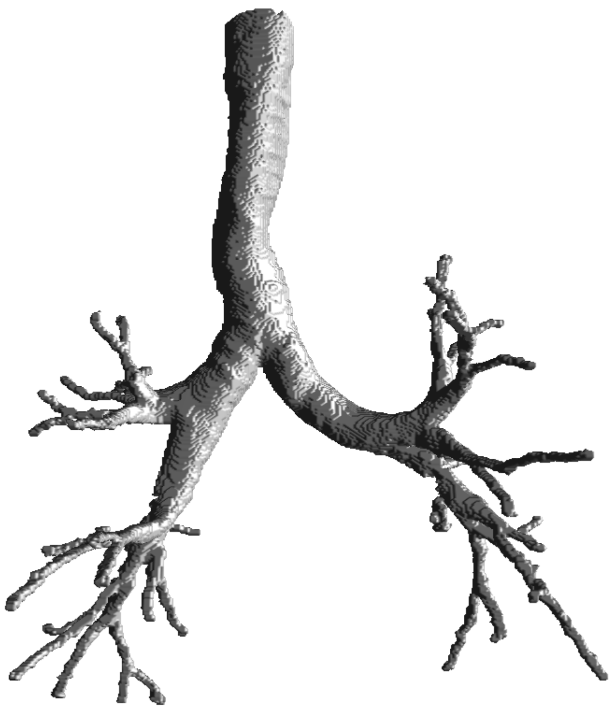
GPU-Based Airway Tree Segmentation and Centerline Extraction

Erik Smistad
Advisor: Frank Lindseth
Co-advisor: Anne C. Elster

April 23, 2012



Abstract

Lung cancer is one of the deadliest and most common types of cancer in Norway. Early and precise diagnosis is crucial for improving the survival rate. Diagnosis is often done by extracting a tissue sample in the lung through the mouth and throat. It is difficult to navigate to the tissue because of the complexity of the airways inside the lung and the reduced visibility. Our goal is to make a program that can automatically extract a map of the Airways directly from X-ray Computer Tomography(CT) images of the patient. This is a complex task and requires time consuming processing.

In this thesis we explore different methods for extracting the Airways from CT images. We also investigate parallel processing and the usage of modern graphic processing units for speeding up the computations. We rate several methods in terms of reported performance and the possibility of parallel processing. The best rated method is implemented in a parallel framework called Open Computing Language.

The results shows that our implementation is able to extract large parts of the Airway Tree, but struggles with the smaller airways and airways that deviate from a perfect circular cross-section. Our implementation is able to process a full CT scan using less than a minute with a modern graphic processing units. The implementation is very general and is able to extract other tubular structures as well. To show this we also run our implementation on a Magnetic Resonance Angio dataset for finding blood vessels in the brain and achieve good results.

We see a lot of potential in this method for extracting tubular structures. The method struggles the most with noise handling and tubes that deviate from a circular cross-sectional shape. We believe that this can be improved by using another method than ridge traversal for the centerline extraction step. Because this is a local greedy algorithm, it often terminates prematurely due to noise and other image artifacts.

Acknowledgments

The author wish to express gratitude for the guidance from the advisors of this project, Frank Lindseth and Anne C. Elster as well as to the lung surgeons at St. Olavs hospital for their input and acquisition of data. Great thanks also goes to the people of the High Performance Computing Lab (HPC-Lab) at the Norwegian University of Science and Technology (NTNU) for all their assistance and especially their help with debugging. The author would also like to convey thanks to the Department of Computer and Information Science (IDI), NVIDIA and AMD. Without their hardware contributions to the HPC-Lab, this project would not have been possible.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Lung cancer has one of the highest mortality rates in Norway and is also one of the most common types of cancer [10]. Early and precise diagnosis is crucial for improving the mortality rate. Currently, at St. Olavs Hospital in Trondheim, Norway, diagnosis is done by taking a tissue sample of the tumor using a bronchoscope. With a bronchoscope, tissue samples are extracted through the airways that conduct air in and out of the lungs. One of the major challenges with doing such a biopsy with a bronchoscope is to actually find the tumor inside the lung. The Airways of the lungs form a complex tree structure with many branches. The bronchoscope has a camera and light for visual guidance, but physicians still find it difficult to navigate using this camera. Also, the airways become very small in the periphery of the lungs. And in these small airways there is no visibility. This makes it very hard to find tumors that are located in the periphery of the lungs.

SINTEF Medical Technology, St. Olavs Hospital and the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway has a joint project on image guided bronchoscopy. Image Guided Surgery is the field of using images, such as those acquired through Computer Tomography scans, for planning and navigation before and during surgical procedures such as bronchoscopy. The goal of this joint project is to develop new methods by combining the clinical bronchoscopic expertise of the lung department at St. Olavs Hospital with SINTEF and NTNUs experience in Image Guided Surgery. This master thesis is part of this joint project. The goal of this master thesis is to develop a method for automatically creating maps of the airways from CT images. These maps are needed to plan a route before the procedure to the region of interest such as a tumor site and to navigate during surgery. The location of the bronchoscope can be set on the airway map by combining the airway map and electromagnetic tracking of the tip of the bronchoscope. Much like using a GPS in a car to find the path from A to B. Before navigation is possible, the map has to be registered to the patient. A set of coordinates that are known to be inside the airway can be established by moving the bronchoscope through the airways. These coordinates are then used to register the patient to the map.

The map of the Airway Tree will consist of a segmentation and a centerline. Many different methods for extracting the Airway Tree exists in the literature. Most of these methods are very computationally expensive and require a long time to finish processing. Also, many of the methods require several runs with different parameters before a satisfactory results is achieved. We wish to create an implementation of Airway Tree segmentation and centerline extraction that is fast so that the physicians does not have to wait for the result needed to do the planning and the guidance during bronchoscopy. We also believe that increased speed can be traded off to do additional processing that can increase the accuracy and quality of the segmentation. To increase the speed of our implementation we will to investigate parallel processing and especially the use of modern graphic processing units (GPUs). GPUs are ideal for data parallel computations where several elements in a collection of data need to be processed with the same instructions. Generally, image processing tasks are data parallel because they work on large datasets and often run the same instructions on each pixel/voxel. We will use Open Computing Language (OpenCL) for the implementation. OpenCL is a new and popular framework for parallel programming on different processors from different manufactures.

1.2 Project Goals

The purpose of this project is to explore Airway Tree segmentation and centerline extraction and implement a program that performs these tasks and utilizes the computational power of modern graphic processing units (GPUs) to speed up the calculations. Thus the main goals of this project are:

- Explore state-of-the-art methods for Airway Tree segmentation and centerline extraction

- Evaluate each method in terms of accuracy, performance and the potential for improving speed by utilizing a GPU
- Implement one of these methods using OpenCL and document the results

1.3 Outline

The following is an outline for the remainder of this thesis.

Chapter 2 - Background

In the next chapter a background study will be conducted. This study will include an introduction to the anatomy and terminology of the lungs, how images of the lungs are created, image guided bronchoscopy, parallel and GPU computing and a review of different Airway Tree Segmentation and Centerline extraction methods. The last section in this chapter contains some conclusions drawn from the background study which lay the foundations to the methods that will be used to create our implementation.

Chapter 3 - Methodology

The Methodology chapter discusses our implementation in further detail with pseudocode for each step. This chapter also contains details on how the implementation was adapted and optimized for parallel processing on the GPU.

Chapter 4 - Results

Chapter 4 presents images of the extracted centerlines and segmentation result of our implementation as well as runtime measurements on different datasets and processors.

Chapter 5 - Discussion

Chapter 5 discusses the quality of the extracted Airway Tree and the runtime performance of our implementation.

Chapter 6 - Conclusions

The last chapter contains conclusions on this project and suggestions for future work.

Chapter 2

Background

In this chapter a background study will be conducted. This study will include an introduction to the anatomy and terminology of the lungs, how images of the lungs are created, image guided bronchoscopy, parallel and GPU computing and a review of different Airway Tree Segmentation and Centerline extraction methods. The last section in this chapter contains some conclusions drawn from the background study which lay the foundations to the methods that will be used to create our implementation.

2.1 The Lungs and Airways

We start by giving a short introduction to the lung anatomy and terminology. Then we proceed to describing how images of the lungs and airways are created using Computer Tomography.

2.1.1 Anatomy and Terminology

The pulmonary airway tree, also called bronchial tree, is a large tree that distributes air flowing in and out from the nose and mouth down into the lungs and to the respiratory membranes where air and blood exchange gases like oxygen and carbon dioxide. From the nose and mouth the air is conducted through *trachea*, a wide tube that is reinforced with cartilage. *Trachea* goes down into the chest and divides into two smaller tubes called the primary*bronchi*. These two tubes proceeds in to the left and right lung respectively. Inside both lungs the primary *bronchi* divide again into the secondary *bronchi* and then again to the tertiary *bronchi*. The *bronchi* continues to branch out many times as shown in figure 2.1. The final destination is the *alveoli*, which are small air sacs. It is in these air sacs that the oxygen is diffused into the blood. Blood vessels run along these branches all the way to the *alveoli* where the blood receives the oxygen and then continues back to the heart.

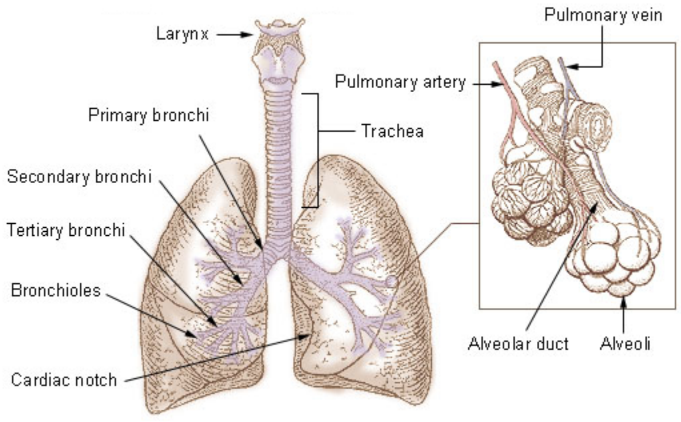


Figure 2.1: The lungs and the pulmonary airway tree. Image is public domain and taken from the U.S. National Cancer Institute (http://upload.wikimedia.org/wikipedia/commons/d/db/Illu_bronchi_lungs.jpg).

2.1.2 Lung Cancer

According to the Cancer Registry of Norway [10], cancer in the lung and trachea is one of the most common types of cancers in Norway. For men it is the second most common type of cancer and for women the third most common type with around 2500 new cases per year. This type of cancer is most common for people above 50 and has one of the highest mortality rates of up to 60-65% mortality 1 year after diagnosis.

2.1.3 Imaging of the Lungs

Chest X-ray and Computer Tomography (CT) are the primary tools for creating images of the lungs. Ultrasound is not used, because the ribs make it difficult to send and receive ultrasound waves. Also, the air inside the lung reflect large amounts of the waves thus making it very difficult to see far inside the lungs. Magnetic Resonance Imaging (MRI) is also not used to image the lungs because MRI is very dependent on water in the tissue and the lung consist mostly of air and not water as the rest of the body. Nevertheless, Lewis et al. [20] has shown that it is possible to image the airways of the lung using MRI and a hyperpolarized helium gas which the patient inhales.

X-ray is one of the oldest image modalities used in the clinic. X-rays are created by bombarding a plate of metal with electrons. The electrons will interact with the atoms in the metal plate and photons with a very high frequency are emitted. These photons are directed toward the body. Some of the photons will pass through the entire body while others will be absorbed by the tissue in the body. The photons that pass through the body hits a plate. Locations where photons hit this plate will become darker, and the more photons hitting at a specific location the darker the plate becomes at that location. This plate becomes the X-ray image such as the one in figure 2.2.

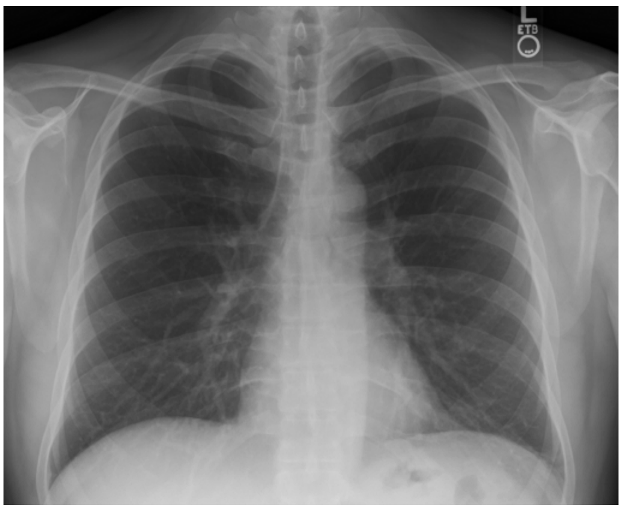


Figure 2.2: Chest X-ray. Public domain image from (http://en.wikipedia.org/wiki/File:Chest_Xray_PA_3-8-2010.png)

The amount of photons that are absorbed depends on the density of the tissue. Bone and teeth absorbs the most, and regions where bones are present becomes white in the final image. X-rays creates shadows or silhouettes of parts inside the body. The resulting brightness is the sum of absorption through the body at a specific direction. X-rays are most often used to display teeth and bone fractures. This is because X-rays don't provide much contrast for soft-tissue, but instead provides a large contrast between soft-tissue and hard-tissue like bone.

Computer Tomography (CT) is a technique which gives the ability to create images that display more than just shadows and thus better at creating contrasts between different soft-tissue. CT can image the absorption of each location in a 2D slice of the body and not just the sum of all absorptions at a specific direction through the body. CT achieves this by sending several X-ray beams through the body at different angles. By interpolating all these signals in the frequency domain, a 2D frequency image can be created. The actual image can then be retrieved by applying the inverse fourier transform on the frequency image.

The output of the CT scanner is a set of 2D slices forming a 3D volume. Each element in a 3D volume is called a voxel. Each voxel in a CT volume has a number indicating the amount of X-ray absorption at that specific location in the body. This number is usually stored in the form of Hounsefield Units (HU). The HU scale is a linear transformation of the original absorption value α and is defined so that distilled water at standard pressure and temperature has 0 HU. The equation below is the transformation formula for the HU scale and table 2.1 shows a list of different substances in the body and their HU value.

$$\text{HU} = 1000\frac{\alpha - \alpha_{\text{water}}}{\alpha_{\text{water}}} \tag{2.1}$$

Substance	Air	Lung	Soft Tissue	Water	Blood	Muscle	Bone
HU	-1000	-700	-300 to -100	0	30 to 45	40	700 to 3000

Table 2.1: Table of different substances and their HU value

The main disadvantage of using X-rays and CT to create images of patients is that the high frequency waves are dangerous to the patient. The X-rays are categorized as ionizing radiation, meaning that it can knock electrons out of molecules making them ions. This injures the tissue. This type of radiation can lead to cancer. The advantages of X-ray and CT imaging is that it is a fast imaging technique that provides good resolution and intensity values that can be mapped to physical tissue types.

Imaging The Airways

Because the airways contains air which absorbs very little x-ray radiation it will appear as black on CT images. The airways will thus appear as black tubes surrounded by white blood vessels and cartilage. Figures 2.3 and 2.4 shows two CT images of the lung from different orientations. Figure 2.3

clearly shows the main *bronchi* as a black tube that divides in two and goes to the left and right. Figure 2.4 shows some smaller airways in the form of black circles surrounded by a grey border.

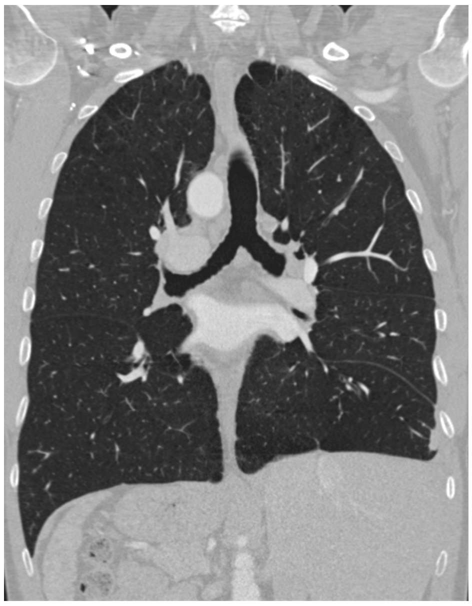


Figure 2.3: Parts of many CT slices put together to create an image from another orientation than the slice angle.

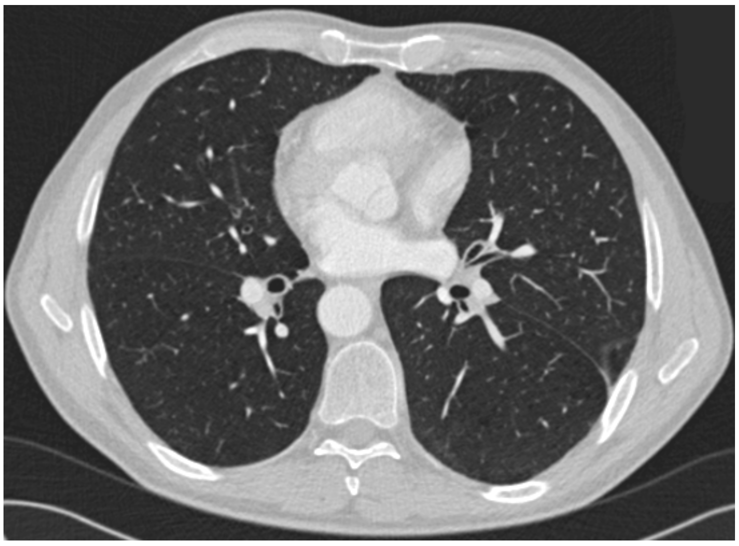


Figure 2.4: One CT slice image of the lungs. Note the small airways depicted as black circular objects with a grey border.

2.2 Image Guided Bronchoscopy

Bronchoscopy is a minimal-invasive diagnostic and surgical procedure. It allows the physician to reach into the airways through the mouth or nose as depicted in figure 2.5. A bronchoscope is a flexible tube with fiberoptic cables that transmit light both ways so that the physician can see inside the airways through the bronchoscope. The bronchoscope also contains a small shaft that can be used for inserting instruments and extracting tissue samples or remove tumors and other unwanted tissue/objects.

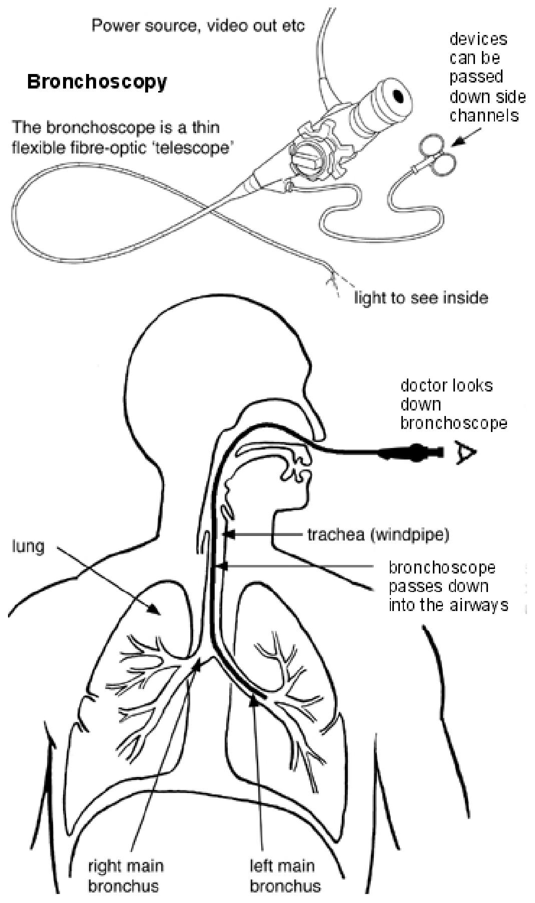

Power source, video out etc
Bronchoscopy
The bronchoscope is a thin flexible fibre-optic 'telescope'
devices can be passed down side channels
light to see inside
doctor looks down bronchoscope
lung
trachea (windpipe)
bronchoscope passes down into the airways
right main bronchus
left main bronchus

Figure 2.5: Figure of the bronchoscope and how it used to reach the airways. Image taken with permission from www.patient.co.uk/health/Bronchoscopy.htm

SINTEF Medical Technology, St. Olavs Hospital and the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway has a joint project on image guided bronchoscopy. The goal is to develop new methods by combining the clinical bronchoscopic expertise of the lung department at St. Olavs Hospital with SINTEF and NTNUs experience in Image Guided Surgery.

This master thesis is part of this joint project. The goal of this master thesis is to develop a method for automatically creating maps of the airways from CT images. These maps are needed to plan a route before the procedure to the region of interest such as a tumor site and to navigate during surgery. To enable navigation during surgery, electromagnetic tracking of the bronchoscope and registration of the map to the patient is needed. A set of coordinates that are known to be inside the airway can be established by moving the bronchoscope through the airways. These coordinates are then used to register the patient to the map for instance by using the Iterative Closest Point algorithm.

Due to the complex branching structure of the airway tree it is hard to find the correct path to the site of interest inside the lung. And the fact that the branches becomes very narrow after a few branches makes it impossible to use the camera on the bronchoscope in the small airways. By using image guided bronchoscopy we hope to improve the success of navigation during bronchoscopic procedures.

Image Guided Surgery is the field of using images, such as the ones acquired through Computer Tomography scans, for planning and navigation before and during surgical procedures. Image Guided Surgery enables minimal-invasive surgery and the overall goal is to improve patient treatment by:

- Reducing the risk of complications
- Avoiding large surgical scars
- Removing more of tumor/unwanted tissue
- Shortening recovery time

2.3 Parallel and GPU Computing

Many image processing operations, such as segmentation, are very computationally demanding. This is mainly due to the large size of the datasets. A volume of size 512x512x512 will have more than 132 million individual elements, called voxels. If the image processing operation has to run a complex routine on each of these voxels it will be very time consuming. But image processing often performs the same set of operations on each pixel. When this is the case, parallel processing can be used to process each pixel in parallel. In this section we will go through the basic concepts of parallel execution, different parallel architectures and programming models that enables parallel execution.

2.3.1 Parallel Execution

Originally, programs were executed in a serial manner. Each instruction was executed in order, one after one. Speedup of these programs was achieved by running on a CPU with a higher clock speed. Around 2004, when Intel's clock speeds reached about 4 GHz, the power consumption and heat dissipation made a physical limit to how much the clock speed could be increased with conventional cooling technology. This power wall turned the focus towards parallel computing for speedup. Parallel computing entails dividing a task into smaller subtask that can be executed in parallel. Several different forms of parallelism exists, but they are usually grouped into two main categories: Task and data parallelism.

Task parallelism

Task parallelism is based on the idea of dividing your program into separate tasks and then running them at the same time on different processing elements. According to Flynn's taxonomy task parallelism is the equivalent of multiple instruction, multiple data (MIMD) computation. It is usually necessary for the tasks to communicate and synchronize with each other, and there exists two main schemes of doing this: through message passing and through a shared region in memory.

The message passing model is often used on distributed memory machines such as large supercomputers where there typically is several thousand nodes / computers each with their own processors and memory connected through some high speed link. As the name indicates, communication and synchronization amongst processing elements is done by sending and receiving messages. A popular Application Programming Interface (API) that use this message passing scheme for parallel computing is the Message Passing Interface (MPI). McCool et al. [24] concluded that this scheme scales well to large parallel computers such as supercomputers, but is generally harder to program.

With the shared memory scheme, the different processing elements communicate trough a region in memory that all the processing elements have access to. Locks and barriers are two very important mechanisms for the shared memory scheme. Locks in the memory system enforce that only one task at a time can modify an element in memory. Barriers block tasks until all tasks have reached a certain point. OpenMP is one popular API that uses the shared memory scheme together with threads for parallel computing. McCool et al. [24] argued that the shared memory scheme is simpler to program and reason with than the message passing scheme, but is more difficult to implement in hardware, especially with a growing number of different processing elements.

Data parallelism

Data parallelism is a less general form of parallelism than task parallelism. With data parallelism the same task is performed on different parts of the data in parallel. For instance if you want a program that multiplies each element in a list with 2. Each element in the list can be multiplied with 2 in parallel. This type of parallelism is often present in image processing tasks where each pixel or voxel is processed using the same instructions.

Single instruction, multiple data, or SIMD, is a term from Flynn's taxonomy that is often referred to with data parallelism. SIMD is the simplest form of data parallelism where a single instruction is performed on each element in a collection of data. Stream processing takes this principle further and instead of performing just one instruction, performs several instructions in a SIMD manner. This gives the possibility to increase the arithmetic intensity compared to vector processing where data is read and stored per instruction. Stream processing is a more efficient data parallel scheme because memory operations generally takes up more time than arithmetic. The set of instruction that is applied to each element in the stream processing model is often referred to in literature as a *kernel*.

Sometimes it is not necessary, or wanted, to perform all the instructions on each element in the stream processing model. With serial processing this is done by the use of branching, typically in the form of if-else sentences. With the SIMD stream processing model all the instructions in the kernel has to be performed on all the elements which makes branching difficult. To this purpose an extended model exists called SPMD: single program, multiple data. This model is similar to the SIMD processing model, but also allows branching.

2.3.2 Parallel architectures

Parallel execution of code requires hardware architectures that enables such execution. Brodtkorb et al. [9] identified 4 layers of parallelism exposed in modern hardware:

- **Multi-chip parallelism:** several physical processor chips
- **Multi-core parallelism:** similar to multi-chip, but the cores are all inside the same chip.

- **Multi-thread parallelism:** several execution threads, or contexts, that can be switched amongst with very little overhead. This enables the processor to reduce idle time. For example if one thread need to perform some I/O operation that takes time, the processor can switch to another thread that then can perform some arithmetic while the other thread is waiting.

- **Instruction parallelism:** a processor that can execute more than one instruction per cycle using multiple instruction units.

Today, processors use a combination of these types of architectural parallelism. Different types of processors give different weight to different types of architectural parallelism depending on the tasks the processor is meant to perform. In the following sections the 4 main processor architectures, CPUs, GPUs, FPGAs and Cell BE will be discussed with emphasis on how they embrace parallel computations.

Central Processing Unit (CPU)

The CPU is the traditional processor found in computers. It is the main processing unit of computers. Originally, programs were executed on the CPU in a serial manner. Each instruction was executed in order, one after one. An internal clock in the CPU is used to control the rate in which instructions are executed and synchronize the various components in the computer. CPUs were made faster by increasing the clock frequency and the amount of transistors. To increase the clock frequency the input voltage has to be increased. Increased voltage allows transistors to charge and discharge more quickly thus allowing a higher clock frequency. The problem with increasing the voltage is that processor's power consumption also increases. Around 2004, when Intel's clock speeds reached about 4 GHz the power consumption and heat dissipation made a physical limit to how much the clock speed could be increased with conventional cooling technology.

This power wall turned the focus towards increasing processing speeds by using parallelism in a much larger extent than before. Early approaches to parallelism had been using Multi-thread and Instruction parallelism. Also, large supercomputers embraced parallel execution by connecting several computers together creating multi-chip parallelism. But this was not good enough and thus the power wall introduced multi-core processors which is the concept of replicating the processor and putting several of them on the same chip. This made the processor able to processes several completely different programs simultaneously on the same chip. At the time of writing, CPUs typically have two, four or six cores.

Most CPUs thus embrace parallelism with the focus on being a fast general purpose MIMD processor making it most suitable for task parallelism.

Graphic Processing Unit (GPU)

The GPU is a specialized processor compared to the more general purpose CPU. Originally made to speed up the memory-intensive calculations needed in demanding 3D computer games, these devices are now increasingly being used to accelerate numerical computations in science. The calculations it was intended for was texture mapping, rendering polygons and transformation of coordinates. All of these types of calculations were very memory intensive, but the calculations to be performed was the same for the entire collection of data. Hence the GPU is a type of SIMD processor, it can perform the same instruction on each element in a collection of data in parallel. The GPUs achieve this by having several hundred functional units. These are usually not called "cores" in the same sense as the multi-core CPUs. McCool et al. [24] defined a core as a processing element with an independent flow of control. All of the functional units on a GPU does not have an independent flow of control as they are grouped together in a SIMD manner, meaning that the functional units in one group has to perform the exact same instruction in a clock cycle. These SIMD groups can thus be referred to as cores with the above definition.

Most GPUs today are SPMD processors, meaning that they also allow branching. The code flow is convergent if all execution threads in one SIMD group follow the same path in a set of branches. When the code flow is convergent, no special treatment is needed and only the code needed is run in a SIMD manner. On the other hand, if the code flow is divergent in a SIMD group the GPU will run the instructions from all the code paths that were used for all of the execution threads. The result is that no time is saved and instead more time is used. To ensure the correct answer produced by each processing element the GPU will use masking techniques.

Modern GPUs are usually located on a graphics/video card connected to the motherboard through a AGP or PCI-express slot. Some motherboards also come with integrated GPUs, but these are not nearly as fast as dedicated graphic cards. At the time of writing this report, both Intel and ARM, two large processor manufactures, have created hybrid processors with both a

CPU core and a GPU core on the same chip.

Field-Programmable Gate Array (FPGA)

An FPGA is a set of configurable logic blocks, signal processing blocks and optional CPU cores that are all connected through a configurable interconnect. By configuring these blocks and the interconnect, the FPGA can be tailored to do a certain task. Brodtkorb et al. [9] argues that the FPGA can be much harder to set up and program than other devices, but as it is tailored for a specific task, it can be made very power efficient and fast. Also they are generally more expensive.

Cell Broadband Engine

The Cell Broadband Engine was created by IBM and can be found in the Playstation 3 gaming console. Cell BE uses a heterogeneous architecture consisting of one traditional CPU core and 8 specialized accelerator cores. All of them are connect through a high speed ring bus called the Element Interconnect Bus. This special architecture makes the Cell BE excellent at task parallel problems where the tasks need to communicate.

2.3.3 Parallel programming

In order for programs to exploit parallel execution, the processor(s) has to know what it can run in parallel. For a long time compiler and processor designers have struggled to automatically extract Instruction Level Parallelism (ILP) from serial code. But as noted by McCool et al. [24] automatic extraction without help from the programmer has provided limited results. To properly run parallel code on parallel architectures, explicit parallel languages and APIs are needed.

This section goes through some of the most important languages and APIs for parallel programming. Note that there exists many parallel languages and APIs, this is only a small subset.

POSIX Threads, or Pthreads, is a simple API used for spawning and managing software threads run on the CPU. All synchronization and data handling is left to the programmer. OpenMP is a more extensive API with a set of directives and library routines that allows the programmer to easily express parallelism in serial code on shared memory system. Message Passing Interface (MPI) is one of the main APIs for writing parallel code on distributed memory machines, such as supercomputers. Todays most popular languages like C and Fortran were not made with parallel computing in mind and so many believe that new languages, and not just APIs, are needed to allow the programmer to write efficient parallel code. Many new languages, such as Chapel, are being developed with this goal.

Programming the GPU has originally been done with shader programming, which refers to programming certain parts of the GPU's rendering pipeline. These shader languages includes OpenGL Shading Language (GLSL), High Level Shader Language (HLSL) and C for Graphics (Cg). The problem of doing general-purpose computations on GPUs using these shading languages is that the problem has to be transformed into a graphics rendering problem. CUDA is a newer language / API that can be used to program NVIDIA's GPUs without transforming the problem to a rendering problem. Performing such general-purpose calculations on the GPU has been termed GPGPU. Inspired by CUDA, a new language / API called Open Computing Language (OpenCL) was developed that could be used to program GPUs, CPUs and other types of processors from different vendors.

The Cell BE engine has traditionally been programmed using the IBM Cell SDK API, though it has also been shown by Breitbart et al. [8] that OpenCL can be used to program the Cell BE. FPGAs are usually programmed using circuit design languages such as VHDL and Verilog. The problem with using such languages is that it requires the knowledge of a logic designer to program.

Because of OpenCLs possibilities to program all these different parallel architectures we decided to use this framework in our implementation and in the next section we will discuss OpenCL in further detail.

2.3.4 Open Computing Language - OpenCL

Brodtkorb et al. [9] defined node-level heterogeneous computing as *"the use of different processing cores to maximize performance"*. In the previous section we saw that some types of computations can be more efficiently executed on one type of processing core and less efficient on another. For instance a large data parallel task can run much more faster and efficiently on a GPU than on a CPU. Heterogeneous computing implies using the devices which are best suited for the computation at hand. This can maximize performance and minimize power usage, but it creates new challenges in terms of programming. OpenCL (Open Computing Language) is a new framework for writing programs that can execute on heterogeneous platforms. With this framework the programmer is able to easily write code that can execute on different types of devices, as well as synchronize and share data between them.

The OpenCL framework standard is ratified by the Khronos group, an industry consortium. But it is up to the vendors themselves to make compilers and drivers for OpenCL so that the code can run as specified in the standard on the vendors hardware. For a more detailed description of the OpenCL framework the reader is referred to the OpenCL website [16] and the book by Tsuchiyama et al. [28].

OpenCL consists of two separate parts:

- **OpenCL C Language** - Extended version of C used to write kernels which will execute on the compute devices
- **OpenCL Runtime API** - An API used to control and synchronize the different devices available on the machine

Platform model

OpenCL's platform model consists of one host and several devices. The host is usually a thread running on the CPU. The devices act as accelerators, or co-processors, to the host. The devices are subdivided, as figure 2.6 depicts, into multiple compute units which are subdivided again into processing elements. Which devices are available on the system can be queried through the OpenCL API as well as their properties, such as number of compute units, max clock frequency etc. How the concepts of compute units and processing elements map to the physical device differs for different types of devices.

Execution model

The host creates, using the OpenCL API, a context and a set of command queues for each device. It also compiles kernels, written in the OpenCL C language, for each specific device. The kernels are the programs that will run on the devices. The command queues can be filled with instructions by the host. These instructions include transfer of data, synchronization and execution of different kernels.

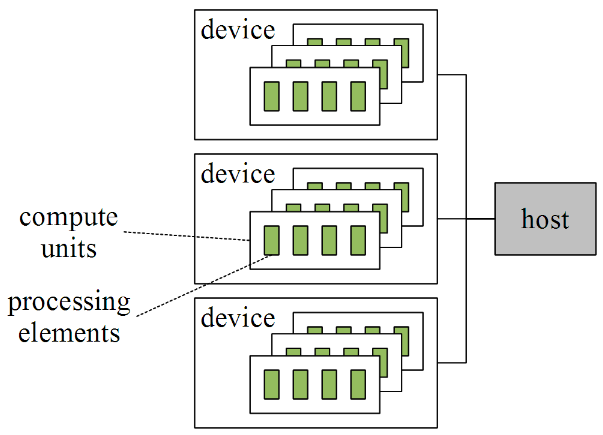


Figure 2.6: OpenCL's platform model

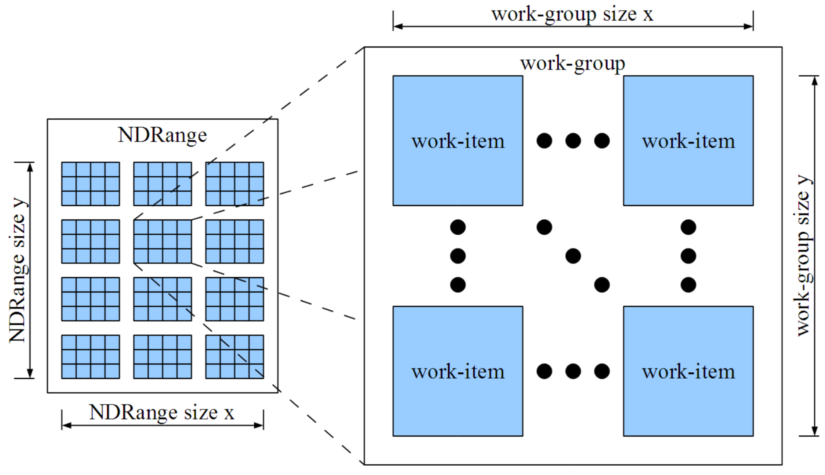


Figure 2.7: OpenCL's execution model

OpenCL supports both task and data parallel execution. The data parallel model used is the SPMD, single-program multiple-data, model. This allows control flow in the kernels.

As figure 2.7 depicts, OpenCL divides the data-parallel tasks into a NDRange hierarchy. An NDRange can be in 1,2 or 3 dimensions and sizes in all dimensions can be set. The NDRange is further divided into work-groups of either 1,2 or 3 dimensions. The works-groups are further divided into work-items. The work-items are run on the processing elements and run the actual contents of the kernels. Each kernel gets a global_id, group_id and local_id which is an N-vector, where N is the number of dimensions. All of the work-items in a work-group are guaranteed to run on the same compute unit.

Memory model

The kernels written in the OpenCL C language has access to 4 different levels of memory. These levels differ in speed, size, physical location and how they are shared. All of the four memory levels reside on the device itself. Figure 2.8 depicts this memory model and how the different levels relate to the different parts of the device. The largest and slowest memory is the global memory. It can be both read from and written to and is the device main memory, usually located off-chip. The constant memory is the same as the global memory, but is read-only and smaller. This memory can therefor be cached for faster access. The local memory is local to a compute unit and often found on-chip and therefor quite fast. The private memory is local to the individual processing elements and is the fastest, but very limited in size.

- **Global** memory is the device main memory. It can be accessed by all work-items.
- **Constant** memory is a read-only memory that can be accessed by all work-items.
- **Local** memory is a read/write memory local to a work-group and is shared by all the work-items in that work-group.
- **Private** memory is a read/write memory local to each processing element.

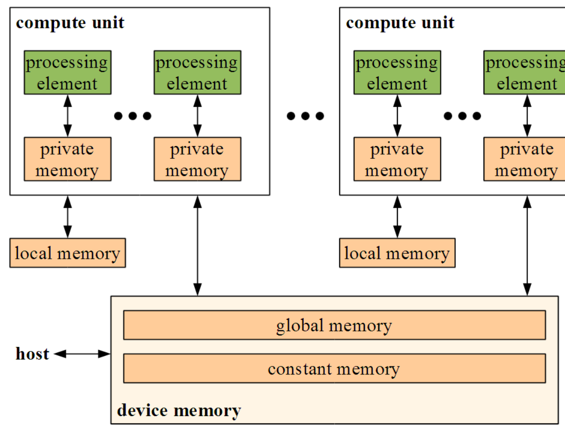


Figure 2.8: OpenCL's memory model

2.4 Airway Tree Segmentation and Centerline Extraction

Our map of the Airway Tree will consist of two things: A segmentation and a centerline. The segmentation is a labeling of each voxel in the CT scan that determines whether the voxel is part of an airway or not. The centerline is a line that runs through the center of each airway. The centerline can be both represented as a graph, with nodes and edges, or as a set of voxels. Thus the centerline represents the Airway Tree on a structural level while the segmentation describes the size and area of the airways.

Note that centerlines can always be extracted from a valid segmentation result with methods called skeletonization. A segmentation result may not necessarily be extracted from a centerline. Still, several methods start by finding a centerline and then the segmentation from the centerline.

The airways are part of a more general structure which is a network of tubular structures. Tubular structures are elongated structures and has often a near circular cross-sectional profile. Blood vessels and neural pathways are two examples of other tubular networks. Methods for creating Airway Tree segmentations and centerlines can often be used on other tubular structures as well and visa versa. In this thesis we will use the terms airways and tubular structures interchangeably.

Two notable reviews on Airway Tree segmentation and centerline extraction from CT images can be found in the works by Sluimer et al. [26] and a newer one by Lo et al. [21].

A larger and more general review on vessel segmentation was done in Lesage et al. [19]. In the next section of this thesis we provide a brief review of tubular structure segmentation and centerline extraction methods based on the categorization of Lesage et al. [19].

2.4.1 Tubular Models, Features and Extraction Schemes

Many different image processing methods have been used for segmentation of tubular structures. Lesage et al. [19] provided a categorization of several of these different methods into three schemes: Models, Features and Extraction. Usually methods from all three schemes are needed. In this section we will go through each scheme and look at some of the most common methods in that scheme.

Tubular Models

Models are used to represent *a priori* knowledge about tubular structures. These models can use image and/or geometric knowledge. Image knowledge can for instance be information about the image intensity values of a tubular structure. *E.g.* we know that airways in CT images appear as dark tubes surrounded by a white wall. Geometric knowledge is information about the shape of the tubes. *E.g.* size, radius, shape of cross-section, branching etc. For instance it is very common to assume that the cross-section of the tubes are circular.

Tubular Features

Features are used to detect possible tubular structures in an image or volume. Image knowledge from the tubular model is often used for the tubular features. Image knowledge used for tubular features are often based on intensity and first- and second-order derivate information. The most plausible detected tubular structures can be used as input to an extraction method such as region growing. For instance van Ginneken et al. [30] detected trachea by looking for large dark circular regions in each slice. The center of this circular region was then used as the seed for a region growing procedure. Aykac et al. [1] used a similar approach with morphological operations to detect the smaller airways in each 2D slice.

A filter runs a set of operations on each voxel and its neighborhood. Filters for detecting tubular structures are often called Tubular Detection Filters (TDFs). One of the most common TDF methods use second-order derivative information in the form of the eigenanalysis of the Hessian matrix. Frangi et al. [11] created a very popular vessel-enhancement filter based on the eigenanalysis of this matrix. Bauer et al. [5] showed how a centerline could be extracted directly from the output of the TDF of Frangi et al. [11]. Another popular TDF based on the eigenanalysis of the Hessian matrix is the Circle Fitting TDF by Krissian et al. [17]. This method was applied on airways in two different works by Bauer et al. [6] and [7].

Tubular Extraction Methods

The extraction methods does the actual segmentation and finds the centerlines. They are based on the assumptions in the tubular model and is guided by the tubular features. The most common method to extract tubular structures is region growing.

Region growing starts with a set of seed voxels. From these seed voxels the regions will expand to the neighboring voxels if they satisfy some predefined properties compared to the seed, for instance specific ranges of intensity or color. The region will continue to expand to the neighbors of the seed's neighboring voxels if they also satisfy the predefined property of the regions. This continues until no more new voxels can be added to the regions. For Airway segmentation the regions can be allowed to continue to grow as long as the intensity remains low. The region growing will then stop at the airway wall which is white. This works fine for the main *bronchi* and *trachea* where the airway is surrounded by a solid white wall. But for the smaller airways the contrast between the wall and lumen can become very small and because of partial volume effects the wall itself can disappear. Using region growing on these smaller airways will lead to what is called segmentation leakage. The leakage occurs because region growing will grow outside of the airway and fill the entire lung. This problem can be addressed by adding additional constraints to the growing procedure or by detecting leakages and stopping them when they occur as introduced by Mori *et al.* [25]. Another approach is to do a conservative region growing on the *trachea* and main *bronchi* and then use another method for the smaller airways such as done by Graham *et al.* [13] and many others. Region growing provides a segmentation and a centerline can be extracted from the segmentation by means of skeletonization methods.

Another method of extracting tubular structures is to directly extract their centerlines and then finding the segmentation from the extracted centerlines for instance by region growing. Direct centerline extraction is usually done by some sort of ridge traversal. Aylward et al. [2] provides a review of different centerline extraction methods and propose an improved method for traversing ridges in the TDF result. Bauer et al. [5] presented a similar ridge traversal method that used Gradient Vector Flow (GVF). Instead of traversing the ridges created by the TDF, Bauer et al. [5] proposed to follow the valleys created by the GVF method. This is based on the fact that the magnitude of the vector field from GVF will decrease towards the center of any closed object as noted earlier by Hassouna et al. [14]. Similar to region growing these ridge/valley traversal methods need seed voxels and thus are all highly dependent on identifying these properly.

Disadvantages of both ridge traversal and region growing is that they are very sensitive to noise and initialization. Ridge traversal is quite fast, while region growing can be slow, depending on the complexity of the growing constraints. Also, both methods have very little parallelity.

2.4.2 Evaluation of different methods

Evaluation of segmentation results of tubular structures is considered to be very hard and time consuming. Automatic methods need a set of pre-segmented datasets which acts as ground truth segmentations. This has to made by experts and this work is very time consuming and thus expensive. Also, because of the complex structure it is hard to calculate metrics that are fair and accurate. Due to the difficulty of automatic evaluation, semi-automatic methods are often used. The evaluation of Lo et al. [21] was part of the Extraction of airways from CT (EXACT) 2009 competition at the Second International Workshop on Pulmonary Analysis. They used a set of 20 chest CT datasets that were segmented manually using about 750 hours. An additional 20 chest CT datasets were used as a training dataset. The main conclusion from the study was that no method was able to extract more than 77% on average of the entire manually segmented reference tree. Thus the problem of Airway Tree segmentation is far from solved. Note that in this comparison study, only the segmentation results were compared and not the centerlines.

In this thesis we selected the Airway Tree method to implement based on this evaluation study. Figure 2.9 is taken from the evaluation paper by Lo et al. [21] and shows each method tree length vs. false positives. The goal is to have has high tree length as possible with few false positives. There is of course a trade-off in these two measures as can be seen on the method that has the largest tree detected (method 4), but also the largest amount of false positives. Based on our project goals, the criteria for select the Airway Tree Segmentation and Centerline extraction method was the following:

- How fast is the method and how much can be gained by running the method in parallel on a GPU (level of data parallelism)
- Large tree length detected
- Small false positive rate
- Method should be fully automatic

Note that having an exact segmentation for all airways detected is not that important for our project. This is because the physicians know, with their expertise in anatomy, approximately how large each airway segment is. The ultimate goal is to detect as many correct airway segments as possible so that the best route from *trachea* to the target site can be established. Thus, some false positives that are the result of inaccurate tube borders can be tolerated, while entire fake airway sections can not.

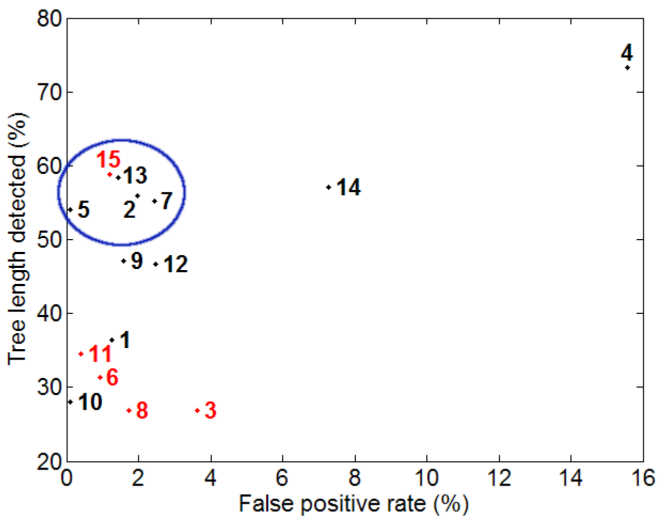


Figure 2.9: Figure from Lo et al. [21] that graphs tree length vs. false positives for each method evaluated in their study. The red methods requires user interaction while the black methods are completely automatic. We investigated each method inside the circle except the semi-automatic one (15).

From figure 2.9 we selected the top-left cluster (the best performing methods) and did an evualation of the potential of each method. This evaluation is summed up in table 2.2. From this short evaluation we see that methods that use the Hessian-based eigenanalysis as tubular features perform quite well (5, 7 and 13). And from those methods that perform well, the ones that don't use seeded region growing, but instead a ridge traversal on a TDF result has the most parallelism (methods 7 and 13). Based on these observations and the fact that method nr. 7 only provides an approximate segmentation we decided to use method nr. 13 by Bauer et al. [6]. In the next section of this chapter we explain Hessian-based Tubular Detection Filters in more detail. We also go trough the methods that Bauer et al. [6] use to extract centerlines and segmentation from the Tubular Detection Filter result.

Nr.	Ref.	Description	Parallelism	Comments
2	[15]	Region growing + Morphological operators for detecting smaller airways	Medium parallelism on morphological operators and low for region growing part	Centerline has to be extracted in addition to segmentation
5	[22]	Region growing + Hessian-based region growing for smaller airways	Low parallelism due to extensive use of seeded region growing	Seed has to be selected manually + Needs training which is very slow + Centerline has to be extracted in addition
13	[6]	Circle Fitting method (Hessian-based TDF) followed by a ridge traversal for centerline and then a growing from this	Very high parallelism, both TDF and GVF is embarissingly parallel, but centerline extraction is serial	Provides centerline and segmentation. Centerline extractions is serial, but fast.
7	[7]	Simular to 13. Circle Fitting method(Hessian-based TDF) + ridge traversal	Very high parallelism for everything except centerline extraction	Segmentation is approximated as a set of circles

Table 2.2: Summary of the potential of the four best methods from the evaluation study by Lo et al. [21].

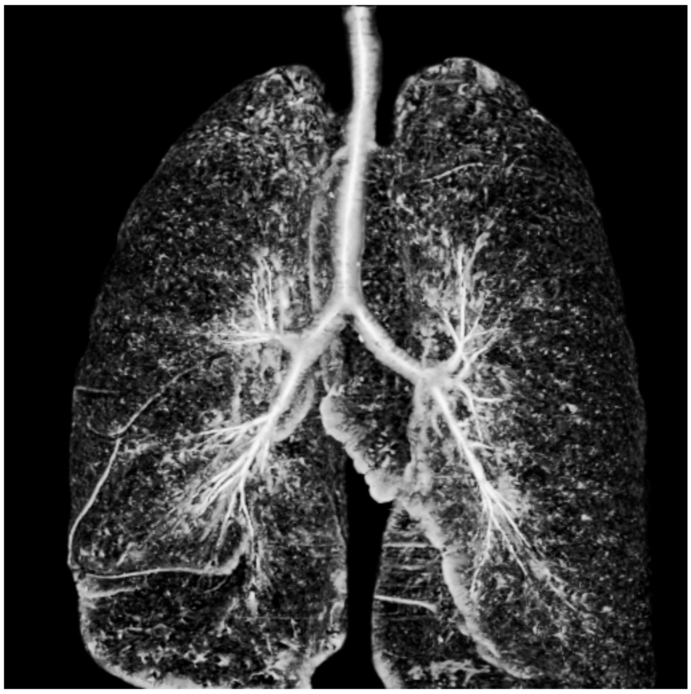


Figure 2.10: Maximum Intensity Projection of the TDF result on CT images of the lungs. The brighter the voxels, the more likely it is that a tube exist in that voxel.

2.4.3 Hessian-based Tubular Detection Filters (TDFs)

Tube Detection Filters (TDFs) are used to detect tubular structures, such as airways, in 3D images. TDFs perform a shape analysis on each voxel and return a value indicating the probability of the voxel belonging to a tubular structure. The output of a TDF is often referred to as its response. Figure 2.10 shows an example of a TDF response on a CT image of the lungs displayed using maximum intensity projection (MIP).

The eigenanalysis of the Hessian matrix is one of the most common methods to determine whether a voxel is part of a tube. The Hessian matrix is a set of variables that represents the second-order derivative information at a specific voxel position $\vec{v}$. First-order derivative image information in each direction is often called the gradient. These image gradients, denoted $\nabla I(\vec{v}) = (\frac{\partial I(\vec{v})}{\partial x}, \frac{\partial I(\vec{v})}{\partial y}, \frac{\partial I(\vec{v})}{\partial z})$, are vectors that says something about the change in intensity values in all three directions at position $\vec{v}$. The magnitude, or length, of the gradient $|\nabla I(\vec{v})|$ describes how strong the intensity change is at a specific voxel and the gradient vector points in the direction of the strongest

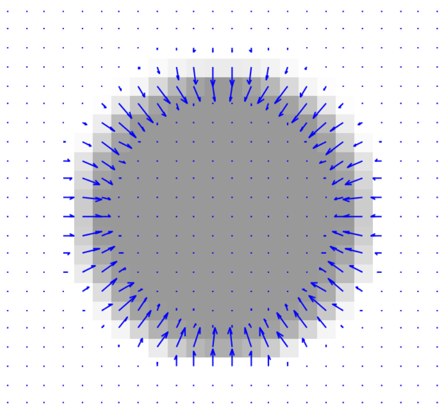


Figure 2.11: Image gradients of 2D slice of a tube

intensity change. Figure 2.11 shows the gradient for each pixel superimposed on an image of a tube's cross-section.

Second-order derivative information can be extracted from the first-order derivate information by calculating the gradients of each component in the image gradient. For instance the second-order derivative information in the x direction is $\nabla(\nabla I(\vec{v})_x)$. The Hessian $\mathbf{H}(\vec{v})$ is a matrix of these three gradients as shown in Eq. 2.2.

$$\mathbf{H}(\vec{v}) = \begin{bmatrix} \nabla(\nabla I(\vec{v})_x) \\ \nabla(\nabla I(\vec{v})_y) \\ \nabla(\nabla I(\vec{v})_z) \end{bmatrix} = \begin{bmatrix} \frac{\partial^2 I(\vec{v})}{\partial xx} & \frac{\partial^2 I(\vec{v})}{\partial xy} & \frac{\partial^2 I(\vec{v})}{\partial xz} \\ \frac{\partial^2 I(\vec{v})}{\partial xy} & \frac{\partial^2 I(\vec{v})}{\partial yy} & \frac{\partial^2 I(\vec{v})}{\partial yz} \\ \frac{\partial^2 I(\vec{v})}{\partial xz} & \frac{\partial^2 I(\vec{v})}{\partial yz} & \frac{\partial^2 I(\vec{v})}{\partial zz} \end{bmatrix} \tag{2.2}$$

The second-order derivative information describes how the first-order derivatives change in the image. Thus it describes the change of the change of intensity values in all three directions.

For an ideal small cylindrical tube this information is as shown in figure 2.12. Graphs of the intensity, first and second-order derivative for two lines through this tube are depicted in figures 2.13 and 2.14. One line goes through the cross-sectional plane (green) and the other one goes inside the middle of the tube (red).

From these figures, we can conclude with the following four observation for tubular structures:

1. The smallest intensity change is in the direction of the tube (see figure 2.14)

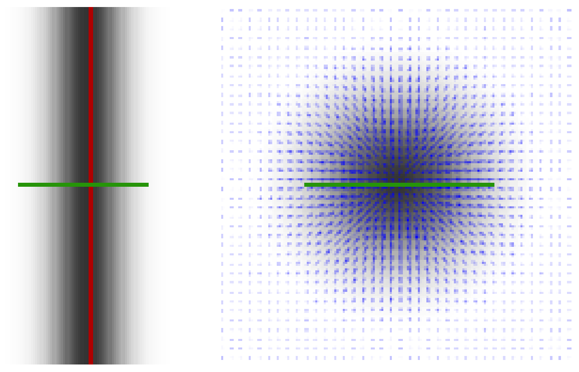


Figure 2.12: Two slice views of an ideal tube with a Gaussian intensity profile

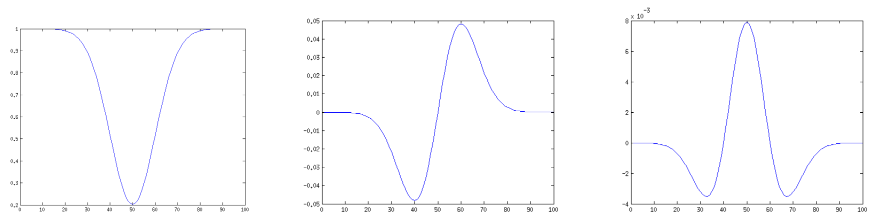


Figure 2.13: Graphs of intensity, first derivative (change in intensity) and second derivative of the green line in figure 2.12. Note that the second derivative is highest at the tube's center

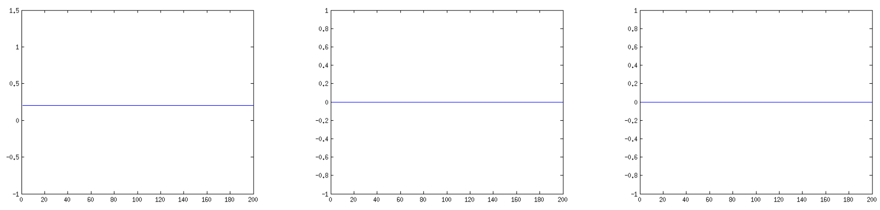


Figure 2.14: Graphs of intensity, first derivative (change in intensity) and second derivative of the red line in figure 2.12

2. The highest intensity change is in the cross-sectional plane of the tube (see figure 2.13)
3. The gradient vector field creates a sink or a source at the center of the tube depending on the tube's intensity (see figure 2.12)
4. The highest change of change in intensity $\partial^2 I(\vec{v})$ is in the center of the tube because the gradients here change direction (see figure 2.13)

Shape analysis by eigenanalysis of the Hessian

The four previous observations about the derivative information of tubes can be used to detect them. This can be done by checking all possible tube directions and checking the derivatives, but this would be very computationally inefficient. Frangi et al. [11] showed how to use the eigenvalues of the Hessian (Eq. 2.2) to efficiently determine locally the likelihood that a tube is present at the current position.

The N eigenvectors of a NxN matrix are non-zero vectors with N components. These eigenvectors have the property that when multiplied with the matrix they remain parallel to that matrix. Each eigenvector $\vec{e_i}$ has a corresponding eigenvalue λ_i that is the factor it is scaled by when multiplied with the matrix as shown in Eq. 2.3.

$$\mathbf{H}\vec{e_i} = \lambda_i \vec{e_i} \tag{2.3}$$

Because the Hessian matrix is a 3x3 symmetric matrix it has 3 eigenvectors that are orthonormal, meaning that they are all normal to each other. The eigenvectors of the Hessian also has a geometric interpretation: The eigenvectors corresponds to the principal directions of the second-order derivatives which are the directions in the volume where the curvature is the maximum and minimum. Recall from the previous section that:

1. The smallest intensity change is in the direction of the tube (see figure 2.14)
2. The highest intensity change is in the cross-sectional plane of the tube (see figure 2.13)

Thus one of the three eigenvectors will be associated with the direction of the tube, and the other two will lay in the cross-sectional plane of the tube. In order to find out which eigenvector this is, one can look at the eigenvalues λ_i.

To do this we sort the three eigenvalues and their corresponding eigenvectors so that we have to following relation: $|\lambda_1| < |\lambda_2| < |\lambda_3|$. The direction of the tube will then be given by $\vec{e}_1$ which is the eigenvector with the eigenvalue of smallest magnitude $|\lambda_1|$. The reason for this is that the eigenvalues corresponds to the principal curvature which means that they represent the amount of curvature, or in our case: change in intensity change. And since we know that the smallest intensity change is in the direction of the tube, the eigenvector with the smallest eigenvalue magnitude will also point in the direction of the tube. The two other eigenvectors will lay in the cross-sectional plane of the tube and have high corresponding eigenvalues. This is because the highest intensity change is in the cross-sectional plane of the tube and because all the eigenvectors have to be orthonormal. Figure 2.15 shows how these different eigenvectors relate to a tube and its orientation.

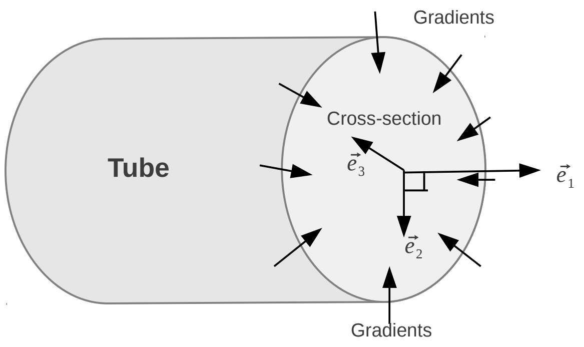


Figure 2.15: Diagram of an ideal tube and its eigenvectors

The reason why we look at the absolute value of the eigenvalues is that the sign of the eigenvalues only indicate the direction of the gradient. In other words: if the tube is white and the background is black the eigenvalues λ_2 and λ_3 will be negative. And if the tube is black and the background white they will be positive.

Thus for an ideal tube the following relations of the eigenvalues should hold, and can be used to detect tubular structures:

$$|\lambda_1| \approx 0 \tag{2.4}$$

$$|\lambda_1| << |\lambda_2| \tag{2.5}$$

$$\lambda_2 \approx \lambda_3 \tag{2.6}$$

Table 2.3 show which type of structures different value configurations of eigenvalues of the Hessian correspond to.

λ_1	λ_2	λ_3	Structure
L	L	-H	Plate (bright)
L	L	+H	Plate (dark)
L	-H	-H	Tubular (bright)
L	+H	+H	Tubular (dark)
+H	+H	+H	Blob (dark)
-H	-H	-H	Blob (dark)

Table 2.3: Different value configurations of eigenvalues and their corresponding structure/shape. H means high value and L low value.

Scale Invariance and TDFs

Note that in larger tubes, such as the one in figure 2.11 the gradients does not exist in the center, because there is only intensity change at the border. Thus it is not possible to calculate the Hessian at the center. For smaller tubes, where the center is right next to the border, the gradients will exist in the center. Solving this problem for larger tubes can be done by propagating the gradient information from the border to the center. In the literature there exist two main methods of doing this: Gaussian Scale-Space and Gradient Vector Flow. The next two sections discusses these two methods in further detail.

Gaussian Scale-Space

Gaussian Scale-Space uses Gaussian smoothing of the image at different scales to propagate the gradient information to the center. Gaussian smoothing will spread the gradient information in all directions. Different scales are processed by using different standard deviations σ for the Gaussian smoothing filter G. Thus for each scale the image is smoothed by convolution with a Gaussian filter G with standard deviation σ. After each smoothing, the TDF is run on the result and the final TDF result is the maximum TDF response of each scale as shown in equation 2.7. Figure 2.16 depicts a cross-section of tubes with two different sizes and the same tubes smoothed with one scale for each of the two sizes. The vector field in figure 2.17 shows how the gradient information is propagated to the center of each tube for each of the two scales.

$$T(\vec{v}) = \max_{\sigma \in \mathbf{S}} \text{TDF}(I(\vec{v}) * G_\sigma) \tag{2.7}$$

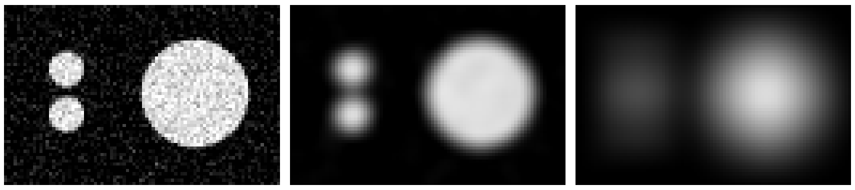


Figure 2.16: From left to right: Original image of tube cross-section; original image with a low scale smoothing; original image with a high scale smoothing. Note how the two small tubes to the left diffuse together when a high scale smoothing is used.

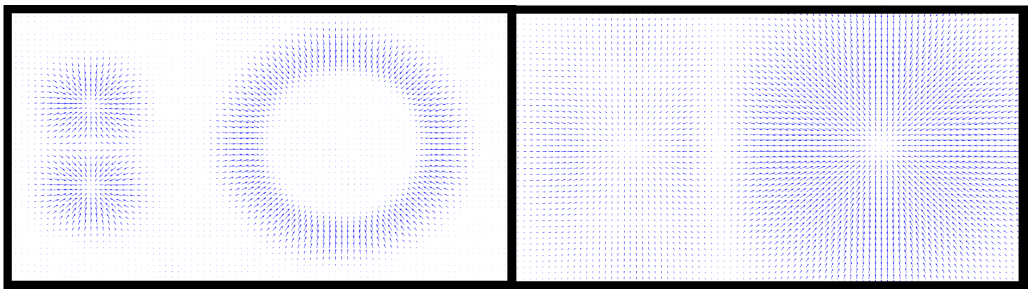


Figure 2.17: Gradient vector field of image in figure 2.16 after Gaussian smoothing of two different scales. To the left low scale and to the right high scale smoothing.

Gradient Vector Flow

Opposite to Gaussian smoothing which diffuses the intensity of the original image, Gradient Vector Flow (GVF) diffuses the image gradients instead. GVF was originally introduced by Xu and Prince [31] as a new external force field for active contours. The resulting gradient vector field $\vec{V}$ of GVF aims to minimize the energy function $E(\vec{V})$:

$$E(\vec{V}) = \int \mu |\nabla \vec{V}(\vec{x})|^2 + |\vec{V}_0(\vec{x})|^2 |\vec{V}(\vec{x}) - \vec{V}_0(\vec{x})|^2 d\vec{x} \tag{2.8}$$

where $\vec{V}_0$ is the initial gradient vector field. Figure 2.18 depicts how the GVF work when run iteratively on a simple image of the letter A. The top row shows the image and the magnitude of the vector field after a set of iterations. The bottom row depicts the vector field superimposed on the image. Notice how the gradients spread in the gradients direction for each iteration.

Bauer and Bischof [4] were the first to point out that GVF could be used to create scale-invariance of TDFs and serve as a replacement to the Gaussian Scale-Space method. This is possible because GVF will propagate the

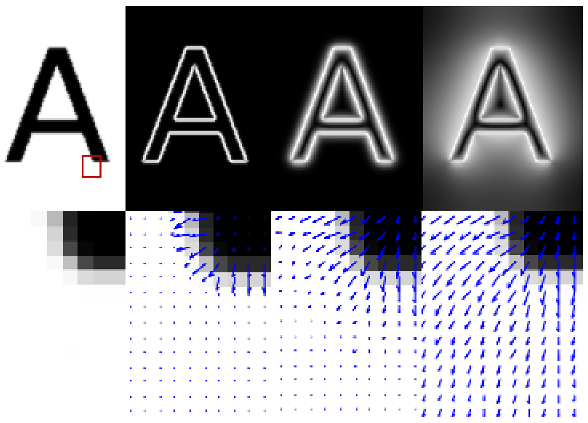


Figure 2.18: Example of GVF execution. From left to right: **Top:** 1) Smoothed image. 2) Magnitude of image gradients $\vec{V}_0$ 3) Magnitude of GVF after 10 iterations, 4) Magnitude of GVF after 400 iterations. **Bottom:** 1)Zoomed area of smoothed image 2, 3 and 4) Image gradients superimposed on zoomed image after 0, 10 and 400 iterations.

gradient information from the tube border to the center just as Gaussian smoothing.

Also, GVF addresses an important problem of Gaussian Scale-Space: Gaussian smoothing is not feature-, or structure-preserving which can lead to two or more structures diffusing into each other. This can give the impression of a false tube of a larger scale as shown in the high-scale smoothing images to the right in figures 2.16 and 2.17.

GVF is a feature-preserving spatial diffusion of gradients and thus solves this problem. Also, GVF does not have to be run for a set of scales such as with Gaussian Scale-Space. Instead a single GVF result can be used and thus it is not needed to know which tube scales are present in the data. Figures 2.19 and 2.20 shows the result of GVF when run on an image of tubes with different sizes.

However, calculating the GVF field is very slow due to the need for many iterations of the algorithm to reach convergence. This has limited its practical usage for large volumes such as CT scans of the lung, but our previous work [27] shows how GVF can be efficiently computed in only a few seconds by running the calculations in parallel on a GPU.

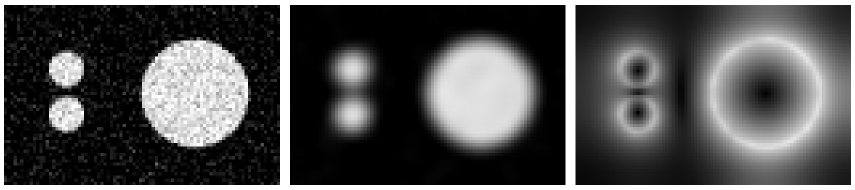


Figure 2.19: From left to right: Original image of tube cross-section; original image with a low scale smoothing to remove image noise and used as input $\vec{V}_0$; magnitude of GVF vector field.

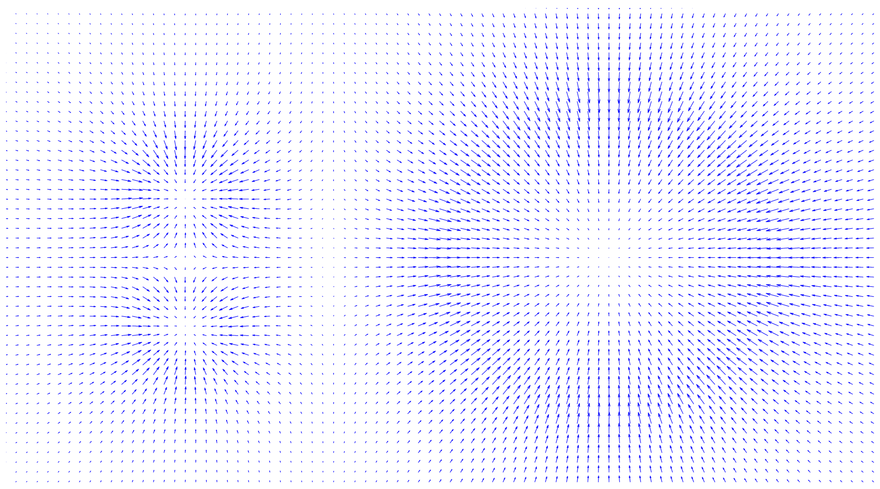


Figure 2.20: Vector field after GVF has been run on the smoothed image in figure 2.19. Note how the gradient information has been propagated to the center on the large tube cross section but still kept the small tubes to the left.

Central TDFs

Unfortunately, determining whether the eigenvalues are high/low and negative/positive is not enough to accurately determine whether a voxel is part of a tubular structure. More information and criterias are needed to increase the accuracy of the TDFs.

Central TDFs are a class of TDFs that only use information at or close to the current voxel, such as the eigenvalues. Frangi et al. [11] designed a very popular central TDF that use two geometric criterias R_A and R_B and one structureness S criteria to deal with noise. The geometric model is based on the expected second order derivative information. This is done by considering an ellipsoid where the length of each of the ellipsoids axis' are defined as the

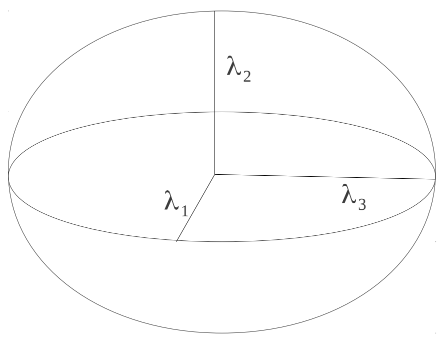


Figure 2.21: Second order ellipsoid

magnitude of the eigenvalues as shown in figure 2.21. For a perfect tubular structure this second order ellipsoid would look like a pancake because $|\lambda_1|$ is very small and $|\lambda_2|$ and $|\lambda_3|$ are large.

The first measure R_A is the largest cross section's area divided by the largest axis' semi length. This ratio will be very small for plate like structures and larger for blob and tubular structures.

$$R_A = \frac{\text{Largest cross section's area}/\pi}{(\text{Largest Axis Semi-length})^2} = \frac{|\lambda_2||\lambda_3|\pi/\pi}{|\lambda_3|^2} = \frac{|\lambda_2|}{|\lambda_3|} \tag{2.9}$$

The second measure R_B is the volume of the second order ellipsoid divided by the largest cross section's area. This ratio should be highest for a blob like structure because there is a change in intensity in all directions creating a large ellipsoid. On the other hand, for a tubular structure the second order ellipsoid would be very small, because there is very little change in the direction of the tube.

$$R_B = \frac{\text{Volume}/(4\pi/3)}{(\text{Largest cross section's area}/\pi)^{3/2}} = \frac{(4\pi|\lambda_1||\lambda_2||\lambda_3|/3)/(4\pi/3)}{((\pi|\lambda_2||\lambda_3|)/\pi)^{3/2}} = \frac{|\lambda_1|}{\sqrt{|\lambda_2||\lambda_3|}} \tag{2.10}$$

The last measure is used to distinguish from background noise. With this Frangi et al. [11] assumes that background noise have low contrast with the actual background and thus the squared sum of all the eigenvalues should be low. They called this the structureness measure S:

$$S = \sqrt{\sum_i \lambda_i^2} \tag{2.11}$$

To sum up: A tubular structure should have a high R_A, a low R_B and a high structureness S. Frangi et al. [11] combined this in the following exponential expression:

$$V = (1 - \exp\left(-\frac{R_A^2}{2\alpha^2}\right)) \exp\left(-\frac{R_B^2}{2\beta^2}\right) (1 - \exp\left(-\frac{S^2}{2c^2}\right)) \tag{2.12}$$

Central TDFs such as the methods presented by Frangi et al. above has the advantage that they are fast to compute because they only use local information. But using only local information may not always be enough to differentiate between noise/non-tubular and tubular structures.

Offset TDFs

Offset TDFs use information at specific offsets from the center as well as at the center. This can give a greater accuracy to the TDFs because they use more information, but it also increases execution time because more information has to be gathered and calculated from the image.

One simple offset TDF is the Circle Fitting (CF) method by Krissian et al. [17]. With this method a circle is constructed in the cross-sectional plane defined by the two eigenvectors $\vec{e}_2$ and $\vec{e}_3$. First a very small radius is used. For a defined number of evenly spaced points on this circle the gradient vector field is sampled using trilinear interpolation. The position of each point i on the circle is found by first calculating the angle as $\alpha = \frac{2\pi i}{N}$ and the direction from the center to the point as $\vec{d_i} = \vec{e}_2 \sin\alpha + \vec{e}_3 \cos\alpha$. The position of point i on a circle with radius r and center $\vec{v}$ is then equal to $\vec{v} + r\vec{d_i}$. As shown in equation 2.13, the average dot product between the sampled gradient and the inward normal $(-\vec{d_i})$ of the circle at each point is calculated for the given radius. This radius is then increased and the average dot product is calculated again. This is done as long as the average increases. The gradients will continue to increase in length until the border is reached. After the tube border, the gradients will decrease in length. Thus this method tries to fit a circle to the gradient information such as shown in figure 2.22.

$$K(\vec{v}, r, N) = \frac{1}{N} \sum_{i=0}^{N-1} \vec{V}(\vec{v} + r\vec{d_i}) \cdot -\vec{d_i} \tag{2.13}$$

The resulting TDF response of the CF method is the largest average dot product. This offset TDF is more selective than Frangi et al. [11] central

TDF, but is slower to compute because it has to sample many points using trilinear interpolation. Nevertheless, this method also outputs a radius for each voxel that can guide further processing and be used to create a soft-segmentation result.

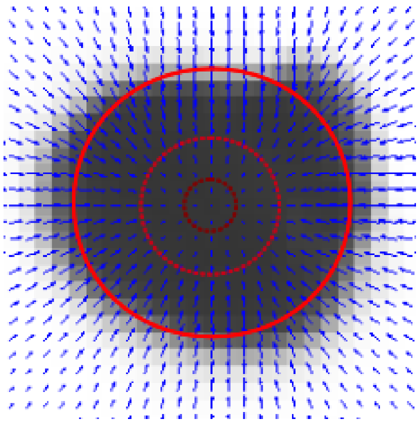


Figure 2.22: Krissian et al. [17] Circle Fitting method: A circle is inflated from the center in the cross-sectional plane. The average dot product between the gradient vector field and the best fit circle normals are used as TDF response.

2.4.4 Centerline Extraction by Ridge Traversal

A common way to perform centerline extraction after tube detection is to perform a ridge traversal on the result of the tube detection filters (TDFs). This is possible when the TDF have the medialness property. Medialness is a measure of how "in the center" a position is inside an object such as a tube. The response from a TDF with this property will be largest in the center of the tube and decreasing from the center to the boundary. Both Frangi et al. [11] and Krissian et al. [17] circle fitting method from the previous section has this property. A ridge traversal procedure can then extract the tube centerline by jumping from point to point inside the tube looking for the next large TDF value. Figure 2.23 shows the extracted centerlines in green superimposed on the TDF result on CT images of the lungs.

Aylward et al. [2] provides a review of different centerline extraction methods and proposed an improved method based on a set of ridge criteria and different methods for handling noise. Bauer et al. [5] presented a similar ridge traversal method with the GVF method. Instead of traversing the ridges created by the TDF, Bauer et al. [5] proposed to follow the valleys created

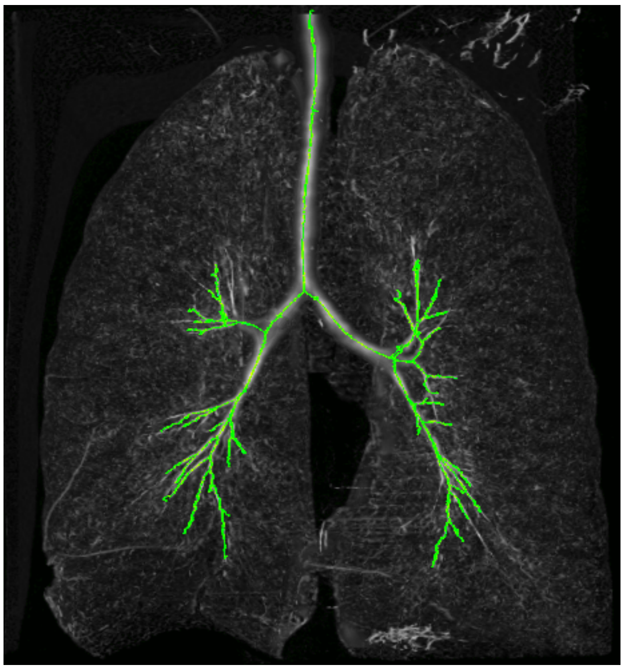


Figure 2.23: Maximum Intensity Projection of the TDF result on CT images of the lungs. The green lines are the centerlines extracted from the underlying TDF using ridge traversal.

by the GVF method. This is based on the fact that the magnitude of the vector field from GVF will decrease towards the center of any closed object as noted earlier by Hassouna et al. [14]. The TDF was used by Bauer et al. [5] to automatically create seed points.

The ridge traversal method we decided to use is based on the method of Aylward *et al.* [2]. The method starts with a seed voxel $\vec{v}_0$. For each voxel $\vec{v}_i$ we know a tube likeliness value $T(\vec{v}_i)$ provided by a tube detection filter such as the ones in the previous section. Also, for each voxel, we have an estimate of the tube's direction $\vec{t}_i$. This direction estimate is based on the eigenvector associated with the smallest eigenvalue $\vec{e}_1$ of the Hessian matrix. The direction of the seed voxel is set to this eigenvalue $\vec{t}_0 = \vec{e}_1$. From this voxel a new voxel is selected as the next point on the centerline. This is done by selecting the neighboring voxel in the direction $\vec{t}_0$ that has the largest TDF value. This procedure is repeated until the TDF value of the next maximum neighboring voxel drops below a certain threshold. During the traversal the tube directions are updated as shown in equation 2.14 by taking the average of the previous direction and the new voxel's estimate $\vec{e}_1$. The average is used so that the sensitivity to noise that will corrupt the eigenvector $\vec{e}_1$ is reduced.

Also, the sign of the dot product between these two are multiplied with the average to maintain the same direction. This is necessary because the sign of the direction estimate $\vec{e}_1$ is not guaranteed to be equal for neighboring voxels.

$$\vec{t}_i = \text{sign}(\vec{t}_{i-1} \cdot \vec{e}_1)\frac{\vec{t}_{i-1} + \vec{e}_1}{|\vec{t}_{i-1} + \vec{e}_1|} \tag{2.14}$$

When the traversal stops, the method returns to the seed voxel $\vec{v}_0$ and continues traversing in the opposite direction $-\vec{t}_0$.

Several seed points are necessary to extract the centerline for complex tubular networks such as the Airway Tree. When a traversal procedure hits a voxel that has already been extracted as part of another centerline the traversal stops.

Multiple seed points can be retrieved by selecting all voxels that have a TDF value above a high threshold and has the highest TDF value amongst its neighbors. But this method requires some way to throw away invalid or unnecessary centerlines as not all seed points will be valid and thus create invalid centerlines. This can be done by rejecting very small centerlines and requiring that the average TDF value of each voxel on the centerline is above a given threshold.

This centerline extraction method has the advantages that it is simple and quite fast. But the method can also easily stop prematurely due to noise or local artifacts. Also this method is very sensitive to its initialization, the seed points. If one tube segment has a low TDF value and there is no seed point on the isolated tube segment its centerline will not be extracted. As this method is completely serial its speed cannot be increased by parallelization, but as the method is so fast compared to the TDF calculations this is not an issue.

2.4.5 Segmentation by Inverse Gradient Flow Tracking

From the extracted centerlines a segmentation result will be created. A soft segmentation can be provided by using the circles for each centerline point from Krissian et al. circle fitting method. But in this work we will create a segmentation result that for each voxel in the volume determine whether the voxel is part of the airway tree or not. Bauer et al. [6] proposed a method for performing such a segmentation from the centerline using the already

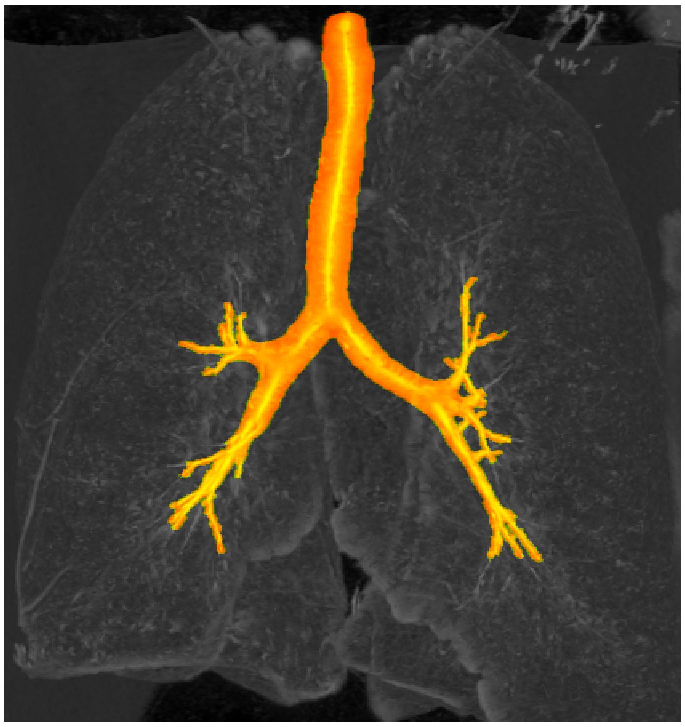


Figure 2.24: Maximum Intensity Projection of the TDF result on CT images of the lungs. The yellow area is the voxels that have been segmented as part of the airways using Inverse Gradient Flow Tracking.

computed GVF vector field. They named this method Inverse Gradient Flow Tracking Segmentation because it for each voxel tracks the centerline using the directions of the GVF vector field, but in the inverse direction.

Basically, the method works by growing the segmentation from the centerlines in the inverse direction of the GVF field as long as the length of the gradient vectors are larger than the previous ones. This makes sense because the magnitude of the gradient vectors should be largest at the border of the tubes/airways. A segmentation result of using this method can be seen in figure 2.24.

2.5 Conclusions from background study

In this chapter we concluded that many image processing tasks are data parallel because each image element can often be processed by the same instructions. We saw that modern graphic processing units (GPUs) are excellent at performing large data parallel tasks. We reviewed several methods for Airway Segmentation and Centerline extraction and concluded that the Hessian-based methods perform very well and are very data parallel. We chose the Hessian-based method by Bauer et al. [6] and explained this method in further detail. In the next chapter we will go through our implementation of this method step by step.

Chapter 3

Methodology

This chapter describes our implementation of Airway Segmentation and Centerline Extraction. As explained in the previous chapter, we choose, based on our extensive background study, to base our implementation on the methods of Bauer et al. (see [6] and Bauer's PhD thesis [3]). Figure 3.1 depicts the pipeline of our implementation. The numbers indicates which section in this chapter that describes the corresponding part of the pipeline.

One of the goals of this project was to exploit data parallelism and the parallel processing power of Graphic Processing Units. For this purpose we chose to program our implementation in C++ and OpenCL. OpenCL is, as described in the previous chapter, a framework for parallel programming on heterogeneous systems.

The pipeline of our implementation starts with some pre-processing which involves some Gaussian smoothing and normalization of the data. This step is explained in section 3.1. The next section, 3.2, explains how we implemented the Circle Fitting tubular detection filter (TDF) introduced by Krissian et al. [17]. Because the Airway Tree are tubular structures with various radius a method to provide scale-invariance is needed. For this we choose to use the Gradient Vector Flow method which was introduced by Xu and Prince [31] and first used for scale-invariant TDFs by Bauer and Bischof [4]. Our GPU implementation of this method is based on our previous work [27] and is desribed in section 3.3. Note that in the pipeline the TDF is run two times, once for small airways and once for large airways. The reason for this is as explained in [6], that small airways with very little contrast will have a tendency to dissapear in in the GVF process. Thus it is necessary to run the TDF on the output of the pre-processing without GVF.

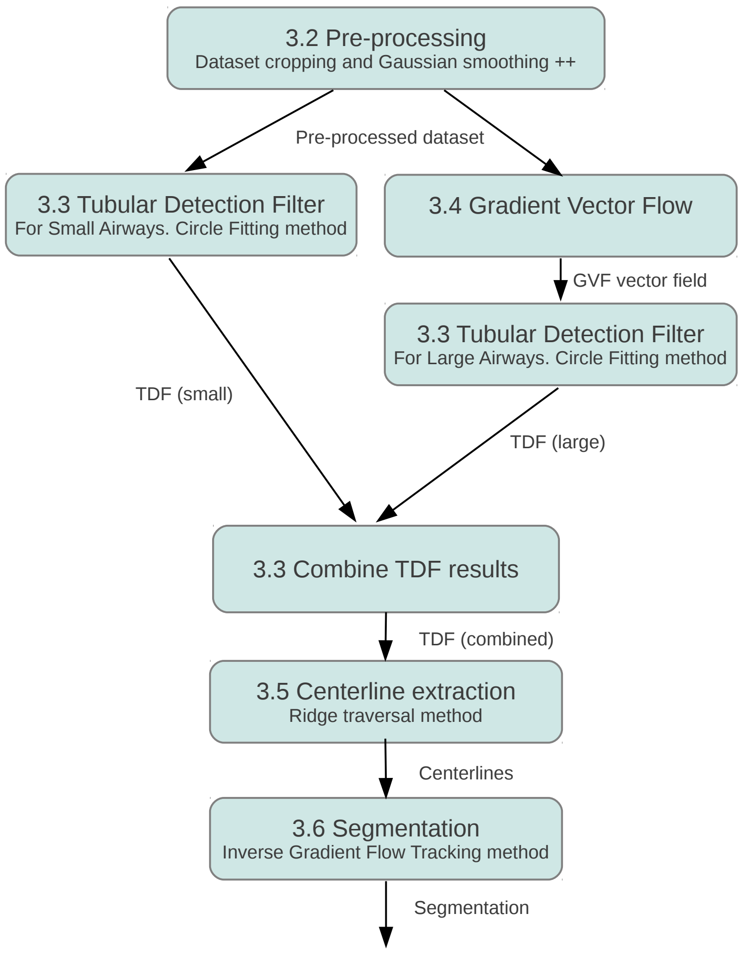

3.2 Pre-processing
Dataset cropping and Gaussian smoothing ++
Pre-processed dataset
3.3 Tubular Detection Filter
For Small Airways. Circle Fitting method
3.4 Gradient Vector Flow
GVF vector field
3.3 Tubular Detection Filter
For Large Airways. Circle Fitting method
TDF (small)
TDF (large)
3.3 Combine TDF results
TDF (combined)
3.5 Centerline extraction
Ridge traversal method
Centerlines
3.6 Segmentation
Inverse Gradient Flow Tracking method
Segmentation

Figure 3.1: Block diagram of the implementation. The number indicates which section in this chapter describes that part.

The next two sections, 3.4 and 3.5, describe how we implemented a ridge traversal and inverse gradient flow tracking method for the centerline extraction and segmentation respectively. These two steps are based on the works by Bauer et al. [6], [3] and Aylward et al. [2].

In section 3.6, we discuss how our implementation was parallelized using OpenCL and optimized for running on a GPU.

Throughout this chapter we will use the notation that I denotes the volume or dataset and that $\vec{v}$ is a specific voxel position.

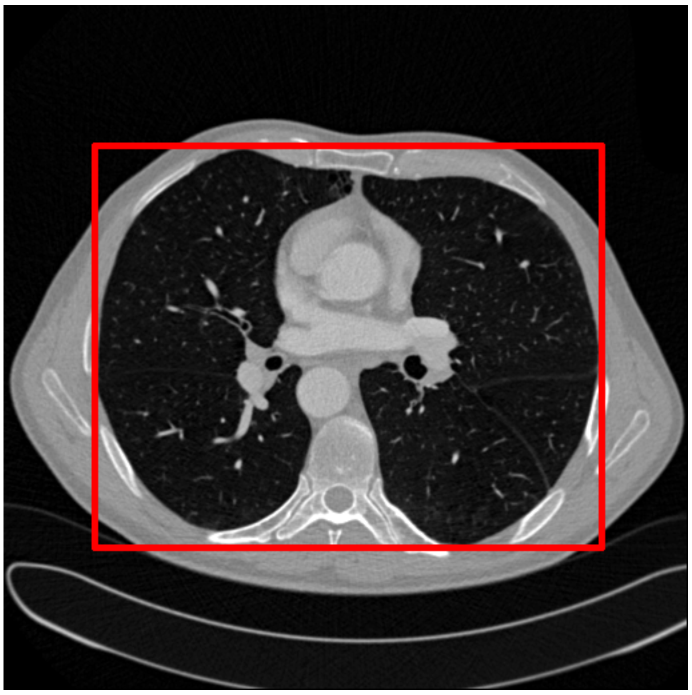


Figure 3.2: One slice from a CT volume and an appropriate cropping shown in red.

3.1 Pre-processing

3.1.1 Dataset cropping

The first step is to crop the dataset so that all of the air and fat surrounding the lungs is removed from the dataset. This cropping saves a lot of processing time because the rest of the pre-processing, the TDF and Gradient Vector Flow all work on all voxels in the dataset. If we are able to prune half of the voxels as not part of the lung, then the processing time can be twice as fast. Figure 3.2 shows one slice from a CT volume and the cropping region in red. Everything outside of the red border is removed and deleted.

In this thesis a novel lung cropping method was created. The cropping method works by scanning one slice at a time in all three directions: x, y and z. Each line in these slices are scanned sequentially. And for each slice the method tries to determine whether the scan line went through the lung or not. The number of scan lines that went through the lung is recorded for each slice. If the number of scan lines that went through the lung is above a threshold called **minScanLines**, we know that that slice has to be part of the dataset. The reason for this threshold is that noise can give false reponses and thus we need a lower limit before we conclude that the scanned slice is inside the lung. For each direction we look for the first and last slice that has the minimum required scan lines inside the lung. These two slices determines the border of the cropping. Algorithm 1 gives a more detailed pseudocode for our cropping method. An advantage of this cropping method is that each scan line can be processed entirly in parallel.

3.1.2 Gaussian smoothing

Gaussian smoothing of the dataset is necessary because of noise in the CT images. If the data is not smoothed Gradient Vector Flow will have problems with propagating the gradient information from the edge of the airways to the center. Recall that this information is necessary at the center to calculate the eigenvectors needed to perform the TDF. Smoothing to much can also be a problem because important edge information might be lost. Smoothing is executed by convolution of the dataset with a small Gaussian kernel of scale/standard deviation σ.

In practice, discrete convolution is performed by calculating a new value of each voxel based on a weighted sum of the neighboring voxels. The size of the neighborhood is chosen so that it increases with increasing σ. We chose to use a neighborhood NxNxN, where $N = 2\lceil 3\sigma \rceil + 1$. The weight for each neighbor voxel and the current voxel is calculated using a Gaussian distribution as

$$(I * G_\sigma)(\vec{v}) = \frac{1}{Z} \sum_{x=-\lceil 3\sigma \rceil}^{\lceil 3\sigma \rceil} \sum_{y=-\lceil 3\sigma \rceil}^{\lceil 3\sigma \rceil} \sum_{z=-\lceil 3\sigma \rceil}^{\lceil 3\sigma \rceil} I(\vec{v}) e^{-\frac{x^2+y^2+z^2}{2\sigma^2}} \tag{3.1}$$

where Z is a normalization constant that is equal to the sum of all the weights. Figure 3.3 shows the effect of the Gaussian smoothing on one CT image slice with $\sigma = 1.0$.

3.1.3 Hounsefield Units Conversion

Recall from the previous chapter that the intensity values of X-ray/CT images are measured in Hounsefield Units that are directly related to the radiation absorption amount of the tissue at a specific position. Since the absorption amount varies for different types of tissue, the Hounsefield Unit

Algorithm 1 Pseudocode of cropping procedure

```
for each slice direction (x,y and z) do
  for each slice in current direction do
    for each scan line do
      for each scan line element do
        if volume[position] > HUthreshold then
          if whiteCount = Wlimit then
            detectedWhite ← detectedWhite + 1
            blackCount ← 0
          end if
          whiteCount ← whiteCount + 1
        else
          if blackCount = Blimit then
            detectedBlack ← detectedBlack + 1
            whiteCount ← 0
          end if
          blackCount ← blackCount + 1
        end if
      end for
      if (detectedWhite = 2 and detectedBlack = 1) or (detectedBlack
      > 1 and detectedWhite > 1) then
        scanLinesInside[sliceNr] ← scanLinesInside[sliceNr] + 1
      end if
    end for
  end for
  for each slice in increasing order do
    if scanLinesInside[sliceNr] > minLinesInside then
      first cropping border for this direction is sliceNr
      break
    end if
  end for
  for each slice in decreasing order do
    if scanLinesInside[sliceNr] > minLinesInside then
      second cropping border for this direction is sliceNr
      break
    end if
  end for
end for
```

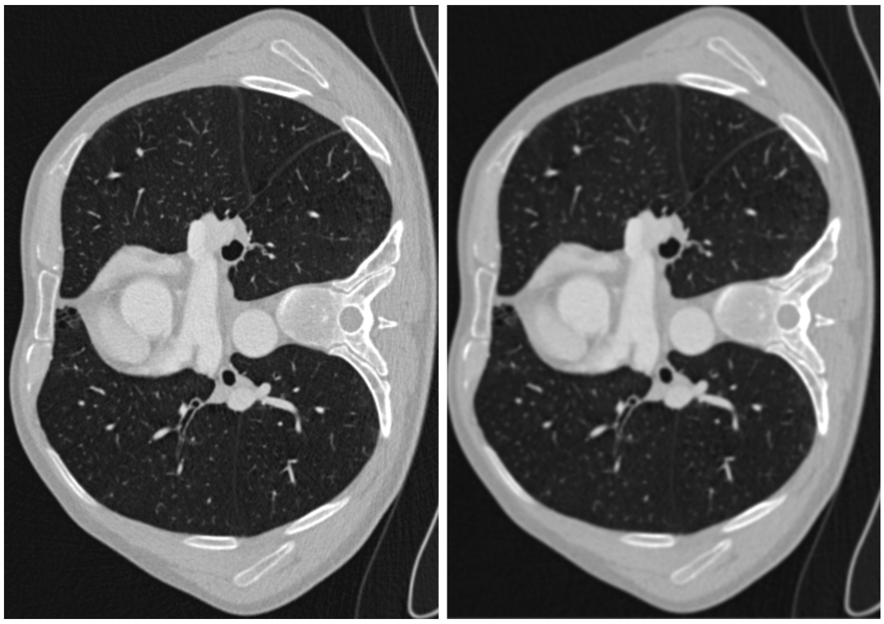


Figure 3.3: The effect of Gaussian smoothing. The image to the left is one CT slice before Gaussian smoothing and the one to the right is after smoothing.

measure can be roughly mapped to the types of tissue or density of the tissue. On the bottom of the scale is air which is around -1000 HU and on the other end is bone with 700 to 3000 HU. Since we only care about differentiating between the air that is inside the airways and all other types of tissue we use a threshold $\mathrm{HU}_{\mathrm{max}}$ of around -100 HU. All voxels with intensity above this threshold is set to the threshold. The other voxels are scaled according to the minimum HU, $\mathrm{HU}_{\mathrm{min}}$, so that the range of intensity values is converted to floating point numbers from 0.0 to 1.0 as shown in equation 3.2. Note that the minimum HU value for X-ray/CT images is almost always $\mathrm{HU}_{\mathrm{min}} = -1024$. Figure 3.4 depicts the effect of this HU scaling and thresholding.

$$I(\vec{v}) = \begin{cases} 1.0 & \text{if } I(\vec{v}) \geq \mathrm{HU}_{\mathrm{max}} \\ \frac{I(\vec{v}) - \mathrm{HU}_{\mathrm{min}}}{\mathrm{HU}_{\mathrm{max}} - HU_{\mathrm{min}}} & \text{else} \end{cases} \tag{3.2}$$

3.1.4 Initialize gradient vector field

The next step in the pre-processing stage is to calculate the initial gradient vector field and normalize it. Recall that the gradient of an image $\nabla I(\vec{v})$ is a vector that describes the intensity change (derivatives of the image intensity)

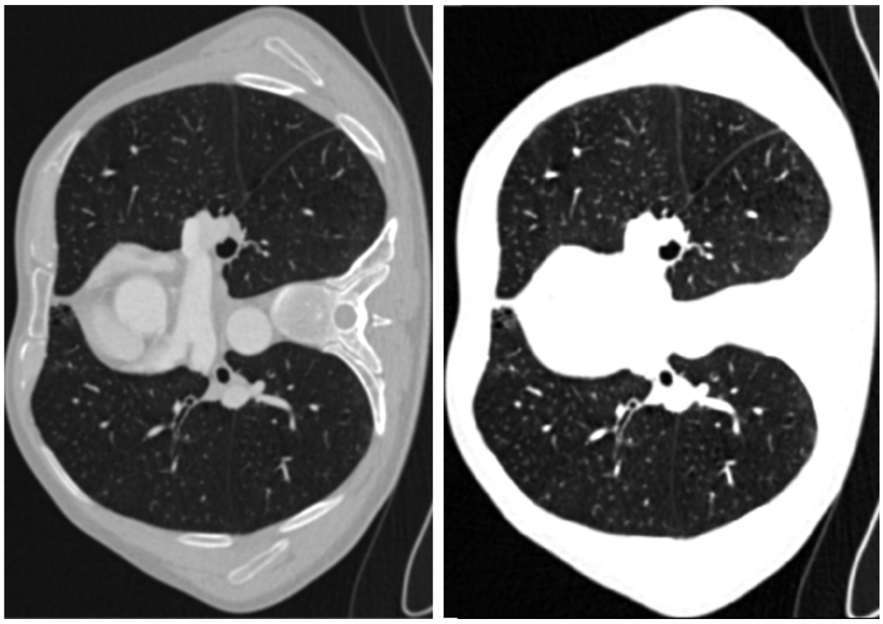


Figure 3.4: The effect of HU threshold. The image to the left is one CT slice after Guassian smoothing and the one to the right is the same image after HU thresholding by equation 3.2 has been performed.

at a specific point $\vec{v}$ in the image. The angle of the vectors will describe the direction of the maximum intensity change and the magnitude of the gradient vector describes how big the change is.

Calculating the derivatives numerically is usually done by using finite difference methods. We used a central difference scheme which takes two neighboring voxel values in each direction and calculates the difference as shown below:

$$\begin{aligned}
\nabla I(\vec{v})_x &= \frac{I(\vec{v}+(1,0,0)) - I(\vec{v}-(1,0,0))}{2} \\
\nabla I(\vec{v})_y &= \frac{I(\vec{v}+(0,1,0)) - I(\vec{v}-(0,1,0))}{2} \\
\nabla I(\vec{v})_z &= \frac{I(\vec{v}+(0,0,1)) - I(\vec{v}-(0,0,1))}{2}
\end{aligned}$$

After the gradient vector field, $\vec{V}(\vec{v}) = \nabla I(\vec{v})$, has been calculated it has to be normalized. The normalization is necessary to ensure contrast-invariance

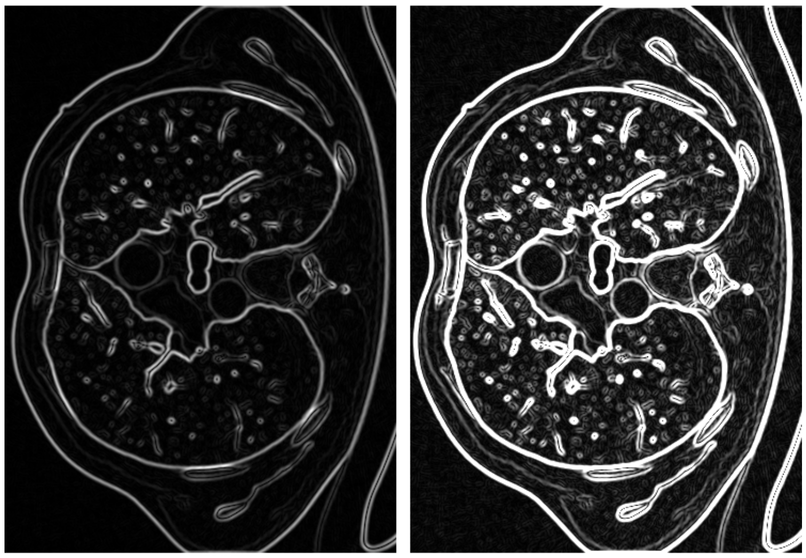


Figure 3.5: Gradient vector field normalization. To the left is the magnitude of the image gradients after the image has been smoothed. To the right is the same magnitude of gradients image after normalization by equation 3.3.

of the TDF. With contrast-invariance we mean that if an airway has low contrast from the lumen to the wall it should get just as good TDF result as an airway with a high contrast. The normalization is also necessary for making sure that the important gradient information is maintained in the GVF process. The normalization uses a parameter $F_{\max}$ and all gradient vectors with magnitude above this value should be scaled so that it has unit length. All gradients vectors below this parameter will be scaled according to this parameter. The reason for not normalizing all vectors to unit length is that this would make GVF impossible and would increase the sensitivity of noise. Equation 3.3 shows how the normalized gradient vector field $\vec{V}^n$ is created. Figure 3.5 depicts the normalization on a CT image.

$$\vec{V}^n(\vec{v}) = \begin{cases} \frac{\vec{V}(\vec{v})}{|\vec{V}(\vec{v})|} & \text{if } |\vec{V}(\vec{v})| \geq F_{\max} \\ \frac{\vec{V}(\vec{v})}{F_{\max}} & \text{else} \end{cases} \tag{3.3}$$

3.2 Tubular Detection Filter

Recall from the previous chapter that a tubular detection filter (TDF) performs a shape analysis on each voxel and return a value indicating the probability of the voxel belonging to a tubular structure. We decided to use the Circle Fitting TDF by Krissian et al. [17]. This method constructs a circle in the cross-sectional plane of the tube and increase the radius until a best-fit is found.

The cross-sectional plane is defined by the two eigenvectors $\vec{e}_2$ and $\vec{e}_3$ of the Hessian matrix. First, the Hessian matrix $\mathbf{H}$ is determined by calculating the gradient of the derivatives $\vec{V}$ in each direction ($\vec{V}(\vec{v})_x$, $\vec{V}(\vec{v})_y$ and $\vec{V}(\vec{v})_z$) as shown in equation 3.4. This is done with the central difference method as explained in the previous section. Note that the TDF is run two times with different input for the gradient $\vec{V}$ as shown in the pipeline diagram 3.1. For the small airways the image gradient is used, thus $\vec{V}(\vec{v}) = \nabla(I * G)(\vec{v})$. While for larger airways GVF has to be used to propagate the gradient information from the tube border to the center. Hence for larger airways $\vec{V}$ is equal to the output of the resulting vector field of GVF.

$$\mathbf{H}(\vec{v}) = \begin{bmatrix} \nabla(\vec{V}(\vec{v})_x) \\ \nabla(\vec{V}(\vec{v})_y) \\ \nabla(\vec{V}(\vec{v})_z) \end{bmatrix} = \begin{bmatrix} \frac{\partial \vec{V}(\vec{v})_x}{\partial x} & \frac{\partial \vec{V}(\vec{v})_x}{\partial y} & \frac{\partial \vec{V}(\vec{v})_x}{\partial z} \\ \frac{\partial \vec{V}(\vec{v})_y}{\partial x} & \frac{\partial \vec{V}(\vec{v})_y}{\partial y} & \frac{\partial \vec{V}(\vec{v})_y}{\partial z} \\ \frac{\partial \vec{V}(\vec{v})_z}{\partial x} & \frac{\partial \vec{V}(\vec{v})_z}{\partial y} & \frac{\partial \vec{V}(\vec{v})_z}{\partial z} \end{bmatrix} \tag{3.4}$$

After the Hessian matrix has been determined the eigenvectors of this matrix has to be calculated.

3.2.1 Calculating Eigenvalues and Eigenvectors

Two of the most common algorithms for calculating eigenvalues and eigenvectors of a matrix is the QR and QL algorithms, which are very similar. In our work we decided to use the QL algorithm. The basic idea of this algorithm is that any real matrix can be decomposed to the form:

$$\mathbf{H} = \mathbf{Q} \cdot \mathbf{L} \tag{3.5}$$

where Q is an orthogonal matrix and L is the lower triangle of the H matrix. The Householder transformation is used to find this decomposition. Recall that an orthogonal matrix is a square matrix where each column contains orthogonal unit vectors.

The QL algorithm is an iterative method that performs a sequence of transformations that will eventually lead to the eigenvalues and vectors. We start with $\mathbf{H_0} = \mathbf{H}$. Then for each iteration s we find the orthogonal and lower triangle matrix of the current matrix $\mathbf{H_s}$ and create the next matrix $\mathbf{H_{s+1}}$ using the following equation:

$$\mathbf{H_{s+1}} = \mathbf{L_s} \cdot \mathbf{Q_s} \tag{3.6}$$

A theorem says that when this is repeated for several iterations, the eigenvalues will appear on the diagonal of the $\mathbf{L_s}$ matrix and the eigenvectors at the columns of the orthogonal matrix $\mathbf{Q_s}$. The proof of this theorem is quite complex and thus will not be repeated here. The time complexity for the QL algorithm on a nxn matrix is $O(n^3)$ per iteration. We used an implicit QL implementation adapted from the tql2 subroutine from the Fortran library EISPACK.

After convergence, the eigenvectors and their eigenvalues are sorted according to their magnitude so that $|\lambda_1| < |\lambda_2| < |\lambda_3|$.

3.2.2 Circle Fitting

The eigenvectors $\vec{e}_2$ and $\vec{e}_3$ associated with the two largest eigenvalues are used to construct the cross-sectional plane of the tube. In this plane we will sample N points on a circle with radius r. The direction $\vec{d}$ to each point i from 0 to N-1 on the circle is calculated in the following manner:

$$\vec{d_i} = \vec{e}_2 \sin\frac{2\pi i}{N} + \vec{e}_3 \cos\frac{2\pi i}{N} \tag{3.7}$$

Point i on the circle with radius r that lays on the cross-sectional plane with center located at $\vec{v}$ is $\vec{v} + r\vec{d_i}$.

The TDF value T is calculated as the average dot product between all the sampled gradients $\vec{V}$ and the inward normals $\vec{n}$ of the circle at each point for a given radius (see equation 3.8). Note that the inward normal $\vec{n}_i$ of the circle is at point i equal to the inverted directions vector $-\vec{d_i}$. The gradient is sampled using trilinear interpolation. First, a small radius of 0.5 is used. Then the radius is increased and the average dot product is calculated again. The radius is increased as long as the TDF value increases or a maximum radius is reached.

$$T(\vec{v}, r, N) = \frac{1}{N}\sum_{i=0}^{N-1} \vec{V}(\vec{v} + r\vec{d_i}) \cdot (-\vec{d_i}) \tag{3.8}$$

The pseudocode below shows step by step how the TDF value is calculated for a given voxel $\vec{v}$:

Algorithm 2 Pseucode of Circle Fitting TDF at position $\vec{v}$

```
Calculate eigenvectors $\vec{e}_1$, $\vec{e}_2$ and $\vec{e}_3$ of the Hessian matrix at position $\vec{v}$
maxSum ← 0
for r from 0.5 to maxRadius do
    sum ← 0
    for i from 0 to N-1 do
        $\vec{d_i} \leftarrow \vec{e}_2 \sin \frac{2\pi i}{N} + \vec{e}_3 \cos \frac{2\pi i}{N}$
        sum ← sum + $\vec{V}(\vec{v} + r\vec{d_i}) \cdot (-\vec{d_i})$
    end for
    if sum > maxSum then
        maxSum ← sum
    else
        break
    end if
end for
return maxSum
```

3.3 Gradient Vector Flow

In the previous chapter, we explained that a method for propagating gradient information from the airway wall to the center was necessary for being able to calculate the Hessian and its eigenvectors at the center of the airway. We chose to use Gradient Vector Flow (GVF) to do this.

The goal of GVF is to find the vector field $\vec{V}$ that minimizes the energy function E defined as:

$$E(\vec{V}) = \int \mu |\nabla \vec{V}(\vec{x})|^2 + |\vec{V}_0(\vec{x})|^2 |\vec{V}(\vec{x}) - \vec{V}_0(\vec{x})|^2 d\vec{x} \tag{3.9}$$

where $\vec{V}_0$ is the input vector field. In our case of tubular detection this is set to the normalized gradients of the volume that is smoothed by a Gaussian ($\vec{V}_0 = \vec{V}^n$ from equation 3.3). Figure 3.6 shows the result of GVF after applied on the normalized gradient vector field of a CT image.

The creators of GVF, Xu and Prince [31], explained how this vector field could be found by iteratively solving the following Euler equation for each

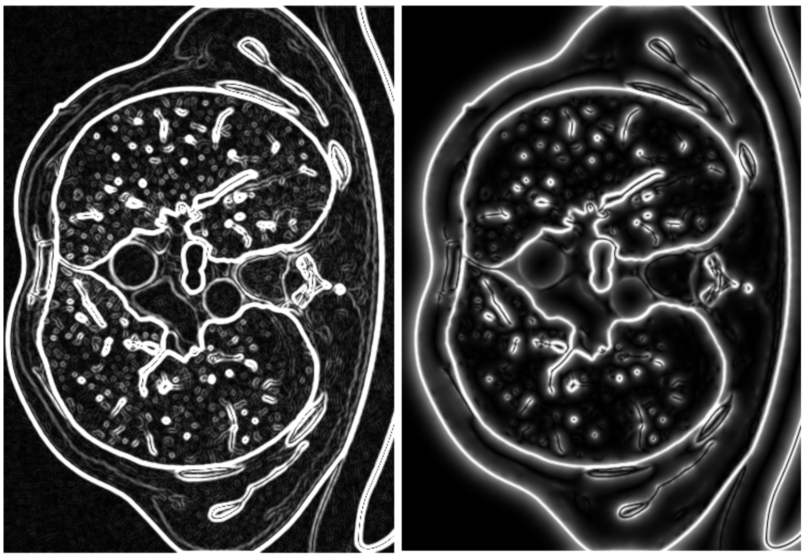


Figure 3.6: To the left is the magnitude of the normalized gradient vector field. To the right is the same image after Gradient Vector Flow has been performed.

vector component independently:

$$\mu\nabla^2\vec{V} - (\vec{V} - \vec{V}_0)|\vec{V}_0|^2 = \vec{0} \tag{3.10}$$

This equations is solved by treating $\vec{V}$ as a function of time and solving the resulting diffusion equations with the numerical scheme shown in algorithm 3. The lapacian $\nabla^2\vec{V}(\vec{v})$ is approximated using a 7 point stencil finite difference scheme.

Algorithm 3 3D Gradient Vector Flow

```
for a predefined number of iterations do
  for all points $\vec{v} = (x, y, z)$ in volume do
    laplacian $\leftarrow -6\vec{V}(\vec{v}) + \vec{V}(x+1,y,z) + \vec{V}(x-1,y,z) + \vec{V}(x,y+1,z) + \vec{V}(x,y-1,z) + \vec{V}(x,y,z+1) + \vec{V}(x,y,z-1)$
    $\vec{V}(\vec{v}) \leftarrow \vec{V}(\vec{v}) + \mu*$ laplacian $-(\vec{V}(\vec{v}) - \vec{V}_0(\vec{v}))|\vec{V}_0(\vec{v})|^2$
  end for
end for
```

3.4 Centerline Extraction

Our program will extract the centerlines from the TDF result using a ridge traversal method. This step is based on the works by Bauer et al. [6], [3] and Aylward et al. [2].

The Centerline Extraction method starts by automatically creating a stack of candidate seed voxels C. The candidate seeds are all voxels that have a TDF value above a certain threshold called T_{high}. Also, the voxel has to have the highest TDF value of all of its closest 26 neighboring voxels. After all voxels that satisfy these criteria have been added to this stack it is sorted according to their TDF value so that the voxel with the highest TDF value is located at the top of the stack C. This is done to ensure that the centerlines of the extracted seeds are in the center of the airway.

Centerlines are then extracted from these candidate seed points as explained earlier in section 2.4.4. The candidate seed points are handled in order, starting with the one with highest TDF value. The traversal procedure continues as long as the TDF value of the next point don't drop below T_{low} or an already extracted centerline is hit. The new centerline is accepted if the following conditions hold:

1. The centerline length is above D_{min}
2. The average TDF value of each point on the centerline is above T_{mean}
3. The centerline is not connected to any other centerline OR connected to one other centerline OR connected to two **different** previously extracted centerlines

If the new centerline hit some previously found centerlines, they are all connected. The output is the largest connected centerline.

The pseudocode in algorithm 4 gives a detailed description of how our implementation works.

3.5 Segmentation

The segmentation method works by growing out from the extracted centerlines and is based on the Inverse Gradient Flow Tracking method by Bauer et al. [6], [3]. The method starts by dilating the centerline and adding it to the segmentation result. This is done because the centerline might not be

Algorithm 4 Pseudocode of centerline extraction by ridge traversal

```
input: a set of candidate seed points $\mathbf{C}$, sorted on TDF value
for each seed point $\vec{x}_0 \in \mathbf{C}$ do
  $\vec{t}_0 \leftarrow \vec{e}_1(\vec{x}_0)$
  for each direction $\vec{t}_0$ and $-\vec{t}_0$ do
    while current point $\vec{x}_i$ is not in existing centerline and $T(\vec{x}_i) > T_{\text{low}}$
    do
      max $\leftarrow 0$
      $\mathbf{L} \leftarrow \emptyset$
      for each neighbor pixel $\vec{y}$ of $\vec{x}_i$ do
        if $\frac{\vec{y}-\vec{x}_i}{|\vec{y}-\vec{x}_i|} \cdot \vec{t}_i > 0$ and $|\vec{V}(\vec{y})| > \max$ then
          max $\leftarrow |\vec{V}(\vec{y})|$
          $\vec{x}_{i+1} \leftarrow \vec{y}$
        end if
      end for
      $\vec{t}_{i+1} \leftarrow \text{sign}(\vec{t}_i \cdot \vec{e}_1(\vec{x}_{i+1}))\frac{\vec{t}_i+\vec{e}_1(\vec{x}_{i+1})}{|\vec{t}_i+\vec{e}_1(\vec{x}_{i+1})|}$
      $\mathbf{L} \leftarrow \mathbf{L} \cup \vec{x}_{i+1}$
    end while
  end for
  if $|\mathbf{L}| > D_{\text{min}}$ and $T(\mathbf{L}) > T_{\text{mean}}$ then
    if an existing centerline(s) was found then
      connect current centerline to that centerline(s)
    end if
  end if
end for
output: the largest connected centerline
```

exactly at the center. And for this method to work, the center voxels where the gradient vectors change direction has to be part of the initial segmentation. The next step is to create a queue and add the neighbors of the intial segmentation to this queue. Then this queue is processed one voxel at the time.

A voxel $\vec{x}$ is added to the segmentation if there exists a neighbor voxel $\vec{y}$ that is not part of the segmentation, has a larger GVF magnitude than $\vec{x}$ and its GVF vector points towards voxel $\vec{x}$.

Centerlines that are far away from the actual center of the tube may produce holes and gaps in the segmentation result. We implemented a morphological closing procedure to remove these holes. Morphological closing is dilation followed by erosion. We used a simple 3x3 box structering element for this operation.

Algorithm 5 Pseudocode of segmentation method

```
input: set of dilated centerline points C and the GVF vector field $\vec{V}$
set S ← C
queue Q ← C
while $Q \neq \emptyset$ do
  $\vec{x}$ ← Q.pop()
  for each voxel $\vec{y} \in \text{Adj}(\vec{x})$ do
    if $\vec{y} \notin S$ and $|\vec{V}(\vec{y})| > |\vec{V}(\vec{x})|$ and $\text{argmax}_{\vec{z} \in \text{Adj26}(\vec{y})} \frac{(\vec{z}-\vec{y}) \cdot \vec{V}(\vec{y})}{|(\vec{z}-\vec{y})||\vec{V}(\vec{y})|} = \vec{x}$
    then
      $S \leftarrow S \cup \{\vec{x}\}$
      Q.push($\vec{y}$)
    end if
  end for
end while
output: segmentation S
```

3.6 Parallelization and GPU Optimizations

In this section, we present the details of how OpenCL was used to accelerate our implementation on the GPU. All steps of our implementation was implemented in OpenCL to run on the GPU except the actual centerline extraction and segmentation step. These steps were not implemented using

OpenCL because they are serial in nature and thus these steps are run on the CPU serially using C++.

The Gaussian smoothing, Hounsefield Unit Conversion, vector field initialization, TDF, morphological closing and GVF calculations are all completely data parallel on the voxel level. An OpenCL kernel was made for each of these steps that process one voxel each. The position of the voxel that each kernel should process is found by using the *get_global_id* function in OpenCL. For the dataset cropping method each scan line can be run in parallel in all three directions.

3.6.1 Texture Cache Optimizations

Caching is a mechanism of storing data in a memory closer to the processor, usually on the chip itself, so that future requests for that data can be serviced faster. Most caching algorithms use the two following principles:

- **Temporal locality:** Data requested now is likely to be requested again soon.
- **Spatial locality:** Data located near the requested data is likely to be requested soon.

Most modern GPUs have a separate texture cache. These texture caches exists on GPUs because a lot of video games and 3D applications use texture mapping to map an image to 3D objects to create a realistic 3D scene. Textures are simply images, either 1, 2 or 3 dimensional.

When a specific pixel in the texture is requested, the GPU will store the data and the neighboring data in a special buffer that is located near to where the actual calculations are performed. Unlike regular linear storage buffers which only have caching in one dimension, textures can cache neighboring data in 2 or 3 dimensions. Thus when a pixel is requested, the neighboring pixels above and below as well as those to the left and right are cached.

In our implementation, we have many 3-dimensional structures such as the dataset itself, the vector fields and the TDF result. We store all of these structures in textures, or images as they are called in OpenCL. A texture can also have up to four channels. These channels exists to support color textures and transparency. These channels are perfect for packing the x, y and z components of the vector fields.

Note that writing to 3D textures inside a kernel is not enabled by default in OpenCL. To support writing to 3D textures, the OpenCL implementation has to have the extension *cl_khr_3d_image_writes*. At the time of writing, only AMD support this extension. Because of this restrictions, 3D data is written to regular 1-dimensional buffers inside the kernels. After the kernels are finished the content of the buffers are copied into 3D textures so that they can be cached in the next kernel.

3.6.2 Trilinear Interpolation

3D textures on the GPU have another big advantage. Data from textures are fetched with a specific unit that can also perform datatype conversion and interpolation in hardware which is much faster than doing it in software. For our TDF we have to sample very many points on a circle. This sampling is done with trilinear interpolation which is a technique to approximate a continuous point in a discrete grid by using the 8 closest neighboring points in the grid. Thus this requires access to 8 points in the texture and many arithmetic operations to compute the sample. Using the texture interpolation sampler in OpenCL removes the burden of doing this explicitly in software and utilizes the caching mechanisms making sampling of continuous points much faster.

3.6.3 Work-group Optimization

Recall that in OpenCL, work-items are an instance of a kernel and that the work-items are executed on the GPU in groups. AMD calls these units of execution *wavefronts* while NVIDIA calls them *warps*. The units are executed atomically and has at the time of writing the size of 32 or 64 work-items for NVIDIA and AMD respectively. If the work-group sizes are not a multiple of this size some of the GPUs stream processors will be idle for each work-group that is executed. There is also a maximum number of how many work-items that can exists in one work-group. On AMD GPUs this limit is currently 256 and on NVIDIA higher. Also the total number of work-items in one dimension has to be dividable by the size of the work-group in that dimension. So, if we have a volume of size 400 in the x direction, the work-group can have the size 2 or 4 in the x direction, but not 3, because 400 is not dividable by 3.

To summarize we have the following constraints for optimal work-group size:

- The number of work-items should be a multiple of 64 (which is also a multiple of 32)
- The number of work-items must be equal or lower than 256
- The number of work-items in each dimension must be dividable by the size of the work-group in each dimension

The optimal work-group size can vary a lot from device to device so we decided to use the fixed work-group size 4x4x4 (=64 total work-items in a group) which satisfies all of the constraints above. To make sure that the cropped volume is dividable by 4 in each direction we increase the size of the cropping until the new size is dividable by 4.

3.7 Parameters

In this last section we provide a short summary of all the parameters that are present in our implementation and the values that was used.

Symbol	**Description**	**Values**
$F_{\max}$	Gradient Scaling parameter	0.2
μ	GVF regularization parameter	0.05
σ	Standard deviation of Gaussian Smoothing	0.5
HU_{max}	Maximum HU value	-100
HU_{min}	Minimum HU value	-1024
D_{min}	Min centerline distance	7
T_{mean}	Mean TDF value to accept centerline	0.6
T_{high}	Threshold TDF to check as candidate seed points	0.6
T_{low}	Treshold for how low the TDF can be on centerline	0.4

Chapter 4

Results

Six anonymized Computer Tomography datasets of the lungs were provided by St. Olavs Hostpital and SINTEF Medical Technology. In this chapter our implementation is run on each of these six datasets.

Each of the next pages will present results of the tubular detection filter step, centerline extraction and segmentation for each patient. The TDF response and centerlines are depicted using maximum intensity projection (MIP). MIP sends one ray for each pixel in the image through the volume and sets its value equal to the maximum intensity on the ray's path through the volume. The viewer3d plugin for Matlab was used to create these MIP images. To illustrate the segmentation we used our own Marching Cubes implementation together with OpenGL. Marching Cubes is an isosurface extraction method by Lorensen and Cline [23].

The method we have implemented is a general method for segmenting and extracting centerlines from tubular structres and not just airways. To illustrate this we also provide results of extracting blood vessels from a Magnetic Resonance Angio image and an Ultrasound Doppler image of a patient's head.

4.1 Airway Results

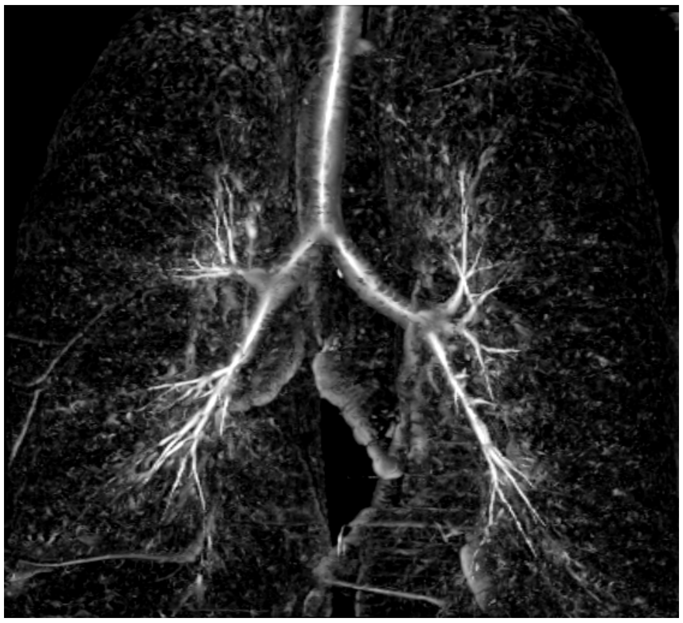


Figure 4.1: Patient 1: Tubular Detection Filter

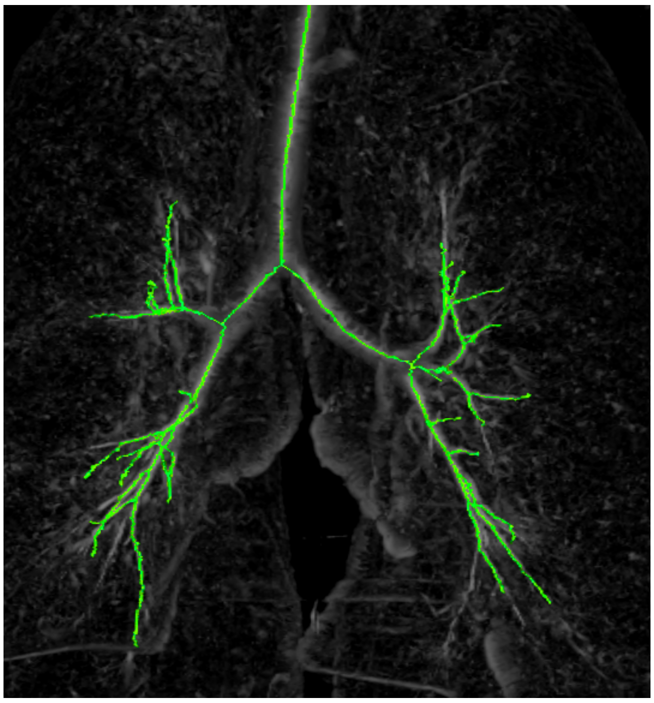


Figure 4.2: Patient 1: Centerlines

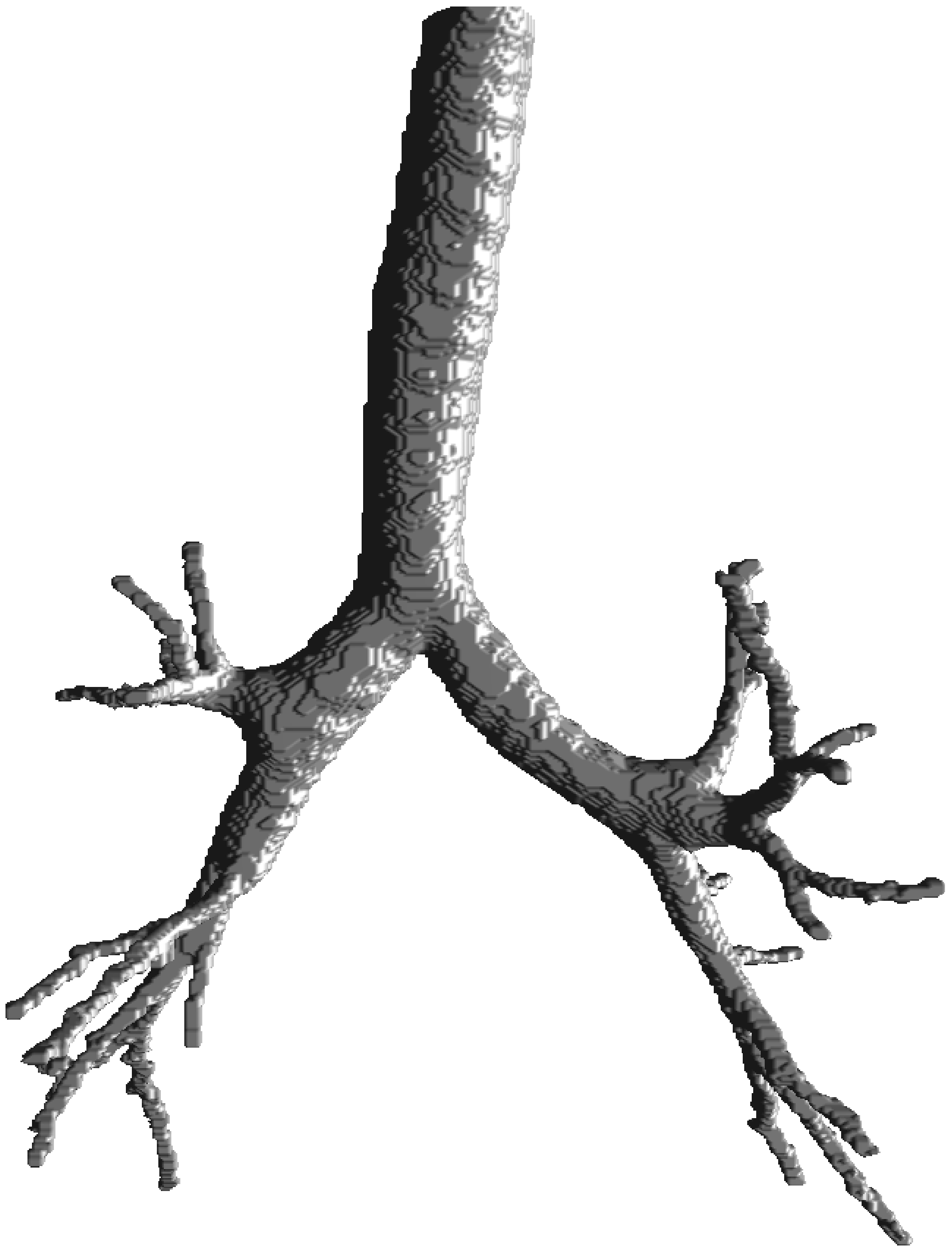


Figure 4.3: Patient 1: Segmentation

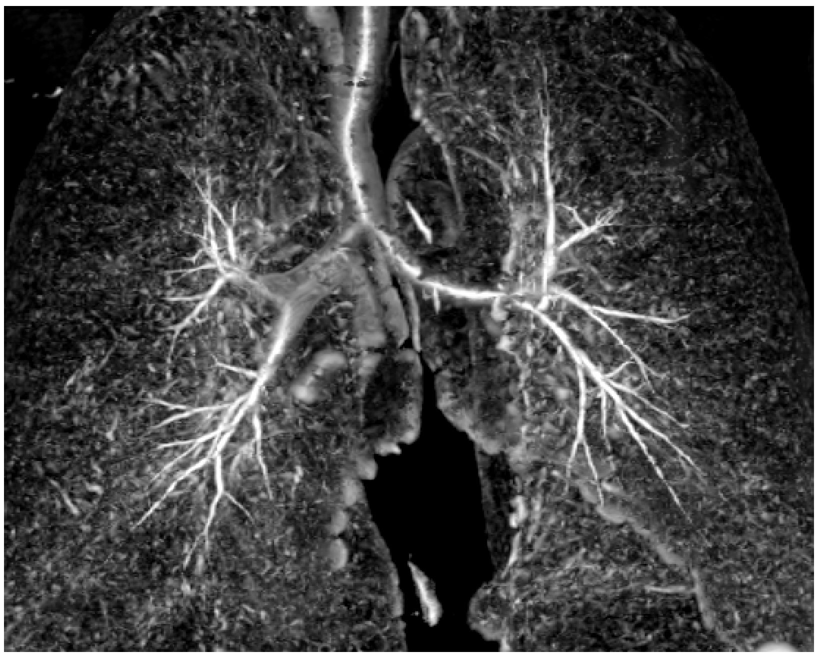


Figure 4.4: Patient 2: Tubular Detection Filter

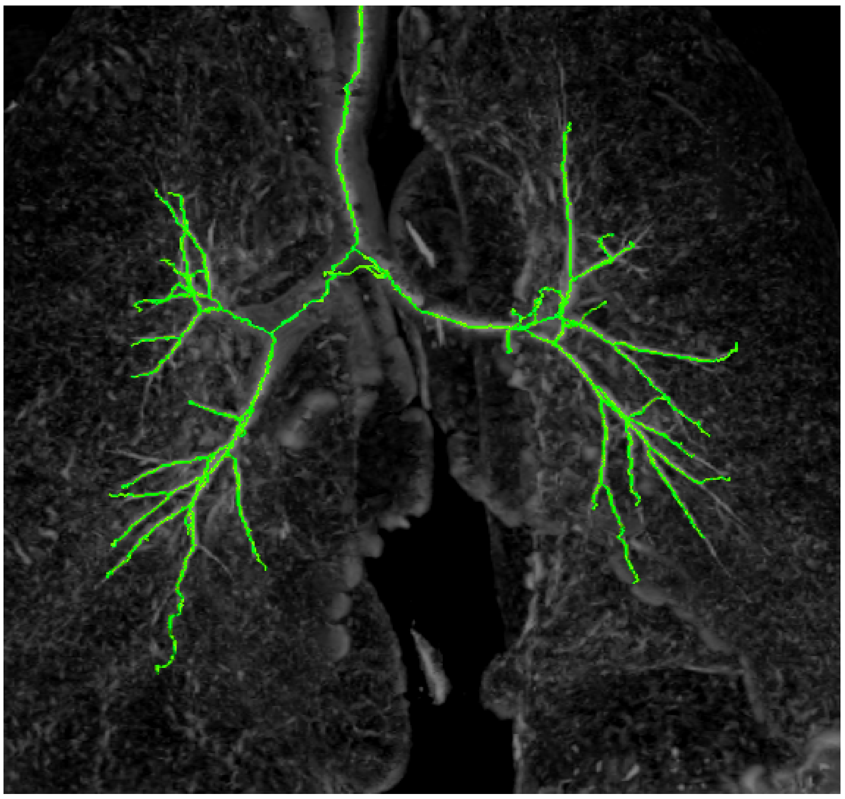


Figure 4.5: Patient 2: Centerlines

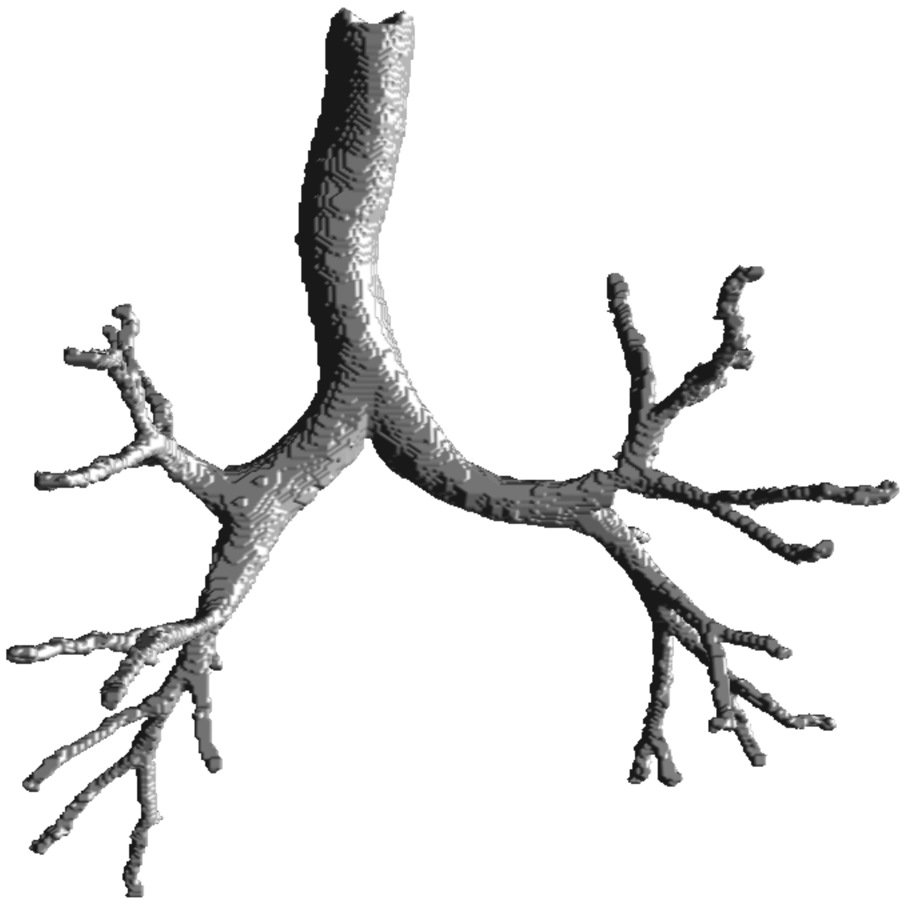


Figure 4.6: Patient 2: Segmentation

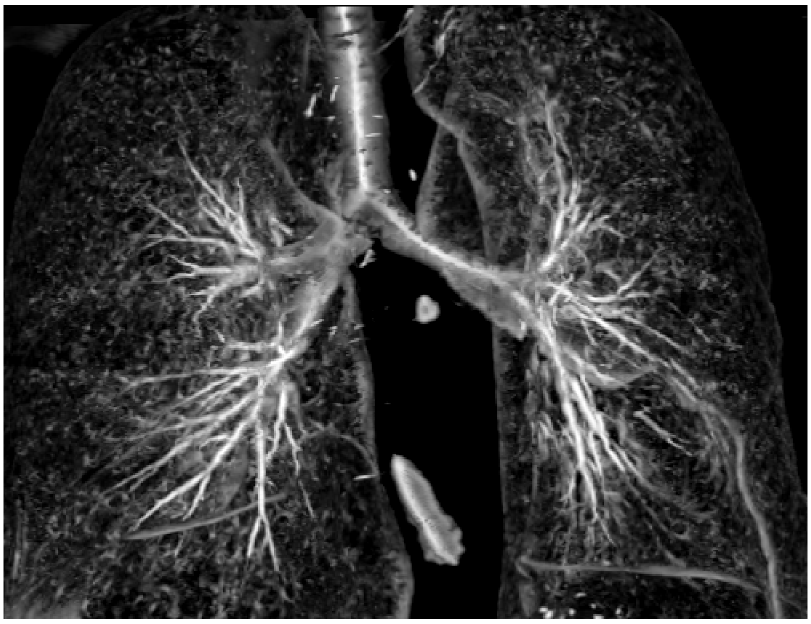


Figure 4.7: Patient 3: Tubular Detection Filter

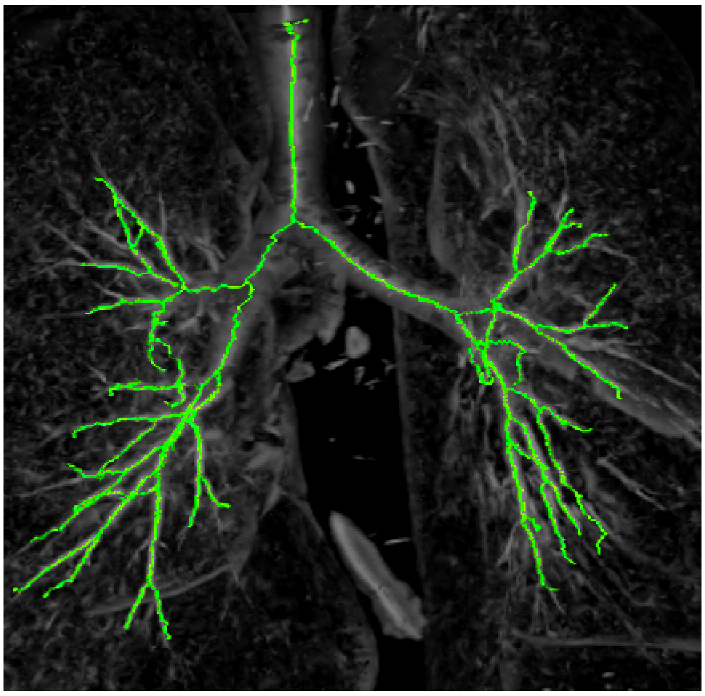


Figure 4.8: Patient 3: Centerlines

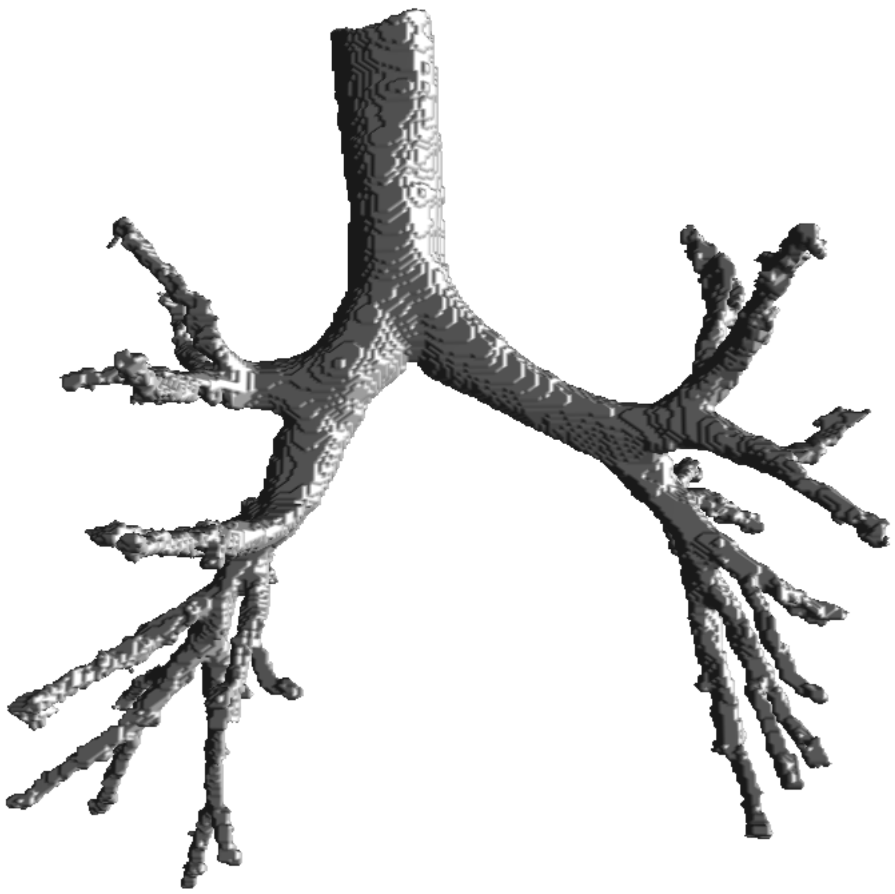


Figure 4.9: Patient 3: Segmentation

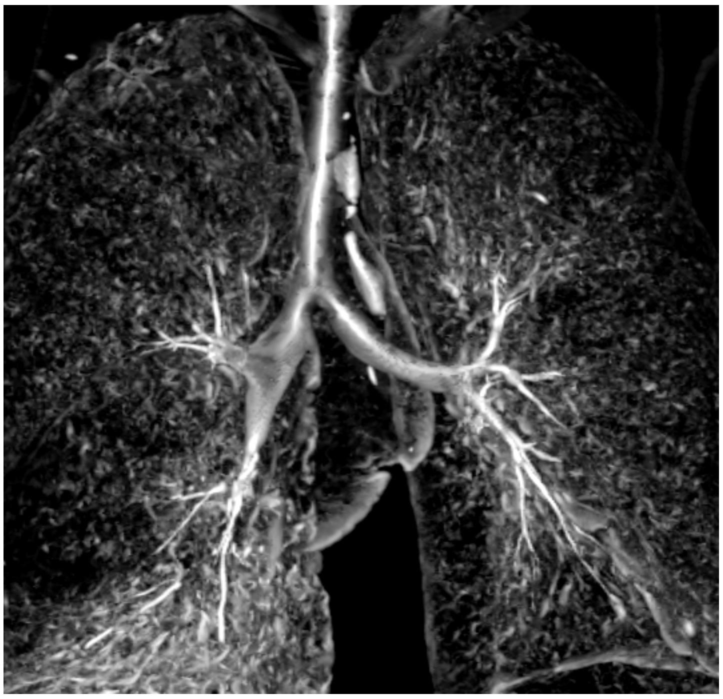


Figure 4.10: Patient 4: Tubular Detection Filter

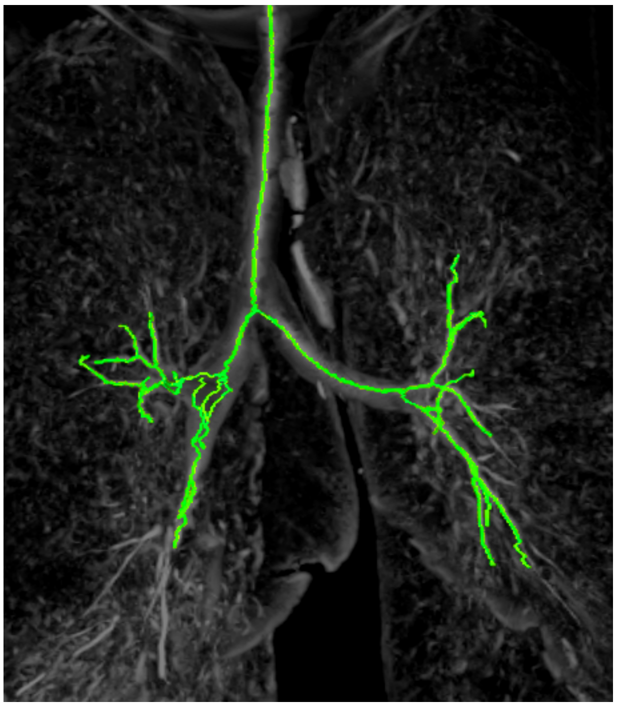


Figure 4.11: Patient 4: Centerlines

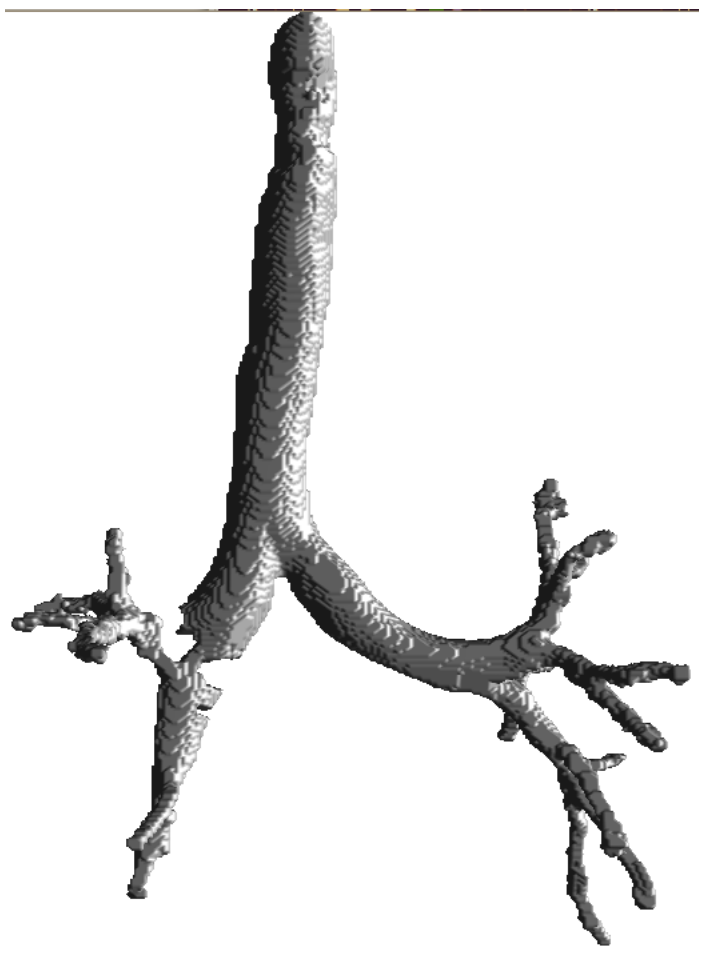


Figure 4.12: Patient 4: Segmentation

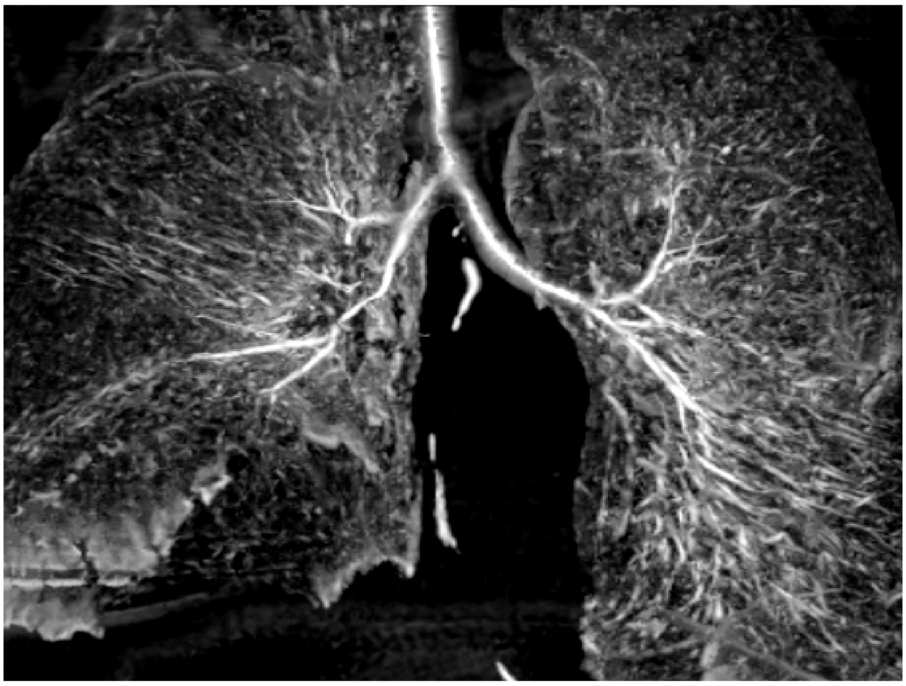


Figure 4.13: Patient 5: Tubular Detection Filter

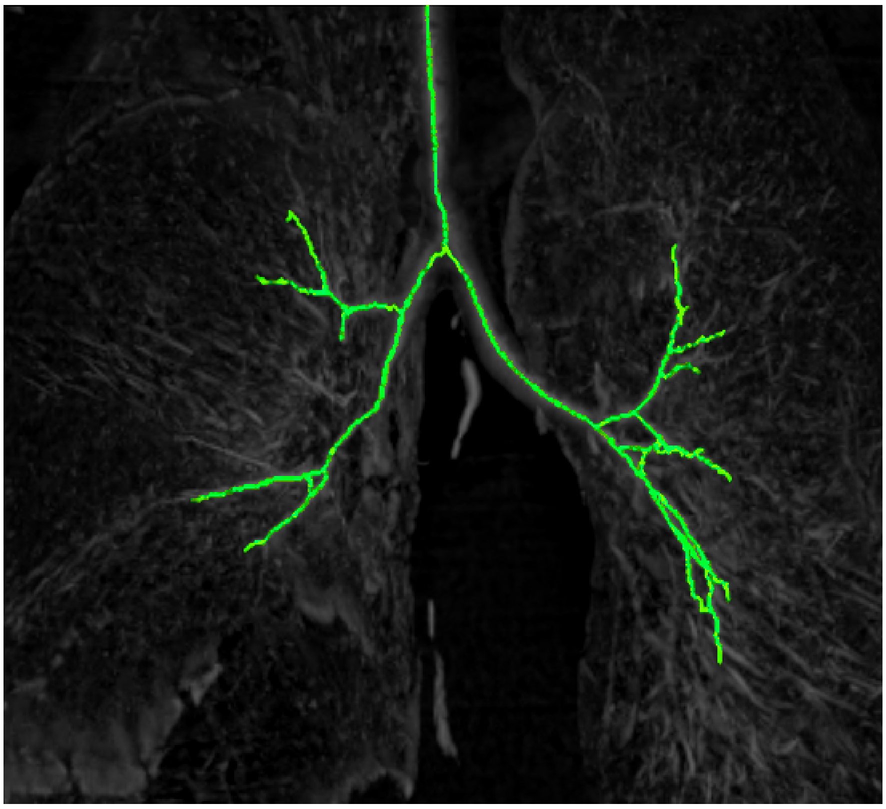


Figure 4.14: Patient 5: Centerlines

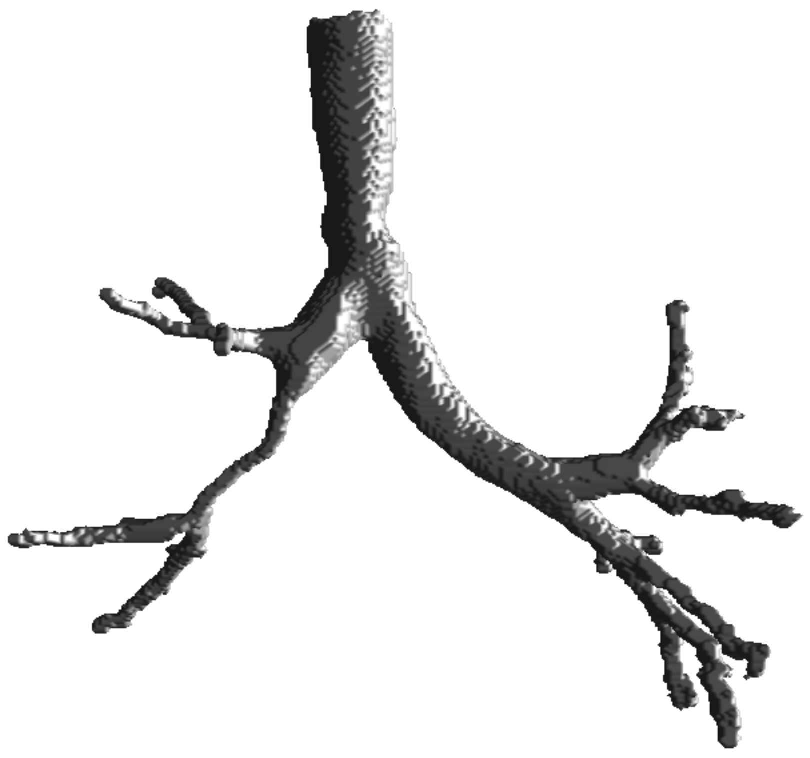


Figure 4.15: Patient 5: Segmentation

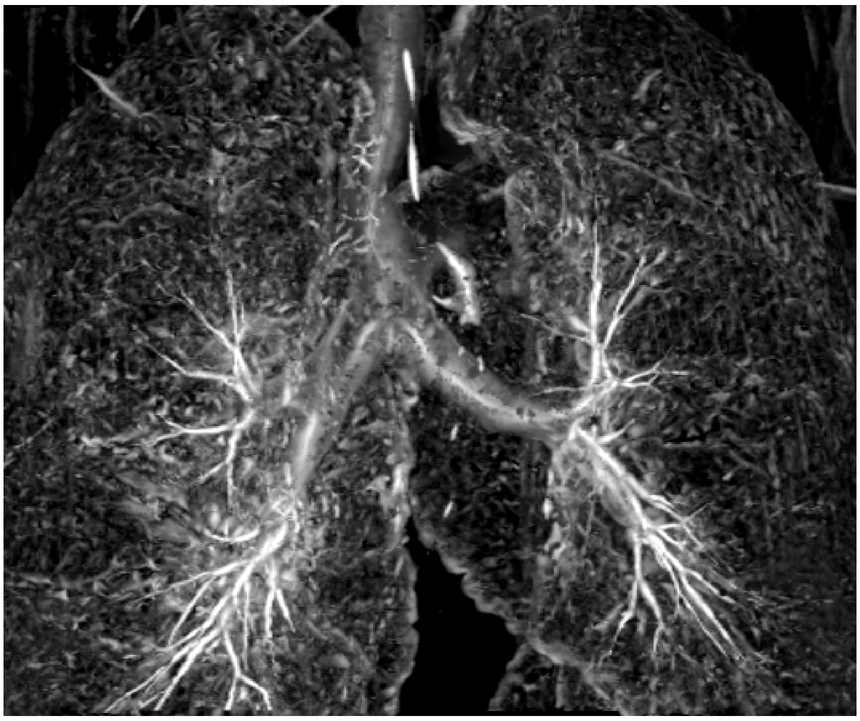


Figure 4.16: Patient 6: Tubular Detection Filter

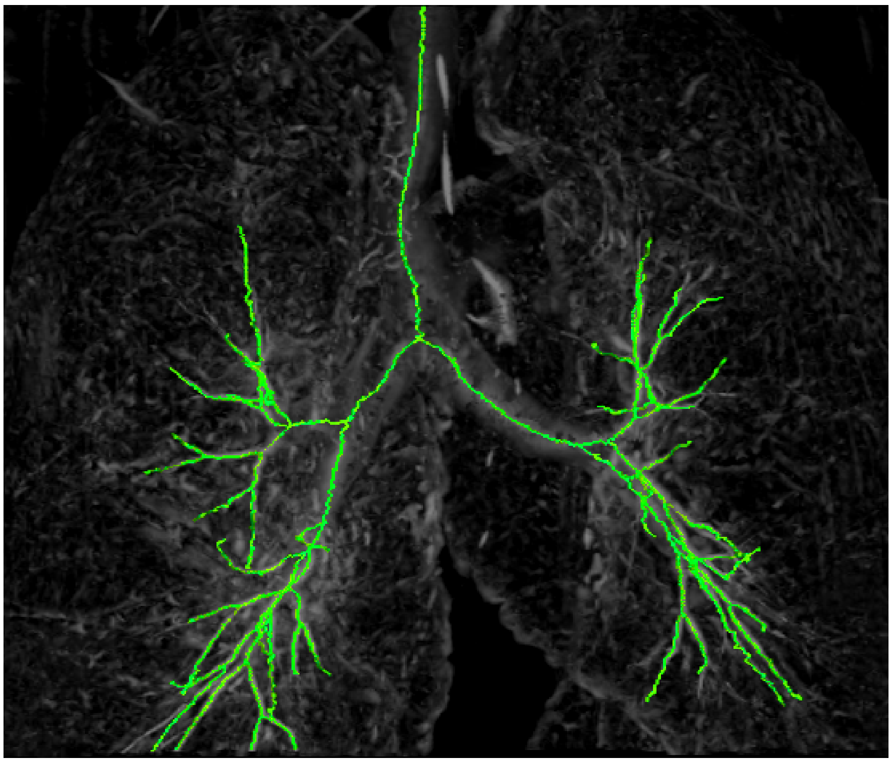


Figure 4.17: Patient 6: Centerlines

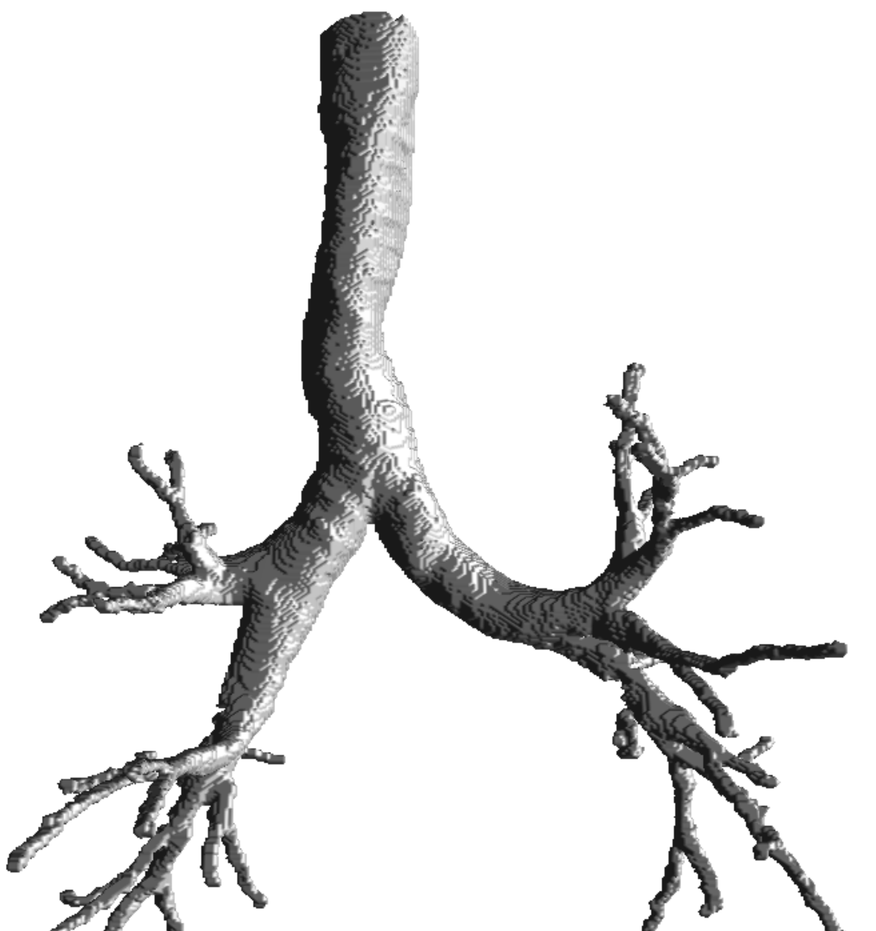


Figure 4.18: Patient 6: Segmentation

4.2 Blood Vessel Results

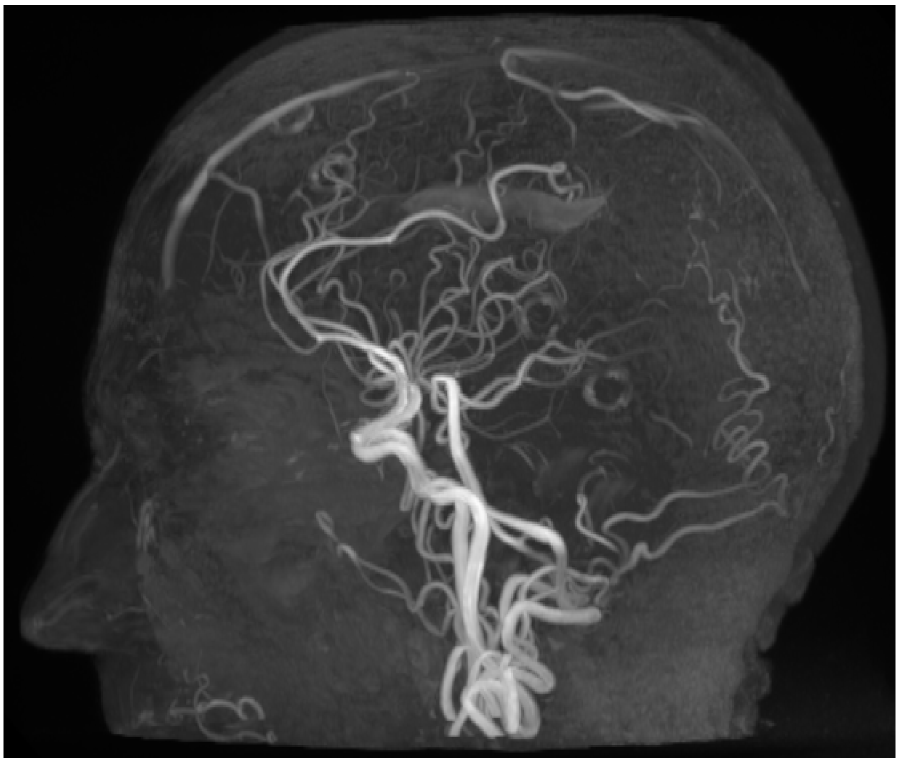


Figure 4.19: The MR Angio dataset depicted with MIP

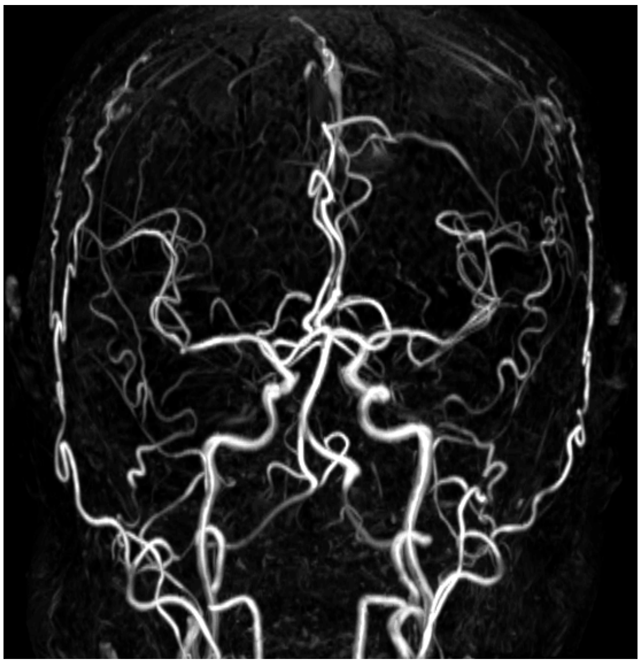


Figure 4.20: Tubular Detection Filter

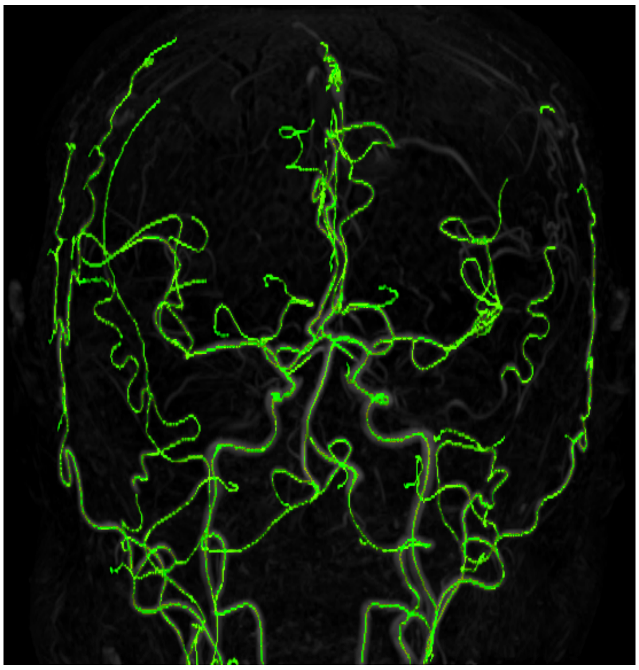


Figure 4.21: Centerlines

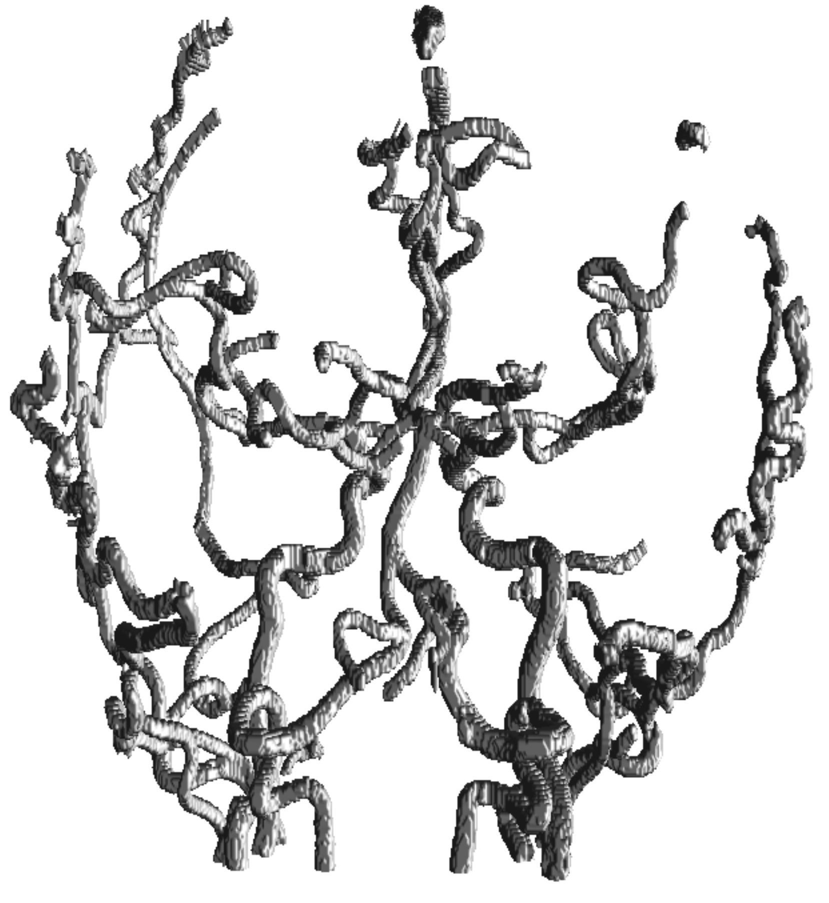


Figure 4.22: Segmentation

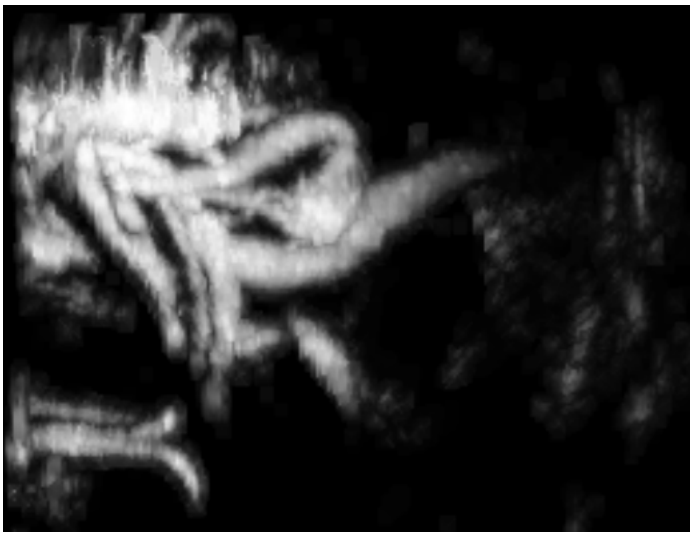


Figure 4.23: The Ultrasound Doppler dataset depicted with MIP

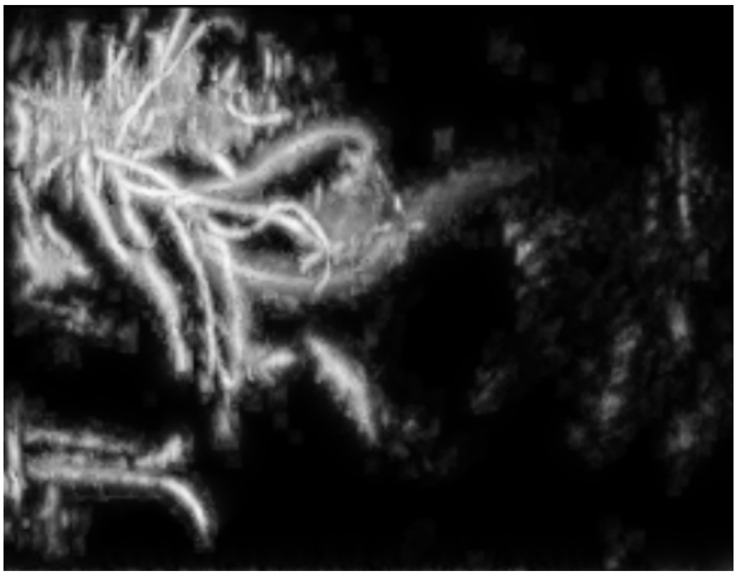


Figure 4.24: Tubular Detection Filter

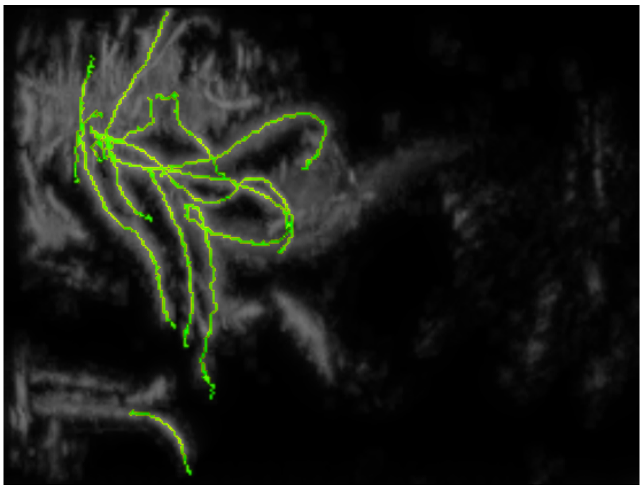


Figure 4.25: Centerlines

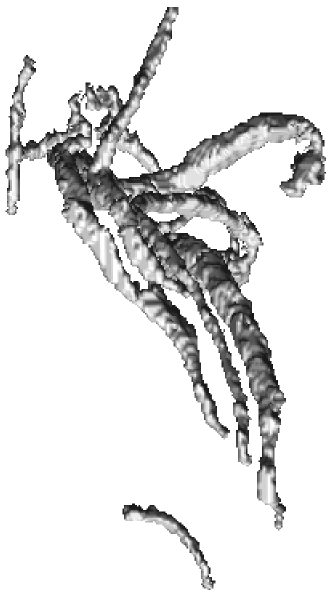


Figure 4.26: Segmentation

4.3 Speed

To analyze the speed of our implementation the six airway datasets were run on two different processors, one NVIDIA Tesla C2070 GPU and one Intel i7 720 CPU with 4 cores. For each dataset and processor we ran it 10 times and calculated the average runtime. Note that the runtime includes everything including loading the dataset from disk and storing all the results (centerline and segmentation) on disk. The results are summarized in table 4.1. The six airway datasets were run with the same parameters.

We also measured the runtime for each part of our implementation on the NVIDIA Tesla C2070 GPU. Figure 4.27 depicts the runtime in seconds of each step when performed on patient 1. The runtime for the different steps varies from patient to patient, but the calculation of the GVF is always the most time demanding step. Reading the dataset (I/O) uses much less time if it has been read recently, due to caching.

Dataset	**Size after cropping**	**GPU runtime**	**CPU runtime**
Patient 1	376x280x496	46 secs	12 min 52 secs
Patient 2	400x288x456	49 secs	14 min 43 secs
Patient 3	432x264x392	49 secs	10 min 44 secs
Patient 4	392x256x472	45 secs	14 min 4 secs
Patient 5	376x264x360	33 secs	10 min 5 secs
Patient 6	448x312x424	60 secs	17 min 25 secs

Table 4.1: Speed measurements

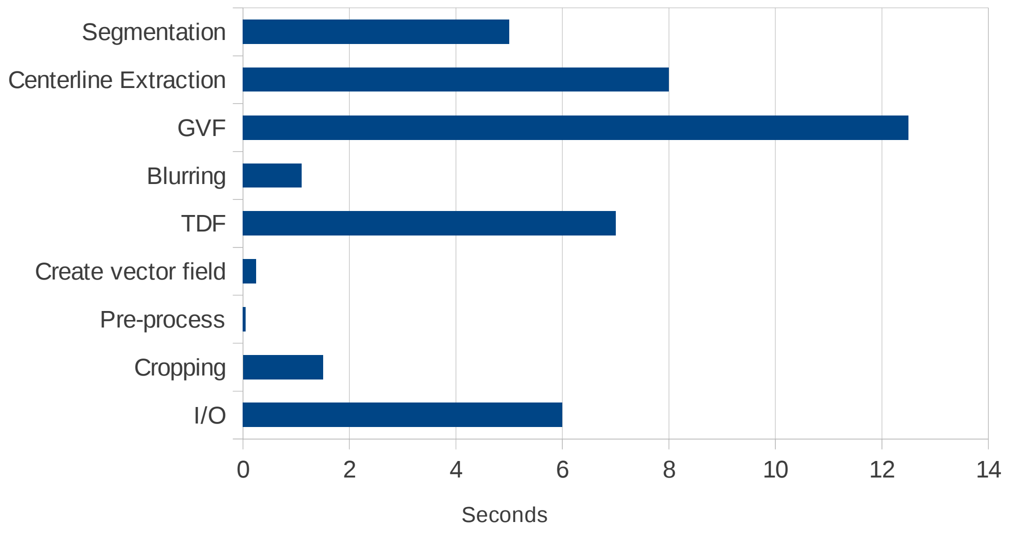


Figure 4.27: Runtime for the different steps of the implementation in seconds when run on Pasient 1. GVF is run with 250 iterations and R_{max} is 15.

Chapter 5

Discussion

In this chapter, the performance and speed of the TDF, centerline extraction and segmentation will be discussed based on the results presented in the previous chapter.

5.1 Tube Detection Filter

The Circle Fitting TDF that was implemented has the ability to detect large parts of the airway tree from the largest part, *trachea* to the smaller *bronchioles*. But the TDF also gives a lot of false responses inside the entire lung. Still, these responses have generally a lower value than correctly identified airways. The main challenge with this TDF is that it has problems with detecting tubes that deviates from a perfect circular cross-section. This is very clear in branch points, especially in the larger branches such as the one in figure 5.1. This drop in the TDF response creates problems for the centerline extraction method because the ridge that it is supposed to traverse disappears.

5.2 Extracted Centerlines

The centerline extraction method that uses ridge traversal is probably the weakest method in this implementation. The ridge traversal method has large problems dealing with noise and local artifacts. And this is due to the local greedy nature of the ridge traversal algorithm. Branches that are not

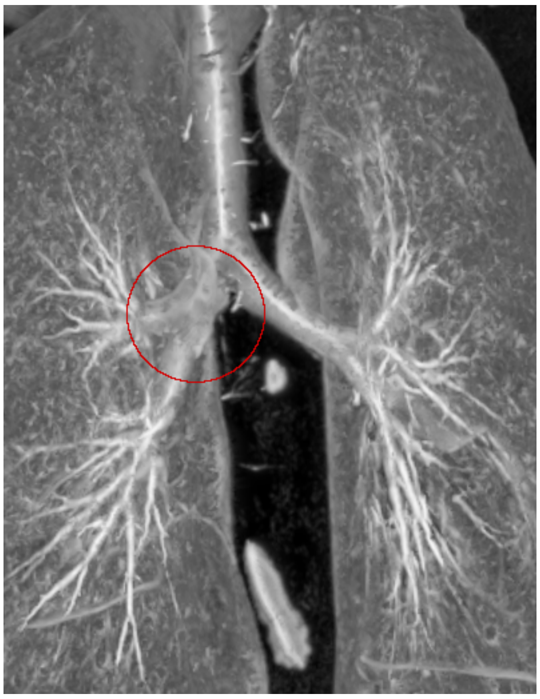


Figure 5.1: TDF response created with the Circle Fitting method. Note the low TDF response in the primary left bronchi inside the red circle. This low TDF response is because this branch has a shape that deviates a lot from a circle.

detected properly by the TDF thus presents a big challenge for this method and may lead to gaps and lines that are not in the center of the airway. Also, small branches at the end of the detected tree from the TDF are often discarded. This is because very short centerlines must be discarded due to noise. Patient 4 and 5 in figures 4.10, 4.11, 4.13 and 4.14 clearly show several centerlines that are not detected on the lower left side. In the TDF we can see that this is due to gaps in the TDF of the airways.

5.3 Segmentation Results

The quality of the segmentation is very dependent on the extracted centerlines. If a centerline on the smaller airways go outside of the actual airway the segmentation may create a small leakage. Also when the centerline is not exactly in the center of the airway the growing procedure may not be able to segment the entire airway in the cross-sectional at that point. Most of these

”holes” in the segmentation are fixed using morphological closing, but if the gap is to big it might not be able to fix it. Such a gap is present on the left main *bronchi* in the segmentation of patient 4 in figure 4.12.

5.4 Speed

Our implementation uses about 45 to 60 seconds on a full CT scan when run on a NVIDIA Tesla GPU. This is a major improvement from the 3-6 minutes reported by Bauer et al. [6] that only used a GPU for the GVF calculations. Though less than a minute is fast, an even faster implementation would be preferable. We also ran our implementation on a multi-core CPU which clearly shows that this application benefits a lot from the GPUs data parallel processing power. The OpenCL/C++ implementation is an enormous improvement over the serial Matlab implementation that was made in the early stages of this thesis. The runtime of the Matlab implementation was many hours.

Runtime analysis of each step of our implementation showed that the Gradient Vector Flow calculation was the most expensive step and was very dependent on the dataset size and number of iteration. The runtime of the segmentation and centerline extraction steps are highly dependent on how large the detected airway tree is, but generally they and the TDF calculation are the three most expensive steps after the GVF. The I/O part is dependent on whether the dataset has been read recently. Reading is much faster if the dataset was recently read, due to caching.

We were not able to exploit the GPU’s texture system in the Gradient Vector Flow computation because NVIDIA’s GPUs doesn’t support writing to a 3D texture. AMD GPUs, on the other hand, support writing to 3D textures and may thus be able to run the implementation even faster. Our previous work [27] showed that AMD GPUs could calculate the 3D GVF several times faster than NVIDIA GPUs. Unfortunately we did not have an AMD GPU with enough memory to test this.

Chapter 6

Conclusions

6.1 Goal achievement

The purpose of this project was to create a system for Airway Tree Segmentation and Centerline Extraction that exploits the computational power of modern graphic processing units (GPUs) to speed up the processing of the large CT scans. We did a wide background study to identify the best methods for achieving this goal. We also investigated how GPUs could be used to speed up these methods. We chose the most promising method and implemented it using OpenCL and C++. We have shown that our implementation uses less than a minute on GPUs and is able to extract large parts of the Airway Tree from CT scans. We also showed that our implementation is very general and can extract other tubular structures from other imaging modalities, such as blood vessels from a MR Angio scan and Ultrasound Doppler image of the brain.

6.2 Future work

In the previous chapter we identified several problems with the implementation. The TDFs problems of identifying tubes with non-circular cross-section and the centerline extractions lack of robustness to noise were two of the main problems identified in the previous chapter. In this section we present the identified problems and suggest how they can be improved.

6.2.1 Pre-processing

In our implementation the dataset is first smoothed by a Gaussian filter. Gaussian smoothing has the ability to reduce the effect of noise in the dataset, but it also destroys important edge information in the image. For small low-contrast airways the Gaussian smoothing can have the effect of reducing the contrast even further or eliminate the airway entirely.

A possible solution to this problem can be to replace the Gaussian smoothing with some sort of anisotropic smoothing instead as suggested by Bauer in his thesis [3]. An anisotropic smoothing filter will vary the smoothing for different directions. Thus an anisotropic smoothing filter can smooth more in the direction of the tube where there should be none or very small intensity change and smooth less or nothing in the cross-sectional plane of the airway where there should be higher intensity change. Krissian [18] suggested a smoothing filter that does exactly this.

6.2.2 Tubular Detection Filter

The main problem of the Tubular Detection Filter used in our implementation is that it creates a lot of false positives and gives a lower response to airways that don't have circular cross-section. This is quite evident around each branch point and especially in the left primary bronchi as shown in figure 5.1. We suggest to remove the assumption from the Circle Fitting method of Krissian et al. [17] that the cross-sectional profile is circular. A more appropriate assumption is that the profile of the airway is a closed smooth bright border with black inside. Such as profile can be modeled with a spline and its control points found using line searches in different directions in the cross-sectional plane as shown in figure 6.1.

6.2.3 Centerline extraction

Inaccuracies in the eigenanalysis of the Hessian, Gradient Vector Flow and TDF create problems for the centerline extraction method. If any part of the airway has a very low TDF response, the centerline extraction can easily just stop there. Also if the airway directions that are based on the eigenvectors are very inaccurate the extraction can stray of from the airway. Generally speaking, centerline extraction by ridge traversal is a greedy local search algorithm that is not very robust. Thus we believe that this method is not

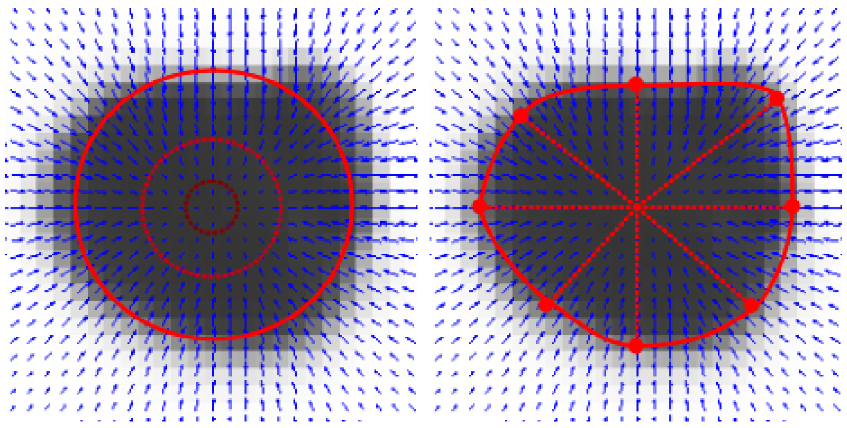


Figure 6.1: To the left the Circle Fitting model and to the right the suggested spline model with a set of control points in the cross-sectional plane.

well suited for this problem. The only real advantage of this method is that it is fast. One very interesting alternative is tree reconstruction with Ant Colony Optimization (ACO). ACO is a type of evolutionary algorithm for graphs. Two recent articles (2011) by Türetken and González et al. ([29], [12]) shows very promising results in extracting centerlines from tree structures like neural pathways and retinal blood vessels. We believe that because this method tries to find the global optimal tree in the dataset instead of finding a local greedy tree it will work better than ridge traversal. Still, this method is very complex and not very fast, but may benefit a lot from parallel processing.

6.2.4 Segmentation

We explained earlier how inaccurate centerlines can create small segmentation leaks and gaps when using the inverse gradient flow tracking method. A valid segmentation can be created by collecting all the splines of the proposed TDF in section 6.2.2 from the extracted centerline points and displaying a surface from these. This is possible because the spline TDF can model the tube's cross-sections more accurately than the circle fitting TDF we used in this thesis. This should also be faster as it avoids the processing necessary for the growing procedure. Graham et al. [13] developed a method where they established links between many candidate airway points and then grew, using regular region growing, the segmentation inside the surface defined by the best-fit circles at the two points. Such a method avoids segmentation leakage and can be run in parallel on a GPU.

6.2.5 Parallelization and Optimization

The calculation of the Gradient Vector Flow was the most expensive step of our implementation. Textures were not used in this step, since NVIDIA GPUs doesn't support writing to 3D textures. A solution to this problem can be to pack the 3D texture in a 2D texture. This might improve speed due to caching of the textures, but address calculations has to be done to map the 3D coordinates to 2D. Also, after each kernel the result of the kernel has to be copied into a new 3D texture. This can be avoided using the packed 3D to 2D texture method.

The two serial steps of our implementation: centerline extraction and inverse gradient flow tracking segmentation limits the speed of our implementation. They might be improved by transforming them into data parallel algorithms that can benefit from the computationally power of the GPUs. Doing all the calculations on the GPU also brings the benefit of avoiding to transfer the large amounts of data back to the main memory of the CPU.

GVF is necessary for the larger airways, but the large airways have a lot of contrast and is very easy to extract with other methods such as a conservative region growing with explosion control. Such a region growing can be faster even when run serially. But without GVF ridge traversal for centerline extraction becomes difficult. A hybrid solution where the large airways are extracted by a simple region growing and the smaller airways with Hessian-based techniques was used by Graham et al. [13] with great success.

References

[1] D. Aykac, E. a. Hoffman, G. McLennan, and J. M. Reinhardt. Segmentation and analysis of the human airway tree from three-dimensional X-ray CT images. *IEEE transactions on medical imaging*, 22(8):940–50, Aug. 2003.

[2] S. R. Aylward and E. Bullitt. Initialization, noise, singularities, and scale in height ridge traversal for tubular object centerline extraction. *IEEE transactions on medical imaging*, 21(2):61–75, Feb. 2002.

[3] C. Bauer. *Segmentation of 3D Tubular Tree Structures in Medical Images.* PhD thesis, Graz University of Technology, 2010.

[4] C. Bauer and H. Bischof. A novel approach for detection of tubular objects and its application to medical image analysis. *Pattern Recognition*, pages 163–172, 2008.

[5] C. Bauer and H. Bischof. Extracting curve skeletons from gray value images for virtual endoscopy. *Medical Imaging and Augmented Reality*, pages 393–402, 2008.

[6] C. Bauer, H. Bischof, and R. Beichel. Segmentation of airways based on gradient vector flow. In *International Workshop on Pulmonary Image Analysis, Medical Image Computing and Computer Assisted Intervention*, pages 191–201. Citeseer, 2009.

[7] C. Bauer, T. Pock, H. Bischof, and R. Beichel. Airway tree reconstruction based on tube detection. In *Proceedings of the 2nd International Workshop on Pulmonary Image Analysis*, pages 203–214. Citeseer, 2009.

[8] J. Breitbart and C. Fohry. OpenCL - An effective programming model for data parallel computations at the Cell Broadband Engine. *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, Apr. 2010.

[9] A. R. Brodtkorb, C. Dyken, T. R. Hagen, and J. M. Hjelmervik. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18:1–33, 2010.

[10] Cancer Registry of Norway. *Cancer in Norway 2009 - Cancer indicdence, mortality, survival and prevalence in Norway.* Cancer Registry of Norway, Oslo, 2011.

[11] A. Frangi, W. Niessen, K. Vincken, and M. Viergever. Multiscale vessel enhancement filtering. *Medical Image Computing and Computer-Assisted Interventation—MICCAI'98*, 1496:130–137, 1998.

[12] G. González, E. Turetken, F. Fleuret, and P. Fua. Delineating trees in noisy 2D images and 3D image-stacks. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2799–2806. IEEE, 2010.

[13] M. W. Graham, J. D. Gibbs, and W. E. Higgins. Robust system for human airway-tree segmentation. *Proceedings of SPIE*, 6914:69141J–69141J–18, 2008.

[14] M. Hassouna and A. Farag. On the extraction of curve skeletons using gradient vector flow. *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8, 2007.

[15] B. Irving, P. Taylor, and A. Todd-pokropek. 3D segmentation of the airway tree using a morphology based method. In *Second International Workshop on Pulmonary Image Analysis*, pages 297–307, 2009.

[16] Khronos. OpenCL Official Website, 2011.

[17] K. Krissian. Model-Based Detection of Tubular Structures in 3D Images. *Computer Vision and Image Understanding*, 80(2):130–171, Nov. 2000.

[18] K. Krissian. Flux-Based Anisotropic Diffusion Applied to Enhancement of 3-D Angiogram. *IEEE TRANSACTIONS ON MEDICAL IMAGING*, 21(11):1440–1442, Jan. 2002.

[19] D. Lesage, E. D. Angelini, I. Bloch, and G. Funka-Lea. A review of 3D vessel lumen segmentation techniques: models, features and extraction schemes. *Medical image analysis*, 13(6):819–45, Dec. 2009.

[20] T. a. Lewis, Y.-S. Tzeng, E. L. McKinstry, A. C. Tooker, K. Hong, Y. Sun, J. Mansour, Z. Handler, and M. S. Albert. Quantification of airway diameters and 3D airway tree rendering from dynamic hyperpolarized 3He magnetic resonance imaging. *Magnetic resonance in medicine*

: official journal of the Society of Magnetic Resonance in Medicine / Society of Magnetic Resonance in Medicine, 53(2):474–8, Feb. 2005.

[21] P. Lo, B. V. Ginneken, J. M. Reinhardt, and M. de Bruijne. Extraction of Airways from CT (EXACT ' 09). In *Second International Workshop on Pulmonary Image Analysis*, pages 175–189, 2009.

[22] P. Lo, J. Sporring, and M. D. Bruijne. Multiscale Vessel-guided Airway Tree Segmentation. In *Second International Workshop on Pulmonary Image Analysis*, pages 323–332, 2009.

[23] W. Lorensen and H. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, volume 21, pages 163–169. ACM, 1987.

[24] M. D. McCool. Scalable Programming Models for Massively Multicore Processors. *Proceedings of the IEEE*, 96(5):816–831, May 2008.

[25] K. Mori, J. Hasegawa, and J. Toriwaki. Recognition of bronchus in three-dimensional X-ray CT images with applications to virtualized bronchoscopy system. *Pattern Recognition,*, pages 528–532, 1996.

[26] I. Sluimer, A. Schilham, M. Prokop, and B. van Ginneken. Computer analysis of computed tomography scans of the lung: a survey. *IEEE transactions on medical imaging*, 25(4):385–405, Apr. 2006.

[27] E. Smistad, A. C. Elster, and F. Lindseth. Real-Time Gradient Vector Flow on the GPU using OpenCL. 2012. Manuscript submitted for publication.

[28] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, and S. Miki. *The OpenCL Programming Book*. Fixstars, 2010.

[29] E. Türetken, G. González, C. Blum, and P. Fua. Automated reconstruction of dendritic and axonal trees by global optimization with geometric priors. *Neuroinformatics*, 9(2-3):279–302, Sept. 2011.

[30] B. van Ginneken, W. Baggerman, and E. M. van Rikxoort. Robust segmentation and anatomical labeling of the airway tree from thoracic CT scans. *Medical image computing and computer-assisted intervention : MICCAI ... International Conference on Medical Image Computing and Computer-Assisted Intervention*, 11(Pt 1):219–26, Jan. 2008.

[31] C. Xu and J. Prince. Snakes, shapes, and gradient vector flow. *Image Processing, IEEE Transactions on*, 7(3):359–369, 1998.