GPU-Based Airway Segmentation and Centerline Extraction for Image Guided Bronchoscopy

Erik Smistad¹, Anne C. Elster¹, Frank Lindseth¹,²

¹) Norwegian University of Science and Technology
²) SINTEF Medical Technology. Trondheim, Norway

{smistad,elster,frankl}@idi.ntnu.no

Abstract

Bronchoscopy is an important minimal-invasive procedure for both diagnosis and therapy of several lung disorders, including lung cancer. However, narrow airways and complex branching structure increases the difficulty of navigating to the target site inside the lungs. It is possible to improve navigation by extracting a map of the airways from CT images and tracking the tip of the bronchoscope. Most of the methods for extracting such maps are computationally expensive and have a long runtime. In this paper, we present an implementation of airway segmentation and centerline extraction, which utilizes the computational power of modern graphic processing units. We also present a novel parallel cropping algorithm which discards over 70% of the dataset as non-lung tissue, thus significantly reducing memory usage and processing time.

1 Introduction

Lung cancer is one the most common type of cancer in Norway and has one of the highest mortality rates [5]. Early and precise diagnosis is crucial for improving the mortality rate. Bronchoscopy is an important minimal-invasive procedure for both diagnosis and therapy of several lung disorders, including lung cancer. Currently, at St. Olav’s University Hospital in Trondheim, Norway, diagnosis is done by extracting a tissue sample of the tumor using a bronchoscope. The bronchoscope is a flexible tube with a camera and light source that enables the physician to see inside the lungs. It is inserted through the mouth and the airways of the lungs. Tissue samples can then be extracted through a shaft in the bronchoscope. The airways of the lungs is a complex tree structure, where each branch is smaller than the previous. After several divisions, the airways becomes very small. The small airways and complex branching structure increases the difficulty of navigating to the target site inside the lung. Thus, one of the major challenges with bronchoscopy is to actually find the tumor inside the lungs.

Together with SINTEF Medical Technology and St. Olav’s University Hospital, our main goal is to increase the success rate of bronchoscopy procedures by using images and electromagnetic tracking of the bronchoscope. By registering a map of the airways to the patient, the surgeon is able to see the location of the bronchoscope on the map, and use this to navigate. The map is automatically extracted from Computer Tomography (CT) images of the lungs and consists of two things: A segmentation and a centerline.

This paper was presented at the NIK-2012 conference; see http://www.nik.no/.
The segmentation is a classification of each voxel in the CT volume which determines whether the voxel is part of the airways or not. The centerline is a line that goes through the center of each branch of the airways and is used to register the CT data to the patient. The registration is done by matching the centerline with the positions of the tip of the bronchoscope. Deguchi et al. [6] performed a study on performing such a navigated bronchoscopy using phantom airways and achieved an accuracy of 2.0 - 3.5 mm.

Several methods for extracting the airway tree exist in the literature. Two notable reviews on airway tree segmentation and centerline extraction from CT images can be found in the works by Sluimer et al. [12] and a newer one by Lo et al. [11]. A larger and more general review on vessel segmentation was done by Lesage et al. [10]. Most of these methods are very computationally expensive and require a long runtime. Many of the methods, also require several runs with different parameters before satisfactory results are achieved. The CT image is acquired right before the procedure or the day before. In either case, the image is processed right before the bronchoscopy. To reduce waiting time, it is essential that the airway extraction and registration goes as quickly as possible. Also, the method presented in this paper is applicable for extracting blood vessels from intraoperative 3D ultrasound. In this application, time is even more crucial.

Several image processing techniques are data parallel because each pixel can often be processed in parallel using the same instructions. Graphic Processing Units (GPUs) allow many pixels to be processed in the same clock cycle enabling massive speedups.

In this paper, we present a GPU-based implementation of the airway segmentation method introduced by Bauer et al. [4], [2]. Their method showed very promising results in the Extraction of Airways from CT challenge in 2009 (EXACT’09) [11]. We also present a novel data parallel algorithm for cropping the large CT datasets. The CT images are often very large and include a lot of voxels which are not part of the lungs, such as background and body fat. Our cropping algorithm automatically crops the CT volume, thus reducing the amount of irrelevant voxels. With this cropping algorithm, processing time and memory usage can be reduced significantly.

2 Methodology

In this section, we present our implementation of the airway segmentation and centerline extraction methods of Bauer et al. [4], [2]. The implementation was created using C++ and the Open Computing Language (OpenCL). OpenCL is a new framework for writing parallel programs for heterogeneous systems. This framework allows execution of parallel code directly on the GPU and CPU concurrently. Figure 1 depicts the main steps of our implementation. First, the dataset is cropped. Then the dataset is pre-processed with Gaussian smoothing, Gradient Vector Flow (GVF) and vector normalization to make the following Tube Detection Filter (TDF) invariant to scale and contrast of the airways. The TDF use the Hessian matrix to detect tubular structures in the dataset. Centerlines
Figure 2: **Left:** A typical CT image of the lungs. The border indicates the minimal rectangular area that includes the lungs. **Right:** Thresholded CT image and two scan lines in the x and y directions. For each scan line the number of detected white and black regions are listed.

are extracted from the TDF result using a ridge traversal approach and finally, the airways are segmented from the centerlines.

**Cropping**

A typical CT image of the thorax will contain a lot of data which is not part of the lungs, such as space outside the body, body fat and the bench which the patient is resting on. Figure 2 shows a typical CT image of the thorax. The rectangle is the smallest rectangular area in which the lungs are contained. For this slice, more than half of the image is not part of the lungs and thus not relevant for further processing. As several of the methods used to perform segmentation and centerline extraction of the airways will process each voxel in the entire volume, removing this unnecessary data not only reduce memory usage, but also execution time.

A common way to remove the unwanted data, is to perform a lung segmentation first and then crop the data to the segmentation. Such methods usually needs one or two seeds to be set manually and can be time consuming.

In this paper, we introduce a novel cropping algorithm that do not need a lung segmentation. The algorithm is data parallel and very efficient on GPUs. The pseudocode for the cropping procedure is shown in Algorithm 1 below. The cropping algorithm works by scanning slices in all three directions: x, y and z. And for each slice, the method determines how many scan lines went through the lungs. The number of scan lines that went through the lungs, $L_s$, is recorded for each slice in the function $\text{CALCULATE}\_L$. A pixel on the scan line is categorized as black or white based on the intensity in the CT volume using a threshold $T_{\text{HU}} = -150\,\text{HU}$. A scan line is considered to have intersected the lungs if the number of detected black regions $B_d$ and white regions $W_d$ are both above 1. A white or black region is detected if the number of consecutive black or white pixels has reached a threshold $T_r$. Figure 2 shows a CT image thresholded using $T_{\text{HU}}$, two scan lines in the x and y directions and the number of detected black and white regions. If $L_s$ is above a threshold called $L_{\text{min}}$, we know that slice $s$ has to be part of the dataset. This
threshold is necessary due to noise in the dataset. The function \texttt{FINDCROPBORDERS} locates the cropping borders \((c_1, c_2)\) in a specific direction. The dataset is assumed to be oriented so that the patient’s back is parallel with the z direction. For directions \(x\) and \(y\), we look for the first and last slice with the minimum required scan lines \((L_{\min})\) inside the lung. These two slices determine the borders of the cropping. For the z direction, we start at the center of the dataset and locate the first slices below the threshold \(L_{\min}\).

Each direction and slice can be processed in parallel using the same instructions in the \texttt{CALCULATEL} function. This creates many threads and is ideal for GPU execution. The \texttt{FINDCROPBORDERS} function is run serially on the CPU. Using an NVIDIA Tesla GPU, this algorithm uses only 1-2 seconds on regular CT datasets. The parameters \(L_{\min} = 128\) and \(T_c = 30\) were chosen through experimentation.

\begin{algorithm}
\begin{algorithmic}
\Function{CROP}{volume}
\State \(L \leftarrow \text{CALCULATEL}(volume, x)\)
\State \(x_1, x_2 \leftarrow \text{FINDCROPBORDERS}(L, x)\)
\State \(L \leftarrow \text{CALCULATEL}(volume, y)\)
\State \(y_1, y_2 \leftarrow \text{FINDCROPBORDERS}(L, y)\)
\State \(L \leftarrow \text{CALCULATEL}(volume, z)\)
\State \(z_1, z_2 \leftarrow \text{FINDCROPBORDERS}(L, z)\)
\State \text{crop volume according to } x_1, x_2, y_1, y_2, z_1 \text{ and } z_2
\Return volume
\EndFunction

\Function{CALCULATEL}{volume, direction}
\State \(L_v \leftarrow 0\)
\For{each slice \(s\) in direction}
\State \(L_v \leftarrow \text{WINDING}(L_v, s)\)
\EndFor
\For{each scan line \(d\) in \(L_v\)}
\State \(B_{d} \leftarrow B_{d} + 1\)
\EndFor
\Return \(L_v\)
\EndFunction
\end{algorithmic}
\end{algorithm}

\textbf{Pre-processing and Gradient Vector Flow}

Before the TDF can be calculated, some pre-processing is necessary. First the dataset is blurred using Gaussian smoothing. Smoothing is done by convolution of the dataset with a small Gaussian kernel of scale/standard deviation \(\sigma = 0.5\). After the smoothing, the gradient vector field \(\vec{V}\) is created and normalized. The normalization is done according to equation 1 and is necessary to ensure contrast invariance for the TDF. The parameter \(F_{\max}\) controls the normalization. All gradients with a length above this parameter will be set to unit length and the others will be scaled accordingly.

All of these pre-processing steps are completely data parallel and are implemented as separate kernels that process the entire dataset.

\[
\tilde{V}^a(\vec{v}) = \begin{cases} 
\frac{\vec{V}(v)}{|\vec{V}(v)|} & \text{if } |\vec{V}(\vec{v})| \geq F_{\max} \\
\tilde{V}(\vec{v}) & \text{else}
\end{cases} 
\]  

(1)
To be able to calculate the Hessian matrix at a certain voxel, the image gradients has to exist. In large tubular structures, such as trachea in the airways, the gradients will only exist at the edge and not in the center. Thus, to detect tubular structures that are larger than a few voxels, the gradient information has to be propagated from the edge to the center. There exists two main methods of doing this: The Gaussian scale space method, where the image is blurred using Gaussian smoothing at different scales. And the Gradient Vector Flow (GVF) method, in which the gradient vectors are diffused iteratively. Bauer and Bischof [3] were the first to point out that GVF could be used to create scale-invariance of TDFs and serve as an alternative to the Gaussian scale space method. The GVF method has the advantage that it is feature-preserving and avoid the problem of two or more tubular structures diffusing together to create the illusion of a larger, false tube. The disadvantage of this method is that it is very time consuming.

GVF was originally introduced by Xu and Prince [14] as a new external force field for active contours. The resulting gradient vector field $\vec{V}$ of GVF aims to minimize the energy function $E(\vec{V})$:

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

(2)

where $\vec{V}_0$ is the initial gradient vector field and $\mu$ a weighting constant of the two terms. Xu and Prince [14] developed a method for calculating the GVF by iteratively solving the following Euler equation for each vector component independently:

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

(3)

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

**Algorithm 2 3D Gradient Vector Flow**

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

With this iterative numerical scheme, each voxel can be processed in parallel using the same instructions. This makes the calculations ideal for data parallel executions on GPUs. He and Kuester [8] presented a GPU implementation of GVF and Active Contours using OpenGL Shading Language (GLSL). They reported that their GPU implementation was up to 4 times faster than a CPU implementation. Their implementation was for 2D images only and used the texture memory system to speed up data retrieval. In our recent work [13], we presented a highly optimized 3D GPU implementation of GVF which we have used in this implementation.

**Hessian-based Tube Detection Filters**

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 the probability of the voxel belonging to a tubular structure.

We assume that, for an ideal tubular structure, the smallest intensity change is in the direction of the tube and the highest intensity change is in the cross-sectional plane of the
tube. Such a tubular structure can be detected by checking all possible tube directions and calculating the derivatives. However, this would be very inefficient. Frangi et al. [7] showed how to use the eigenvalues of the Hessian matrix to efficiently determine the likelihood that a tube is present without having to check all directions. The Hessian is a matrix of the second-order derivative information at a specific voxel position \( \vec{v} \). The three eigenvectors of the Hessian matrix correspond to the principal directions of the second-order derivatives. These are the directions where the curvature is the maximum and minimum. 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. The direction of the tube is 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 \( \vec{e}_2 \) and \( \vec{e}_3 \), 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 the eigenvectors has to be orthonormal.

By assuming that the airway cross-section is circular, Krissian et al. [9] showed that a TDF response for each voxel can be calculated by fitting a circle to the gradient information in the cross-sectional plane defined by the eigenvectors \( \vec{e}_2 \) and \( \vec{e}_3 \). This method starts by creating a circle with a very small radius in the cross-sectional plane. For a defined number of evenly spaced points, \( N \), on the 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 \( \alpha = \frac{2\pi}{N} \) 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 4, 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.

The circle fitting TDF is more selective than the TDF of Frangi et al. [7], but is slower to compute because it has to sample many points.

\[
T(\vec{v},r,N) = \frac{1}{N} \sum_{i=0}^{N-1} \vec{V}(\vec{v} + r \vec{d}_i) \cdot -\vec{d}_i 
\]  

(4)

Centerline Extraction by Ridge Traversal

Centerlines can be extracted from a valid segmentation using skeletonization and 3D thinning techniques. Another method is to extract the centerlines directly, without a segmentation, by traversing a ridge in the TDF result. 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.

Aylward et al. [1] provides a review of different centerline extraction methods and proposed an improved ridge traversal method based on a set of ridge criteria and different methods for handling noise. Their ridge traversal method starts with a seed voxel \( \vec{v}_0 \). For each voxel \( \vec{v}_i \), a tube likeliness value \( T(\vec{v}_i) \) and an estimate of the tube’s direction
Figure 3: Inverse Gradient Flow Tracking Segmentation. **Left:** The GVF vector field superimposed on the cross section of a tube. The dot in the middle is the dilated centerline. From this centerline, the segmentation is grown in the inverse direction of the vectors as long as the length of the vectors increase. **Right:** The magnitude of the GVF vector field. The border shows the final segmentation.

\[ \vec{t}_i \] is available. The 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. 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. However, this method requires some way to throw away invalid or unnecessary centerlines as some seed points will be invalid 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.

As this method is completely serial its speed cannot be increased by parallelization.

**Segmentation by Inverse Gradient Flow Tracking**

Bauer *et al.* [4] proposed a method for performing a segmentation from the centerline using the already 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. First, the centerlines are dilated and added to the segmentation result. The rest of the segmentation is gradually grown in the inverse direction of the GVF field as long as the magnitude 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 airways. Figure 3 depicts a cross section of a tube with the GVF vector field superimposed and the magnitude of the GVF.
Texture Cache Optimizations

Most modern GPUs have a separate texture cache. These texture caches exist on GPUs because video games and 3D applications use texture mapping to map images to 3D objects to create realistic 3D scenes. Textures are simply images, either 1, 2 or 3 dimensional. The texture caches are optimized for 2D and 3D spatial locality. Regular buffers on the other hand, have only caching in one dimension. Using textures can thus increase cache hits which will increase the speed of global memory access.

In our implementation, we have several 3D 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 exist to support color textures and transparency, and are perfect for storing 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. However, it is possible with the extension cl_khr_3d_image_writes. At the time of writing, only AMD support this extension. The alternative is to only read from textures and write to regular buffers instead. However, this entails having to copy the contents of buffers to textures explicitly.

Trilinear Interpolation

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. The circle fitting TDF has to sample 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.

Work-group Optimization

Work-items are instances of a kernel and 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 and 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. This leads to very inefficient use of the GPU. There is also a maximum number of 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 same direction, but not 3, because 400 is not dividable by 3. 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 the constraints above. To make sure that the cropped volume is dividable by 4 in each direction, the size of the cropping is increased until the new size is dividable by 4.
3 Results and Discussion

Six anonymized Computer Tomography datasets of the lungs were provided by St. Olav's University Hospital and SINTEF Medical Technology. To analyze the speed of our implementation, the six airway datasets were run on two different processors, one NVIDIA Tesla C2070 GPU with 6GB memory and one Intel i7 720 CPU with 4 cores. For each dataset and processor the implementation was executed 10 times and the average runtime calculated. 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 1. The six datasets were processed with the same parameters. The segmentation and centerline for two of the datasets are depicted in Figure 4.

The runtime for each part of the implementation was also measured on a NVIDIA Tesla C2070 GPU. Figure 5 depicts the runtime in seconds of each step when performed on patient 1.

Table 2 shows the original sizes of the datasets, the sizes they were cropped to and the percentage of the original dataset that was removed. Also, the peak memory usage is measured in MBs for both the original and cropped volume. Peak memory usage occurs in the GVF step.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>GPU Runtime</th>
<th>Multi-threaded CPU Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patient 1</td>
<td>31 secs</td>
<td>12 min 52 secs</td>
</tr>
<tr>
<td>Patient 2</td>
<td>31 secs</td>
<td>14 min 43 secs</td>
</tr>
<tr>
<td>Patient 3</td>
<td>26 secs</td>
<td>10 min 44 secs</td>
</tr>
<tr>
<td>Patient 4</td>
<td>27 secs</td>
<td>14 min 4 secs</td>
</tr>
<tr>
<td>Patient 5</td>
<td>20 secs</td>
<td>10 min 5 secs</td>
</tr>
<tr>
<td>Patient 6</td>
<td>38 secs</td>
<td>17 min 25 secs</td>
</tr>
</tbody>
</table>

Table 1: Speed measurements

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Original size</th>
<th>Cropped size</th>
<th>Removed</th>
<th>Peak memory usage (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patient 1</td>
<td>512x512x829</td>
<td>376x280x496</td>
<td>76%</td>
<td>1793 (7461)</td>
</tr>
<tr>
<td>Patient 2</td>
<td>512x512x714</td>
<td>400x288x456</td>
<td>72%</td>
<td>1803 (6426)</td>
</tr>
<tr>
<td>Patient 3</td>
<td>512x512x846</td>
<td>432x264x392</td>
<td>80%</td>
<td>1535 (7614)</td>
</tr>
<tr>
<td>Patient 4</td>
<td>512x512x619</td>
<td>392x256x472</td>
<td>71%</td>
<td>1626 (5571)</td>
</tr>
<tr>
<td>Patient 5</td>
<td>512x512x696</td>
<td>376x264x360</td>
<td>80%</td>
<td>1227 (6264)</td>
</tr>
<tr>
<td>Patient 6</td>
<td>512x512x843</td>
<td>448x312x424</td>
<td>73%</td>
<td>2035 (7587)</td>
</tr>
</tbody>
</table>

Table 2: Result of cropping for each dataset. Peak memory usage is at the GVF step. The number in parentheses is the memory usage without cropping.

Cropping algorithm

The cropping algorithm discards on average 75% of the original dataset as non-lung tissue. In fact, without this cropping, there would not be enough memory on the GPUs to process the dataset in its entirety. The GVF step is the most memory demanding step and the first patient would require 512*512*829 * 3 components * 3 vector fields * 4 bytes = 7461 MB of memory, which is more memory than any GPU has onboard at the time of writing. On the other hand, with the cropping algorithm this memory usage is reduced to 1793 MB.
As the airways are contained within the lungs the cropping does not affect the result of airway segmentation and centerline extraction as long as the $L_{\text{min}}$ variable has a reasonable value. If the value is too high, the upper part of the trachea and some of the distal airways might be lost. However, decreasing this value will result in less cropping. In the six patient datasets that were tested, no airways were lost with the value $L_{\text{min}} = 128$.

**Speed**

The implementation uses about 20 to 40 seconds on a full CT scan when run on a modern NVIDIA Tesla GPU. This is a major improvement from the 3-6 minutes reported by Bauer et al. [4] that only used a GPU for the GVF calculations. The standard deviation in runtime for each patient on the GPU was found to be 0.5-1.5 seconds. Thus the runtime is very stable. The implementation was also run on a multi-core CPU which clearly shows
that this application benefits a lot from the GPUs data parallel processing power.

Runtime analysis of each step of the implementation showed that the GVF calculation was the most expensive step and was very dependent on the dataset size and number of iterations. 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.

We were not able to exploit the GPU’s texture system in the GVF 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 [13] 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.

Accuracy

Lo et al. [11] concluded from their evaluation of 15 different algorithms for segmentation of airways from 20 CT images, that none of the methods were able to extract more than 77% of the manually segmented references on average. Thus the problem of airway segmentation is far from solved. With our implementation of the method of Bauer et al. [4], we conclude that the extraction of the centerlines is the weakest step of the method. And because the segmentation is created using the centerlines, the segmentation is only as good as the centerline extraction is. The ridge traversal method has large problems dealing with noise and local artifacts. This is due to the local greedy nature of the ridge traversal algorithm. Branches that are not detected properly by the TDF thus present 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 are often discarded as noise.

4 Conclusions & Future Work

We have presented an airway segmentation and centerline extraction implementation that utilizes the computational power of modern GPUs. A novel data parallel cropping algorithm was also presented. This algorithm significantly reduces memory usage, thus allowing an entire CT scan to be processed even faster on the GPU in one single pass. The implementation uses about 20 to 40 seconds on a full CT scan when run on an NVIDIA Tesla GPU. This allows the image guided bronchoscopy to start almost immediately after the CT scan is acquired.

Future work will include creating a parallel centerline extraction method and improving it to better extract the small airways. Also, we will test this method on other applications such as blood vessel segmentation.

Acknowledgements

Great thanks goes to the people of the High Performance Computing Lab at NTNU for all their assistance and St. Olav’s University Hospital for the datasets. The authors would also like to convey thanks to NTNU, NVIDIA and AMD. Without their hardware contributions to the HPC Lab, this project would not have been possible.
References


