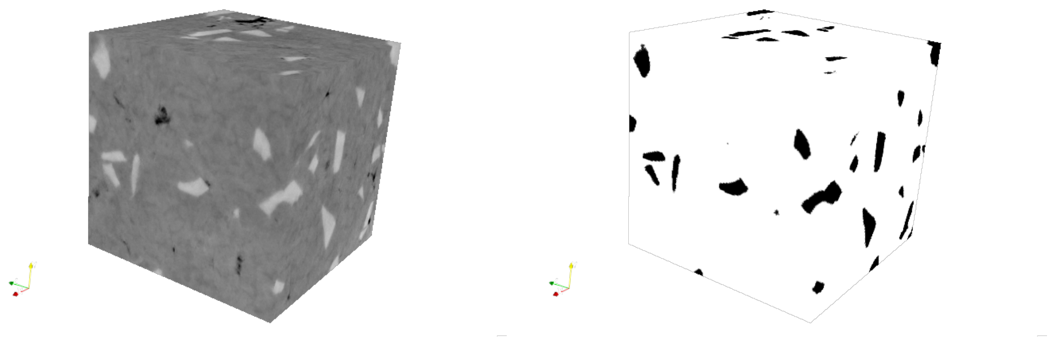

(a) Before (b) After

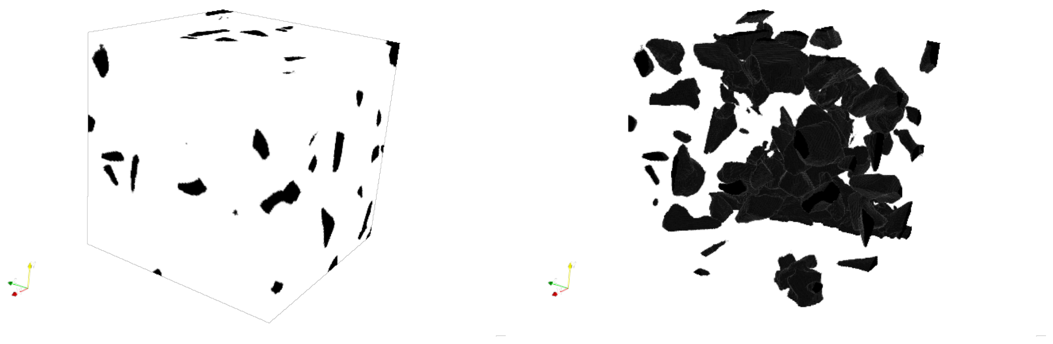
Figure 2.6: Image cleaning of a CT

### 2.2.2 Segmentation

Image segmentation refers to the division of an image into the multiple regions that compose it. The ending point is that the different regions represent separate and meaningful areas of the same image. These regions can correspond to separate groups of voxels in a 3D object, such as the CT images being transformed. By decomposing these images it is possible to analyse its components and characteristics separately, since the voxels get organized in a more meaningful manner. [9]

After the image binarization mentioned in the last section, the composite material is now easier to visualize and able to be segmented. Each fragment, represented in black, is a particle of the material being studied.

The identification of the different particles of the material will be based on notion of neighbouring voxels, already described above. It will start with one known voxel belonging to the composite material. The voxels next to that target voxel are either white (non-material) or black (material). If a neighbouring voxel is black, it will then be added a tag meaning that the voxel in question belongs to the same particle set as the neighbour. This procedure will be repeated until all voxels in a voxel neighbourhood are either white voxels or black voxels that already have a tag assigned to them. When this happens, an independent set of voxels is found and it can be said that a particle of the material was found. This will be repeated until all black voxels have a label assigned, in order to find all the particles that the CT image contain. In the end, each tag will represent a particle, that form a fragment composed by a set of continuous black voxels. Segmentation can be observed in the image 5.1. In the (b) image, a larger portion of the material can be seen due to material particles overlap.



(a) Before (b) After

Figure 2.7: Image segmentation of a CT

## 2.3 Parallelism

While running a program on a computer, by default only one core of the Central Processing Unit (CPU) is used. This will lead to heavy computation that will take a long period of time to terminate. One approach to make heavy programs run faster is to parallelize them if possible.

The parallelization technique consists in splitting the work or/and tasks among workers. Those workers will then execute the operations simultaneously and merge the results in the end. This means that the minimum time that a parallel program takes to execute is equal to or higher than the maximum amount of time the workers take to finish it.

Despite the existence of multiple parallel programming architectures, SIMD is the one that the solution to be developed will focus on.

SIMD means that the initial data set will be split in a way that all workers execute the same instructions on the slice of data that is assigned to them. In the current problem, the initial data set will be the matrix of voxels of the tomographic image, then the matrix will be sliced and assigned to a pool of workers. This way, the handling of the extensive amount of voxels will be processed simultaneously and consequently faster [3] [11].

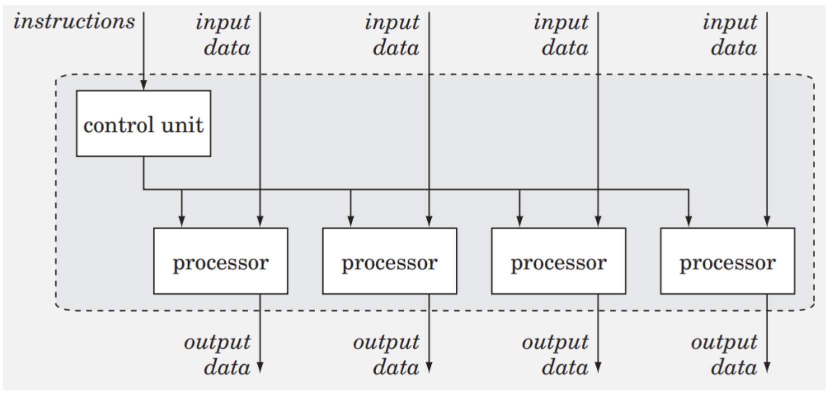


Figure 2.8: Architecture of SIMD. [12]

### 2.3.1 Shared Memory

In a shared memory system, each CPU core has access to the same memory. Parallelism benefits from a system like this, since data can be centralized in the global shared memory and different CPU cores are able to read and modify it. In parallel programming each core can work on top of a portion of the global shared memory, splitting the processing work. Two solutions to build parallel programs using CPU cores are PThreads and OpenMP. [13]

Another technique commonly used in the image processing area to make parallel programs is to make use of Graphics Processing Unit (GPU) processing power, which can benefit from having more cores than CPU and still process the data in a SIMD approach.[3]

This method can be employed through the use of OpenCL or CUDA, the latter in case an NVIDIA GPU is being utilized.[14]

#### 2.3.1.1 CUDA

CUDA is a platform developed by NVIDIA that allows programmers to execute regular functions on their GPUs. When CUDA code is compiled it generates machine code for the CPU and GPU. The portion of code that runs on the CPU is responsible for sending and receiving the data from GPU, initializing variables, and defining the execution of kernel functions. The portions of code that run on GPU are called kernels. These can be composed of operations over arrays or matrices of data, such as 2D and 3D image representations. When a kernel function is executed multiple instances of that function are created. Different instances are assigned to different threads, and run in parallel, resulting in a SIMD processing pattern. GPU parallelism potential will be explored in this project since GPUs provide a significantly larger amount of cores when compared to CPU. This allows for image processing algorithms to run faster. There are plenty of programming languages that support interaction with the CUDA platform, and Julia programming language is one of them. A closer look into how Julia interacts with CUDA will take place in section 2.4 since the machine where the project will be developed on contains an NVIDIA GPU. [3, 11]

### 2.3.2 Distributed Memory

In a distributed memory system, each CPU has its own memory. These memories have their addresses shared among the other CPUs and can be accessed by them through an interconnection network This allows to obtain more computation power and be able to process bigger data in memory, in opposition to shared memory in a local computer, where the there is a limit in the amount of CPUs and memory that computer has. However, this system poses a challenge: since different machines are being used, there is time expanded in inter-machine communication. This needs to be taken in consideration when making the decision between a shared memory and distributed memory system, to opt for the most time efficient option. Two good examples of this are MPI and Spark.

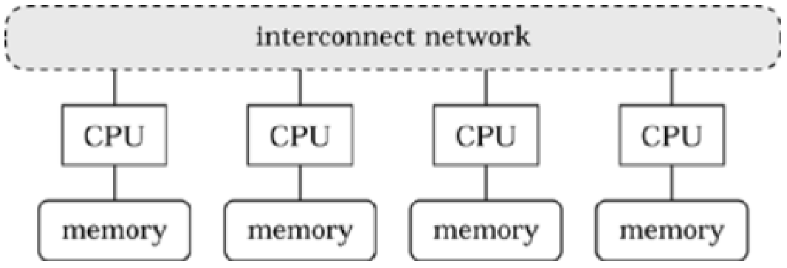


Figure 2.9: Architecture of distributed memory. [12]

## 2.4 Julia

Depending on the problem nature, different programming languages are more appropriate to different solution scenarios. Looking at a problem where the speed of the solution is the most important, a low level and statically typed language like "C"would probably be the best approach. One of the problems of building programs in low level languages is that sometimes it can be difficult to write, read and maintain when compared to high level and dynamically typed languages like "Python"for example. Trying to get the advantages of both worlds was how Julia programming language was born.[5]

Julia is a dynamically typed high level programming language with several favorable characteristics that led to it being chosen. First, Julia's remarkable speed was possible owing to two main factors: being a Just In Time (JIT) compiled language and using a Low Level Virtual Machine (LLVM) compilation infrastructure. A JIT compilation consists on a technique where high level languages are converted to machine code when they are being run on a CPU. This makes them independent from the source languages runtime, unlike what happens in interpreted languages. The LLVM is a compiling technology and toolkit that automates many of the tasks involved in language creation. This compiler does not make a language faster by itself, but Julia's design and the way it employs types allowed this to happen. One example lies in the ˆ (power) operator that can be used with either an integer or floating points argument. Julia will compile both versions of the code and call the appropriate one, avoiding type checks and making it faster. [5]

Julia is a programming language built by a group of scientists, engineers, and mathematicians to overcome their needs. Often time, researchers need to opt between a user-friendly syntax or a faster language. Julia is characterized by performing nearly as quickly as C programming languages, while at the same time maintain good code readability, similarly to Python. This programming language uses a Read Evaluate Print Loop (REPL) console, that allows for faster code compilation by only compiling the code once. This is possible through a mechanism where the first time the code is executed, it is compiled and stored in memory, thus the following times it is executed there is no need to wait for the compilation to happen. In this manner, Julia allows for the best of both worlds, by providing exceptional speed and a pleasant and familiar interface. [15]

REPL is a computer programming environment that, as the name suggests, takes individual user inputs and reads, evaluates and then returns them to the user. This mechanism creates a loop that ends when the user closes the program.

In recent years Julia has gone through an increase of popularity and continuous development of new packages. These factors aroused the curiosity of recreating the previously developed image processing solution, described in 1, in Julia. By doing this it would be possible to evaluate how Julia performs in image processing efficiency and efficacy. It is expected to improve existing work and save a significant amount of time lost during compilation. 2.4.

### 2.4.1 Julia Code Example

The Julia code snippet available below was used to generate the CT image transformation shown in image 2.6. The material voxels were made black (represented as value 0 in the code) and the remaining voxels were made white (represented as value 255 in the code).

```julia
using FileIO
using NRRD

function image_transform(img)
    for idx = 1:length(img)
        if img[idx] > 60 && img[idx] < 90
            img[idx] = 0
        else
            img[idx] = 255
        end
    end
    return img
end


function main()
    img = load("base_image.nrrd")
    img = image_transform(img)
    save("new_image.nrrd", img)
end

main()
```

### 2.4.2 Parallelism Support

From the vast list of features that Julia provides, one that will be essential for the development of the proposed project is parallelism support. Julia supports several categories of parallel programming, including multi-threading, distributed computing and GPU computing.

SIMD, as described above, constitutes a parallelizing method where a task data is split and processed by several processing units simultaneously. LLVM compiler allows the code to run in parallel even in the absence of external hints, except in the presence of some limitations.

To guarantee that certain operations will run as SIMD instructions the use of *SIMD.jl* package provides the necessary types and functions to specify this.[5]

Threads refers to the independent and simultaneous execution of code on multiple CPUs. The number of threads Julia can run on is set at startup, unless if it remains undefined, in which case the default number of threads is 1. In Julia the @threads macro can be used before a loop, which will result in that loop being parallelized in different threads. [5]

Code can also be accelerated through the use of a GPU, by running thousands of threads simultaneously. Julia's interoperability with C makes it possible to call GPU libraries like *CUDA.jl* package. This package resulted from merging several previously existing packages, including *CUDAnative.jl* and *CuArrays.jl*. From all the parallelism options Julia offers. CUDA will certainly be used since the machine where the program will be developed uses an NVIDIA GPU. [5]

Other relevant packages to mention include *FileIO.jl* alongside the *NRRD.jl* package, since together they will simplify the interaction with NRRD files. The built-in package manager is a very useful Julia feature on this topic, since it simplifies the process where the user obtains the latest and correct packages. [5]

Now that there is a complete understanding of the relevant basic concepts, it is possible to move forward and develop the intended solution.

# Chapter 3

# Solution

In the present chapter, the solution organization, as well as the decisions behind it, will be described thoroughly.

## 3.1 Organization

In order to accomplished what was proposed, the project will need to be divided in two main portions. Firstly, the initial image will be split into two parts. Each voxel will be assigned one of two possible values: 0 or 255. The value 0 corresponds to the target material that we intend to study, while 255 corresponds to the base material. There are a lot of ways in which this can be done. Some of them, including the method chosen and applied in this scenario, will be covered in the next section.

To begin the second step it is necessary that the target material and base are already properly separated. Following that first step, the different material particles will have to be uniquely identified. To achieve this, contiguous groups of voxels will be attributed an unique value or label each. The unique label of each particle will allow researchers to analyse the characteristics and properties of each fragment individually.

It is to be noted that this whole process will be performed resorting to a GPU. The relevant choices that had to be made regarding its implementation will be explained and justified in the subsection bellow.

## 3.2 Specification

In the present section specifications of the solution developed, namely of the GPU, material labeling and particle labeling, are presented.

### 3.2.1 GPU

Expanding on the topic above, Julia contains three main packages that assist with the GPU interaction. They are *oneAPI.jl*, *AMDGPU.jl* and *CUDA.jl* packages.

The first runner up, *oneAPI.jl* package, makes use of the oneAPI framework developed by intel.

This package, according to the most recent documentation regarding Julias implementation, *oneAPI.jl* is currently only supported on 64-bit Linux. This is a downside, considering that the machine used for project development and testing runs a Windows operating system. Furthermore, it currently only supports a limited set of Intel GPUs. This package is still under significant development, so it is expected to have some bugs and missing features. Hardware support is also limited, and the package has not been extensively tested, making it possible that performance issues might be present. For the reasons listed above, this package was not put into use in this current project.

Another package alternative is the *AMDGPU.jl* package. This package, however, is also only supported by 64-bit Linux. A necessary prerequisite to use this Julia package is to have a working ROCm stack installed. ROCm, short for Radeon Open Compute platform, is AMD's open-source GPU computing platform. This platform is only supported by most modern AMD GPUs and some AMD Accelerated Processing Unit (APU)s. Currently, ROCm works solely on Linux, with no future plans to support either Windows or macOS announced by AMD until present time. Since the machine used to develop the project is a 64-bit windows machine, it was not possible to utilize the *AMDGPU.jl* package. Additionally, this package did not have all the features and performance level of the *CUDA.jl* package.

This leads us to our choice, the *CUDA.jl* package. As mentioned previously, the machine chosen to develop and test this project has a Windows 10 64-bit operating system and an NVIDIA GPU. CUDA, or Compute Unified Device Architecture, is a technology developed mainly to be used alongside NVIDIAs graphic cards. This package is, additionally, the most stable and complete Julia package available for users at the present time.

In summary, the GPU interaction package chosen to for this project was *CUDA.j*. Despite the fact that the machine in which the program was developed contains two GPUs: An NVIDIA and Intel GPU, the NVIDIIA graphics card is more powerful. Furthermore, even though *CUDA.jl* was developed for NVIDIA GPUs, Julia's package implementation can also interact with both Intel and NVIDIA graphic cards. Additionally, *CUDA.jl* is a more stable, bug-fee version of the other packages, according to the documentation available.

### 3.2.2 Material Labeling

To separate the base material from the material synthesized by the researchers, it was decided to start by thresholding. Thresholding corresponds to a simplified way to differentiate whether or not a voxel belongs to out target material, based on a set of values attributed to voxels. This method is simpler but consequently less precise. It was concluded that by performing an imprecise method at first the number of iterations needed for the hysteresis algorithm would decrease. This allows to decrease the execution time for hysteresis while maintaining the quality of the classification of different materials. This was inferred through trial and error.

#### 3.2.2.1 Thresholding

To perform thresholding, it is necessary to define a minimum and maximum, according to the histogram of the voxel value distribution of the CT image.

In the histogram, each material corresponds to a peak, since each peak represents a conglomerate of voxels with a value close to each other. In this scenario, there are two materials, the base and study material, and thus there are two peaks. It can be assumed that starting the beginning of the horizontal axis until the first peak we have the first material, and from the second peak onward we have the second material. This leaves us with an interval of voxels between the two peaks that need to be classified as belonging to one material or the other. To do this, it is required a more precise classification mechanism to precisely identify the border that separates the two. The function below can exemplify the voxel value classification, assuming that the first peak would be the material that is being processed.

$$F(x) = \begin{cases} 0 & 0 \leq x \leq \text{min} \\ x & \text{min} < x < \text{max} \\ 255 & \text{max} \leq x \leq 255 \end{cases}$$

#### 3.2.2.2 Hysteresis

At this point it would become beneficial to apply an algorithm that would allow to separate the two materials with greater precision

The method found to achieve the correct labelling of the voxels in the "gray area"was to iterate through these voxels and classify them based on their neighbours. Meaning that a voxel will be attributed the same value, as the absolute majority of the neighbouring voxels that surround it. This process could not label every single voxel in a single execution, since voxels with a majority of unassigned neighbours might exist. However, this could be solved with more iterations, since with each one more and more voxels get labels assigned to them, and those could be neighbours to other unassigned voxels. By this logic, the algorithm would be executed an X amount of times, where X corresponds to the number of times at least a single voxel of the image had its value altered.

When the iteration cycle ends and no voxels were altered in the last iteration, there might still be unassigned voxels remaining. This might happen because no neighbour has an assigned value, or there is no absolute majority of a label in the assigned neighbours.

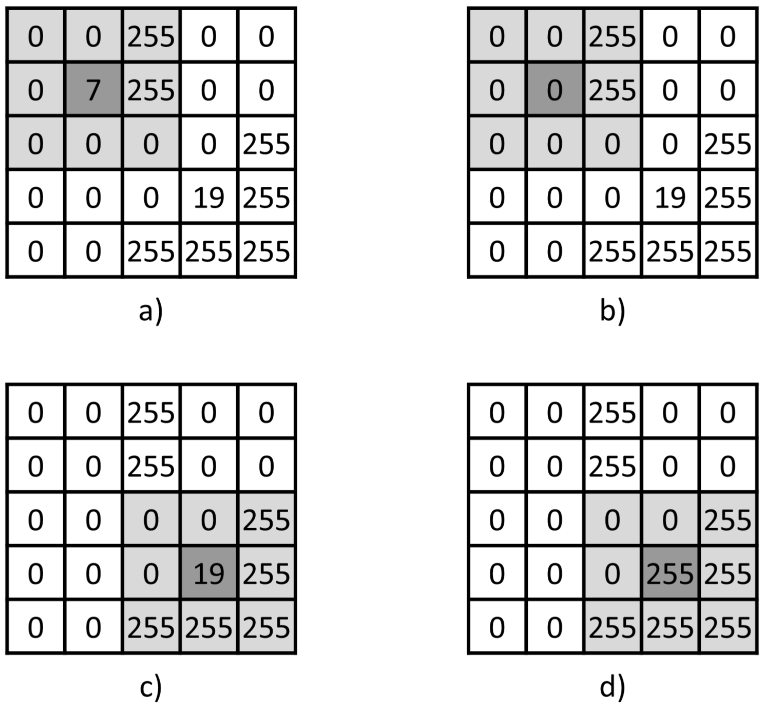


Figure 3.1: Example of how absolute majority voxel assignment works in a 2D image slice.

To continue the label attribution to each voxel, the process will need to be slightly altered and the precision of the classification will decrease as well. Instead of the label attribution being based on the absolute majority of the neighbouring pixels, it will be based on the relative majority. This means that irrespective of the number of neighbours, the label assigned to the voxel will correspond to their majority. This can be better comprehended by observing the figure bellow. Even if only one of the neighbours has a label assigned, that voxel will be assigned that same label.

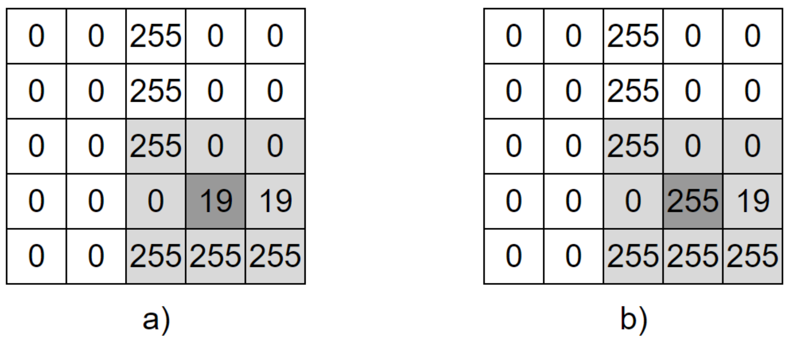


Figure 3.2: Example of how the simple majority voxel assignment works in a 2D image slice.

When the image doesn't suffer any changes after an execution, it is to be expected that at least 99 percent of pixels are properly labelled. The remaining minority exists due to the number of neighbour voxels assigned to the background material and the target material being the same.

In this scenario the most extreme and imprecise method for voxel classification is used. Through a single iteration, voxels are randomly assigned as belonging to the background or target material. By the end of this step, it is guaranteed that every voxel in the image is labelled either 0 or 255. Considering the image is completely labelled, it is possible to proceed to the next step.

### 3.2.3 Particle Labeling

Now every voxel in the image was attributed one of two possible values. The easiest way to distinguish every particle of the target material from the background is to look at every voxel with the corresponding label, and detect contiguous voxel groups. In this case in particular, the study material was labeled as 0. Each contiguous voxel group will correspond to a material particle.

Connected Component Labeling (CCL), or connected component labeling algorithms, constitute a way to achieve the separation described above. The CCL algorithms provide an analysis of every pixel present in an image, or voxel when applied to a CT scan. It is assumed that the input image is a binary image composed of a background and foreground. In this case, the foreground corresponds to the study material. These algorithms have the purpose of labeling every pixel or voxel of an image based on its neighbours, and to do so it usually requires multiple iterations. In the first run, if the voxel is considered to be part of the background, it is attributed the label -1. If a voxel is considered part of our study material it is attributed an unique numerical label. In the second run, the algorithm will iterate through each volume slice, with the thickness of one voxel, first by row then by column.

In this step, different labels attributed to voxels in the same particle acquire the same label. This is done analysing voxel's neighbourhood. If the labels of the neighbours differ from the label attributed to the current voxel, the voxels with the highest label will be assigned the smallest label in the neighbourhood.

Since this previous step can generate label changes based , the algorithm does multiple reruns until there aren't label changes. The end goal of CCL is to label every connected voxel, by attributing them the same label. Often this is done after a segmentation algorithm is applied to the input image. A variety of CCL algorithms exist, such as Accelerated Connected Component Labeling (ACCL), Block-based Union-Find (BUF) and Block-based Komura Equivalence (BKE). ACCL in particular is a known algorithm in the image processing field. As the name suggests, it corresponds to a faster version of CCL that can run on NVIDIA's CUDA framework, among others. It is used to decrease the time needed to run an image analysis, compared to CCL

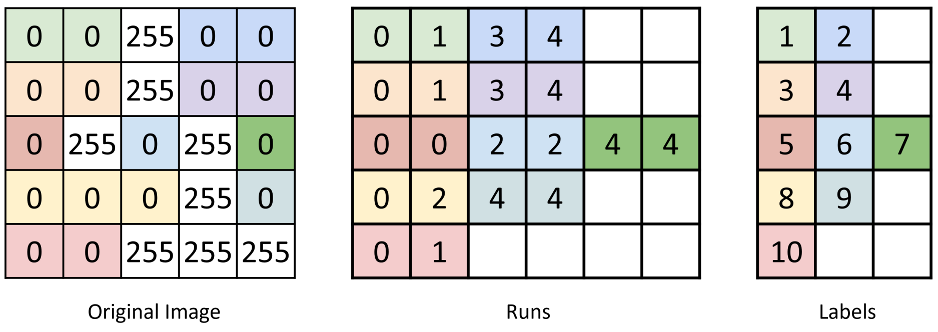
Based on the dissertation written by João Ribeiro [4] on the performance of the three algorithms cited above, it was concluded that ACCL demonstrated superior performance in almost every case, except for some highlighting run-based algorithm deficiencies. For this reason, the algorithm chosen for particle labelling was ACCL.

#### 3.2.3.1 Image representation

To ensure ACCL performance is magnified, the initial CT image matrix is divided into two. This is done after the first step of this algorithm. From this point onward the CT image will be split into the two following matrices: The *Runs* matrix and the *Labels* matrix, as can be observed in picture 3.3.

The index corresponds to the voxel position in a certain line, where the count begins at 0. The *Runs* matrix is where the indexes that represents where the contiguous voxel portions begin and end are stored. It is to be noted that the maximum size the *Runs* matrix corresponds at most to the size of the initial matrix plus one. In this scenario, the line would need to have an odd number of elements that are all discontinuous voxels, meaning if one belongs to the material, the following voxel wouldn't, and so forth. The size of this matrix will correspond at most to X/2 contiguous voxel portions, rounded up, where X equals the number of columns of the original matrix. This is due to the run matrix registering two positions for each contiguous voxel group (the index where it begins and the index where it ends). X / 2 rounded up times 2 equals at most X + 1, therefore, the maximum number of elements per row in the *Runs* matrix would be X+1.

The *Labels* matrix, as the name suggests, is the matrix where the labels are stored. Each contiguous voxel group will have an unique label attributed. Since, as demonstrated above, a row could have a maximum of x / 2 labels rounded up, the maximum size of this matrix will correspond to half the size of the original matrix, rounded up.



| | | | | |
|---|---|---|---|---|
| 0 | 0 | 255 | 0 | 0 |
| 0 | 0 | 255 | 0 | 0 |
| 0 | 255 | 0 | 255 | 0 |
| 0 | 0 | 0 | 255 | 0 |
| 0 | 0 | 255 | 255 | 255 |

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 3 | 4 | | |
| 0 | 1 | 3 | 4 | | |
| 0 | 0 | 2 | 2 | 4 | 4 |
| 0 | 2 | 4 | 4 | | |
| 0 | 1 | | | | |

| | | |
|---|---|---|
| 1 | 2 | |
| 3 | 4 | |
| 5 | 6 | 7 |
| 8 | 9 | |
| 10 | | |

Figure 3.3: New method of image representation through two matrices in a 2D image slice.

#### 3.2.3.2 Find Runs

Now that the image restructuring is understood, lets more forward to the first part of the ACCL algorithm: *Find runs*. This algorithm will only be executed once and will attribute a value to each voxel that belongs to our target material. A counter is present to indicate the value of the current label being attributed. The label attribution is performed in the following manner: - If the current voxel belongs to the material, the the value of the label attributed to it corresponds to the current counter value - If the last voxel processed belongs to the material and the current voxel doesn't, the label counter increments by one. This means the previous contiguous voxel group has ended. - If both the last voxel processed and the current voxel don't belong to the material, then the current voxel is ignored and the algorithm moves forward to the next voxel.

By the end of the execution, every contiguous voxel group in each row has a unique label attributed. However, when observing the image by columns or slices, it is possible for contiguous voxel groups to have different labels attributed. For this reasons, the next step will consist in normalizing that situation and assuring only one label is attributed to each group.

#### 3.2.3.3 Merge Runs

In this step, much like hysteresis, multiple executions will be needed until all labels converge and there are no image changes during a full iteration.

The algorithm will go through each voxel and look at the neighbours that have a label attributed to them, meaning they belong to our material. When the algorithm locates a neighbouring voxel with a label different from the label of our current voxel, it will assign the smallest label of the group to voxels with different labels. This will make all voxels in the same particle to have same label.

Note that due to the new image representation, explained above, if one of the elements in the "Labels"matrix is changed, every voxel in that row of neighbouring slice that are continuous with the voxel will be altered.

When the algorithm is executed and no labels are changed, it means that all contiguous voxel groups have had their labels corrected and every particle is properly identified. However, the image processing is still not complete, since the matrix needs to be represented as it was before, and not by 2 matrices.

#### 3.2.3.4 Decompression

The last step of ACCL algorithm, decompression, is executed in a single iteration. This step consists in filling a matrix, the same size of the original image, based on the *Runs* and *Label* matrices values. By the end of this step, a new image will be generated, where every particle is individualized and separated from the others.

Described above are all the steps that make up the developed algorithm. This theoretical basis will ease the understanding of the code implementation described in the next chapter.

# Chapter 4

# Implementation

In this chapter all the previously described concepts and methods will be put into practice. Code snippets alongside explanations provide a full overview on how the developed solution works.

## 4.1 Image Cleaning

The first implementation step consists of image cleaning, as explained in chapter 3. Firstly, the chosen tomographic image, in NRRD format, needs to be imported. To import the image the *NRRD.jl* package was used. This package implements the *FileIO.jl* interface, allowing the user to interact with the file in a simple and efficient manner. An alternative method would be to open the CT image with the *FileIO.jl* directly. This, however, would read the image as a text document that would later need to be interpreted and analysed. The *NRRD.jl* package will be used again later to save the image after its processing stage.

After importing the target image, its data then needs to be processed by a GPU. The machine where this project was developed has an NVIDIA graphic card, and the *CUDA.jl* package was chosen to handle the interaction with the graphic card. Another package could have been used, but this one was chosen based on its specificity for NVIDIA graphics cards.

The data needs to be converted to a format the GPU can read. Then, after data processing, the GPU output will then be re-converted to a CPU readable. To convert the data type between CPU and GPU data, the functions "gpu"and "cpu"from the *Flux.jl* package were used.

The following code snippet loads the "image_file.nrrd"image, converts it to a GPU readable datatype and creates an auxiliary variable with the same size of the image where the GPU output will be place.

```
img = load("image_file.nrrd")
gpu_img =  img |> gpu
gpu_img_aux = CUDA.zeros(size(img))
```

### 4.1.1 Thresholding

The algorithm used for thresholding was relatively simple. Through the *map!* function, the *get_value* function was applied on each voxel. Afterward the resulting image was saved as an auxiliary image with the same size as the original.

The function *get_value* receives the three following parameters: voxel, min and max. The voxel parameter corresponds to the singular intensity of the voxel being processed. As explained in chapter 2, each voxel has a value assigned to it. The min parameter corresponds to the maximum voxel value accepted for the value 0 to be assigned to that voxel. In opposition, the max corresponds to the minimum voxel value accepted for the value 255 to be assigned to that voxel.

This way, every voxel with a value between 0 and the min, excluding those with a value equal to min, will be assigned the value 0. The same way, every voxel with a value between the max and 255, excluding those with a value equal to max, will be assigned the value 255. This will create a gray area, where the voxels with a value between min and max, including those with a value equal to min and max, won't be assigned a new value and will maintain their initial value. This grey area was previously explained in chapter 2.

```
function get_value(voxel, min, max)
    if voxel < min
        final_v = 0
    elseif voxel > max
        final_v = 255
    else
        final_v = voxel
    end
    return final_v
end

map!(elem -> get_value(elem, 85, 110), gpu_img_aux ,gpu_img)
```

By the end of this stage, our initial image will be divided in three components: material A with an assigned value of 0, material B, with an assigned value of 255, and the gray zone.

To finalize image cleaning in a precise manner, voxels from the gray zone need to be assigned either 0 or 255. To separate the gray zone voxels hysteresis is used..

### 4.1.2 Hysteresis

Hysteresis can be split into three main steps. In the first step, the decision algorithm will run for each voxel in the gray area. This algorithm will decide the new value attributed to those voxels. This decision will be based, as explained in chapter 3, in the neighbouring voxels that surround the target voxel. If the absolute majority of the neighbouring voxels is 0 or 255, our target voxel will be assigned that respective value. The algorithm will run iteratively until no voxel value is changed during a full iteration.

The second step is only faintly different to the previous one, and is applied when the absolute majority of the neighbouring voxels can't be used as a deciding factor to assign a gray area voxel a value. For example, if a target voxel has less than or equal to half of it's neighbours with 0 value and 255 value assigned. In this situation, the relative majority of neighbouring voxels is used. Meaning that to make a decision the algorithm will assign the value that corresponds to the largest number of neighbouring voxels, regardless if there are more than fifty percent of the neighbours with the same value assigned, as it was done in the previous step

In the function bellow there are 3 parameters: img, that corresponds to the image input; img_out, that corresponds to the processed image, with the changed voxels; and maj, the flag to indicate if the function is going to user either the absolute or relative majority as deciding factor.

```
while gpu_img_aux != gpu_img
    copyto!(gpu_img, gpu_img_aux)
    @cuda threads=config.threads blocks=config.blocks
    hysteresis!(gpu_img, gpu_img_aux, 1)

end
```

```
function hysteresis!(img, img_out, maj)
    (x_size, y_size, z_size) = size(img)
     x = (blockIdx().x -1) * blockDim().x + threadIdx().x
     y = (blockIdx().y -1) * blockDim().y + threadIdx().y
     z = (blockIdx().z -1) * blockDim().z + threadIdx().z

    if x <= x_size && y <= y_size && z <= z_size
        voxel = img[x, y, z]
        if voxel % 255 != 0
            count_mat1 = 0
            count_mat2 = 0
            total_neigh = 0
            for x_idx = max(1, x - 1):min(x + 1, x_size)
                for y_idx = max(1, y - 1):min(y + 1, y_size)
                    for z_idx = max(1, z - 1):min(z + 1, z_size)
                        total_neigh = total_neigh+ 1
                        neighbour = img[x_idx, y_idx, z_idx]
                        if neighbour == 0
                            count_mat1 = count_mat1 + 1
                        elseif neighbour == 255
                            count_mat2 = count_mat2 + 1
                        end
                    end
                end
            end

            if maj == 1
                if count_mat1 > Int(ceil(total_neigh/2))
                    img_out[x, y, z] = 0
                elseif count_mat2 > Int(ceil(total_neigh/2))
                    img_out[x, y, z] = 255
                end
            else
                if count_mat1  > count_mat2
                    img_out[x, y, z] = 0

                elseif count_mat1  < count_mat2
                    img_out[x, y, z] = 255
                end
            end
        end
    end
    return
end
```

The third and final step of hysteresis is performed when after the previous steps it wasn't possible to attribute all the gray area voxels to either material A or B. This happens, for example, when a target voxel has the same amount of neighbouring voxels with value 0 as with value 255. In this instance each remaining unassigned voxel will be randomly assigned either the value 0 or the value 255. This will be performed during a single iteration with the *map!* function, as can be observed in the code snippet bellow.

```
function hysteresis_random(voxel)
    if voxel % 255 != 0
        return rand((0,255))
    end
    return voxel
end

map!(elem -> hysteresis_random(elem), gpu_img ,gpu_img_aux)
```

## 4.2 Segmentation

From this point forward there are 2 distinct materials, fully separated. This makes it possible to now identify isolated voxel groups, and each voxel group is classified as a particle of material.

Numerous Connected Component Labeling, or CCL algorithms are available, but the one used in this project was ACCL. This algorithm didn't have a significant performance difference when compared to other algorithms available that are more complicated and difficult to maintain as explained in chapter 3.

### 4.2.1 ACCL

ACCL, as explained in chapter 3, is a labeling algorithm. This algorithm has the goal of identifying groups of connected voxels that compose each particle. The ACCL implementation is divided into three portions: "Find Runs", "Merge Runs"and "Decompression". The decompression is only needed for optimization purposes, as explained bellow.

#### 4.2.1.1 Find Runs

The "Find Runs"function consists of finding voxel sequences of the study material line by line in an image slice with the thickness of one voxel. Each sequence found will increment the next label value by one.

To improve machine performance two matrices were used instead of working with a full image. These matrices have position indexes representative of the original position of the elements and the according labels in the original matrix. These matrices are the runs matrix and the labels matrix. Only one iteration is required to achieve both matrices. This matrix structure is explained in detail in chapter 3.

```
function find_runs!(img, runs, labels)
    (x_size, y_size, z_size) = size(img)

    # slice to be processed by this thread
    slice = (blockIdx().x -1) * blockDim().x + threadIdx().x
    # row to be processed by this thread
    row = (blockIdx().y -1) * blockDim().y + threadIdx().y

    # runs and labels matrix current index initialization
    pos_runs, pos_labels = 1, 1
    # does previous voxel belong to the material in study?
    run_found = false

    # column to be iterated by this thread
    column = 1

    # ensure that index are within matrix bounds
    if slice <= x_size && row <= y_size
        # iterate all columns of the the current slice and row
        for idx = 1:z_size
            column = idx
            # if the voxel belongs to the material in study
            if img[slice, row, column] == 0
                # if previous voxel does not belong to the material
                if !run_found
                    # mark previous voxel as belonging to the study material
                    run_found = true
                    # calculate an unique identifier based on the
                    #current slice, row and column combination and
                    # assigned to the current particle.
                    labels[slice, row, pos_labels] =
                    y_size * z_size * (slice - 1) + z_size * (row - 1) + column
                    # assign starting index of the particle to the
                    # correspondent runs matrix index
                    runs[slice, row, pos_runs] = column
                    # increment labels matrix insert position
                    pos_labels = pos_labels + 1
                    # increment runs matrix insert position
                    pos_runs = pos_runs + 1
                end
            # if the voxel does not belong to the material in study
            else
                # if the previous voxel belongs to the material in study
                if run_found
                    # assign previous voxel as not belonging to the
                    # material in study
                    run_found = false
                    # assign ending index of the particle to the
                    # correspondent runs matrix index
                    runs[slice, row, pos_runs] = column - 1
                    # increment runs matrix insert position
                    pos_runs = pos_runs + 1
                end
            end
        end
        # If last processed voxel belongs to the material, the ending position of
        # the particle must be assigned in runs matrix
        if run_found
            runs[slice, row, pos_runs] = column
        end
    end
    return
end
```

#### 4.2.1.2 Merge Runs

The second step, "Merge runs"consists of finding runs with different labels that neighbour each other and merge them into a single label. The label chosen to keep is always the one with lowest value among them. This is a iterative algorithm that runs until there are no further changes in the labels values.

```
function merge_runs!(runs, labels)
    # slice to be processed by this thread
    slice = (blockIdx().x -1) * blockDim().x + threadIdx().x
    # row to be processed by this thread
    row = (blockIdx().y -1) * blockDim().y + threadIdx().y
    (x_size, y_size, z_size) = size(labels)

    # ensure that index are within matrix bounds
    if slice <= x_size && row <= y_size
        # iterate all columns of the the current slice and row
        for column = 1:z_size
            # obtain the current label of the labels' matrix
            curr_label = labels[slice, row, column]

            # if the current label does not belong to the material
            # in study, end execution of this thread.
            if curr_label == 0
                return
            end

            # get start index of the particle
            run_start = runs[slice, row, column*2 - 1]
            # get end index of the particle
            run_end = runs[slice, row, column*2]

            # x_idx - x position of the neighbourhood
            for x_idx = max(1, slice - 1):min(slice + 1, x_size)
                # y_idx - y position of the neighbourhood
                for y_idx = max(1, row - 1):min(row + 1, y_size)
                    # z_idx - z position of the neighbourhood
                    for z_idx = max(1, column - 1):min(column + 1, z_size)

                        # get previous slice label
                        prev_slice_label = labels[x_idx, y_idx, z_idx]

                        # stop checking neighbourhood if the previous slice
                        # does not belong to the material in study
                        if prev_slice_label == 0
                            break
                        end

                        # get start index of the neighbour particle
                        prev_slice_run_start = runs[x_idx, y_idx, z_idx*2 - 1]
                        # get end index of the neighbour particle
                        prev_slice_run_end = runs[x_idx, y_idx, z_idx*2]

                        # if neighbour particle and current one connect
                        if  (prev_slice_run_start >= run_start - 1
                        && prev_slice_run_start <= run_end + 1) ||
                        (prev_slice_run_end >= run_start - 1
                        && prev_slice_run_end <= run_end + 1)

                            # if neighbour particle label is lower than
                            # the current
                            if prev_slice_label < curr_label
                                # assign the neighbour label value to
                                # the current particle
                                labels[slice, row, column] = prev_slice_label

                             # if current particle label is lower than
                             # the neighbour
                            elseif  curr_label < prev_slice_label
                                # assign the current label value to
                                # the neighbour particle
                                labels[x_idx, y_idx, z_idx] = curr_label
                            end

                        end
                    end
                end
            end
        end
    end
    return
end
```

### 4.2.1.3 Decompression

As a final step, the initial matrix will need to be recreated taking the previous two matrices, runs and labels, as a starting point. Note that this step is only needed because the initial matrix was initially split into two in order for performance reasons. A single matrix is generated, with the same size as the initial image and with all the individual particles identified. Only a single iteration is needed to achieve this.

```
function decompression!(runs, labels, out_img)
    slice = (blockIdx().x -1) * blockDim().x + threadIdx().x
    row = (blockIdx().y -1) * blockDim().y + threadIdx().y
    (x_size, y_size, z_size) = size(labels)

    if slice <= x_size && row <= y_size
        for column = 1:z_size
            curr_run_label = Int(labels[slice, row, column])
            if curr_run_label == 0
                return
            end
            run_pos = column * 2
            curr_run_start = Int(runs[slice, row, run_pos-1])
            curr_run_end = Int(runs[slice, row, run_pos ])
            for i = curr_run_start:curr_run_end
                out_img[slice, row, i] = curr_run_label
            end
        end
    end
    return
end
```

Now, with a solution fully developed, it is possible to proceed to the assessment of its applicability. To do this, the program will be tested on several CT image examples.

CHAPTER 5

# Results and Discussion

In this chapter the solution developed will be tested on several tomographic images. Some of these images are synthesised through a synthetic image generation method. Others, are real life examples of CT images obtained by material science researchers. As the results are presented, they're discussed and explained throughout this chapter.

## 5.1 Processing Time

When analysing the processing time, several variables need to considered, such as sample size, machine specifications, and the method with which these values were obtained. In terms of size, the samples have a variable size, and consequently a variable number of voxels. The machine used to run the program had the following specifications:

| | |
|---|---|
| **CPU** | Intel core i7 4710HQ 2.5GHz up to 3.5GHz |
| **GPU** | NVIDIA GTX 850M 2GB |
| **RAM** | 8GB |

Table 5.1: Machine Specifications

An analysis of time and memory metrics in relation to the solution steps over a sample is shown in the table below 5.2. The results displayed are divided in the same way as discussed in chapter 4. The "Main"line in the table consists of the time it took for the complete program to run, including loading and image saving time.

In order to retrieve a non biased metrics, the machine ran the code over the samples a total of six times. The first time the @time function is called, the time measured will include the compilation time. This causes the first run of code to display larger times and space allocations than what was expected. For this reason, the first run was neglected. In the five following runs, the best (shortest time) and worse (longest time) runs were discarded.This was done to normalize the values and minimize errors. The remaining 3 runs were selected and averaged, giving the value displayed in table 5.2.

| | Time (s) | Space (MB) |
|---|---:|---:|
| **Thresholding** | 2.19 | 141.94 |
| **Hysteresis** | 5.80 | 175.26 |
| **Segmentation** | 2.06 | 66.43 |
| **Main** | 9.33 | 410.90 |

Table 5.2: Time and Space Metrics

## 5.2 Image Output

An example of the image output along all the intermediate steps between the initial unprocessed image and the final result, is displayed in this subsection.

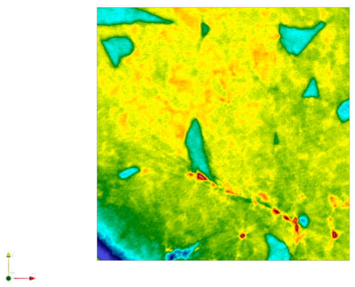
In the initial image (a), can be observed an extensive palette of colors, where the greens and yellows correspond to the base material and the blues to the target sample material.

After thresholding (b), can be seen a more distinct color separation between blue, our sample, and red, the base material is wished to remove. Here, there are also present other colors, that belong to the intermediate gray zone, as described in chapter 2. These will be assigned to either sample or base material through hysteresis, as described in chapter 4.

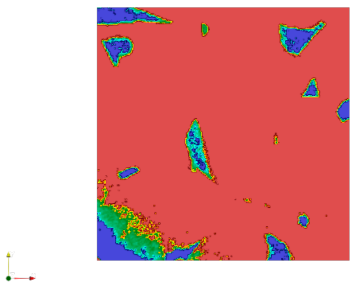
In (c) a duality between red and blue is visible after hysteresis.

Bellow, in picture (d), each particle of the sample material was attributed a unique color, representing a label, to isolate each particle.

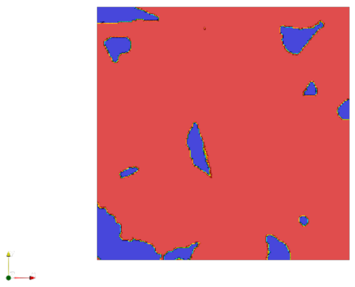
Finally, in figure (e), can be observed the final result, where the base material was removed and the whole sample particles can clearly be visualized.



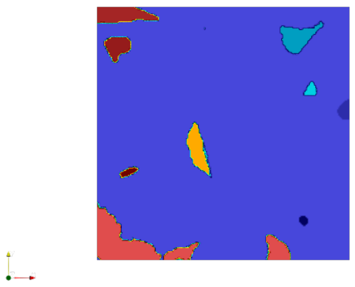
(a) Initial Image



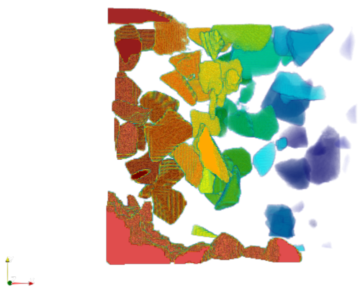
(b) After Thresholding



(c) After Hysteresis



(d) After Segmentation



(e) The Study Material

Figure 5.1: Implementation Output

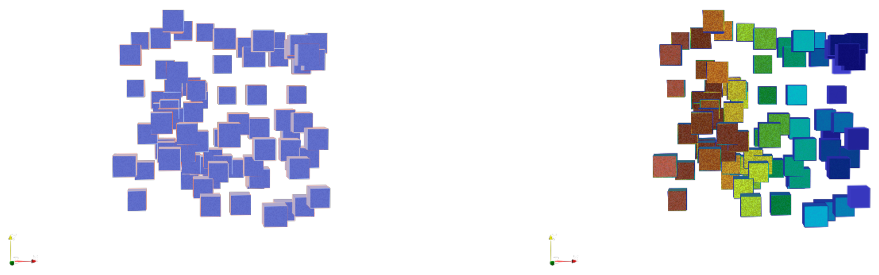
## 5.3 Test Cases

As a means to test the developed solution, test cases will be performed. These can be of two types: real or synthetic synthetic images. These images have varying material pixel percentages, material particle numbers and image dimensions.

### 5.3.1 Synthetic Images

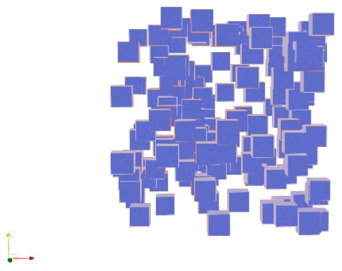
The first tests were performed on the synthetic images. These images were generated through a program written in C language that generates cubes within a volume of 400 x 400 x 400 voxels Each cube generated intends to represent a material particle from a material synthesized by researchers. The CT image generator allows the user to set strict parameters, therefore one can be certain of the volume each cube occupies, by comparing it to the sample. This allows to have certainty on the results obtained after processing the image with the solution developed.

Three synthetic samples were generated, with an escalating percentage of volume occupied by the material, represented by cubes. The first sample is filled by five percent of material, then ten percent and finally fifteen percent. These three values were chosen because they are similar the usual percentage values of the material in study that occupies a real-life sample. A real sample usually contains around that ten percent of material volume, in relation to its total volume.

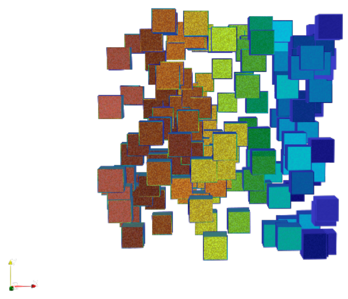


(a) Initial Image (b) After Processing

Figure 5.2: Pre- and post-processing synthetic CT image with 5% sample volume

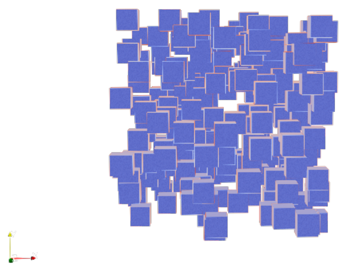


(a) Initial Image

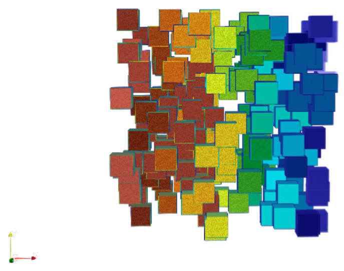


(b) After Processing

Figure 5.3: Pre- and post-processing synthetic CT image with 10% sample volume



(a) Initial Image



(b) After Processing

Figure 5.4: Pre- and post-processing synthetic CT image with 15% sample volume

As it can be observed in the examples above, in all the three tests, the particles were correctly identified.

Regarding the processing time of the developed solution, it is possible to observe in figure 5.5 that it slightly increases as the amount of voxels belonging to the study material also increases. The specific values are specified in the graph bellow. This can be explained by the fact that the ACCL algorithm transformed the original matrix into a new representation. The algorithm needs to work upon that new representation and process more voxels, leading to an increased processing time. This mechanism was explained in depth in subsection 3.2.3.1.

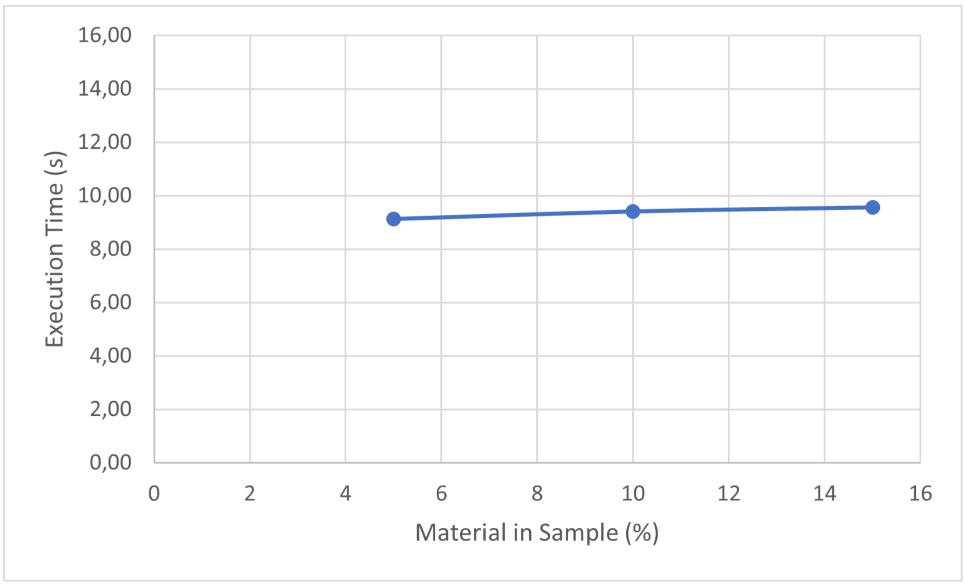
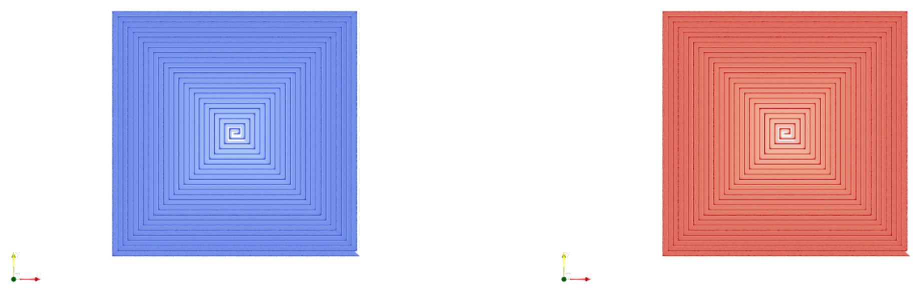


Figure 5.5: Processing time according to material percentage in a 64 million voxel Sample

A test was conducted on an 100x100x100 voxel image that consisted of a single spiral covering the whole sample, as can be observed in the image bellow. This test took 7.26 seconds to run and is particularly interesting due to the existence of only a single large particle within the whole sample. It is visible that after image processing the resulting image only has one color, meaning that the solution created was able to identify successfully the single particle. This example acts as evidence that the merge runs step of ACCL algorithm is working properly, regardless of the number of material particles present in a sample.



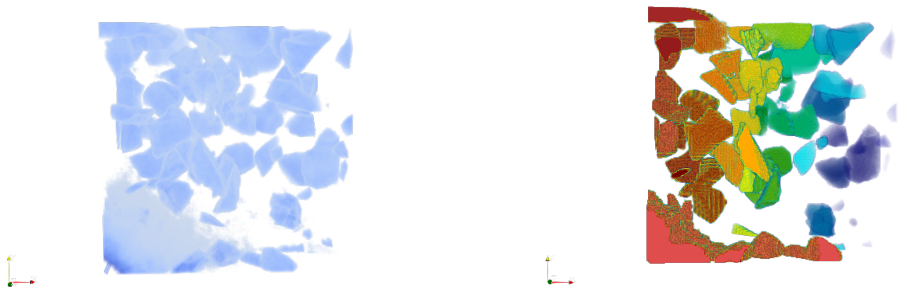
(a) Initial Image

(b) After Processing

Figure 5.6: Pre- and post-processing artificial CT image consisting of a continuous spiral-shaped particle

### 5.3.2 Real Images

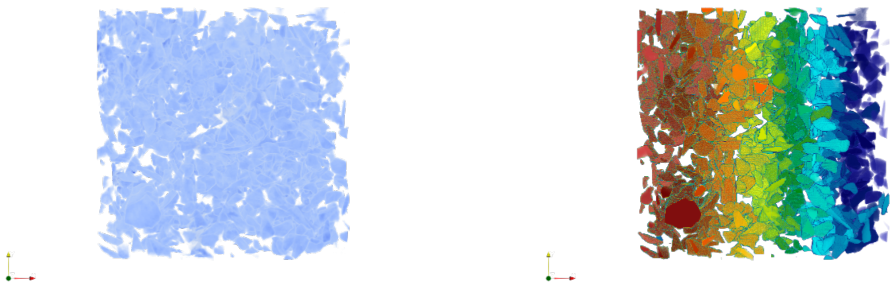
As explained above, it is possible to be certain that the program is correctly classifying every voxel in each particle of the synthetically generated material. Now the developed solution can be tested on real life examples. Next, two sections of the sample C63C2 results are going to be presented and discussed. The CT image bellow represents the first section. This section is composed by 200x200x200 voxels and contains 107 particles in it. The time it took to run the solution on this sample is presented in table 5.2.



(a) Initial Image (b) After Processing

Figure 5.7: Pre- and post-processing of a real CT image fragment, cut from sample C63C2 with 8 million voxels

Additionally, the full C63C2 image from the previous sample was tested, with the intent to verify how the program would perform when faced with an extremely large voxel number. Unfortunately, due to the testing machine specifications, the program was not able to run properly. In order to process a 719 x 701 x 900 CT image, more than 2GB of memory dedicated are needed, more than what the GPU used has. Alternatively, a larger slice of the C63C2 sample with A size of 500 x 500 x 500 was generated. This sample was 15.63 times bigger than the previous sample, with a total of 1013 particles took 55.46 seconds to be processed. This time, the program was able to run and the output image presented a positive result. As seen in the picture below, the whole volume voxels were labelled correctly.



(a) Initial Image (b) After Processing

Figure 5.8: Pre- and post-processing of a real CT image fragment, cut from sample C63C2 with 125 million voxels

The solution developed was confirmed to be functioning on all images it was tested on, regardless of the intrinsic characteristics of said images. However, this will be further discussed in the next chapter.

Chapter 6

# Conclusions

Material science researchers are in the constant search for new methods to improve upon previously developed software, be it in terms of speed or result acuity. For this thesis in particular, it was proposed to develop a program able to take an NRRD image format and distinguish the different materials contained within it. Thus, the goal was to be able to separate the different materials as well as identify the individual particles of the material being studied. This project expands upon previously developed work, by utilizing Julia, a new language, as well as the use of a GPU to accelerate the image processing algorithms. This newer language poses some advantages, such as a high processing speed and elevated comprehensibility, making it an interesting option to develop new work.

The program was developed as intended originally, and every algorithm presented was written in Julia and built through the use of a GPU. Research was continuously conducted to choose which algorithms to implement, as to maximize program speed and efficacy.

Testing was performed, on both real and artificial samples, of varying size and particle complexity. In all the tests performed, the program was able to separate and identify every single material particle correctly. Additionally, this was performed with acceptable processing time. With this, it is possible to conclude that the proposed goals for this dissertation were fulfilled in full.

In terms of future developments, it would be interesting to utilize distributed computational resources. The computational power is limited to the resources the machine it runs on can allocate. Considering this, an alternative to process extremely large tomographic images in a more efficient manner would be to slice the image into sections and send those sections into a cluster with several machines. These machines could process those sections simultaneously, reducing processing time considerable. After the sections were processed, they would be returned to the machine that made the processing request, to join them and compose the final processed image.

# Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf.

[2] F. Birra et al. "Tomographic image analysis of reinforcement distribution in composites using a flexible and material's specialist-friendly computational environment". English. In: *Materials Letters: X* 7 (2020-09). ISSN: 2590-1508. DOI: 10.1016/j.mlblux.2020.100046.

[3] S. Sarangi. *Basic Computer Architecture*. White Falcon Publishing, 2021. ISBN: 9781636403038.

[4] J. Ribeiro. *3D GPU-enabled Connected-Component Labeling Algorithms with Numba*. 2021.

[5] J. Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671. URL: https://epubs.siam.org/doi/10.1137/141000671.

[6] M. A. ALI. *CT scan generated material twins for Composites Manufacturing in industry 4.0*. SPRINGER, 2020. ISBN: 9811580200.

[7] G. Kindlmann, D. Weinstein, and M. Ikits. *NRRD*. URL: http://teem.sourceforge.net/nrrd/index.html.

[8] J. Ahrens, B. Geveci, and C. Law. *ParaView: An End-User Tool for Large Data Visualization*. Elsevier, 2005. ISBN: 978-0123875822.

[9] G. Stockman and L. G. Shapiro. *Computer Vision*. 1st. Prentice Hall PTR, 2001. ISBN: 0130307963.

[10] F. Rossi et al. "A 3D Voxel Neighborhood Classification Approach within a Multiparametric MRI Classifier for Prostate Cancer Detection". In: (2015). DOI: 10.1007/978-3-319-16483-0_24.

[11] R. J. McCool M. Arch M. *Structured Parallel Programming: Patterns for Efficient Computation*. First. Morgan Kaufmann, 2012. ISBN: 978-0-12-415993-8.

[12] T. G. Mattson, B. A. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley, 2005. ISBN: 0321228111.

[13] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*. 2008-05. URL: http://www.openmp.org/mp-documents/spec30.pdf.

[14] NVIDIA, P. Vingelmann, and F. H. Fitzek. *CUDA, release: 10.2.89*. 2020. URL: https://developer.nvidia.com/cuda-toolkit.

[15] J. Perkel. “Julia: come for the syntax, stay for the speed”. In: *Nature* 572 (2019-08), pp. 141–142. DOI: 10.1038/d41586-019-02310-3.

2022 TOMOGRAPHIC IMAGE PROCESSING USING JULIA AND GPU Bruno Brito

NOVA SCHOOL OF SCIENCE & TECHNOLOGY