Medical Image Segmentation for Improved Surgical Navigation

PhD Thesis

Erik Smistad

Abstract

Image guided surgery (IGS) aims to enhance minimal invasive surgery with images and computer technology. In IGS, preoperative images are used to plan the procedure. During the procedure, the surgeon is guided by intraoperative images and tracking of the instruments. Image segmentation is an important image processing step in IGS. Segmentation enables visualization of the structures of interest, removing unnecessary information from the images. Segmentation is also useful for registration and structure analysis, such as calculating the volume of a tumor.

Segmentation of images acquired just before the operation as well as during the operation, has to be fast and accurate in order to be useful in a clinical setting. Many segmentation methods are computationally expensive, especially when run on large medical datasets. There is often a trade-off between speed and accuracy, in which accuracy may be reduced for increased speed and vica versa. Furthermore, the amount of data available for each patient is steadily increasing, making fast segmentation algorithms even more important.

Today, consumer computers contain processors capable of running many operations in parallel at a low cost. In addition to the central processing unit (CPU), most modern computers also contain a specialized processor, the graphic processing unit (GPU). GPUs were originally created for rendering graphics, and are primarily used for computer games. In the last ten years, GPUs have also become popular for general-purpose high performance computing, including medical image processing.

Robust and accurate medical image segmentation is challenging due to low image quality, tissue intensity inhomogeneity and other image artifacts. Model-based segmentation methods incorporate prior knowledge such as the shape, location and appearance of the structure of interest to increase performance.

The work documented in this thesis investigates the use of parallel and GPU computing to accelerate model-based segmentation methods in image guided surgery. A comprehensive review show that most of the common medical image segmentation methods can benefit from running on GPUs. A fast segmentation method for extracting tubular structures is proposed. This segmentation method is able to extract airways, blood vessels and abdominal aortic aneurysms in a few seconds, using a modern GPU and a tubular shape model. Ultrasound segmentation methods for the left ventricle of the heart and blood vessels are also proposed. It is a challenge to segment these images at the same speed as they are produced, as ultrasound is a real-time imaging modality. The proposed methods achieve high accuracy and real-time performance by using GPUs, and a model-based Kalman filtering approach which combines temporal information with shape and appearance information to segment the ultrasound images.

During this project, developing image segmentation software for different types of processors was found to be challenging due to several factors, such as driver errors, processor differences, and the need for low level memory handling. Therefore a novel framework for heterogeneous medical image computing and visualization was developed. This framework aims to make it easier to simultaneously process and visualize medical images efficiently on heterogeneous computer systems.

Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) for partial fulfillment of the requirements for the degree of philosophiae doctor. The thesis is organized as a collection of articles, with an Part I focusing on the broad lines of the work, the research goals and how the articles are related. The articles are found in Part II of the thesis.

This doctoral work has been financed by and performed at the Department of Computer and Information Science, NTNU, Trondheim, with Frank Lindseth as the main supervisor, and Anne C. Elster and Toril A. Nagelhus Hernes as the co-supervisors.

Acknowledgements

First of all, I would like to thank the Department of Computer Science and Information Technology at NTNU for giving me the opportunity to follow my dream of doing research in medical imaging. These five years have been great and challenging. I could not have done it without the continued support and guidance from my supervisor Frank Lindseth, as well as the help, support and motivation from my girlfriend Helene, friends, family, colleagues and co-supervisors Anne C. Elster and Toril A. Nagelhus Hernes.

Research can be lonesome work, but it helps to have good colleagues such as Kai, Gleb, Lars, Hans, Lester, Axel, Boye, Ana Paula, Vegard and Jean-Marc. Thank you for all the entertaining and enlightening lunch discussions over the years.

During my PhD study, I have attended several great conferences organized by MedIm, the Norwegian Research School in Medical Imaging. Thank you for organizing these conferences, as well as funding my travels and awarding me the MedIm Travel & Research Grant. This has been a huge help to my research, and I hope the MedIm organization will continue their work of building a Norwegian network in medical imaging. Finally, I would like to thank NVIDIA and AMD for their contributions to the HPC-Lab, and my colleagues at the HPC-Lab, SINTEF Medical Technology and St. Olavs hospital. Your help and cooperation have been much appreciated and have had a great impact on this work.

Contents

Part I

Research overview

1 Introduction

This chapter starts with an introduction to the key concepts of image segmentation and image guided surgery, which is followed by a discussion on the challenges of image segmentation. Finally, the research goals are presented.

1.1 Image segmentation

Image segmentation is the process of dividing the individual elements of an image or volume into a set of groups, so that all elements in a group have a common property. In the medical domain, this common property is usually that elements belong to the same tissue type or organ. Medical images contain a lot of information as well as noise and other image artifacts. Usually only one or two structures are of interest. Segmentation allows visualization of the relevant structures, removing unnecessary information as shown in Figure 1.1. Segmentation is also useful for registration and structure analysis such as calculating the volume of a tumor.

There are many different segmentation methods from simple intensity-based methods like thresholding and region growing (Adams and Bischof, 1994), to more advanced model-based approaches such as statistical shape models (Heimann and Meinzer, 2009). No segmentation method is considered to be the best, and the method of choice depends on the application, structure to be segmented, and the type of images. Article **A** in Part II of this thesis provides an overview of common medical image segmentation methods.

1.2 Image guided surgery

The term open surgery refers to any surgical technique where the incision in itself is enough to enable the procedure. These methods may involve large wounds and unnecessary damage to healthy tissue. Minimal invasive surgery (MIS) is an alternative to open surgery which aims to improve patient treatment. It has been shown in several surgical procedures that MIS reduces the risk of complications, the amount of postoperative pain and shortens recovery time (Darzi and Munz, 2004). In MIS, surgical instruments are placed through small incisions, thus avoiding large surgical scars. Endoscopes equipped with a camera and a light, are often used allowing the surgeon to see inside the body.

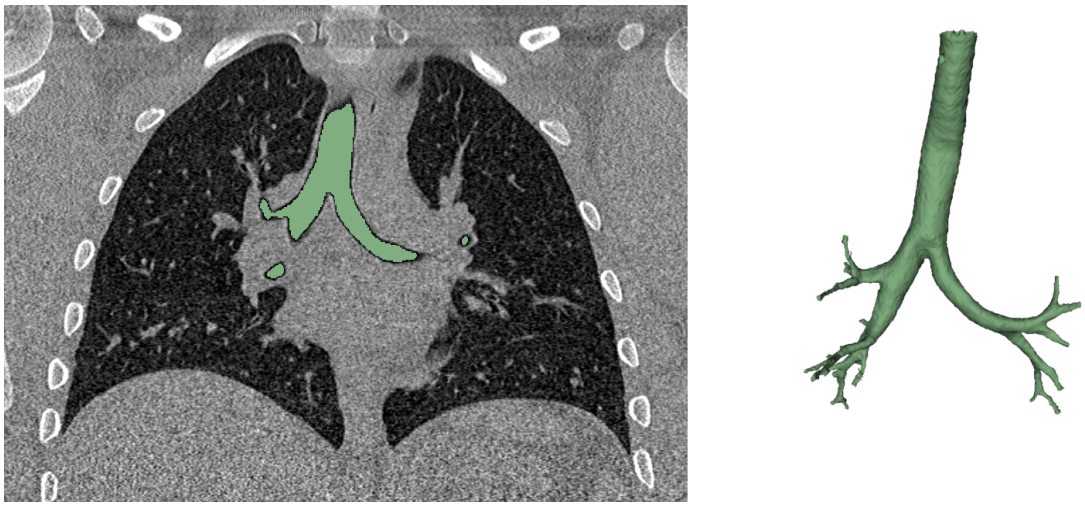

Figure 1.1: Example of airway segmentation of a CT scan. The green pixels of the segmentation highlights the airways, while the rest of the pixels are identified as other tissue. The image to the right shows a 3D visualization of the segmentation.

The goal of image guided surgery (IGS) is to enhance MIS with computer technology and images acquired by ultrasound, X-ray, computer tomography (CT), magnetic resonance imaging (MRI) and cameras. A typical computer assisted image guided intervention uses the following sequence (Cleary and Peters, 2010). First, preoperative images are acquired and used to plan the procedure. During the procedure, instruments are tracked using an optical or electromagnetic tracking system. Using the tracked instruments, the preoperative images are registered to the patient. This enables the instruments to be displayed in relation to the preoperative images, thus guiding the surgeon during the procedure. Intraoperative images are often acquired during the procedure to provide updated image data of the patient.

IGS has been used in several applications. One such application is neurosurgery, in which image and computer guidance has been practiced in more than 30 years, and its success has made it a standard method in most centers (Cleary and Peters, 2010). Many of the neurosurgical procedures require a craniotomy, which results in a brain deformation called brain shift (Nimsky et al., 2001). This is a major challenge in neurosurgery, and the conclusion is that these procedures can only be performed accurately with intraoperative imaging (Cleary and Peters, 2010), using methods such as intraoperative MRI (Hall and Truwit, 2008) and ultrasound (Unsgaard et al., 2002).

Another application of IGS is abdominal laparoscopy, which involves insufflation of a gas into the abdominal cavity, and insertion of instruments through small incisions in the abdomen (Perrin and Fletcher, 2004). Procedures such as liver tumor resection and ablation can be performed in this manner. The abdominal organs shift and deform due to respiration, surgical manipulation and pneumoperitoneum making accurate image guidance challenging (Vijayan et al., 2014).

Bronchoscopy is a minimal invasive technique for diagnosis and treatment of the airways,

where a bronchoscope is inserted into the airways. Today, the bronchoscope is usually flexible containing fiberoptic cables and light, which allows the surgeon to see inside the airways through the bronchoscope (Leira, 2012). Due to the breathing motion of the patient and the complex and narrow structure of the airways, it is difficult for the surgeon to navigate to the target site using this procedure. Also, the bronchoscope is too wide to pass through the smallest airways.

Anaesthesia is an important step in most surgical procedures. The use of regional anaesthesia (RA) is increasing due to the benefits over general anaesthesia (GA), such as reduced morbidity and mortality (Rodgers et al., 2000; Beattie et al., 2001; Urwin et al., 2000), reduced postoperative pain, earlier mobility, shorter hospital stay, and lower costs (Chan et al., 2001). Despite these clinical benefits, RA remains less popular than GA. One reason for this is that GA is far more successful and reliable than RA. Ultrasound imaging has been employed to increase the success rate of RA (Griffin and Nicholls, 2010; Dolan et al., 2008). In ultrasound-guided RA, the nerve and other important structures, such as blood vessels and fascias, are located using the ultrasound images. After a good view of the target nerve has been achieved, a needle is inserted and used to inject local anaesthesia around the nerve.

Several computer technologies are important in IGS, such as image segmentation, registration, tracking and visualization. The role of image segmentation in IGS is to extract the structures of interest from the pre- and intraoperative images. In all of the applications mentioned above, it may be beneficial to use segmentation to extract the important structures, and thereby create a map which can be used to plan and guide the procedure. Segmentation can also be used for structure analysis and feature-based registration which can deal with anatomical shift.

1.3 Image segmentation challenges

Gonzalez and Woods (2008) argued that segmentation of nontrivial images is one of the most difficult tasks in image processing. In this section, the challenges of segmentation will be discussed.

1.3.1 Accuracy

Human experts are in many cases still superior to computer algorithms in terms of accuracy when it comes to image segmentation of nontrivial images. For instance, Lo et al. (2009) evaluated 15 different algorithms for segmentation of airways from CT images, and concluded that none of the methods were able to extract more than 77% of the manually segmented references on average.

Sharma and Aggarwal (2010) argued that image segmentation accuracy is affected by the following factors:

- **Partial volume effect** - If the image resolution is lower than the structure of interest, multiple tissue types contribute to the intensity value of the pixel. This results in the structure boundary being either blurred or not visible at all in the image (Pham et al., 2000). This effect can be addressed by allowing segmentation regions to overlap, often called soft-segmentation.
- **Tissue intensity inhomogeneity** - Many segmentation methods use the pixel intensity value to identify the tissue type. However, in many cases the same tissue has different intensity values and different tissue have the same intensity. This is especially the case in magnetic resonance and ultrasound images were the intensity of the same tissue varies with the location in the image.
- **Low contrast** - Different tissue may have similar intensity values. This is common for soft-tissue in CT images, which can make it hard to separate different soft-tissues based on intensity values alone.
- **Other image artifacts** - Different image modalities may also have other image artifacts. One example is motion during image capture which can introduce blurring or inconsistencies from one image slice to another. This occurs in CT due to the heart beat which moves tissue around the heart while each slice is captured.

Given the challenges above, one may argue that relying on pixel intensity alone is not sufficient for segmentation. To counter these challenges, prior knowledge about the anatomy can be used to make the segmentation more robust, such as the shape and location of the structure of interest. Segmentation methods using prior knowledge are often referred to as model-based segmentation methods. One example is using a circle to model the cross-section of a blood vessel as done by Krissian et al. (2000). Statistical shape models (SSMs) are more complex examples of model-based segmentation. These models are trained to capture the average shape and variation of anatomical structures (Heimann and Meinzer, 2009). Appearance models describe how a structure appears in an image, and can also be used to make segmentation more robust. One example is active appearance models (AAMs) (Cootes et al., 2001), which can generate synthetic images of how a structure should look in an image based on training data.

1.3.2 Speed

Segmentation of images acquired just before the operation as well as during the operation, has to be both fast and accurate in order to be useful in image guided surgery. Although machines are generally faster than humans at image segmentation, several segmentation methods are still not fast enough. Ideally, the result should be ready instantly, but many segmentation methods may require several minutes of processing. The amount of data available for each patient is steadily increasing (Scholl et al., 2010), making fast segmentation algorithms even more important.

The following are factors affecting the speed of segmentation algorithms.

- **Segmentation algorithm complexity** - The time-complexity of an algorithm quantifies the amount of time an algorithm uses as a function of the input size.
- **Segmentation algorithm implementation** - An algorithm can be implemented and optimized in different ways, for instance by using different data structures, intrinsic functions and parallelization.
- **Computer hardware** - The speed and capacity of the computer hardware on which the segmentation is executed.
- **Image size** - Larger images tend to require more processing time as well as memory. Thus, speed may be increased by cropping the image before segmentation, use image compression, or use a smaller and less accurate data type.

There is usually a trade-off between speed and accuracy, in which accuracy may be reduced for increased speed and vica versa.

1.3.3 Automation

Semi-automatic segmentation methods may be used to improve the accuracy over automatic segmentation, while being faster than manual segmentation. Most segmentation algorithms require some sort of initialization. This initialization may be difficult to do automatically, while easy to do manually. For instance, region growing segmentation requires one or more seed pixels, which may be selected by a user with the computer mouse and anatomical domain knowledge. Although semi-automatic segmentation methods can be used to increase accuracy, user interactions during surgical procedures are not desired as it takes time, distracts the surgeons from the actual procedure, require expert knowledge, and is non-repeatable and subjective.

1.4 Research goals

The main goal of this thesis is to develop image segmentation methods for image guided surgery that are accurate, fast and automatic, with primary focus on increasing the speed of image segmentation. Two image segmentation domains which are important in image guided surgery have been investigated. These domains are segmentation of tubular structures, and segmentation of ultrasound images. In this section, these domains and the related research goals are presented.

1.4.1 Accelerating segmentation with parallel and GPU computing

As discussed previously, speed is one of the main challenges of image segmentation for image guided surgery. There are several ways of increasing the speed of an image seg-

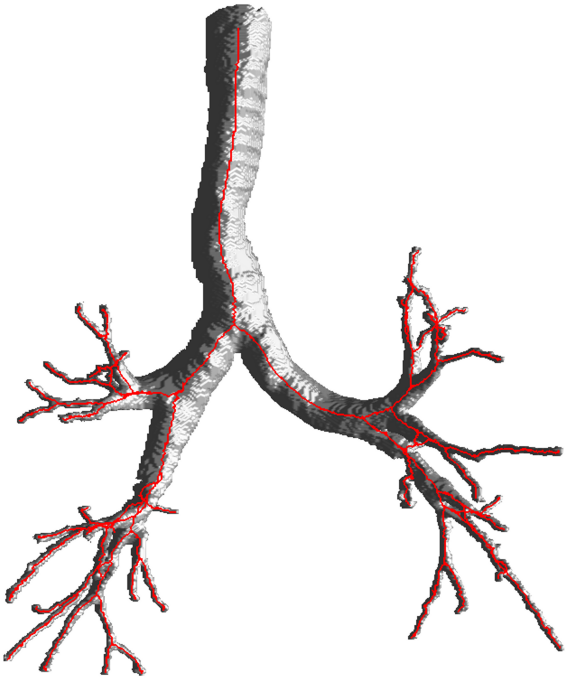

Figure 1.2: Segmentation and centerline of the airways.

mentation algorithm. One way is to exploit the parallel processing capabilities of modern processors. Interconnected machines and processors were used to run image segmentation in parallel already in the 1980's (Tilton, 1988; Morioka et al., 1990), but it required expensive hardware. Today, consumer computers contain processors capable of running many operations in parallel at a low cost. In addition to the central processing unit (CPU), most modern computers also contain another processor called the graphic processing unit (GPU). The main difference between a CPU and a GPU, is that a GPU has a lot more arithmetic logic units (ALUs) than the CPU (McCool, 2008), allowing the GPU to process many different data elements in parallel. However, many of the ALUs share a control unit and therefore have to run the same instruction for each data element. GPUs were originally created for rendering graphics and are primarily used for computer games. However, in the last ten years, GPUs have become popular for general-purpose high performance computation, including medical image processing (Eklund et al., 2013). This leads to the first research goal of this thesis:

Research goal 1: Investigate the use of parallel and GPU computing to accelerate image segmentation.

1.4.2 Segmentation of tubular structures

Blood vessels, airways, bones, neural pathways and intestines are all examples of important tubular structures in the human body. In addition to segmentation, it can be useful to extract the centerline of these structures. The centerline is a line going through the center

of the tubular structures, providing a structural representation of the tubular structures as shown in Figure 1.2. The extraction of these structures can be essential for planning and guidance of several surgical procedures such as bronchoscopy, laparoscopy, and neurosurgery. Tubular structures extracted from preoperative images can be matched to similar intraoperative structures, such as a set of positions inside the airways generated by a tracked bronchoscope, or brain vessels extracted from ultrasound. It has been shown that registration of blood vessels from pre- and intraoperative image data can also be used to detect and correct organ-shift, such as brain-shift (Reinertsen et al., 2007).

There are several methods for extracting tubular structures from medical images. A recent and extensive review on blood vessel extraction was done by Lesage et al. (2009), and an older one was done by Kirbas and Quek (2004). Two reviews on the segmentation of airways were done by Lo et al. (2009) and Sluimer et al. (2006).

A common strategy for extracting tubular structures is to grow the segmentation from an initial point or area, using methods such as region growing (Adams and Bischof, 1994), active contours (Xu and Prince, 1998) and level sets (Sethian, 1999). A centerline can be extracted from a binary segmentation using iterative morphological thinning, also called skeletonization (Palàgyi and Kuba, 1998; Palágyi and Kuba, 1999; Xie et al., 2003). Iterative thinning removes voxels from the segmentation in a particular order until the object can not be thinned anymore. Another approach is to use a distance transform or gradient vector flow (GVF), as done by Hassouna and Farag (2007). Direct centerline extraction without a prior segmentation is also possible using methods such as shortest path and ridge traversal. The segmentation can be grown afterwards using region growing with the centerline as seeds. Aylward and Bullitt (2002) presented a review of different centerline extraction methods. They proposed an improved ridge traversal algorithm based on a set of ridge criteria, and different ways of handling noise. Bauer and Bischof (2008c) showed that ridge traversal could be used together with GVF. Direct centerline extraction usually needs some initial centerpoints and the direction of the tubular structure. This can be acquired with tube detection filters (TDFs). TDFs are used to detect tubular structures by calculating a probability of each voxel being inside a tubular structure. Most TDFs use gradient information, often in the form of the eigenanalysis of the Hessian matrix. Frangi et al. (1998) presented an enhancement and detection method for tubular structures based on the eigenvalues of this matrix. Krissian et al. (2000) created a model-based detection filter that fits a circle to the cross-sectional plane of the tubular structure. These TDFs have the potential of detecting different types of tubular structures such as blood vessels and airways. There are several examples of methods claiming to be robust enough to segment and extract centerlines of tubular structures of different types (e.g. vessels and airways), organs and modalities (Bauer, 2010; Bauer and Bischof, 2008b,a,c; Bauer et al., 2009a,b; Krissian et al., 2000; Aylward and Bullitt, 2002; Benmansour and Cohen, 2010; Li and Yezzi, 2007; Behrens et al., 2003; Cohen and Deschamps, 2007; Lorigo et al., 2000; Spuhler et al., 2006). However, most of these present results for only a few datasets of one or two organs/modalities. The PhD thesis of Bauer and related articles (Bauer, 2010; Bauer and Bischof, 2008b,a,c; Bauer et al., 2009a,b) are exceptions that present results for several different organs (e.g. lung, heart and liver), however only from CT scans.

Bauer et al. use different methods to perform the major steps (tube detection, centerline extraction and segmentation) for each organ. Instead, a generic method able to extract different tubular structures would be beneficial as it would reduce code duplication. Also, the future improvement of a generic method would benefit a wider range of applications.

Studies have shown that the extraction of tubular structures can be accelerated using GPUs. Erdt et al. Erdt et al. (2008) performed TDF and region growing segmentation on the GPU, and reported 15 times faster computation of the gradients and up to 100 times faster TDF. Narayanaswamy et al. (2010) did vessel luminae region growing segmentation on the GPU, and reported a speedup of 8 times. Bauer et al. presented a GPU acceleration for airway segmentation by computing the GVF on the GPU in Bauer et al. (2009a), and the TDF calculation on the GPU in Bauer et al. (2009b). Helmberger et al. (2013) performed region growing for airway segmentation on the GPU, and a lung vessel segmentation on the GPU using a TDF. They reported a runtime of 5-10 minutes using a modern GPU and CUDA compared to a runtime of up to an hour using only the CPU. These studies on tube detection filters and GPU acceleration lead to the second research goal of this thesis:

Research goal 2: Create a segmentation method for tubular structures that is:

- Generic - Able to extract different tubular structures such as blood vessels and airways from various image modalities using Hessian-based tube detection filters.
- Automatic - No user interaction needed.
- Fast - Accelerate processing using GPUs.

1.4.3 Segmentation of ultrasound images

Ultrasound is a key intraoperative imaging modality in image guided surgery, due to its real-time imaging capability, low cost and small footprint in the operating room. It can be used intraoperatively to look inside the patient during the procedure or to update preoperative images and models. Ultrasound has shown significant potential in several procedures, such as neurosurgery (Gronningsaeter et al., 2000) and laparoscopy (Langøet al., 2008).

Segmentation of ultrasound images is challenging due to noise and image artifacts such as tissue inhomogeneity, reverberations and acoustic shadowing. Thus, relying on pixel intensity alone for may not be sufficient for robust and accurate ultrasound segmentation. Noble and Boukerroui (2006) conducted a review of current segmentation methods for ultrasound images, where they concluded that a good ultrasound image segmentation method needs to make use of all task-specific constraints and prior knowledge due to the low image quality.

It is a challenge to segment ultrasound images at the same speed as they are produced, as ultrasound is a real-time imaging modality, delivering several images per second. Several different segmentation methods have been used on ultrasound images including region

growing (Abdel-Dayem and El-Sakka, 2005), level sets (Xie et al., 2005; Li et al., 2006), active contours (Mao et al., 2000; Mikić et al., 1998; Chen et al., 2004), tube detection filters (Hennersperger et al., 2015) and statistical shape models (Bosch et al., 2002). However, most of these do not target real-time ultrasound segmentation.

One way to segment dynamic images, is to apply a segmentation method on each image frame, but this may not satisfy real-time constraints. Another approach, often called tracking, is to use the segmentation of the previous frame as prior knowledge to segment the next frame, and thus reduce the computational cost. The Kalman and particle filters are examples of such tracking methods. The Kalman filter (Kalman, 1960) is an algorithm that estimates a state using a series of noisy measurements over time. In image segmentation, the state may be a set of parameters describing the shape of the structure of interest and its position in the image. One type of measurement for object tracking, is the offset from each point on the shape to the object's edges in the current image frame. Previous work has shown that the Kalman filter can track structures such as blood vessels (Guerrero et al., 2007) and the left ventricle of the heart (Orderud, 2006; Orderud et al., 2007; Orderud and Rabben, 2008) in real-time. The Kalman filter algorithm consists mostly of a set of matrix operations. Linear algebra libraries such as ATLAS, Eigen and boost can accelerate these type of operations. The particle filter method (Arulampalam et al., 2002) estimates the posterior density of the state variables given the measurements using Monte Carlo simulations. Generally many samples are required, but each sample can be evaluated in parallel using a GPU (Montemayor et al., 2006; Mateo Lozano and Otsuka, 2008; Lozano and Otsuka, 2008; Murphy-Chutorian and Trivedi, 2008; Lenz et al., 2008; Brown and Capson, 2012).

The review of Noble and Boukerroui (2006) showed that most segmentation methods for ultrasound depend on user interaction such as the selection of a seed point or region of interest. As ultrasound is an intraoperative image modality, user interaction may distract the physician from the procedure or require additional personnel, which is why automation of ultrasound segmentation methods is important.

The studies mentioned above on ultrasound segmentation lead to the third and last research goal of this thesis:

Research goal 3: Create segmentation methods for ultrasound images which are:

- Robust - Using model-based segmentation methods.
- Automatic - Initialize methods using a priori knowledge, thus eliminating the need for user interaction.
- Real-time - Accelerate processing using GPUs, state estimation methods and fast linear algebra libraries.

2 Summary of contributions

The thesis is organized as a collection of articles. In this chapter, the articles are listed and the results of these articles are discussed in relation to the research goals of the thesis.

2.1 Publication list

2.1.1 Selected publications

The following is a list of the articles, sorted after the research goal they contribute most to. The full text of the articles can be found in Part II of the thesis.

Research goal 1: Parallel and GPU accelerated image segmentation

A Medical image segmentation on GPUs - A comprehensive review
Erik Smistad, Thomas L. Falch, Mohammadmehdi Bozorgi, Anne C. Elster and Frank Lindseth
Medical Image Analysis, volume 20, February 2015, pages 1-18. Elsevier B.V.

B FAST: framework for heterogeneous medical image computing and visualization
Erik Smistad, Mohammadmehdi Bozorgi, and Frank Lindseth
International Journal of Computer Assisted Radiology and Surgery, 2015. Springer. Not assigned to an issue yet.

Research goal 2: Segmentation of tubular structures

C GPU accelerated segmentation and centerline extraction of tubular structures in medical images
Erik Smistad, Anne C. Elster and Frank Lindseth
International Journal of Computer Assisted Radiology and Surgery, volume 9, issue 4, July 2014, pages 561-575.

D A new tube detection filter for abdominal aortic aneurysms
Erik Smistad, Reidar Brekken and Frank Lindseth

Proceedings of MICCAI 2014 Workshop on Abdominal Imaging: Computational and Clinical Applications. Lecture Notes in Computer Science, volume 8676, September 2014, pages 229-238.

E Multigrid gradient vector flow computation on the GPU
Erik Smistad and Frank Lindseth
Journal of Real-Time Image Processing, 2014. Springer. Not assigned to an issue yet.

Research goal 3: Segmentation of ultrasound images

F Real-time Tracking of the Left Ventricle in 3D Ultrasound Using Kalman Filter and Mean Value Coordinates
Erik Smistad and Frank Lindseth
Proceedings MICCAI Challenge on Echocardiographic Three-Dimensional Ultrasound Segmentation (CETUS), September 2014, pages 65-72. Midas Journal.

G Real-Time Automatic Vessel Segmentation and Model Registration for Improved Ultrasound-Guided Regional Anaesthesia of the Femoral Nerve
Erik Smistad and Frank Lindseth
Submitted to IEEE Transactions on Medical Imaging, April 2015.

2.1.2 Other publications

The following publications were also produced during the work on this thesis, but are not included in the thesis. However, the full text of these publications can be found in the Appendix.

A1 Real-time surface extraction and visualization of medical images using OpenCL and GPUs
Erik Smistad, Anne C. Elster and Frank Lindseth
Norsk informatikkonferanse 2012, pages 141-152.

A2 GPU-Based Airway Segmentation and Centerline Extraction for Image Guided Bronchoscopy
Erik Smistad, Anne C. Elster and Frank Lindseth
Norsk informatikkonferanse 2012, pages 129-140.

A3 Real-time gradient vector flow on GPUs using OpenCL
Erik Smistad, Anne C. Elster and Frank Lindseth
Journal of Real-Time Image Processing, volume 10, issue 1, March 2015, pages 67-74.

Contributions to two challenges were submitted and resulted in the two following publications, which are not included in this thesis.

C1 Comparing algorithms for automated vessel segmentation in computed tomography scans of the lung: the VESSEL12 study
R. D. Rudyanto, S. Kerkstra, E. M. van Rikxoort, C. Fetita, P-Y. Brillet, C. Lefevre, W. Xue, X. Zhu, J. Liang, I. Öksüz, D. Ünay, K. Kadipaşaoğlu, R. S. J. Estépar, J. C. Ross, G. R. Washko, J-C. Prieto, M. H. Hoyos, M. Orkisz, H. Meine, M. Hüllebrand, C. Stöcker, F. L. Mir, V. Naranjo, E. Villanueva, M. Staring, C. Xiao, B. C. Stoel, A. Fabijanska, E. Smistad, A. C. Elster, F. Lindseth, A. H. Foruzan, R. Kiros, K. Popuri, D. Cobzas, D. Jimenez-Carretero, A. Santos, M. J. Ledesma-Carbayo, M. Helmberger, M. Urschler, M. Pienn, D. G. H. Bosboom, A. Campo, M. Prokop, P. A. de Jong, C. Ortiz-de-Solorzano, A. Muñoz-Barrutia and B. van Ginneken
Medical Image Analysis, volume 18, issue 7, October 2014, pages 1217-1232.

C2 Online system for standardized evaluation of algorithms for left ventricular segmentation in 3D echocardiography
O. Bernard, J. G. Bosch, B. Heyde, M. Alessandrini, D. Barbosa, S. Camarasu-Pop, F. Cervenansky, S. Valette, O. Mirea, M. Bernier, P. M. Jodoin, J. S. Domingos, R. V. Stebbing, K. Keraudren, O. Oktay, J. Caballero, W. Shi, D. Rueckert, F. Milletari, S. A. Ahmadi, E. Smistad, F. Lindseth, M. van Stralen, C. Wang, Ö. Smedby, A. Papachristidis, M. L. Geleijnse, E. Galli and J. D'hooge
Submitted to Medical Image Analysis, March 2015.

2.2 Parallel and GPU accelerated image segmentation

The first research goal was to investigate parallel and GPU computing to accelerate image segmentation. Article **A** is a comprehensive review on the use of GPUs to accelerate medical image segmentation. The review first describes the properties of GPUs in comparison to CPUs. With this description of the GPU, five key factors affecting the algorithm's suitability for GPU computation were identified:

- **Data parallelism** - An algorithm is data parallel if multiple data elements can be processed using the same instructions in parallel. The degree of speedup achieved by parallelization is limited by how much of the algorithm can be run in parallel. Amdahl (1967) showed that the maximum speedup of a program where 95% is executed in parallel is a factor of 20, regardless of the number of cores or thread processors being used. Thus, an important GPU suitability factor is the percentage of the algorithm that is data parallel.

- **Thread count** - Thread count is how many individual parts the calculations can be divided into and executed in parallel. To obtain a substantial speedup of a data parallel algorithm on the GPU, the number of threads has to be high.

- **Branch divergence** - Branches, such as "if-else" statements, are problematic because all thread processors sharing a control unit have to perform the same instructions. A branch is divergent if two or more threads in an atomic unit of execution

(AUE, also called warp or wavefront) choose different execution paths. The threads in an AUE with a divergent branch must perform all execution paths, which can reduce performance.

- **Memory usage** - The GPUs limited amount of memory may not be enough to process large images. Most image processing algorithms for the GPU are memory-bound. Their speed therefore primarily depend on the memory bandwidth of the GPU, and reducing memory usage can improve the speed of the algorithm.

- **Synchronization** - Many parallel algorithms require some form of synchronization which can lower performance.

Several common medical image segmentation algorithms are explained and discussed in article **A** in terms of GPU computation using the five key factors listed above. Most of these segmentation methods process each pixel using the same instructions and data from a small neighborhood around the pixel. Thus, the amount of data parallelism and thread count is usually high. Typical sizes of medical datasets are 512×512 for images, and 512^3 for volumes, which amount to over 262 thousand pixels and more than 134 million voxels respectively. However, as seen in this review, some segmentation methods do not process each pixel. Examples include active contours which move a contour consisting of a set of points, and statistical shape models that model shapes using a set of landmark points. These methods may only benefit from using GPUs when the number of points is high.

Segmentation methods are often iterative, running a set of instructions several times. The iterative processing may require double buffering, as global memory writes are not coherent within one kernel execution. With double buffering, data is written back and forth to the slow global memory of a GPU. The GPU has a specialized memory system for images, called the texture system. This system specializes in fetching and caching data from 2D and 3D textures and may improve performance when processing image data (NVIDIA Corporation, 2010; Advanced Micro Devices Inc., 2012). Double buffering is currently required when using textures, as a texture can only be read from or be written to in a thread, not both. Double buffering also doubles the amount of memory used, which can be problematic for some methods such as 3D gradient vector flow. Branch divergence is a challenge for several methods such as region growing and narrow-band level sets, as not all pixels need to be processed in these methods. The performance loss due to branch divergence can be reduced using stream compaction methods (Billeter et al., 2009; Ziegler et al., 2006). However, this comes at the cost of more processing, and will not necessarily improve performance if it has to be used for each iteration, which is the case for region growing. Some GPU methods may not provide a large speedup over an optimized serial method because the GPU method implies more processing. This is true for methods such as region growing and watershed. With region growing, the total number of pixels processed in each iteration is much higher in the data parallel GPU implementation than in the serial CPU implementation.

Review article **A** concludes that most segmentation methods can benefit from GPU acceleration. However, factors such as synchronization, branch divergence and memory usage may limit the speedup over serial execution.

2.2.1 FAST - A framework for heterogeneous medical image computing and visualization

Computer systems are becoming increasingly heterogeneous in the sense that they consist of different processors, such as multi-core CPUs and GPUs. As the amount of medical image data increases, it is crucial to exploit the computational power of these processors. However, the programming of this hardware is still difficult due to several factors. One factor is that the software needed to use the hardware, such as GPU drivers and OpenGL/OpenCL implementations, may contain errors which are hard to debug. The programmer may have to write separate code for different hardware and software versions, as the programmer can not change proprietary software such as GPU drivers. This results in increased software development overhead, and fragmented source code. Most GPU programming tools such as shaders, CUDA and OpenCL expose the programmer to several hardware details. For instance, most GPUs have their own memory that is separate from the computer's main memory. This memory is often divided into the different memory spaces global, texture and constant memory (Owens et al., 2008). The programmer has to explicitly move data between these memory spaces during execution.

Article **B** presents a novel FrAmework for heterogeneouS medical image compuTing and visualization (FAST). The goal of this framework is to simplify the efficient processing and visualization of medical images on heterogeneous systems. The insight toolkit (ITK) (Ibanez and Schroeder, 2004; Kitware, 2014a) and the visualization toolkit (VTK) (Schroeder et al., 2006; Kitware, 2014b) are two of the most commonly used frameworks for medical image analysis and visualization. While these frameworks provide GPU accelerated processing more as an extension and as an optional feature, the FAST framework presented in this article was designed with heterogeneous accelerated processing in mind from the start, and it is part of the core of the framework. We believe this will result in a framework that is faster and easier to use. One framework that aims to aid the design of image processing algorithms for different GPUs, is the Heterogeneous Image Processing Acceleration Framework (HIPAcc) (Membarth et al., 2012, 2015). This framework copes with the complexity of programming GPUs for medical image processing algorithms by using a high-level domain specific language. Code written in this language is translated to low-level OpenCL and CUDA code. HIPAcc targets only the design of image processing algorithms. However, medical imaging pipelines consist of several other steps such as visualization and registration. FAST considers all parts of the medical image computing and visualization pipeline.

In article **B**, code examples and performance measurements demonstrates that the framework is both easy to use, and performs better than ITK and VTK for several common medical image algorithms. The FAST framework achieves this by using common pro-

gramming paradigms, and hiding the details of memory handling from the user, while still enabling the use of all processors and cores on a system. FAST aims to provide a large set of tests and benchmarks to detect and report errors in the underlying hardware and software (e.g. GPU drivers). This enables a user to check if there are any problems for a specific setup. The framework is open-source, cross-platform and available online[1].

The performance of four different image processing pipelines were evaluated and compared to that of ITK and VTK. The pipelines include algorithms such as Gaussian smoothing, surface extraction, region growing, thresholding, skeletonization and iterative closest point. The results show that FAST is faster for the four pipelines with speedups of up to 20 times. This speedup is mainly due to the fact that FAST is able to use the GPU for processing and rendering, while ITK and VTK rely on multi-threading for acceleration.

2.2.2 Fast visualization of segmentation - Surface extraction

In most segmentation applications, visualization of the segmentation result is required. To visualize the segmentation result in 3D, a surface mesh of the segmentation must be extracted. The main method of surface extraction is the marching cubes algorithm by Lorensen and Cline (1987). This algorithm is highly data parallel, processing each voxel independently. However, the algorithm has divergent branches which must be handled efficiently. Ziegler et al. (2006) introduced a data structure called histogram pyramid (HP), which can handle divergent branches in logarithmic time complexity on the GPU. This data structure uses the texture system of the GPU to increase memory access speed. In article **A1**, a GPU-based marching cubes method was presented. This method was inspired by Dyken et al. (2008), which used HPs to accelerate marching cubes on the GPU. Their method stores the HP in a single 2D texture with all the HP levels as mipmap levels. A disadvantage of using single texture is that the same data type and format has to be used for all levels. The proposed method in article **A1** has a 3D texture for each level, enabling the use of 8 and 16 bit data types when sufficient. This reduces the memory usage and enables 3D spatial caching. The method proposed in article **A1** is able to extract and visualize surfaces from large datasets (512^3 and 1024^3 voxels) faster than the implementation of Dyken et al. (2008), and was later integrated into FAST. In the article on FAST (**B**), it was shown that this GPU method was more than 10 times faster than the surface extraction method in VTK, and enables real-time surface extraction and visualization of ultrasound image sequences.

2.3 Segmentation of tubular structures

A segmentation method able to extract different types of tubular structures from various imaging modalities, has to handle variations in intensity, size, shape and contrast. Com-

[1] http://github.com/smistad/FAST/

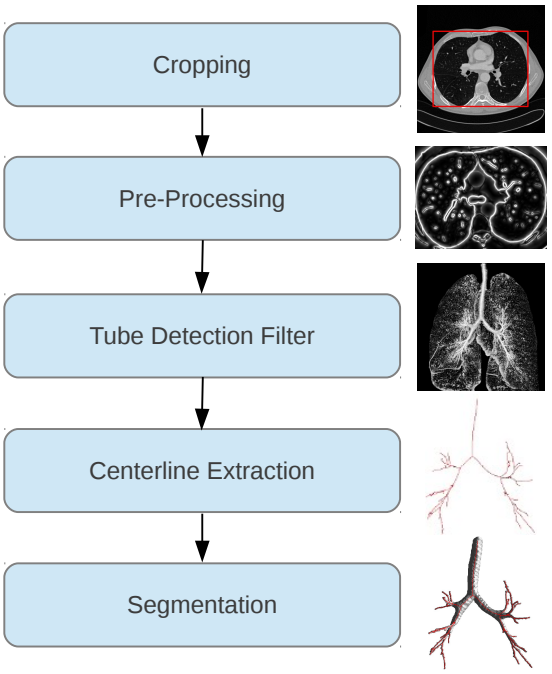

Figure 2.1: GPU-based airway segmentation and centerline extraction

mon to all tubular structures is their elongated shape with a closed cross-sectional profile. The results from the challenges EXACT'09 (Lo et al., 2009) and VESSEL'12 (Rudyanto et al., 2014) showed that the best performing methods for both airway and vessel segmentation of the lungs are Hessian-based tube detection filters. Article **A** of this thesis described how these Hessian-based methods have high potential for GPU acceleration. These results inspired the first work towards research goal 2, which used Hessian-based TDFs and GPUs to create a fast airway segmentation and centerline extraction method for image guided bronchoscopy.

2.3.1 Airways

In article **A2**, a GPU-based airway segmentation and centerline extraction method was presented. This method extracts the airways of the lungs from CT images, using five main steps as shown in Figure 2.1.

The first step is cropping of the input dataset. A CT thorax image usually contains 500-850 slices with dimensions 512×512. As the memory on the GPU is limited (at that time 1-3 GB), the image size had to be reduced. A typical CT image of the thorax contains a lot of data which is not part of the lungs, such as space outside the body, body fat and the bench the patient is resting on. Article **A2** introduced a novel parallel cropping algorithm which removes this unnecessary data. On the six CT images used in this article, the algorithm discarded over 70% of the image data, thus significantly reducing memory usage and processing time.

The next step of the method is pre-processing, which involves smoothing of the dataset to remove noise, and gradient vector flow (GVF) to detect airways of different sizes. GVF was found to be the most time-consuming part of the method. However, this algorithm is very suited for GPU computation as each voxel is processed independently using the same instructions. Several GPU implementation of 2D GVF have been reported (Eidheim et al., 2005; He and Kuester, 2006; Zheng and Zhang, 2012; Alvarado et al., 2013). Bauer et al. (2009a) used a GPU implementation of GVF for airway segmentation, but few details on the implementation were provided. Article **A3** presented a GPU-based GVF method. This method was used in article **A2** for the airway segmentation and centerline extraction. GVF is a memory bound iterative algorithm, and article **A3** explored three different memory optimization techniques to improve the performance. These techniques involved using the texture memory, shared memory and a 16-bit compressed floating point storage format. The results showed that using the texture memory with the 16-bit format and without shared memory was fastest on GPUs and can double the performance. Accuracy measurements revealed a small difference in the average vector magnitude, and a large difference in the vector angle between the default 32-bit and compressed 16-bit storage formats. However, the large angle differences were only present on small vectors, and thus may not be a problem for most applications. The 16-bit storage format also allows much larger volumes to reside completely in the GPU memory.

The tube detection filter (TDF) step used the circle fitting method by Krissian et al. (2000). This TDF models the cross-section of a tubular structure as a circle. The cross-sectional plane was determined by an eigenanalysis of the Hessian matrix for each voxel. A circle was then fitted to the GVF gradients, and the TDF response was calculated. This TDF response indicates how well the circle fits the GVF gradients.

A direct centerline extraction method was used to extract the centerlines using the result of the TDF (Bauer et al., 2009a). This serial algorithm starts with the voxel with the highest TDF response from the previous step. From this voxel a centerline is created by traversing from one voxel to a neighbor voxel using the direction of the tubular structure and the TDF responses. This is continued until no more centerlines can be added.

The segmentation was grown from the centerlines using a region growing approach. The growing procedure is constrained by the GVF field so that the segmentation grows in the inverse direction of the vectors, as long as the vector length increases (Bauer et al., 2009a).

The proposed method in article **A2** used about 20 to 40 seconds to process a full CT scan using an NVIDIA Tesla C2070 GPU. This was a major improvement from the 6 minutes reported by Bauer et al. (2009a) which used a GPU for the GVF calculations. In comparison, a multi-threaded CPU implementation of the same method used between 10 and 17 minutes. Due to the lack of ground truth for the CT images, the accuracy was not properly evaluated.

2.3.2 Blood vessels and multi-modality

The airway extraction method from the previous section was later extended in article **C** to extract other types of tubular structures such as blood vessels from different modalities including CT, MRI and 3D ultrasound. This was done by parametrization of the method, where the different parts of the method was tuned to different modalities and structures. The cropping method from article **A2** was extended to crop other types of datasets as well.

All steps in the previous implementation (article **A2**) except centerline extraction, were executed on the GPU. In article **C**, a new parallel centerline extraction method was developed, enabling a complete GPU-based method for segmentation of tubular structures. The previous implementation was tested with only one GPU, but in this study more GPUs were available. The GPUs used included the NVIDIA Tesla C2070, which was the one used in article **A2**, and two newer GPUs, the AMD HD7970 and the NVIDIA Tesla K20. The newer GPUs had 4 times higher theoretical peak performance and were able to extract the tubular structures faster than the C2070 GPU. Of the two newer GPUs, the AMD GPU performed better, as NVIDIA's OpenCL implementation currently does not support writing directly to 3D textures. Because of this restriction, buffers have to be used in the most computationally expensive step, gradient vector flow. Using buffers instead of textures means no 3D cache optimization and hardware data type conversion, both of which can increase performance. With the proposed methods, the AMD GPU was able to extract airways from CT scans and blood vessels from MRI scans in about 5 seconds.

To show the general applicability of the method, clinical images from three different modalities and two different organs were used:

- CT scans of the lungs (Airways, 12 datasets)
- MR images of the brain (Blood vessels, 4 datasets)
- 3D ultrasound Doppler images of the brain (Blood vessels, 7 datasets)

The results showed that the method is able to extract tubular structures from several modalities and organs with comparable quality by only changing a few parameters.

The quality of the extracted centerlines and the segmentation were measured using realistic synthetic vascular tree volumes, and their ground truth segmentation and centerlines. These synthetic volumes and ground truth were created using the VascuSynth software by Hamarneh and Jassi (2010). One of these synthetic volumes is depicted in Figure 2.2. Three generated datasets were used, each with a different amount of Gaussian additive noise. This was done to show how well the different methods performs with increasing amounts of noise. The average centerline error was higher for the proposed parallel centerline (PCE) algorithm, than the ridge traversal and skeletonization methods. This increased centerline error was a result of the straight lines created by the PCE algorithm between centerpoints. However, the error was below 0.7 voxels, which we argue is not problematic for most applications. Also, the proposed PCE algorithm was able to extract over 10% more of the synthetic vascular tree compared to the ridge traversal algorithm

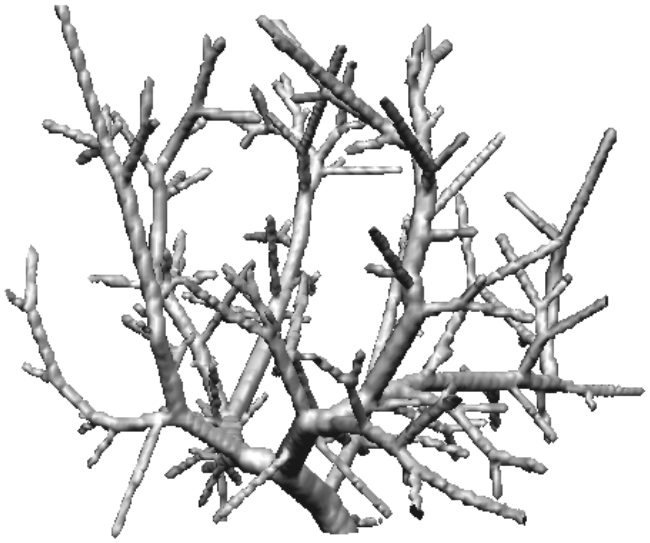

Figure 2.2: A synthetic vascular tree generated by the VascuSynth software (Hamarneh and Jassi, 2010).

for large noise levels (0.3).

2.3.3 Abdominal aortic aneurysms

An abdominal aortic aneurysm (AAA) is a vascular disease resulting in a permanent local dilation of the abdominal aorta. AAAs can eventually rupture, a condition associated with high mortality (85-95%) (Kniemeyer et al., 2000). When the risk of rupture exceeds the risk associated with repair, AAA can be treated by open surgery, or by endovascular placement of a stentgraft inside the aneurysm lumen to reduce the pressure on the arterial wall. The segmentation and centerline of AAAs may be useful for visualization, volume estimation, registration, and planning and guidance of stentgraft placement.

The method proposed in article **C** was able to extract blood vessels. However, AAAs are larger than most other blood vessels, and this posed a challenge in terms of GVF. The most common way to calculate GVF is to use Euler's method as demonstrated by Xu and Prince (1998). This method converges slowly (Han et al., 2007), which can be a problem for large tubular structures where the gradients at the edges have to diffuse a long way to the center. To solve this problem, Han et al. (2007) used multigrid methods to calculate GVF, achieving a better convergence rate. However, non-GPU implementations of multigrid GVF are not as fast as the Euler GVF method when executed on the GPU. In article **E**, a GPU-based multigrid method was proposed to calculate GVF. The same memory optimization techniques as in article **A3** were used, such as texture memory and compressed 16-bit storage format. Using the proposed multigrid GPU GVF method, 6 iterations and 1-2 seconds of processing were sufficient to process the AAA datasets as shown in Figure 2.3. In comparison, the Euler GPU implementation of article **A3** required

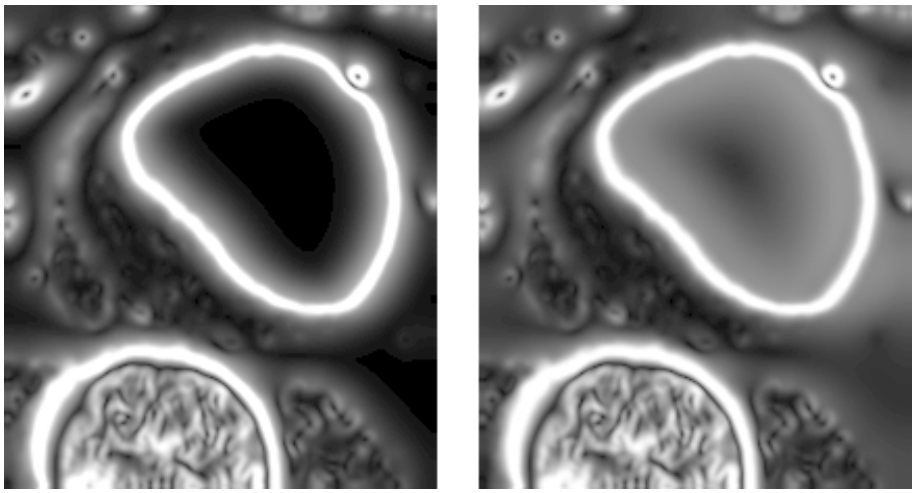

Figure 2.3: Magnitude of the vector field after running gradient vector flow (GVF) on a AAA CT dataset. **Left:** Euler's method with 1000 iterations. **Right:** Multigrid method with 6 iterations. The image to the left shows that GVF with Euler's method has problems diffusing the gradients on the edge of the aneurysm to the center, which is necessary for the TDFs.

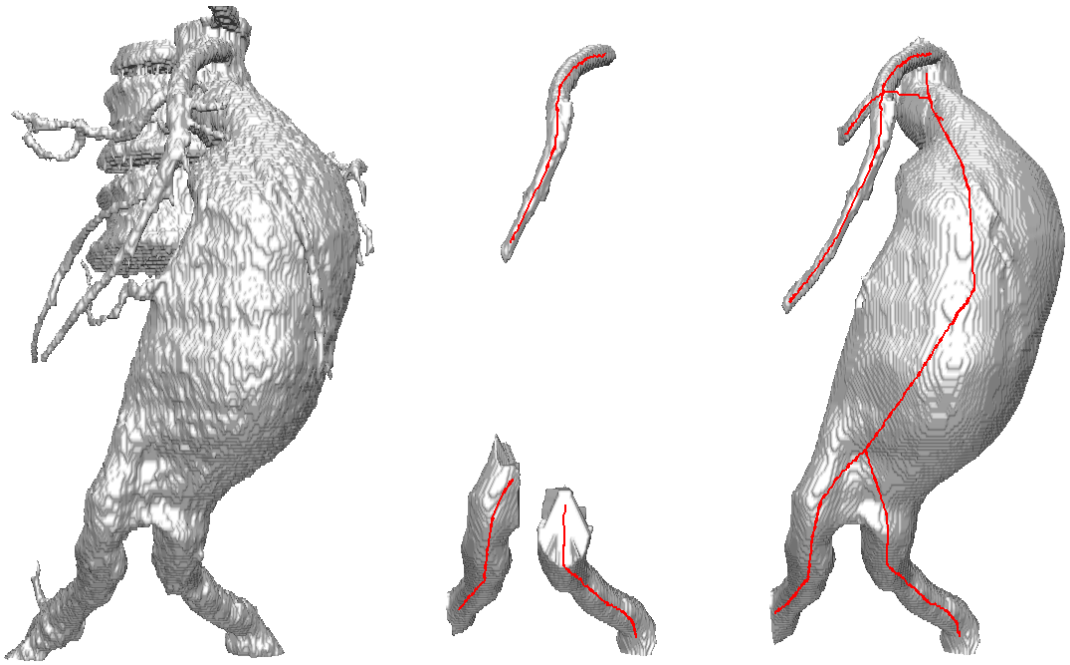

Figure 2.4: Segmentation result of using region growing (left), the circle fitting TDF (middle), and the proposed TDF (right).

more than 10,000 iterations and several minutes of processing to achieve the same result.

In addition to the large size, AAAs often have a non-circular cross-sectional profile. In article **C**, a circle fitting TDF was used to detect tubular structures. This TDF assumes a circular cross-sectional profile, which leads to a lower response for non-circular tubular structures. It may also respond to voxels where tubular structures are absent, for instance semi-circles with high contrast can give a medium response. In article **D**, a new TDF was proposed as a replacement for this filter, to improve detection of large non-circular tubular structures such as AAAs. Instead of assuming a circular cross-sectional profile, the proposed TDF only assumes a closed profile. The proposed TDF does N line searches in the cross-sectional plane at different angles, where the line search stops when an edge is detected or the maximum radius is reached. If edges are found for all line searches, a TDF response is calculated based on the centralness and the GVF vectors. In article **D**, this new TDF, together with the multigrid GVF, was tested on both synthetic and clinical data. The results showed that the proposed TDF was able to properly detect large non-circular tubular structures, such as AAAs. This was tested on four AAA CT datasets, and compared to algorithms such as seeded region growing and the circle fitting TDF. Figure 2.4 shows the results for one of these datasets. The seeded region growing failed to segment the AAAs due to segmentation leakage to the spine, and the circle fitting TDF was unable to properly detect all the AAAs deviating from a circular cross-sectional profile. The runtime of the proposed methods including the novel TDF, multigrid GVF, centerline extraction and segmentation for these datasets was 4-10 seconds using an AMD Radeon HD7970 GPU.

2.4 Segmentation of ultrasound images

The third research goal was to create real-time, robust and automatic segmentation methods for ultrasound images. As ultrasound is a real-time imaging modality, it is a challenge to segment these images at the same speed they are produced. However, one can assume that the segmented structure in one image frame has only moved slightly in the next frame. The Kalman filter is a method which can use the segmentation result of the previous frames to predict the segmentation of the next. In this thesis, a Kalman filter was used together with a shape model to automatically segment the left ventricle of the heart and blood vessels from ultrasound image sequences in real-time.

2.4.1 Left ventricle of the heart

Volume assessment of the left ventricle (LV) of the heart throughout the cardiac cycle is a routine task in diagnostic cardiology, and important in terms of patient management, outcome, and long-term survival (Norris et al., 1992; White et al., 1987). LV border identification can be a challenging task, because of the low image quality and image artifacts.

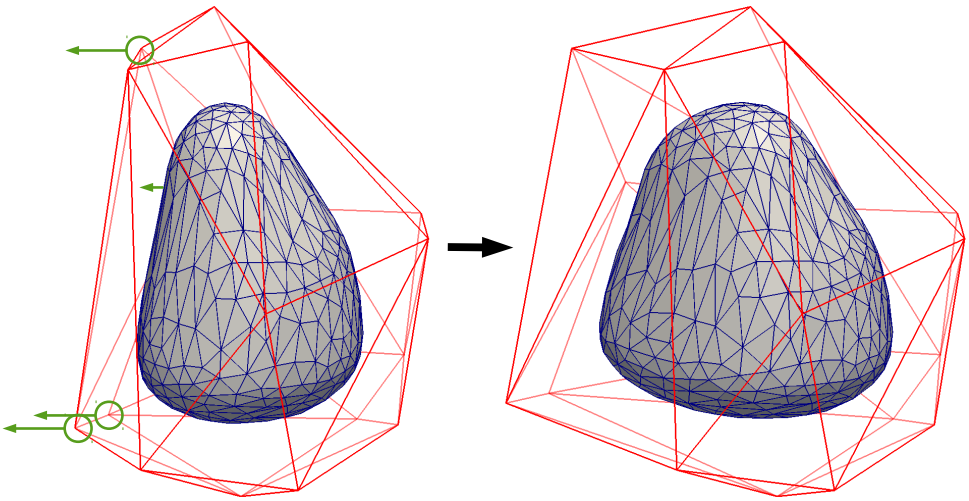

Figure 2.5: Shape model of the left ventricle of the heart. The mesh is deformed with mean value coordinates by moving the control points of the control mesh (red) within the circles (green) to the left. **Left:** The model before deformation. **Right:** The model after deformation.

In article **F**, a method for real-time segmentation of the LV in 3D ultrasound was presented. This is based on the method of Orderud and Rabben (2008), where a shape model is used and transformed with a set of parameters which are estimated using the Kalman filter. The speed of the method is dependent on the number of parameters to be estimated. Orderud et al. used subdivision surfaces for local deformation of the shape model while keeping the number of state parameters low. The proposed method used mean value coordinates (Ju et al., 2005) to deform the shape model, and is able to deform a complex shape with few control points as shown in Figure 2.5. This may prove useful when tracking more complex shapes, in which traditional models such as B-spline and subdivision surfaces will have to use many control points which reduce speed. Mean value coordinates simplifies the shape modelling as only a surface consisting of a set of points is required, and some calculations such as generating surface points for edge detection are avoided. The proposed segmentation algorithm was evaluated in the Challenge on Endocardial Three-dimensional Ultrasound Segmentation (CETUS) at the Medical Image Computing and Computer Assisted Intervention (MICCAI) conference 2014. In this challenge, the accuracy was compared to 8 other automatic and semi-automatic methods. Before the challenge, 30 sequences were made available. The proposed method was able to track the LV in all these sequences, and achieved a mean mesh difference of about 2.5 mm. At the challenge, 15 additional sequences were handed out. After all sequences were processed by all contestants, the proposed method was ranked second in terms of clinical relevance of the fully automatic algorithms, with a mean mesh difference of about 2.72 mm (Bernard et al., 2015). The best algorithm had a mean mesh difference of 2.35 mm. The average runtime per image of the proposed method was measured to be 65 ms, and was the fastest in the CETUS challenge. Most of the proposed Kalman filter method consists of matrix operations. For these operations the Eigen linear algebra library was used. This library uses streaming SIMD extensions (SSE) which allows multiple data to

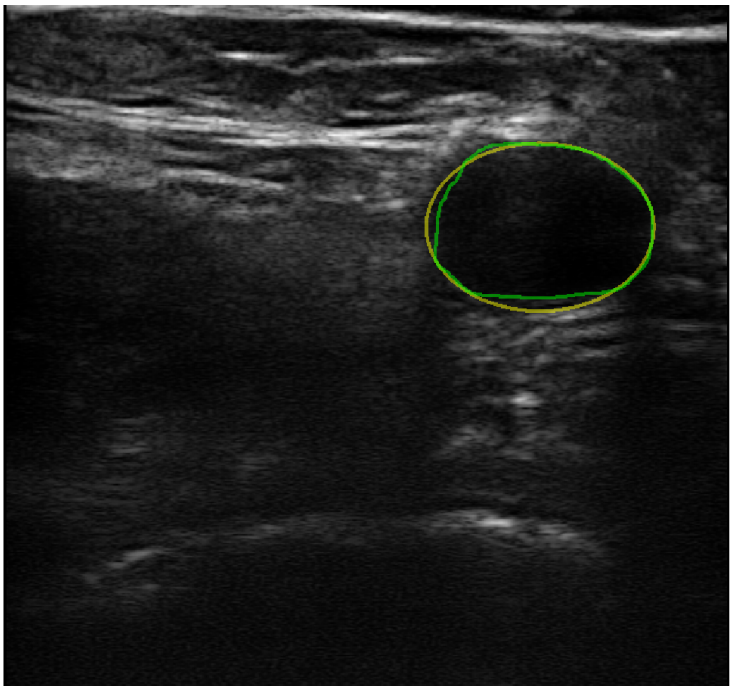

Figure 2.6: Result of the vessel segmentation method.

be processed in parallel.

2.4.2 Blood vessels

Blood vessels appear as black spots in B-mode ultrasound images. Guerrero et al. (2007) presented a method for vessel segmentation and tracking in ultrasound images, using an extended Kalman filter. Their method was fast and accurate, but it had to be manually initialized with a seed point inside the vessel. Article **G** of this thesis proposes a vessel segmentation method, which is initialized automatically using a novel GPU-based vessel detection algorithm. This segmentation method uses a Kalman filter based tracking algorithm, similar to that of Guerrero et al. (2007) and article **F**. The proposed vessel segmentation method is intended for ultrasound-guided regional anaesthesia of the femoral nerve. By segmenting and visualizing the important surrounding structures such as the femoral artery, we hope to improve the success of these procedures.

The vessel cross-section in the ultrasound images is modelled as a compressed circle. The state parameters for the Kalman filter consists of the position, radius and flattening factor. For each pixel, the vessel detection method fits the compressed circle to the image gradients, which is is similar to the circle fitting method in articles **A2** and **C**. A vessel score is calculated for each pixel based on how well the circle fits the gradients. The parameters of the best fitting circle are used as the initial state of the tracking if the vessel score is above a specified threshold. Using a GPU for the circle fitting, real-time performance of about 42 ms on average was achieved for the detection. The tracking was done on the CPU with

the Eigen linear algebra library using about 5 ms on average. A total of 12 ultrasound image sequences from 3 subjects were collected. The number of images per sequence ranged from 110 to 524. For each sequence, the vessel was manually segmented in 4 randomly selected frames. The proposed femoral artery detection and tracking achieved an average dice similarity coefficient of 0.90, mean absolute distance of 0.42 mm, and Hausdorff distance 1.17 mm. Figure 2.6 shows the result of the vessel tracking on ultrasound image of the femoral artery.

2.5 Other contributions

2.5.1 Posters and presentations

In addition to the published articles and proceedings, the following posters and presentations were presented during the work on this thesis:

- Fast Surface Extraction and Visualization of Medical Images Using OpenCL and GPUs
  The Joint Workshop on High Performance and Distributed Computing for Medical Imaging. MICCAI 2011, Toronto, Canada.
- Fast Surface Extraction and Visualization of Medical Images Using OpenCL and GPUs
  Joint National Ph.D. Conference in Medical Imaging and MedViz Conference 2011, Bergen, Norway.
- A New Tube Detection Filter using Gradient Vector Flow, Line Search and Splines
  3rd National PhD Conference in Medical Imaging 2011, Oslo, Norway.
- Airway Tree Segmentation and Centerline Extraction for Image Guided Bronchoscopy
  4th National PhD Conference in Medical Imaging 2012, Trondheim, Norway.
- Segmentation of Abdominal Aortic Aneurysms from CT Images
  5th National PhD Conference in Medical Imaging 2013, Tromsø, Norway.
- FAST - Framework for Heterogeneous Medical Image Computing and Visualization
  Joint National PhD Conference in Medical Imaging and MedViz Conference 2014, Bergen, Norway.

2.5.2 Challenges

Contributions to the following segmentation challenges were submitted including articles based on the result of the challenges:

- VESsel SEgmentation in the Lung 2012 (VESSEL 12) - http://vessel12.grand-challenge.org/. Results published in Rudyanto et al. (2014).
- Challenge on Endocardial Three-dimensional Ultrasound Segmentation 2014 (CETUS) http://www.creatis.insa-lyon.fr/Challenge/CETUS/. Article based on results submitted to Medical Image Analysis (Bernard et al., 2015).

2.5.3 Source code

Most of the source code developed during this study is available at the GitHub page http://github.com/smistad/. The following is a list of some of the source code used in the different publications.

- FAST - Used in articles **B** and **G**.
  http://github.com/smistad/FAST/
- GPU gradient vector flow - Used in articles **A3** and **C**.
  http://github.com/smistad/OpenCL-GVF/
- GPU multigrid gradient vector flow - Used in articles **E** and **D**.
  http://github.com/smistad/Multigrid-GPU-GVF/
- Tube segmentation framework - Used in articles **A2**, **C** and **D**.
  http://github.com/smistad/Tube-Segmentation-Framework/
- GPU-based marching cubes - Used in article **A1**.
  http://github.com/smistad/GPU-Marching-Cubes/

3 Discussion and future work

3.1 Parallel and GPU accelerated image segmentation

Review article **A** showed that most common medical image segmentation methods may benefit from running on GPUs. In the articles on GVF (**A3**, **E**), segmentation of tubular structures (**A2**, **C**, **D**), segmentation of ultrasound images (**G**) and surface extraction (**A1**) it was shown how these algorithms can be accelerated with GPUs.

The number of articles which report using GPUs for acceleration is increasing (Eklund et al., 2013). As the amount and quality of GPU frameworks, drivers and libraries continue to improve, the number of GPU users will probably increase even more. Libraries and frameworks aiding in writing image processing algorithms as well as scheduling, memory management and streaming of dynamic image data, will probably become more important as more algorithms and image data are processed on the GPU. Hybrid solutions using GPUs for the massively data parallel parts, and the CPU for the less parallel parts are possible. One challenge with these hybrid solutions is efficient sharing of data, which currently has to be done explicitly by memory transfer over the PCI express bus. The proposed framework FAST removes the burden of explicit memory transfer by handling it in the core of the framework. The programmer simply request an image or other data for a specific device, and the framework handles the rest. The Heterogeneous System Architecture (HSA) is a new processor architecture proposed by the HSA Foundation, which includes major processor manufacturers such as AMD and ARM. The aim of HSA is to reduce communication latency between CPUs, GPUs and other processors, and make these devices more compatible from a programmer's perspective (HSA Foundation, 2015). AMD has already released a processor named Kavari which uses this new architecture as well as a Linux kernel to utilize the full potential of this new type of processors. A new generation of graphic frameworks have also appeared, including AMD's Mantle (Advanced Micro Devices Inc., 2015) and Apple's Metal (Apple, 2015). This growth in proprietary computing frameworks is worrisome, as it forces developers to create separate code for different processors and operating systems. However, the next generation of the cross-platform OpenGL framework, Vulkan, was just released and is similar to AMD's Mantle framework (The Khronos Group, 2015b). Vulkan offers lower overhead, more direct control over the GPU, lower CPU usage and better support for multi-threading. Thus, Vulkan will be a good candidate for replacing OpenGL in FAST, as high performance concurrent visualization and computation is the goal of this framework. To summarize, a lot

is happening in the field of heterogeneous computing at the moment. When these new programming architectures are in place as well as their low-level programming frameworks HSA, OpenCL and Vulkan, high-level frameworks exploiting these new developments are needed. This is where FAST may play its part in the field of medical image computing and visualization.

Most image processing algorithms are memory bound, thus their speed is highly dependent on the memory bandwidth of the GPUs. The GPU manufacturer NVIDIA has reported working on a solution to boost the memory bandwidth considerably. This solution is called stacked DRAM, and promises several times greater bandwidth, and NVIDIA plans to ship this new technology in 2016 (NVIDIA Corporation, 2014). The speed of data parallel algorithms developed for the GPU also depend on the number of thread processors, which has been steadily increasing. These further developments of GPUs suggest that the speed of algorithms developed for GPUs will increase even more as new GPUs arrive. In article **C**, the same GPU segmentation algorithm was executed on several GPUs. The performance on the newer GPUs were several times higher.

The amount of memory on the GPU has been challenging for medical imaging algorithms. For instance, in the 2004 article Lefohn et al. (2004) had to use a complex streaming method to segment 3D images with a level set method, as the GPU they used only had 128 MB of memory. The proposed methods for GPU-based segmentation of tubular structures included a cropping method which reduced the amount of memory used. The memory on GPUs has increased over the last years. Most consumer cards now have 4 GB of memory, and high-end GPUs may have as much as 16 GB, indicating that the limited amount of memory on GPUs will be less of a problem in the future. Still, reducing the size of datasets by cropping does increase the performance, as less voxels has to processed.

The runtime measurements in article **B** showed that FAST has the potential to become a high performance medical image computing and visualization framework for heterogeneous computer systems. However, the success of this framework depend on further development. To become a valid tool for image guided surgery, FAST needs more algorithms, importers, exporters, streamers and instrument tracking support. The plan is to integrate an OpenIGTLink receiver into FAST. This standardized network protocol supports tracking and image streaming from several different devices (Tokuda et al., 2009).

To achieve the best performance, it is necessary to support heterogeneous processing in the entire framework. This includes all steps in a pipeline from data import, to processing and visualization. There exist a lot of medical image computing and visualization software such as ITK and VTK. One might argue that it is better to focus on adapting these existing frameworks to the new heterogeneous computing world. However, enabling this kind of support in existing frameworks such as ITK and VTK would most likely mean rewriting the entire core of these toolkits.

While FAST provides a high-level interface for creating medical imaging pipelines and managing the data, developers currently have to write the actual image processing algorithms using the low-level language OpenCL. The HIPAcc framework (Membarth et al.,

2012, 2015) provides a high-level domain specific language and a source-to-source compiler to ease the development of image processing algorithms. OpenVX is another open standard for cross platform acceleration of computer vision applications by the Khronos group (The Khronos Group, 2015a), the same organization which have created the OpenCL and OpenGL standards. One possibility is to integrate HIPAcc and OpenVX in FAST. This would allow developers to do high-level development of the actual image processing algorithms, and high-level programming of the entire pipeline from data import to visualization.

3.2 Segmentation of tubular structures

In this thesis, a fast method for extracting different tubular structures from various image modalities was developed (see articles **A2**, **C** and **D**). The proposed method was integrated into the CustusX image guided surgery platform (Langøet al., 2008) and further clinical research on bronchoscopy is currently being conducted. The method is also being integrated into FAST.

The centerline extraction method used is a local greedy optimization algorithm. It is unable to extract all the tubular structures from the TDF result, especially the small tubular structures. Global optimization algorithms may be used for centerline extraction to improve the amount of structures extracted. Two examples of global optimization centerline methods are the ant colony system method by Türetken et al. (2011), and the graph based optimization technique by Graham et al. (2010). However, these global optimization techniques are computational expensive.

Gaussian smoothing is used to reduce the effect of noise in the image, but also destroys important edge information. For small low-contrast tubular structures, Gaussian smoothing may reduce the contrast further or eliminate the structure completely. A possible solution is to replace the Gaussian smoothing with anisotropic smoothing, as suggested by Bauer in his thesis (Bauer, 2010). An anisotropic smoothing filter varies the amount of smoothing for different directions. Thus smoothing more in the direction of the tube, and smoothing less in the cross-sectional plane thereby preserving the tubular structure. Krissian (2002) and Manniesing et al. (2006) proposed two such smoothing filters.

The amount of false positive response was reduced in the proposed TDF in article **D**, but it may still occur in proximity to high contrast edges such as the spine. Further work is needed to reduce the amount of false positive responses, as it makes the centerline extraction less accurate and may lead to extraction of false tubular structures. The circle fitting and proposed TDFs used the local intensity variations and a simple shape model. One way to improve the accuracy, may be to incorporate more shape and appearance information such as the fact that airways in CT are surrounded by a bright airway wall. More shape information can also be incorporated into the centerline extraction step. Bauer et al. (2009b, 2010) used anatomical constraints on the centerline graph branching angle and the radius before and after a bifurcation.

Graham et al. (2010) reported that the choice of reconstruction kernel used in the CT scanner can have a substantial effect on the extraction of small peripheral airways. Soft kernels have a smoothing effect and may blur small airways, while sharp kernels can enhance small airways, but increase high-frequency noise. In this thesis, different reconstruction kernels were not considered, but it is something that could be investigated and may improve the amount of airways extracted from CT images.

The proposed method is able to extract tubular structures from large 3D images in a few seconds using GPUs. This was achieved mainly by reducing the amount of memory used and optimizing the GVF calculations for the GPU. GVF is an algorithm that is also used for active contour segmentation. Thus, the advancements in this thesis on GPU-based GVF will also benefit these segmentation methods. In article, **C** a parallel centerline extraction algorithm was proposed. However, the results showed that this method was not significantly faster than the serial ridge traversal centerline method. 4D ultrasound probes are capable of capturing several volumes per second. The proposed methods may be able to extract tubular structures from ultrasound volumes, and use the result as a shape model and initialization for a Kalman filter based tracking method similar to the ones proposed in this thesis. This could enable real-time tracking of tubular structures in 3D ultrasound and real-time vessel-based registration.

The proposed tubular extraction method may be extended to provide additional information about tubular structures, such as branches, generation number, segment length and radius. It may also be useful to separate different tubular networks such as the hepatic vein and hepatic artery in the liver. TubeTK is an open-source toolkit based on ITK and VTK for the segmentation, registration, and analysis of tubular structures in images (Kitware, 2015). Combining the proposed tubular extraction methods with this toolkit could provide more functionality.

3.3 Segmentation of ultrasound images

Articles **F** and **G** proposed real-time model-based segmentation methods for the left ventricle in 3D ultrasound, and blood vessels in 2D ultrasound. Both these methods use a general approach, in which a shape and appearance model is plugged into a Kalman filter state estimation pipeline as shown in Figure 3.1. The shape model constrains the shape of the segmentation and incorporates knowledge about the anatomy. The appearance model constrains how the input image affect the state, and incorporates knowledge about how the structure of interest appears in the ultrasound image. The state in the Kalman filter describes the segmentation for the current image frame, and includes the position, orientation and deformation of the shape. This general approach should be applicable for segmenting a wide variety of structures from 2D and 3D ultrasound given a suitable shape and appearance model. The Kalman filter incorporates temporal information of the ultrasound videos and has three different steps which are executed in a loop for each image frame. The first step is prediction, which uses the current state to predict the state for the

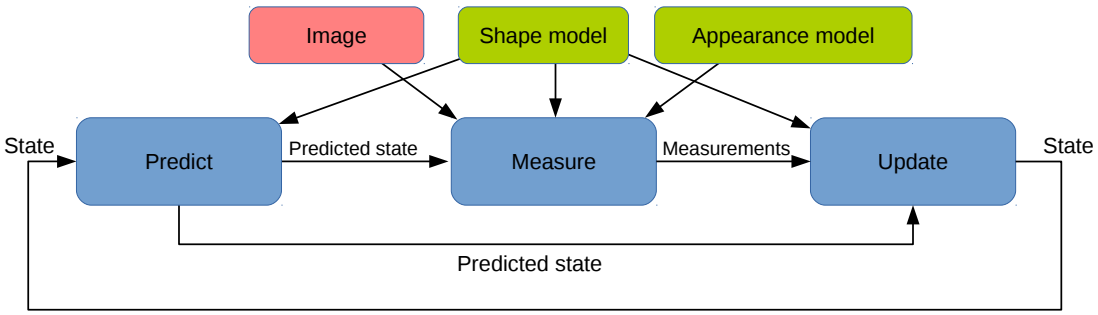

Figure 3.1: Model-based Kalman filter tracking pipeline for segmenting 2D and 3D ultrasound sequences. The pipeline is executed for each frame in the ultrasound image sequence.

next frame. Next is the measurement step that performs measurements on the current image using the predicted state, shape model and appearance model. The final step updates the state using the predicted state, measurements and shape model.

The model-based Kalman filter was tested on 3D ultrasound images of the heart by segmenting the left ventricle (LV). The shape model in this work was a 3D mesh transformed by a linear transformation and deformed with mean value coordinates. The STEP edge detection method was used for the appearance model, requiring the inside of the heart to be darker than the border. While the segmentation of the LV is not necessarily aimed at image guided surgery, the proposed method with the general approach outlined above should be able to efficiently model and track various complex 3D shapes, due to the use of mean value coordinates. This method may prove useful in image guided surgery applications, such as segmentation of the brain ventricles, tumors and blood vessels in 3D ultrasound. More dynamic ultrasound data is required to test this.

An average runtime of 65 ms was achieved on the LV ultrasound datasets from the CETUS challenge. The majority of the runtime was used on performing the edge detection along the mesh model. After the CETUS challenge, GPU acceleration of this step was investigated. The results showed that GPU processing can reduce the average runtime down to about 10 ms on the same data. This was done by doing the edge detection of each line search in parallel. The number of line searches in the LV application was 389. As discussed previously, efficient GPU computation requires many threads. Thus to increase the number of threads, one thread was used per k in the STEP edge detection method, resulting in more than 16 thousand threads. The ultrasound images were also put in the texture memory to optimize memory access.

A fully automatic vessel segmentation method for 2D ultrasound was also developed. In this work, the shape model consisted of a compressed circle, and the appearance model included step and gradient edge detection methods. The method was only tested for tracking a single vessel. However, it should be possible to track multiple vessels by running the vessel detection concurrently with the tracking, and spawning a new Kalman filter and vessel state each time a new vessel is detected. This would also require a way of detecting

when two vessels fuse together at a bifurcation.

One challenge with the model-based Kalman filter segmentation approach is the initialization. On the LV ultrasound data, the LV shape model was initialized to the center of the image. This was possible because the ultrasound probe was positioned in the same way for all sequences. For the blood vessels in 2D ultrasound, an automatic initialization method was developed. This method performs a search for dark circles in the entire image. This was only feasible to do in real-time with a GPU, as each pixel could be processed independently. Doppler image data may be used to improve the vessel detection and tracking methods, but enabling acquisition of Doppler data also reduces the frame rate.

The particle filter (Arulampalam et al., 2002) is another state estimation method, which was not explored in this work. In a linear system with Gaussian noise, the Kalman filter is optimal. However, the measurements used in the proposed methods are not linear, and thus the extended Kalman filter is used which entails approximations. The particle filter may perform better at higher computational cost, as many samples would likely be needed. This method has shown great potential with face tracking, and the tracking can be done in in real-time with GPUs. Also, studies indicate that the unscented Kalman filter may perform better than the extended Kalman filter, in terms of robustness and speed of convergence (Dambreville et al., 2006; Kandepu et al., 2008). The unscented Kalman filter avoids calculation of the Jacobians, which can be difficult to do for a complex model. Thus, the particle filter and the unscented Kalman filter may be viable alternatives to the extended Kalman filter for tracking structures in medical images.

The best performing method in the CETUS challenge used an optical flow tracker with block matching (Barbosa et al., 2014). Block matching is able to track one block of the image to the next by computing a similarity of the intensity values in the blocks. This type of speckle tracking was also used by Orderud et al. (2008) to improve the tracking of the left ventricle. In the proposed methods, speckle tracking was not used. However, these types of measurements may be incorporated into the proposed methods for improved accuracy and can be efficiently computed on a GPU (Kiss et al., 2009).

To summarize, the general model-based Kalman filter approach outlined in Figure 3.1 for segmenting 2D and 3D ultrasound images, is able to track objects in real-time. The challenge lies in designing the shape and appearance model for a given segmentation problem, and to initialize the Kalman filter. Real-time performance can be achieved using fast linear algebra libraries for the Kalman filter, and GPUs for the measurements and initialization.

4 Conclusion

The work documented in this thesis investigates model-based segmentation methods and GPU acceleration for fast, robust and automatic image segmentation within the field of image guided surgery. In this section, the main contributions of this thesis are summarized.

A comprehensive review and several implementations presented in this thesis have demonstrated that most segmentation methods can benefit significantly from GPU computing, due to a high amount of data parallelism. However, the programming of GPUs and multi-core CPUs was found to be challenging due to several factors such as driver errors, explicit memory handling, advanced memory hierarchy and the need for low-level programming. To deal with these challenges, this thesis proposed a framework for efficient medical image computing and visualization on heterogeneous systems consisting of different processors such as multi-core CPUs and GPUs. It was demonstrated how this framework can accelerate several common medical image computing pipelines and outperforms the traditional frameworks ITK and VTK.

A fast and automatic segmentation method for tubular structures was proposed. This method was able to extract different tubular structures such as airways, blood vessels and aneurysms from various image modalities including CT, MRI and ultrasound. Large tubular networks such as the airway tree can be extracted with this GPU-based method in only a few seconds. This was enabled by contributions to acceleration of the time-consuming calculation of gradient vector flow with GPUs.

A general model-based Kalman filtering approach for segmentation of ultrasound images was used to track the left ventricle of the heart and blood vessels. This approach combines temporal information with shape and appearance information to segment ultrasound images in real-time. A method was proposed for tracking objects in 3D ultrasound where the surface deformation method mean value coordinates was used with the model-based Kalman filter approach. This method was evaluated in the Challenge on Endocardial Three-dimensional Ultrasound Segmentation (CETUS) and was ranked second of the fully automatic methods in terms of clinical relevance. A tracking method for the cross-section of blood vessels in 2D ultrasound was also proposed using the same Kalman filter approach. Compared to previous methods which had to be initialized manually, the proposed tracking method was initialized automatically in real-time using a GPU.

The overall conclusion of this thesis is:

- Parallel and GPU computing can significantly accelerate segmentation of medical images.
- A high-level framework for efficient and concurrent medical image computing and visualization on heterogeneous systems is needed.
- Good temporal, appearance and shape models are needed for automatic, robust and accurate segmentation of medical images.

5 Bibliography

Abdel-Dayem, Amr R. and El-Sakka, Mahmoud R. Carotid Artery Ultrasound Image Segmentation Using Fuzzy Region Growing. *Lecture Notes in Computer Science: Image Analysis and Recognition*, 3656:869–878, 2005.

Adams, R. and Bischof, L. Seeded region growing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(6):641–647, June 1994. ISSN 01628828. doi: 10.1109/34.295913.

Advanced Micro Devices Inc., . AMD Accelerated Parallel Processing OpenCL Programming Guide. Technical Report July, 2012.

Advanced Micro Devices Inc., . AMD's Revolutionary Mantle, 2015. http://www.amd.com/en-us/innovations/software-technologies/technologies-gaming/mantle/ - Last accessed 13. April 2015.

Alvarado, Rigo; Tapia, Juan J., and Rolón, Julio C. Medical image segmentation with deformable models on graphics processing units. *The Journal of Supercomputing*, December 2013. ISSN 0920-8542. doi: 10.1007/s11227-013-1042-4.

Amdahl, Gene M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS '67 (Spring)*, pages 483–485, New York, USA, 1967. ACM Press. doi: 10.1145/1465482.1465560.

Apple, . Metal for Developers, 2015. https://developer.apple.com/metal/ - Last accessed 13. April 2015.

Arulampalam, M.S.; Maskell, S.; Gordon, N., and Clapp, T. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2): 174–188, 2002. ISSN 1053587X. doi: 10.1109/78.978374.

Aylward, Stephen R. and Bullitt, Elizabeth. Initialization, noise, singularities, and scale in height ridge traversal for tubular object centerline extraction. *IEEE Transactions on Medical Imaging*, 21(2):61–75, February 2002. ISSN 0278-0062. doi: 10.1109/42.993126.

Barbosa, Daniel; Friboulet, Denis; D'hooge, Jan, and Olivier, Bernard. Fast Tracking of the Left Ventricle Using Global Anatomical Affine Optical Flow and Local Recursive Block Matching. In *Proceedings MICCAI Challenge on Echocardiographic Three-Dimensional Ultrasound Segmentation (CETUS)*, 2014.

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

Bauer, Christian and Bischof, Horst. Edge based tube detection for coronary artery centerline extraction. *The Insight Journal*, 2008a.

Bauer, Christian and Bischof, Horst. A novel approach for detection of tubular objects and its application to medical image analysis. In *Proceedings of the 30th DAGM Symposium on Pattern Recognition*, pages 163–172. Springer, 2008b.

Bauer, Christian and Bischof, Horst. Extracting curve skeletons from gray value images for virtual endoscopy. In *Proceedings of the 4th International Workshop on Medical Imaging and Augmented Reality*, pages 393–402. Springer, 2008c.

Bauer, Christian; Bischof, Horst, and Beichel, Reinhard. Segmentation of airways based on gradient vector flow. In *Proceedings of the 2nd International Workshop on Pulmonary Image Analysis. MICCAI*, pages 191–201. Citeseer, 2009a.

Bauer, Christian; Pock, Thomas; Bischof, Horst, and Beichel, Reinhard. Airway tree reconstruction based on tube detection. In *Proceedings of the 2nd International Workshop on Pulmonary Image Analysis. MICCAI*, pages 203–214. Citeseer, 2009b.

Bauer, Christian; Pock, Thomas; Sorantin, Erich; Bischof, Horst, and Beichel, Reinhard. Segmentation of interwoven 3d tubular tree structures utilizing shape priors and graph cuts. *Medical image analysis*, 14(2):172–184, April 2010. ISSN 1361-8423. doi: 10.1016/j.media.2009.11.003.

Beattie, W. S.; Badner, N. H., and Choi, P. Epidural analgesia reduces postoperative myocardial infarction: a meta-analysis. *Anesthesia and analgesia*, 93:853–858, 2001. ISSN 0003-2999.

Behrens, T.; Rohr, K., and Stiehl, H. S. Robust segmentation of tubular structures in 3-D medical images by parametric object detection and tracking. *IEEE transactions on systems, man, and cybernetics - Part B: Cybernetics*, 33(4):554–561, January 2003. ISSN 1083-4419. doi: 10.1109/TSMCB.2003.814305.

Benmansour, Fethallah and Cohen, Laurent D. Tubular Structure Segmentation Based on Minimal Path Method and Anisotropic Enhancement. *International Journal of Computer Vision*, 92(2): 192–210, March 2010. ISSN 0920-5691. doi: 10.1007/s11263-010-0331-0.

Bernard, O.; Bosch, J. G.; Heyde, B.; Alessandrini, M.; Barbosa, D.; Camarasu-Pop, S.; Cervenansky, F.; Valette, S.; Mirea, O; Bernier, M.; Jodoin, P. M.; Domingos, J. S.; Stebbing, R. V.; Keraudren, K.; Oktay, O.; Caballero, J.; Shi, W.; Rueckert, D.; Milletari, F.; Ahmadi, S. A.; Smistad, E.; Lindseth, F.; van Stralen, M.; Wang, C.; Smedby, Ö.; Papachristidis, A.; Geleijnse, M. L.; Galli, E., and D'hooge, J. Online system for standardized evaluation of algorithms for left ventricular segmentation in 3D echocardiography. 2015.

Billeter, M.; Olsson, O., and Assarsson, U. Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the Conference on High Performance Graphics*, pages 159–166, 2009. ISBN 9781605586038.

Bosch, Johan G.; Mitchell, Steven C.; Lelieveldt, Boudewijn P. F.; Nijland, Francisca; Kamp, Otto; Sonka, Milan, and Reiber, Johan H. C. Automatic segmentation of echocardiographic sequences by active appearance motion models. *IEEE Transactions on Medical Imaging*, 21 (11):1374–1383, 2002. ISSN 02780062. doi: 10.1109/TMI.2002.806427.

Brown, James Anthony and Capson, David W. A Framework for 3D Model-Based Visual Tracking Using a GPU-Accelerated Particle Filter. *IEEE transactions on visualization and computer graphics*, 18(1):68–80, February 2012. ISSN 1941-0506. doi: 10.1109/TVCG.2011.34.

Chan, V. W.; Peng, P. W.; Kaszas, Z.; Middleton, W. J.; Muni, R.; Anastakis, D. G., and Graham, B. A. A comparative study of general anesthesia, intravenous regional anesthesia, and axillary block for outpatient hand surgery: clinical outcome and cost analysis. *Anesthesia and analgesia*, 93:1181–1184, 2001. ISSN 0003-2999.

Chen, C.; Poepping, T.L.; Beech-Brandt, J.J.; Hammer, S.J.; Baldock, R.; Hill, B.; Allan, P.; Easson, W.J., and Hoskins, P.R. Segmentation of arterial geometry from ultrasound images using balloon models. In *2nd IEEE International Symposium on Biomedical Imaging: Nano to Macro*, pages 1319–1322, 2004. ISBN 0-7803-8388-5. doi: 10.1109/ISBI.2004.1398789.

Cleary, Kevin and Peters, Terry M. Image-Guided Interventions : Technology Review and Clinical Applications. *Annual Review of Biomedical Engineering*, 12:119–142, 2010. doi: 10.1146/annurev-bioeng-070909-105249.

Cohen, Laurent D and Deschamps, Thomas. Segmentation of 3D tubular objects with adaptive front propagation and minimal tree extraction for 3D medical imaging. *Computer methods in biomechanics and biomedical engineering*, 10(4):289–305, August 2007. ISSN 1025-5842. doi: 10.1080/10255840701328239.

Cootes, T. F.; Edwards, G. J., and Taylor, C. J. Active appearance models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(6):681–685, 2001.

Dambreville, S.; Rathi, Y., and Tannenbaum, A. Tracking deformable objects with unscented Kalman filtering and geometric active contours. In *American Control Conference*, pages 2856–2861, 2006. ISBN 1-4244-0210-7. doi: 10.1109/ACC.2006.1657152.

Darzi, Sir Ara and Munz, Yaron. The impact of minimally invasive surgical techniques. *Annual review of medicine*, 55:223–237, 2004. ISSN 0066-4219. doi: 10.1146/annurev.med.55.091902.105248.

Dolan, John; Williams, Anne; Murney, Eileen; Smith, Malcolm, and Kenny, Gavin N C. Ultrasound Guided Fascia Iliaca Block: A Comparison With the Loss of Resistance Technique. *Regional Anesthesia and Pain Medicine*, 33(6):526–531, 2008. ISSN 10987339. doi: 10.1016/j.rapm.2008.03.008.

Dyken, Christopher; Ziegler, Gernot; Theobalt, Christian, and Seidel, Hans-Peter. High-speed Marching Cubes using HistoPyramids. *Computer Graphics Forum*, 27(8):2028–2039, December 2008. ISSN 01677055. doi: 10.1111/j.1467-8659.2008.01182.x.

Eidheim, O.C.; Skjermo, J., and Aurdal, L. Real-time analysis of ultrasound images using GPU. *International Congress Series*, 1281:284–289, May 2005. ISSN 05315131. doi: 10.1016/j.ics.2005.03.187.

Eklund, Anders; Dufort, Paul; Forsberg, Daniel, and Laconte, Stephen M. Medical image processing on the GPU - Past, present and future. *Medical image analysis*, 17(8):1073–1094, June 2013. ISSN 1361-8423. doi: 10.1016/j.media.2013.05.008.

Erdt, Marius; Raspe, Matthias, and Suehling, Michael. Automatic hepatic vessel segmentation using graphics hardware. In *Proceedings of the 4th international workshop on Medical Imaging and Augmented Reality*, pages 403–412, 2008.

Frangi, A.; Niessen, W.; Vincken, K., and Viergever, M. Multiscale vessel enhancement filtering. *Medical Image Computing and Computer-Assisted Interventation*, 1496:130–137, 1998.

Gonzalez, Rafael C. and Woods, Richard E. *Digital Image Processing*. Pearson Prentice Hall, third edition, 2008.

Graham, Michael W.; Gibbs, Jason D.; Cornish, Duane C., and Higgins, William E. Robust 3-D airway tree segmentation for image-guided peripheral bronchoscopy. *IEEE transactions on medical imaging*, 29(4):982–997, April 2010. ISSN 1558-0062. doi: 10.1109/TMI.2009.2035813.

Griffin, J and Nicholls, B. Ultrasound in regional anaesthesia. *Anaesthesia*, 65 Suppl 1:1–12, 2010. ISSN 1365-2044. doi: 10.1111/j.1365-2044.2009.06200.x.

Gronningsaeter, Aage; Kleven, Atle; Ommedal, Steinar; Aarseth, Tore Erling; Lie, Torgrim; Lindseth, Frank; Langø, Thomas, and Unsgå rd, Geirmund. SonoWand, an ultrasound-based neuronavigation system. *Neurosurgery*, 47(6):1373–1380, 2000. ISSN 0148396X. doi: 10.1097/00006123-200012000-00021.

Guerrero, Julian; Salcudean, Septimiu E.; McEwen, James a.; Masri, Bassam a., and Nicolaou, Sawakis. Real-time vessel segmentation and tracking for ultrasound imaging applications. *IEEE Transactions on Medical Imaging*, 26(8):1079–1090, 2007. ISSN 02780062. doi: 10.1109/TMI.2007.899180.

Hall, Walter A. and Truwit, Charles L. Intraoperative MR-guided neurosurgery. *Journal of Magnetic Resonance Imaging*, 27:368–375, 2008. ISSN 10531807. doi: 10.1002/jmri.21273.

Hamarneh, Ghassan and Jassi, Preet. VascuSynth: simulating vascular trees for generating volumetric image data with ground-truth segmentation and tree analysis. *Computerized medical imaging and graphics*, 34(8):605–616, December 2010. ISSN 1879-0771. doi: 10.1016/j.compmedimag.2010.06.002.

Han, X.; Xu, C., and Prince, J. L. Fast numerical scheme for gradient vector flow computation using a multigrid method. *Image Processing, IET*, 1(1):48–55, 2007. doi: 10.1049/iet-ipr.

Hassouna, M.S. and Farag, A.A. On the extraction of curve skeletons using gradient vector flow. In *IEEE 11th International Conference on Computer Vision*, pages 1–8. IEEE, 2007. ISBN 978-1-4244-1630-1. doi: 10.1109/ICCV.2007.4409112.

He, Zhiyu and Kuester, Falko. GPU-Based Active Contour Segmentation Using Gradient Vector Flow. In *Advances in Visual Computing*, pages 191–201, 2006.

Heimann, Tobias and Meinzer, Hans-Peter. Statistical shape models for 3D medical image segmentation: a review. *Medical image analysis*, 13(4):543–563, August 2009. ISSN 1361-8423. doi: 10.1016/j.media.2009.05.004.

Helmberger, M.; Urschler, M.; Pienn, M.; Bálint, Z.; Olschewski, A., and Bischof, H. Pulmonary Vascular Tree Segmentation from Contrast-Enhanced CT Images. In *Proceedings of the 37th Annual Workshop of the Austrian Association for Pattern Recognition*, pages 1–10, April 2013.

Hennersperger, Christoph; Baust, Maximilian; Waelkens, Paulo; Karamalis, Athanasios, and Ahmadi, Seyed-ahmad. Multi-Scale Tubular Structure Detection in Ultrasound Imaging. *IEEE Transactions on Medical Imaging*, 34(1):13–26, 2015. doi: 10.1109/TMI.2014.2340912.

HSA Foundation, . What is Heterogeneous System Architecture (HSA), 2015. http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/ - Last accessed 13. April 2015.

Ibanez, Luis and Schroeder, William. *The ITK Software Guide*. Kitware, 2.4 edition, 2004.

Ju, Tao; Schaefer, Scott, and Warren, Joe. Mean Value Coordinates for Closed Triangular Meshes. *ACM Transactions on Graphics*, 24(3):561–566, 2005.

Kalman, R. E. A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering*, 82(1):35–45, 1960.

Kandepu, Rambabu; Foss, Bjarne, and Imsland, Lars. Applying the unscented Kalman filter for nonlinear state estimation. *Journal of Process Control*, 18:753–768, 2008. ISSN 09591524. doi: 10.1016/j.jprocont.2007.11.004.

Kirbas, Cemil and Quek, Francis. A review of vessel extraction techniques and algorithms. *ACM Computing Surveys*, 36(2):81–121, June 2004. ISSN 03600300. doi: 10.1145/1031120.1031121.

Kiss, G.; Nielsen, E.; Orderud, F., and Torp, H.G. Performance optimization of block matching in 3D echocardiography. In *Ultrasonics Symposium (IUS), 2009 IEEE International*, pages 1403–1406, 2009. ISBN 978-1-4244-4389-5. doi: 10.1109/ULTSYM.2009.5441461.

Kitware, . Insight toolkit (ITK), 2014a. http://itk.org/ - Last accessed 18. Aug 2014.

Kitware, . Visualization toolkit (VTK), 2014b. http://www.vtk.org/ - Last accessed 18. Aug 2014.

Kitware, . TubeTK, 2015. http://www.tubetk.org/ - Last accessed 13. April 2015.

Kniemeyer, H. W.; Kessler, T.; Reber, P. U.; Ris, H. B.; Hakki, H., and Widmer, M. K. Treatment of ruptured abdominal aortic aneurysm, a permanent challenge or a waste of resources? Prediction of outcome using a multi-organ-dysfunction score. *European Journal of Vascular and Endovascular Surgery*, 19(October 1995):190–196, 2000. ISSN 10785884. doi: 10.1053/ejvs.1999.0980.

Krissian, Karl. Flux-Based Anisotropic Diffusion Applied to Enhancement of 3-D Angiogram. *IEEE Transactions on Medical Imaging*, 21(11):1440–1442, January 2002. ISSN 1750-8460.

Krissian, Karl; Malandain, Grégoire, and Ayache, Nicholas. Model-Based Detection of Tubular Structures in 3D Images. *Computer Vision and Image Understanding*, 80(2):130–171, November 2000. ISSN 10773142. doi: 10.1006/cviu.2000.0866.

Langø, T.; Tangen, G. A.; Må rvik, R.; Ystgaard, B.; Yavuz, Y.; Kaspersen, J. H.; Solberg, O. V., and Hernes, T. A. N. Navigation in laparoscopy - prototype research platform for improved image-guided surgery. *Minimally Invasive Therapy*, 17(1):17–33, 2008. ISSN 1365-2931. doi: 10.1080/13645700701797879.

Lefohn, Aaron E.; Kniss, Joe M.; Hansen, Charles D., and Whitaker, Ross T. A streaming narrow-band algorithm: interactive computation and visualization of level sets. *IEEE transactions on visualization and computer graphics*, 10(4):422–33, 2004. ISSN 1077-2626. doi: 10.1109/TVCG.2004.2.

Leira, Ho. *Development of an image guidance research system for bronchoscopy*. PhD thesis, Norwegian University of Science and Technology, 2012.

Lenz, Claus; Panin, Giorgio, and Knoll, Alois. A GPU-accelerated particle filter with pixel-level likelihood. In *Vision, Modeling, and Visualization*, pages 235–241, 2008.

Lesage, David; Angelini, Elsa D; Bloch, Isabelle, and Funka-Lea, Gareth. A review of 3D vessel lumen segmentation techniques: models, features and extraction schemes. *Medical image analysis*, 13(6):819–845, December 2009. ISSN 1361-8423. doi: 10.1016/j.media.2009.07.011.

Li, Chunming; Xu, Chenyang; Konwar, Kishori M., and Fox, Martin D. Fast Distance Preserving Level Set Evolution for Medical Image Segmentation. In *9th International Conference on Control, Automation, Robotics and Vision*, pages 1–7. Ieee, 2006. ISBN 1-4244-0341-3. doi: 10.1109/ICARCV.2006.345357.

Li, Hua and Yezzi, Anthony. Vessels as 4-D curves: global minimal 4-D paths to extract 3-D tubular surfaces and centerlines. *IEEE transactions on medical imaging*, 26(9):1213–23, September 2007. ISSN 0278-0062. doi: 10.1109/42.750253.

Lo, Pechin; Ginneken, Bram Van; Reinhardt, Joseph M., and de Bruijne, Marleen. Extraction of Airways from CT (EXACT'09). In *Second International Workshop on Pulmonary Image Analysis*, pages 175–189, 2009.

Lorensen, W.E. and Cline, H.E. 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. ISBN 0897912276.

Lorigo, L. M.; Faugeras, O.; Grimson, W. E. L.; Keriven, R.; Kikinis, R.; Nabavi, A, and Westin, C-F. Codimension-two geodesic active contours for the segmentation of tubular structures. In *Computer Vision and Pattern Recognition*, pages 444–451, 2000.

Lozano, O. M. and Otsuka, K. Simultaneous and fast 3D tracking of multiple faces in video by GPU-based stream processing. In *Acoustics, Speech and Signal Processing*, pages 713–716, 2008. ISBN 1424414849.

Manniesing, Rashindra; Viergever, Max A., and Niessen, Wiro J. Vessel enhancing diffusion. A scale space representation of vessel structures. *Medical Image Analysis*, 10:815–825, 2006. ISSN 13618415. doi: 10.1016/j.media.2006.06.003.

Mao, Fei; Gill, Jeremy; Downey, Donal, and Fenster, Aaron. Segmentation of carotid artery in ultrasound images: Method development and evaluation technique. *Medical Physics*, 27(8): 1961–1970, 2000. doi: 10.1118/1.1287111.

Mateo Lozano, Oscar and Otsuka, Kazuhiro. Real-time Visual Tracker by Stream Processing. *Journal of Signal Processing Systems*, 57(2):285–295, July 2008. ISSN 1939-8018. doi: 10.1007/s11265-008-0250-2.

McCool, Michael D. Scalable Programming Models for Massively Multicore Processors. *Proceedings of the IEEE*, 96(5):816–831, May 2008. ISSN 0018-9219. doi: 10.1109/JPROC.2008.917731.

Membarth, Richard; Hannig, Frank; Teich, Jürgen; Körner, Mario, and Eckert, Wieland. Generating Device-specific GPU code for Local Operators in Medical Imaging. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 569–581, 2012.

Membarth, Richard; Reiche, Oliver; Hannig, Frank; Teich, Jürgen; Körner, Mario, and Eckert, Wieland. HIPAcc: A Domain-Specific Language and Compiler for Image Processing. *IEEE Transactions on Parallel and Distributed Systems*, 2015. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2394802.

Mikić, I; Krucinski, S, and Thomas, J D. Segmentation and tracking in echocardiographic sequences: active contours guided by optical flow estimates. *IEEE transactions on medical imaging*, 17(2):274–284, 1998. ISSN 0278-0062. doi: 10.1109/42.700739.

Montemayor, A.S.; Pantrigo, J.J.; Cabido, R., and Payne, B. Bandwidth improved GPU particle filter for visual tracking. In *Ibero-American Symposium on Computer Graphics SIACG*, 2006.

Morioka, Craig; Chan, K. K., and Huang, H. K. Development of an Image Segmentation and Registration Algorithm on a SIMD Parallel Processor. In *Medical Imaging*, volume 1233, pages 51–57. International Society for Optics and Photonics, 1990.

Murphy-Chutorian, Erik and Trivedi, Mohan M. Particle filtering with rendered models: A two pass approach to multi-object 3D tracking with the GPU. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–8, 2008. ISBN 978-1-4244-2339-2. doi: 10.1109/CVPRW.2008.4563102.

Narayanaswamy, Arunachalam; Dwarakapuram, Saritha; Bjornsson, Christopher S.; Cutler, Barbara M.; Shain, William, and Roysam, Badrinath. Robust adaptive 3-D segmentation of vessel laminae from fluorescence confocal microscope images and parallel GPU implementation. *IEEE Transactions on Medical Imaging*, 29(3):583–597, March 2010. ISSN 1558-254X. doi: 10.1109/TMI.2009.2022086.

Nimsky, Christopher; Ganslandt, Oliver; Hastreiter, Peter, and Fahlbusch, Rudolf. Intraoperative compensation for brain shift. *Surgical Neurology*, 56(01):357–364, 2001. ISSN 00903019. doi: 10.1016/S0090-3019(01)00628-0.

Noble, J A. and Boukerroui, D. Ultrasound image segmentation: A survey. *IEEE Transactions on Medical Imaging*, 25(8):987–1010, 2006. ISSN 0278-0062. doi: 10.1109/TMI.2006.877092.

Norris, R. M.; White, H. D.; Cross, D. B.; Wild, C. J., and Whitlock, R. M. L. Prognosis after recovery from myocardial infarction: the relative importance of cardiac dilatation and coronary stenoses. *European Heart Journal*, 13(12):1611–1618, 1992.

NVIDIA Corporation, . OpenCL Best Practices Guide. Technical report, 2010.

NVIDIA Corporation, . NVIDIA Updates GPU Roadmap; Announces Pascal, 2014. http://blogs.nvidia.com/blog/2014/03/25/gpu-roadmap-pascal/ - Last accessed 6. Feb 2015.

Orderud, F. A Framework for Real-Time Left Ventricular Tracking in 3D+T Echocardiography , Using Nonlinear Deformable Contours and Kalman Filter Based Tracking. *Computers in Cardiology*, 33:125–128, 2006.

Orderud, Fredrik and Rabben, Stein Inge. Real-time 3D segmentation of the left ventricle using deformable subdivision surfaces. In *Computer Vision and Pattern Recognition*, pages 1–8, June 2008. ISBN 978-1-4244-2242-5. doi: 10.1109/CVPR.2008.4587442.

Orderud, Fredrik; Hansgå rd, Jø ger, and Rabben, Stein I. Real-time tracking of the left ventricle in 3D echocardiography using a state estimation approach. *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 10(Pt 1):858–865, January 2007.

Orderud, Fredrik; Kiss, Gabriel; Langeland, Stian; Remme, W.; Torp, Hans G., and Rabben, Stein I. Combining edge detection with speckle-tracking cardiac strain assessment in 3D. In *Proceedings - IEEE Ultrasonics Symposium*, pages 1959–1962, 2008. ISBN 978-1-4244-2428-3. doi: 10.1109/ULTSYM.2008.0483.

Owens, J.D.; Houston, M.; Luebke, D.; Green, S.; Stone, J.E., and Phillips, J.C. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. ISSN 0018-9219. doi: 10.1109/JPROC.2008.917757.

Palàgyi, Kàlmàn and Kuba, Attila. A 3D 6-subiteration thinning algorithm for extracting medial lines. *Pattern Recognition Letters*, 19:613–627, 1998. ISSN 01678655. doi: 10.1016/S0167-8655(98)00031-2.

Palágyi, Kálmán and Kuba, Attila. A Parallel 3D 12-Subiteration Thinning Algorithm. *Graphical Models and Image Processing*, 61(4):199–221, July 1999. ISSN 10773169. doi: 10.1006/gmip.1999.0498.

Perrin, M. and Fletcher, A. Laparoscopic abdominal surgery. *Continuing Education in Anaesthesia, Critical Care & Pain*, 4(4):107–110, 2004. ISSN 1472-2615. doi: 10.1093/bjaceaccp/mkh032.

Pham, Dzung L; Xu, Chenyang, and Prince, Jerry L. Current Methods in Medical Image Segmentation. *Biomedical Engineering*, 2:315–337, 2000.

Reinertsen, I; Lindseth, F; Unsgaard, G, and Collins, D L. Clinical validation of vessel-based registration for correction of brain-shift. *Medical image analysis*, 11(6):673–84, December 2007. ISSN 1361-8415. doi: 10.1016/j.media.2007.06.008.

Rodgers, Anthony; Walker, Natalie; Schug, S; Mckee, A; Kehlet, H; Zundert, a Van; Sage, D; Futter, M; Saville, G; Clark, T, and Macmahon, S. Reduction of postoperative mortality and morbidity with epidural or spinal anaesthesia: results from overview of randomised trials. *British Medical Journal*, 321:1493–1497, 2000.

Rudyanto, Rina D; Kerkstra, Sjoerd; van Rikxoort, Eva M; Fetita, Catalin; Brillet, Pierre-Yves; Lefevre, Christophe; Xue, Wenzhe; Zhu, Xiangjun; Liang, Jianming; Öksüz, Ilkay; Ünay, Devrim; Kadipaşaoğlu, Kamuran; Estépar, Raúl San José; Ross, James C; Washko, George R; Prieto, Juan-Carlos; Hoyos, Marcela Hernández; Orkisz, Maciej; Meine, Hans; Hüllebrand, Markus; Stöcker, Christina; Mir, Fernando Lopez; Naranjo, Valery; Villanueva, Eliseo; Staring, Marius; Xiao, Changyan; Stoel, Berend C; Fabijanska, Anna; Smistad, Erik; Elster, Anne C; Lindseth, Frank; Foruzan, Amir Hossein; Kiros, Ryan; Popuri, Karteek; Cobzas, Dana; Jimenez-Carretero, Daniel; Santos, Andres; Ledesma-Carbayo, Maria J; Helmberger, Michael; Urschler, Martin; Pienn, Michael; Bosboom, Dennis G H; Campo, Arantza; Prokop, Mathias; de Jong, Pim a; Ortiz-de Solorzano, Carlos; Muñoz Barrutia, Arrate, and van Ginneken, Bram. Comparing algorithms for automated vessel segmentation in computed tomography scans of the lung: the VESSEL12 study. *Medical image analysis*, 18(7):1217–32, October 2014. ISSN 1361-8423. doi: 10.1016/j.media.2014.07.003.

Scholl, Ingrid; Aach, Til; Deserno, Thomas M., and Kuhlen, Torsten. Challenges of medical image processing. *Computer Science - Research and Development*, 26(1-2):5–13, December 2010. ISSN 1865-2034. doi: 10.1007/s00450-010-0146-9.

Schroeder, Will; Martin, Ken, and Lorensen, Bill. *Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Kitware, 4th edition, 2006.

Sethian, J.A. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, second edition, 1999.

Sharma, Neeraj and Aggarwal, Lalit M. Automated medical image segmentation techniques. *Journal of Medical Physics*, 35(1):3–14, 2010. doi: 10.4103/0971-6203.58777.

Sluimer, Ingrid; Schilham, Arnold; Prokop, Mathias, and van Ginneken, Bram. Computer Analysis of Computed Tomography Scans of the Lung: A Survey. *IEEE transactions on medical imaging*, 25(4):385–405, April 2006. ISSN 0278-0062. doi: 10.1109/TMI.2005.862753.

Spuhler, Christoph; Harders, Matthias, and Székely, Gábor. Fast and Robust Extraction of Centerlines in 3D Tubular Structures Using a Scattered-Snakelet Approach. *Proc. SPIE*, 6144, March 2006. doi: 10.1117/12.653169.

The Khronos Group, . OpenVX, 2015a. https://www.khronos.org/openvx/ - Last accessed 13. April 2015.

The Khronos Group, . Vulkan, 2015b. http://www.khronos.org/vulkan/ - Last accessed 13. April 2015.

Tilton, J.C. Image segmentation by iterative parallel region growing with applications to data compression and image analysis. In *2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 357–360. IEEE, 1988.

Tokuda, Junichi; Fischer, Gregory S.; Papademetris, Xenophon; Yaniv, Ziv; Ibanez, Luis; Cheng, Patrick; Liu, Haiying; Blevins, Jack; Arata, Jumpei; Golby, Alexandra J.; Kapur, Tina; Pieper, Steve; Burdette, Everette C.; Fichtinger, Gabor; Tempany, Clare M., and Hata, Nobuhiko. OpenIGTLink: an open network protocol for image-guided therapy environment. *The International Journal of Medical Robotics and Computer Assisted Surgery*, 5(4):423–434, 2009. ISSN 1478-596X. doi: 10.1002/rcs.

Türetken, Engin; González, Germán; Blum, Christian, and Fua, Pascal. Automated reconstruction of dendritic and axonal trees by global optimization with geometric priors. *Neuroinformatics*, 9(2-3):279–302, September 2011. ISSN 1559-0089. doi: 10.1007/s12021-011-9122-1.

Unsgaard, Geirmund; Ommedal, Steinar; Muller, Tomm; Gronningsaeter, Aage, and Hernes, Toril A. Nagelhus. Neuronavigation by intraoperative three-dimensional ultrasound: Initial experience during brain tumor resection. *Neurosurgery*, 50(4):804–812, 2002. ISSN 0148396X. doi: 10.1097/00006123-200204000-00022.

Urwin, S C; Parker, M J, and Griffiths, R. General versus regional anaesthesia for hip fracture surgery: a meta-analysis of randomized trials. *British journal of anaesthesia*, 84(4):450–455, 2000. ISSN 0007-0912.

Vijayan, Sinara; Reinertsen, Ingerid; Hofstad, Erlend Fagertun; Rethy, Anna; Hernes, Toril A Nagelhus, and Langø, Thomas. Liver deformation in an animal model due to pneumoperitoneum assessed by a vessel-based deformable registration. *Minimally Invasive Therapy*, 23(5): 279–286, 2014. doi: 10.3109/13645706.2014.914955.

White, H D; Norris, R M; Brown, M a; Brandt, P W; Whitlock, R M, and Wild, C J. Left ventricular end-systolic volume as the major determinant of survival after recovery from myocardial infarction. *Circulation*, 76(January 1977):44–51, 1987. ISSN 0009-7322. doi: 10.1161/01.CIR.76.1.44.

Xie, Jun; Jiang, Yifeng, and Tsui, Hung-tat. Segmentation of kidney from ultrasound images based on texture and shape priors. *IEEE transactions on medical imaging*, 24(1):45–57, 2005. ISSN 0278-0062. doi: 10.1109/TMI.2004.837792.

Xie, Wenjie; Thompson, Robert P., and Perucchio, Renato. A topology-preserving parallel 3D thinning algorithm for extracting the curve skeleton. *Pattern Recognition*, 36:1529–1544, 2003. ISSN 00313203. doi: 10.1016/S0031-3203(02)00348-5.

Xu, Chenyang and Prince, J.L. Snakes, shapes, and gradient vector flow. *IEEE Transactions on Image Processing*, 7(3):359–369, 1998. ISSN 1057-7149.

Zheng, Zuoyong and Zhang, Ruixia. A Fast GVF Snake Algorithm on the GPU. *Research Journal of Applied Sciences, Engineering and Technology*, 4(24):5565–5571, 2012.

Ziegler, Gernot; Tevs, Art; Theobalt, Christian, and Seidel, Hans-Peter. On-the-fly point clouds through histogram pyramids. In *Vision, modeling, and visualization*, pages 137–146. IOS Press, 2006. ISBN 3898380815.

Part II

Selected publications

A

Medical image segmentation on the GPU - A comprehensive review

Authors

Erik Smistad, Thomas L. Falch, Mohammadmehdi Bozorgi, Anne C. Elster and Frank Lindseth

Published in

Medical Image Analysis, volume 20, February 2015, pages 1-18. Elsevier B.V.

Medical Image Segmentation on GPUs - A Comprehensive Review

Erik Smistad[1,2], Thomas L. Falch[1], Mohammadmehdi Bozorgi[1], Anne C. Elster[1], Frank Lindseth[1,2]

1) Norwegian University of Science and Technology

2) SINTEF Medical Technology. Trondheim, Norway

Abstract

Segmentation of anatomical structures, from modalities like computed tomography (CT), magnetic resonance imaging (MRI) and ultrasound, is a key enabling technology for medical applications such as diagnostics, planning and guidance. More efficient implementations are necessary, as most segmentation methods are computationally expensive, and the amount of medical imaging data is growing. The increased programmability of graphic processing units (GPUs) in recent years have enabled their use in several areas. GPUs can solve large data parallel problems at a higher speed than the traditional CPU, while being more affordable and energy efficient than distributed systems. Furthermore, using a GPU enables concurrent visualization and interactive segmentation, where the user can help the algorithm to achieve a satisfactory result. This review investigates the use of GPUs to accelerate medical image segmentation methods. A set of criteria for efficient use of GPUs are defined and each segmentation method is rated accordingly. In addition, references to relevant GPU implementations and insight into GPU optimization are provided and discussed. The review concludes that most segmentation methods may benefit from GPU processing due to the methods' data parallel structure and high thread count. However, factors such as synchronization, branch divergence and memory usage can limit the speedup.

1 Introduction

Image segmentation, also called labeling, is the process of dividing the individual elements of an image or volume into a set of groups, so that all elements in a group have a common property. In the medical domain, this common property is usually that elements belong to the same tissue type or organ. Segmentation of anatomical structures is a key enabling technology for medical applications such as diagnostics, planning and guidance. Medical images contain a lot of information, and often only one or two structures

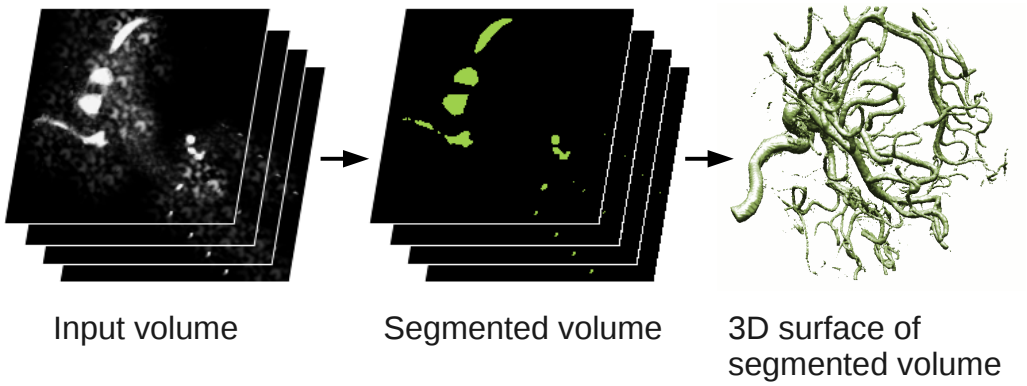

Figure 1: Threshold-based segmentation of a computed tomography (CT) scan. The intensity of each voxel in the input volume is compared to a threshold. If it is higher than the threshold the voxel is segmented as part of the blood vessel. The segmentation result can be used to generate a 3D surface model that can be displayed to the user.

are of interest. Segmentation allows visualization of the structures of interest, removing unnecessary information. Segmentation also enables structure analysis such as calculating the volume of a tumor, and performing feature-based image-to-patient as well as image-to-image registration, which is an important part of image guided surgery. Figure 1 illustrates segmentation of a volume containing blood vessels. The segmentation result, or label volume, is used to create a surface model of the blood vessels using the marching cubes algorithm (Lorensen and Cline (1987)).

Many segmentation methods are computationally expensive, especially when run on large medical datasets. Segmentation of image data, acquired just before the operation as well as during the operation, has to be fast and accurate in order to be useful in a clinical setting. Furthermore, the amount of data available for any given patient is steadily increasing (Scholl et al. (2010)), making fast segmentation algorithms even more important.

Graphic processing units (GPUs) were originally created for rendering graphics. However, in the last ten years, GPUs have become popular for general-purpose high performance computation, including medical image processing. This is most likely due to the increased programmability of these devices, combined with low cost and high performance.

Shi et al. (2012) recently presented a survey on GPU-based medical image computing techniques such as segmentation, registration and visualization. The authors provided several examples on the use of GPUs in these areas. However, only a few segmentation methods are mentioned, and few details on how different segmentation methods can benefit from GPU computing is provided. Pratx and Xing (2011) provided a review on GPU computing in medical physics with focus on the applications image reconstruction, dose calculation and treatment plan optimization, and image processing. A more extensive survey on medical image processing on GPUs was presented by Eklund et al. (2013). They investigated GPU computing in several medical image processing areas such as image

Used in this review	OpenCL	AMD GPUs	NVIDIA(CUDA)
Core	Compute unit	Compute unit	Streaming multiprocessor
Thread processor	Processing element	Stream processor	CUDA Core
Thread	Work-item	Work-item	Thread
Work-group	Work-group	Work-group	Thread block
Atomic Unit of Execution (AUE)	N/A	Wavefront	Warp
Kernel	Kernel	Kernel	Kernel
Shared memory	Local memory	Local data store	Shared memory

Table 1: The different terminology used by different GPU vendors and GPGPU frameworks.

registration, segmentation, denoising, filtering, interpolation and reconstruction.

This review will focus exclusively on medical image segmentation, and thus provide more references and details as well as a comprehensive comparison of the different segmentation algorithms. The goals of this review are to:

1. Give the necessary background information regarding GPU computing, provide a framework for rating how suitable an algorithm is for GPU acceleration, and explain how segmentation methods can be optimized for GPUs. (Section 2)

2. Explain and rate the most common segmentation methods using this framework and provide a survey of how others have accelerated these segmentation methods using GPUs. (Section 3)

2 GPU computing

This section explains the basics of GPUs, and their potential and limitations related to medical image segmentation. An overview of GPU computing, including examples of applications, can be found in Owens et al. (2008). This section may be skipped by readers with a good understanding of GPU computing.

Modern GPUs used for general-purpose computations have a highly data parallel architecture. They are composed of a number of cores, each of which has a number of functional units, such as arithmetic logic units (ALUs). One or more of these functional units are used to process each thread of execution, and these groups of functional units are called thread processors throughout this review. All thread processors in a core of a GPU perform the same instructions, as they share a control unit. This means that GPUs can perform the same instruction on each pixel of an image in parallel. The terminology used in the GPU domain is diverse, and the architecture of a GPU is complex and differs from one model and manufacturer to another. For instance, the two GPU manufacturers NVIDIA and AMD refer to the thread processors as CUDA cores and stream processors, respectively. Furthermore, the thread processors are called CUDA cores in

the CUDA programming language and processing elements in OpenCL (Open Computing Language). Because of this diversity, an overview of the terminology used in this review and by OpenCL, AMD and NVIDIA/CUDA is collected in Table 1.

Thread processors are sometimes referred to as cores, giving the false impression that these cores are similar to the cores of a CPU. The main difference between a thread processor and a CPU core, is that each CPU core can perform different instructions on different data in parallel. This is because each CPU core has a separate control unit. McCool (2008) defined a core as a processing element with an independent flow of control. Following these definitions, this review will refer to the group of thread processors that share a control unit, as cores. GPUs are generally constructed to fit many thread processors on a chip, while CPUs are designed with advanced control units and large caches. At the time of writing, high-end GPUs have several thousand thread processors and around 20 to 40 cores (Advanced Micro Devices (2012)). On the other hand, modern CPUs have around 4 to 12 cores. Figure 2 shows the general layout of a GPU and its memory hierarchy.

The first adopters of GPUs for general-purpose computing had to use frameworks and languages originally designed for graphics, such as OpenGL Shading Language (GLSL) and C for graphics (Cg). As the popularity of GPU programming increased, general-purpose GPU (GPGPU) frameworks such as CUDA and OpenCL were introduced. As opposed to graphic frameworks, these do not require knowledge about the graphics pipeline, and are therefore better suited for general-purpose programming. OpenCL is an open standard for parallel programming on different devices, including GPUs, CPUs and field-programmable gate arrays (FPGAs). OpenCL is supported by many processor manufacturers including AMD, NVIDIA and Intel, while CUDA can only be used with GPUs from NVIDIA.

Image processing libraries that provide GPU implementations of several low-level image processing algorithms are emerging. However, most libraries still lack high-level algorithms such as segmentation methods. Two of the largest image processing libraries, OpenCV and the Insight Toolkit (ITK), both provide a GPU module with support for basic image processing algorithms. A difference between the two toolkits is that OpenCV supports both CUDA and OpenCL, while ITK only supports OpenCL. Accelerated segmentation methods are so far limited to threshold-based segmentation in these libraries. Other GPU-based image processing libraries include NVIDIA Performance Primitives (NPP), ArrayFire, Intel Integrated Performance Primitives (IPP), CUVILIB and OpenCL Integrated Performance Primitives (OpenCLIPP). At the time of writing, these libraries mainly provide GPU accelerated low-level image processing routines.

Several aspects define the suitability of an algorithm towards a GPU implementation. In this review, five key factors have been identified: Data parallelism, thread count, branch divergence, memory usage and synchronization. The following sections will discuss each of these factors, and explain why they are important for an efficient GPU implementation. Furthermore, several levels are defined for each factor (e.g. low, medium, high and none/dynamic), thereby creating a framework for rating to what extent an algorithm can benefit from GPU acceleration.

2.1 Data parallelism

An algorithm that can perform the same instructions on multiple data elements in parallel is said to be *data parallel*, and the set of instructions to be executed for each element is called a *kernel*. Task parallelism on the other hand, is a less restrictive type of parallelism in which algorithms execute different instructions in parallel. As previously discussed, an important characteristic of GPUs is the highly data parallel architecture. Hence, an algorithm has to be data parallel in order to benefit from execution on a GPU. In comparison, task parallel algorithms are more suited for multi-core CPUs.

The degree of speedup achieved by parallelization is limited by the sequential fraction of the algorithm. According to Amdahl's law (Amdahl (1967)), the maximum theoretical speedup of a program where 95% is executed in parallel is a factor of 20, regardless of the number of cores or thread processors being used. The reason for this is that the processing time for the serial part of the code will remain constant. However, in practice the speedup measured and reported in the literature is often much higher than the theoretical limit. There are many reasons for this, one is that the serial version of the program is not fully optimized. Another reason is that the parallel version of the program may use the memory cache more efficiently. Lee et al. (2010) discussed how to make a fair comparison between a CPU and GPU program. Throughout this review the degree of parallelism in a segmentation method is rated as follows:

High: Almost entire method is data parallel (75% - 100%)

Medium: More than half of the method is data parallel (50% - 75%)

Low: None or up to half of the method is data parallel (0% - 50%)

2.2 Thread count

A *thread* is an instance of a kernel. To obtain a substantial speedup of a data parallel algorithm on the GPU, the number of threads has to be high. There are two main reasons for this. Firstly, the clock speed of the CPU is higher than that of the GPU, and secondly global memory access may require several hundred clock cycles (Advanced Micro Devices (2012)), potentially leaving the GPU idle while waiting for data. CPUs attempt to hide such latencies with large data caches. GPUs on the other hand, have a limited cache, and attempt to hide memory latency by scheduling another thread. Thus, a high number of threads are needed to ensure that some threads are ready while the other threads wait. Data parallelism as previously described, is the percentage of the algorithm that is data parallel. Thread count is how many individual parts the calculation can be divided into and executed in parallel.

For most image processing algorithms, each pixel or voxel can be processed independently. This leads to a high thread count, and is a major reason why GPUs are well suited for image processing. For example, an image of size 512x512 would result in 262,144

threads, and a volume of size 256x256x256, almost 17 million threads. The rating of the thread count is defined as follows:

High: The thread count is equal to or more than the number of pixels/voxels in the image

Medium: The thread count is in the thousands

Low: The thread count is less than a thousand

Dynamic: The thread count changes during the execution of the algorithm

2.3 Branch divergence

Threads are scheduled and executed atomically in groups on the GPU. AMD calls these groups *wavefronts* while NVIDIA calls them *warps*. However, in this review they will be referred to as an atomic unit of execution (AUE). An AUE is thus a group of threads that are all executed atomically on thread processors in the same core. The size of these groups may vary for different devices, but at the time of writing it is 32 for NVIDIA GPUs (NVIDIA (2010)) and 64 for AMD GPUs (Advanced Micro Devices (2012)).

Branches (e.g. if-else statements) are problematic because all thread processors that share a control unit have to perform the same instructions. To ensure correct results, the GPU will use masking techniques. If two or more threads in an AUE execute different execution paths, all execution paths have to be performed for all threads in that AUE. Such a branch is called a divergent branch. If the execution paths are short, this may not reduce performance by much.

The following levels are used for branch divergence:

High: More than 10% of the AUEs have branch divergence and the code complexity in the branch is substantial

Medium: Less than 10% of the AUEs have branch divergence, but the code complexity is substantial

Low: The code complexity in the branches is low

None: No branch divergence

2.4 Memory usage

At the time of writing, GPUs with 2 to 4 GB memory are common while some high-end GPUs have 6 to 16 GB. Nevertheless, not all of this memory is accessible from a GPU program, as some of the memory may be reserved for system tasks (e.g. display) or used by other programs. This amount of memory may be insufficient for some segmentation methods that operate on large image datasets, such as dynamic 3D data. The system's main memory can be used as a backup, but this will degrade performance due

to the high latency of the PCIe bus. For iterative methods, this limit can be devastating for performance as data exceeding the limit would have to be streamed back and forth for each iteration. Defining N as the total number of pixels/voxels in the image the rating of memory usage is:

High: More than $5N$

Medium: From $2N$ to $5N$

Low: $2N$ or less

2.5 Synchronization

Most parallel algorithms require some form of synchronization between the threads. One way to perform synchronization is by atomic operations. An operation is atomic if it appears to happen instantaneously for the other threads. This means the other threads have to wait for the atomic operation to finish. Thus, if each thread performs an atomic operation, the operations will be executed serially and not in parallel. Global synchronization is synchronization between all threads. This is not possible to do inside the kernels on the GPU except using atomic operations. Thus global synchronization is generally done by executing multiple kernels which can be expensive. This is due to the need for global memory read and write, double buffering and the overhead of kernel launches. Local synchronization is to perform synchronization between threads in a group. This can be done by using shared memory, atomic operations or the new shuffle instruction (NVIDIA (2013a)). The rating of synchronization is defined in this review as follows:

High: Global synchronization is performed more than hundred times. This is usually true for iterative methods.

Medium: Global synchronization is performed between 10 and 100 times

Low: Only a few global or local synchronizations

None: No synchronization

2.6 Framework

The previous sections covered five criteria, which we argue represent the most important factors affecting GPU performance. Generally, for an algorithm to perform efficiently on a GPU it has to be data parallel, have many threads, no divergent branches, use less memory than the total amount of memory on the GPU and use as little synchronization as possible. However, there are several other factors affecting GPU performance, such as kernel complexity, ALU to fetch ratio, bank conflicts etc. The rating of each segmentation algorithm is summarized in Table 2, along with relevant references. The overall rating of a segmentation algorithm is given by:

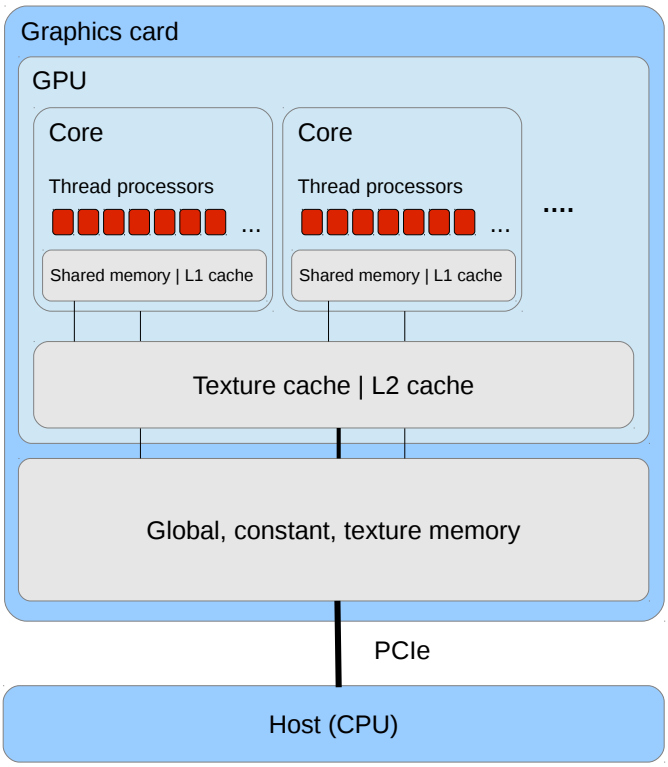

Figure 2: General layout of a GPU and its memory hierarchy. The registers are private to each thread processor, the shared memory is private to each core, and the global, constant and texture memory is accessible from all thread processors. Note that the actual layout is much more complex and differ for each GPU.

High: Large speedup (10 times faster or more)

Medium: Some speedup (2 - 10 times faster)

Low: No substantial speedup (0 - 2 times faster)

2.7 GPU optimization

This section provides some insight on how segmentation methods can be optimized for GPUs.

2.7.1 Grouping

As mentioned in the previous section, threads are scheduled and executed atomically on the GPUs in groups (AUE). GPUs also provide grouping at a higher level, enforced in software and not in hardware like AUEs. These are called thread blocks in CUDA,

and are referred to as work-groups in OpenCL. One benefit of these higher level work-groups is that they are able to access the same shared memory, and thus synchronize among themselves. The size of these work-groups can impact performance, and should be set properly according to guidelines provided by the GPU manufacturers (see Advanced Micro Devices (2012); NVIDIA (2013a)).

2.7.2 Texture, constant and shared memory

In addition to global memory, GPUs often have three other memory types, which can be used to speed up memory access. These memory types are called texture, constant and shared (also called local) memory. They are cached in different ways on the GPU, however, the size of these caches on the GPU are small compared to that of the CPU. Figure 2 show how this memory hierarchy is typically organized on a GPU.

The GPU has a specialized memory system for images, called the texture system. The texture system specializes in fetching and caching data from 2D and 3D textures (NVIDIA (2010); Advanced Micro Devices (2012)). It also has a fetch unit which can perform interpolation and data type conversion in hardware. Using the texture system to store images and volumes can improve performance. Most GPU texture systems support normalized 8 and 16-bit integers. With this format, the data is stored as 8 or 16-bit integers in textures. However, when requested, the texture fetch unit converts the integers to 32-bit floating point numbers with a normalized range. This decreases the memory usage, but also reduces accuracy, and may not be sufficient for all applications.

The constant memory is a cached read-only area of the global off-chip memory. This memory is useful for storing data that remains unchanged. However, the benefit of caching is only achieved when threads in an AUE read the same data elements (Advanced Micro Devices (2012)). On AMD and NVIDIA GPUs the constant cache is smaller than the cache used by the texture system (L1) (Advanced Micro Devices (2012); NVIDIA (2013a)).

The shared memory is a user-controlled cache, also called a scratchpad or local memory. This memory is shared amongst all threads in a group and is local to each core (compute unit) of the GPU.

Generally, the GPU memory that is fastest to access is registers, followed by shared memory, L1 cache, L2 cache, constant cache, global memory and finally host memory (via PCI-express) (Advanced Micro Devices (2012)). The number of registers per core is limited, and exceeding this limit causes register spill, which will reduce performance. To give an impression of the typical size of these memory spaces, the AMD Radeon HD7970 has a 128 kB constant cache for the entire GPU and 64 kB shared memory and 256 kB of registers for each core (Advanced Micro Devices (2012)).

Using as few bits as possible can also speed up processing considerably. Using 8 and 16-bit integers when the range is sufficient instead of the default 32-bit, not only reduces the memory needed, but also memory access latency.

2.7.3 Stream compaction

Some applications may only require a part of the dataset to be processed. This will lead to a branch in the kernel, where one execution path does processing while another does nothing. If threads in the same AUE follow both execution paths, a divergent branch occurs and no time is saved. In these cases, it may be more efficient to remove the unnecessary elements in advance, thus removing the divergent branch. This is called stream compaction, and two such methods are parallel prefix sum (see Billeter et al. (2009) for an overview) and histogram pyramids by Ziegler et al. (2006).

3 Segmentation methods

In this section, several commonly used image segmentation methods are presented and discussed in terms of GPU computing. All of these segmentation methods can be used on both 2D and 3D images, and the terms pixel and voxel are used interchangeably throughout the review.

3.1 Thresholding

Thresholding segments each voxel based on its intensity using one or more thresholds, as shown in Figure 1. In its simplest form, the method performs a binary segmentation using a single threshold T:

$$S(\vec{x}) = \begin{cases} 1 & \text{if } I(\vec{x}) \geq T \\ 0 & \text{else} \end{cases} \tag{1}$$

Where T is the threshold, $I(\vec{x})$ is the intensity of the volume at position $\vec{x}$ and $S(\vec{x})$ is the resulting label or class of the voxel at position $\vec{x}$. As seen in this equation, the method is completely data parallel, since each voxel can be classified independently of all others, and has no need for synchronization. The number of threads needed is equal to the total number of pixels or voxels. While the method contains a divergent branch (a branch were both paths are executed for some AUEs), its simplicity enables the branch to be reduced to a single instruction. The memory usage of the method is low, as only storage for the actual segmentation result is needed, which has the same size as the input image. No references on GPU implementation of this segmentation method are provided as it is trivial to implement on the GPU. An example of a threshold kernel is provided in Algorithm 1. This example uses a single threshold T and a 2D thread ID. It is important to note that this kernel is memory bound because it performs one read and write operation to global memory, which is slower than the comparison operation. The performance may be increased by minimizing the number of global memory accesses. This can be achieved by reading several pixels per thread in each read operation, while at the same time increasing the number of compute operations per memory operation.

Algorithm 1 Thresholding kernel

```
function THRESHOLDINGKERNEL(image, result, T)
    if image(threadID.x, threadID.y) ≥ T then
        result(threadID.x, threadID.y) ← 1
    else
        result(threadID.x, threadID.y) ← 0
    end if
end function
```

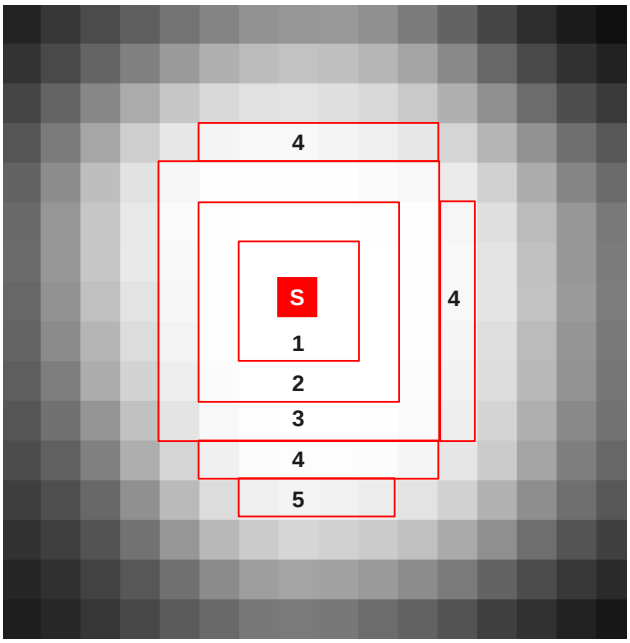

Figure 3: Illustration of parallel region growing with double buffering. The pixel labeled S is the seed pixel. The numbers indicate at which iteration the pixels in the red regions are added to the final segmentation.

3.2 Region growing

Seeded region growing (Adams and Bischof (1994)) is another commonly used segmentation method. This method starts with a set of seed pixels known to be inside the object of interest. The seeds are either set manually using a graphical user interface or automatically using a priori knowledge. From these seeds, regions containing the object of interest will expand to the neighboring pixels if they satisfy one or more predefined criteria. These criteria compare the current pixel to the seed or the pixels already included, using attributes such as intensity, gradient or color. The region will continue to expand as long as there exist neighboring pixels that satisfy the criteria. This method is similar to breadth first search and flood fill algorithms.

Region growing is especially useful when the background and the region of interest have overlapping pixel intensities, and are separated spatially by some wall or region. One

example is thorax CT, where the voxels of the airways and the parenchyma both have low intensities, and are separated by a blood filled tissue with high intensities.

Region growing is a data parallel method as all pixels along the border of the evolving segmentation region are checked using the same instructions. However, as the border expands, the number of threads change. This is problematic because changing the number of threads typically involves restarting the kernel, and this requires reading all the values from global memory again. Nevertheless, the method can be executed on the GPU by having one thread for each pixel in the entire image in each iteration. Figure 3 depicts how the data parallel version of region growing works when double buffering is used. This involves adding more work and introduces branch divergence, limiting the potential speedup over an optimized serial implementation. Furthermore, as this is an iterative method, global synchronization is needed, which also limits the speedup. The memory usage is low (2N), as only the input data and the segmentation result are needed.

An example of a region growing implementation is shown in Algorithm 2. This is based on the parallel breadth first search algorithm by Harish and Narayanan (2007). Segmented voxels are marked with 1, queued voxels with 2 and others 0, in a result data structure Swhich has the same size as the input image. The function $C(\vec{x})$ checks the growing criteria for voxel $\vec{x}$. In this algorithm, texture memory can be used to speed up the global memory access. However, this requires double buffering which increases the memory usage. Shared memory may also be used by first reading global data to shared memory, then grow the region in the area covered by the shared memory and finally write the result back to global memory.

Schenke et al. (2005) implemented seeded region growing on the GPU using GLSL, but provided little description on the implementation. Pan et al. (2008) presented an implementation using CUDA and suggested increasing the number of seeds to make full use of the GPU. Sherbondy et al. (2003) presented a different type of seeded region growing implemented on the GPU with GLSL, which uses diffusion to evolve the segmentation. To reduce unnecessary computations due to branch divergence, their implementation uses a computational mask of active voxels which is updated in each iteration. Chen et al. (2006) presented an implementation of interactive region growing on a GPU. In this implementation, the user marks a region of interest in 2D, which is extruded to 3D. This region of interest is used to create a computational mask that constrains the segmentation. Their implementation also uses GLSL and they reported real-time speeds for medical 3D datasets.

3.3 Morphology

Morphological image processing is often used in combination with other segmentation algorithms such as thresholding, and is therefore included in this review. Examples of morphological techniques include filling holes, and finding the centerline of a segmented tubular structure. See Serra (1986) for a detailed introduction to mathematical morphol-

ogy in computer vision.

Morphological techniques use a mask called a *structuring element* to investigate each pixel. The value of each pixel is determined by the neighboring pixels inside the structuring element. The simplest morphological operations are dilation and erosion. For a binary image, dilation adds all pixels in the structuring element if the current pixel under

Algorithm 2 Parallel region growing

```
function REGIONGROWING(seeds)
    initialize segmentation result S to all zeros
    for each seed voxel s⃗ in parallel do
        % Add seed voxels to the queue
        S(s⃗) ← 2
    end for
    stopGrowing ← false
    while stopGrowing = false do
        stopGrowing ← true
        GROW(S, stopGrowing)
    end while
    return S
end function

function GROW(S, stopGrowing)
    for each voxel x⃗ in parallel do
        if S(x⃗) = 2 then
            % Check growing criteria for voxel x⃗
            if C(x⃗) = true then
                % Add voxel to segmentation
                S(x⃗) ← 1
                for each neighbor voxel y⃗ of x⃗ do
                    if S(y⃗) = 0 then
                        % Add voxel to queue
                        S(y⃗) ← 2
                        stopGrowing ← false
                    end if
                end for
            else
                % Remove voxel from queue
                S(x⃗) ← 0
            end if
        end if
    end for
end function
```

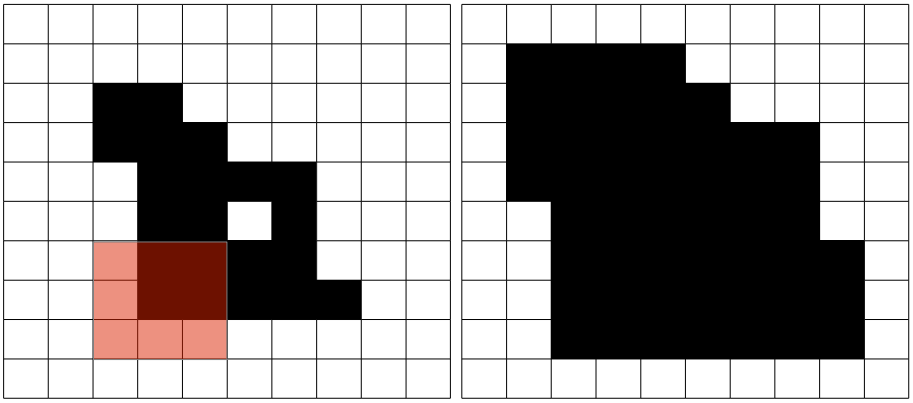

Figure 4: Morphological dilation using a 3x3 square structuring element (shown in red to the left). Since the center pixel is 1, all 0 valued pixels under the structuring element are flipped to 1.

the center pixel in the structuring element is 1 as shown in Figure 4, using a 3x3 square structuring element. Erosion has the opposite effect in which it removes the current pixel with value 1 if there are any pixels in the structuring element that is 0. By combining and repeating these simple operations in addition to other common set operations such as the complement, union and intersection, more advanced operations can be performed.

These morphological operations process each pixel using the same instructions. However, branch divergence limits the speedup, which is also dependent on the size of the structuring element. To avoid reading pixels multiple times from global memory, it can be beneficial to use shared or texture memory. The memory usage is low, as only the image itself and the structuring element is needed for the calculations. Some morphological operations such as thinning, are iterative and therefore require global synchronization.

Morphological operations are a type of stencil operations which can be optimized for GPUs as demonstrated by Holewinski et al. (2012). Eidheim et al. (2005) presented GPU implementations of dilation and erosion using shader programming. They suggested using the shader min and max operations to avoid if-statements. The impact of the structuring element size can be reduced with more advanced methods, such as the Herk-Gil-Werman algorithm (Herk (1992); Gil and Werman (1993)). This was done on the GPU by Thurley and Danell (2012) using CUDA. Morphological operations can be performed on both binary and non-binary images. Karas (2011) presented a GPU implementation of morphological greyscale reconstruction.

3.4 Watershed

The concept of watershed segmentation (Vincent and Soille (1991)) is based on viewing an image as a three dimensional object, where the third dimension is the height of each pixel. This height is determined by the intensity value of the pixel, as shown in Figure 5. In the resulting landscape, there are three types of points. These are determined by the

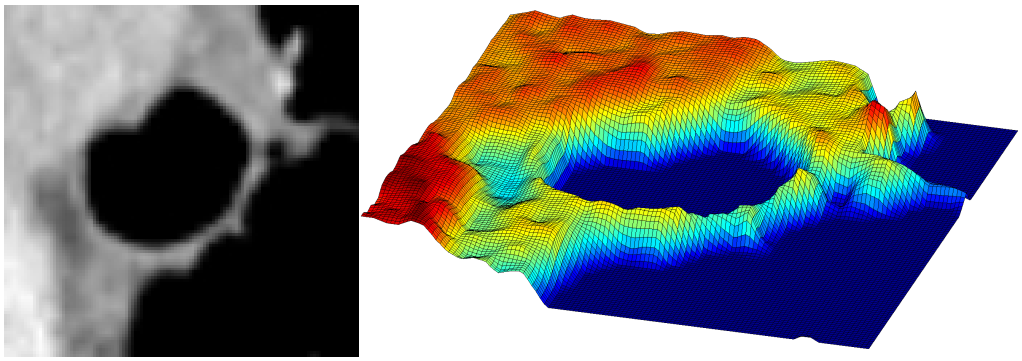

Figure 5: Watershed segmentation. If the intensity values of the pixels of the images on the left are interpreted as heights, it will give create the landscape to the right.

analogy of how a drop of water falling on that specific point would move according to the topographic layout of the landscape:

1. Points that are local minima and where a drop of water would stay in this point

2. Points at which a drop of water would move downwards into one specific local minimum

3. Points at which a drop of water would move downwards into more than one local minimum

The points belonging to type 2 are often called *watersheds* or *catchment basins* and the points belonging to type 3 are often called *divide lines* or *watershed lines*.

The main idea of segmentation algorithms based on these concepts is to find the watershed lines. To find them, another analogy from this topographic landscape is used. Suppose that holes are created in all the points that are local minima, and that water flow through these holes. The watersheds in the topographic landscape will then be flooded at a constant rate. When two watersheds are about to merge, a *dam* is built between them. The height of the dam is increased at the same rate as the water level rises. This process is continued until the water reaches the highest point in the landscape, corresponding to the pixel with maximum intensity. The *dams* then correspond to the *watershed lines*.

For a review of different implementations of watershed segmentation the reader is referred to Roerdink and Meijster (2001). They also investigated parallel implementations of the method, and concluded that parallelization is hard, because of its sequential nature. A parallel implementation is possible by transforming the landscape into a graph, subdividing the image, or flooding each local minimum in parallel. However, Roerdink and Meijster concluded that all of these methods lead to modest speedups. Performing watershed segmentation in a data parallel manner entails adding more work and branch divergence. Thus the speedup over an optimized serial implementation will not be high. This is evident in the literature, where speedups of only 2-7 times are reported.

Algorithm 3 Parallel watershed segmentation using a cellular automaton (Kauffmann and Piche (2008))

for all voxels $\vec{x}$ **in parallel do**
  if $\vec{x}$ is local minima number i **then**
    distance($\vec{x}$) $\leftarrow 0$
    label($\vec{x}$) $\leftarrow i$
  else
    distance($\vec{x}$) $\leftarrow \infty$
    label($\vec{x}$) $\leftarrow 0$
  end if
end for
while convergence is not reached **do**
  for all voxels $\vec{x}$ **in parallel do**
    % $\mathbf{N}$ *is the set of all neighbors of* $\vec{x}$
    $d \leftarrow \min_{\vec{n} \in \mathbf{N}}(\text{distance}(\vec{n}) + \text{cost}(\vec{n}, \vec{x}))$
    $\vec{y} \leftarrow \text{argmin}_{\vec{n} \in \mathbf{N}}(\text{distance}(\vec{n}) + \text{cost}(\vec{n}, \vec{x}))$
    if $d <$ distance($\vec{x}$) **then**
      distance($\vec{x}$)$' \leftarrow d$
      label($\vec{x}$)$' \leftarrow$ label($\vec{y}$)
    end if
  end for
  distance $\leftarrow$ distance$'$
  label $\leftarrow$ label$'$
end while

Kauffmann and Piche (2008) presented a GPU implementation of watershed segmentation using the cellular automaton approach described in Algorithm 3. This method calculates the shortest path from each local minima to all pixels using the Ford-Bellman algorithm. By creating a cost function where the cost of climbing in the landscape is infinite, the shortest path will always lead downwards. Pixels are then assigned the same segmentation label as their closest minima. Using this approach, all the pixels in the image may be processed in parallel using the same instructions. The number of iterations needed to reach convergence depends on the longest path and the branch convergence is high. The memory usage is 4N because of double buffering, and that the distance has to be stored for each pixel. Kauffmann and Piche reported a speedup of 2.5 times, and presented results for 3D images as well.

Pan et al. (2008) presented a CUDA implementation, using a multi-level watershed method. However, few implementation details and results were included. Vitor et al. (2009) created one GPU and one hybrid CPU-GPU implementation. They concluded that the hybrid approach was up to two times faster. Their method initially finds the lowest point from each pixel using a steepest descent traversal. The plateau pixels are then processed to find the nearest border. Finally, the pixels are labeled using a flood fill algorithm from each minimum similar to seeded region growing. Körbes et al. (2009) and Körbes and

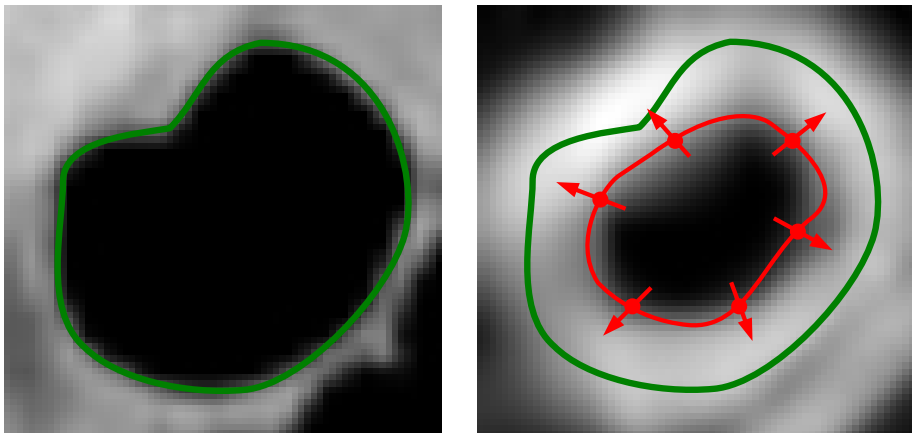

Figure 6: Illustration of active contours. The image to the left is the input image, and the image to the right shows the gradient magnitude of the input image convolved with a Gaussian kernel. The red line superimposed on the right image is the active contour, which is driven towards the high gradient parts of that image, corresponding to the edges in the original image. The green line superimposed on both images show the contour of the lumen.

Vitor (2011) presented an implementation based on the work of Vitor et al. (2009). They also compared performance to the cellular automaton approach by Kauffmann and Piche (2008), and concluded that their implementation was about six times faster than a sequential version. This parallel method also processes each pixel iteratively and suffers from branch divergence. Wagner et al. (2010) processed each intensity level in order starting with the lowest intensity. The labels were merged in each iteration. Their implementation used CUDA, and was 5-7 times faster than a serial implementation on 3D images.

3.5 Active contours

Active contours, also known as snakes, were introduced by Kass et al. (1988). These contours move in an image while trying to minimize their energy, as shown in Figure 6. They are defined parametrically as $v(s) = [x(s), y(s)]$, where $x(s)$ and $y(s)$ are the coordinates for part s of the contour. The energy E of the contour is composed of an internal E_{int} and external energy E_{ext}:

$$E = \int_0^1 E_{\text{int}}(v, s) + E_{\text{ext}}(v(s))ds \tag{2}$$

The internal energy depends on the shape of the contour and can, for example, be defined as:

$$E_{\text{int}}(v, s) = \frac{1}{2}(\alpha|v'(s)|^2 + \beta|v''(s)|^2) \tag{3}$$

where α and β are parameters that control the tension and rigidity of the contour.

The contour can be driven towards interesting features in the image, by having an external energy with low values at the interesting features and high elsewhere. There are

several different choices of external energy. A popular choice is the negative magnitude of the image gradient, i.e. $E_{\text{ext}}(\vec{x}) = -|\nabla[G_\sigma * I(\vec{x})]|^2$, where $G_\sigma *$ is convolution with a Gaussian lowpass filter. This choice of energy drives the contour towards the edges in the image, as depicted in Figure 6. The convolution and gradient calculation can be executed in parallel for each pixel, and optimized using texture or shared memory. A study on how to optimize image convolution for GPUs can be found in the technical report by Podlozhnyuk et al. (2007).

Active contours can be divided into two processing steps. The first is calculating the external energy, and the second is evolving the contour. Both are data parallel operations. The number of threads for calculating the external energy is generally the same as the number of pixels, while the thread count for evolving the contour is lower.

A numerical solution to find a contour that minimize the energy E can be found by making the contour dynamic over time $v(s,t)$.

$$\alpha v''(s,t) - \beta v^{(4)}(s,t) - \nabla E_{\text{ext}} = 0 \tag{4}$$

The Euler equation (4) can be solved on the GPU as done by He and Kuester (2006) and Zheng and Zhang (2012). The thread count is equal to the number of sample points on the contour, which is much lower than the number of pixels in the image. Eidheim et al. (2005) concluded that evolving the active contour on the CPU was faster, as long as the number of points on the contour was below approximately 500. To evolve the contour, each point s has to be extracted from the image using interpolation. Thus, active contours may benefit from using the texture memory, which can perform interpolation efficiently.

Several other formulations of active contours have been implemented on the GPU. Perrot et al. (2011) accelerated a type of active contours that optimizes a generalized log-likelihood function on the GPU. They used a prefix sum algorithm to calculate sums of the image, and shared memory to improve memory access latency. Schmid et al. (2010) implemented a discrete deformable model with several thousand vertices on the GPU using CUDA. Their implementation also allows interactive and concurrent visualization by inserting the vertices into a vertex buffer object, and rendering it with OpenGL. Li et al. (2011) used active contours based on Fourier descriptors implemented on GPUs, for real-time contour tracking in ultrasound video. Kamalakannan et al. (2009) presented a GPU implementation of statistical snakes, which compared the intensity value of each sample point to a seed point. Their implementation was used to assess stains on fabrics.

As shown by Xu and Prince (1998), some different formulations of the external force field ∇E_{ext} may get stuck in local minima, especially if boundary concavities are present. Xu and Prince (1998) introduced a new external force field, gradient vector flow (GVF), which addressed this problem. The GVF field is defined as the vector field $\vec{V}$, that minimizes the energy function E:

$$E(\vec{V}) = \int \mu|\nabla\vec{V}(\vec{x})|^2 + |\vec{V}(\vec{x}) - \vec{V_0}(\vec{x})|^2|\vec{V_0}(\vec{x})|^2 d\vec{x} \tag{5}$$

Algorithm 4 Parallel gradient vector flow using Euler's method

Input: Initial vector field $\vec{V}_0$ and the constant μ.
$\vec{V} \leftarrow \vec{V}_0$
for a number of iterations **do**
  for all voxels $\vec{x}$ **in parallel do**
    $\vec{V}'(\vec{x}) \leftarrow \vec{V}(\vec{x}) + \mu\nabla^2\vec{V}(\vec{x}) -$
      $(\vec{V}(\vec{x}) - \vec{V}_0(\vec{x}))|\vec{V}_0(\vec{x})|^2$
  end for
  $\vec{V} \leftarrow \vec{V}'$
end for

where $\vec{V}_0$ is the initial vector field and μ is an application dependent constant. This equation can be solved using an iterative Euler's method as depicted in Figure 7. This approach differs from other choices of external energy, which are generally not iterative. GVF is thus more time consuming as many iterations are needed to reach convergence. A parallel GPU implementation is possible, as each pixel can be processed independently in each iteration using Algorithm 4. This gives a high thread count and requires global synchronization at each iteration. There is no branch divergence in the calculations, but the memory usage is high, as the method creates a vector for each pixel and requires double buffering. The discrete Laplacian operator in Algorithm 4 is calculated as a stencil operation, which requires access to neighboring pixels. This calculation may benefit from the 2D/3D spatial caching of the texture system. Eidheim et al. (2005), He and Kuester (2006) and Zheng and Zhang (2012) all presented GPU implementations of GVF and active contours for 2D images using shader languages. A GPU implementation of 2D GVF written in CUDA was done by Alvarado et al. (2013). Smistad et al. (2012b) presented an optimized GPU implementation of GVF for 2D and 3D using OpenCL. This implementation use both texture memory and a 16-bit storage format to reduce memory latency.

3.6 Level sets

Similar to active contours, level set methods perform segmentation by propagating a contour in the image (Sethian (1999)). The advantage of level sets compared to the methods in the previous section, is that it allows for splitting and merging of the contours without any additional processing.

Contours in the level set method are represented by the level set function, which is one dimension higher than the contour. Hence, the level set function is a 3D surface when 2D images are being segmented, and a 4D hypersurface for 3D images. The level set function in 2D segmentation, $z = \phi(x, y, t)$, is defined as a function which returns the height z from the position x, y in the image plane to the level set surface at time t. The contour is defined implicitly as the zero level set, which is where the height from the

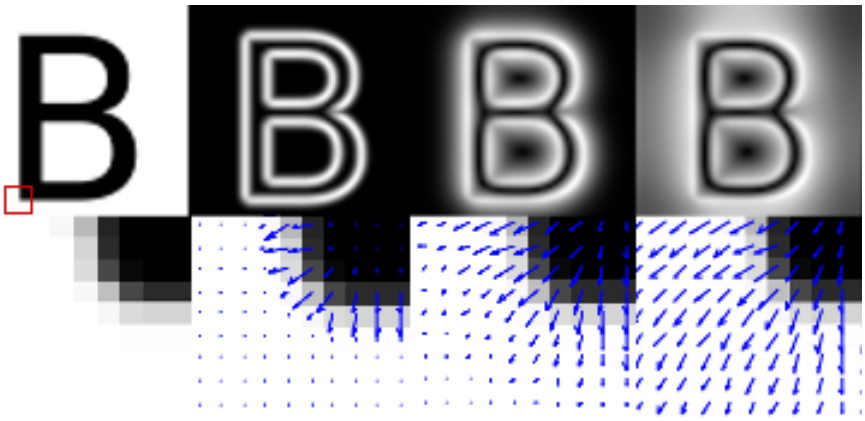

Figure 7: Example of how gradient vector flow diffuses the gradients while preserving the large input gradients. The image to the far left is the input image. The next images depict the magnitude of the vector field after 0, 50 and 500 iterations. The bottom row shows the vector field of the zoomed area indicated by the small red square.

plane to the surface is zero ($\phi(x, y, t) = 0$). This is where the image plane and the surface intersect. To propagate the contour in the x, y plane, the level set surface is moved in the z direction as shown in Figure 8. How fast and in which direction a specific part of the contour moves, is determined by how the level set surface bends and curves. The closer the surface is to being parallel with the image plane, the faster it propagates. When the level set surface is orthogonal to the image plane, the contour does not propagate at all. Assuming that each point on the contour moves in a direction normal to the contour with speed F, the contour can be evolved using the following PDE:

$$\frac{\partial \phi(x, y, t)}{\partial t} = F(x, y, I)|\nabla \phi(x, y, t)| \tag{6}$$

The speed function F varies for different areas of the image I and can be designed to force the contour towards areas of interest and avoid other areas. In image segmentation, the speed function is usually determined by the intensity or gradient of the pixels, and the curvature of the level set function. A negative F makes the contour contract, while a positive F makes it expand.

The level set method starts by setting an initial contour on the object of interest. This is done either manually or automatically using prior knowledge. Next, the level set function is initialized to the signed distance transform of the initial contour. Finally, the contour is updated until convergence.

The PDE above can be solved using an iterative data parallel method, and finite difference methods as shown in Algorithm 5. The thread count is equal to the number of pixels in the image, as the level set function is updated iteratively for each pixel. Rumpf and Strzodka (2001) presented a GPU implementation as early as in 2001. Updating the level set function ϕ for voxels far away from the contour, does not significantly affect the movement of the contour. This observation has lead to two different optimization techniques, known

Algorithm 5 Parallel level sets

Input: Initial segmentation and input image I
Output: Segmentation result S
Initialize ϕ to signed distance transform from the initial segmentation
for a number of iterations or until convergence **do**
  for all voxels $\vec{x}$ **in parallel do**
    Calculate first order derivatives
    Calculate second order derivatives
    Calculate gradient $\nabla\phi(\vec{x})$
    Calculate curvature
    Calculate speed term $F(\vec{x}, I)$
    $\phi'(\vec{x}) \leftarrow \phi(\vec{x}) + \Delta t F(\vec{x}, I)|\nabla\phi(\vec{x})|$
  end for
  $\phi = \phi'$
end for
for all voxels $\vec{x}$ **in parallel do**
  if $\phi(\vec{x}) \leq 0$ **then**
    $S(\vec{x}) \leftarrow 1$
  else
    $S(\vec{x}) \leftarrow 0$
  end if
end for

as *narrow band* and *sparse field*. Both reduce the number of voxels updated in each iteration. The narrow band method updates ϕ only within a thin band around the contour. However, the sparse field method updates ϕ only at the neighbor pixels of the contour. Although these methods reduce the number of threads considerably, they introduce branch divergence. All of these level set methods also require global synchronization after each iteration.

Hong and Wang (2004) used shader programming to create a GPU implementation of level sets for 2D images, and reported a speedup of over 10 times that of a CPU implementation. Cates et al. (2004) presented an interactive application for level set segmentation of 3D images on the GPU. Lefohn et al. (2004) created a GPU implementation for volumes, which was 10-15 times faster than an optimized serial version. They used the narrow band optimization method and streamed only the relevant parts of the volume to the GPU from the CPU. This was done because the GPU memory was too small to fit the entire volume at that time. Jeong et al. (2009) also used the narrow band method. However, they updated the active voxel set on the GPU using atomic operations. Roberts et al. (2010) presented an optimization technique similar to the sparse field method. They used prefix sum scan (see Billeter et al. (2009)) to compact the buffers containing the coordinates of the active voxels on the GPU.

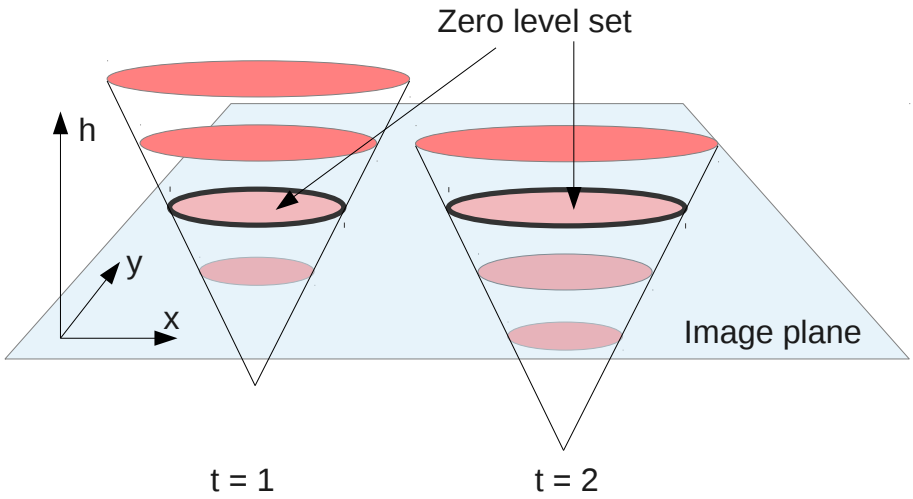

Figure 8: Illustration of level set segmentation. A level set (hyper)surface is moved through the image plane x, y for each time step. The current contour of the segmentation is defined as the location where the height h to the (hyper)surface is zero. This is also called the zero level set. In this example the level set segmentation is a circle that is gradually inflated over time.

3.7 Atlas / registration-based

An atlas is a pre-segmented image or volume. Atlas-based segmentation methods use registration algorithms to find a one-to-one mapping between the atlas and the input image. This mapping is the segmentation result. Each pixel in the input image will have a corresponding pixel and segmentation class in the atlas.

Pham et al. (2000) argued that atlas-based segmentation is generally better suited for segmentation of structures that are stable in the population at study. This makes it easier to create a representative atlas. Still, atlas-based methods can be used as an initialization of other methods, when large variation or pathology (e.g. an MRI scan of a patient with a brain tumor) is present. In addition, atlas-based methods have the advantage that regions may be automatically classified, based on labels from the atlas.

Several registration methods exist, and are often divided into the two categories intensity- and feature-based methods. Intensity-based registration methods use the intensity values in the two images (or image and atlas), and a similarity measure to perform the registration. Feature-based registration methods first extract some common features from the images, and then register the images by matching these features. Mutual information and iterative closest point are the most common intensity- and feature-based registration methods respectively, and both are discussed in more detail below. For even more details on how to accelerate registration methods on the GPU, the reader is referred to Shams et al. (2010a) and Fluck et al. (2011).

Intensity-based registration - Mutual Information

Mutual information (MI) is a measure that can be used to assess how well one image is registered to another. This measure is based on the assumption that regions of similar intensity distribution in one image, correspond to regions with similar intensity distribution in the other image (i.e. a dark region in one image can be similar to a bright region in another image). The MI measure M is based on Shannon's entropy H and is defined as:

$$M(A, B) = H(B) - H(B|A) \tag{7}$$

where A and B are two images. Shannon's entropy is defined as:

$$H(A) = \sum_{i \in A} p_i log(\frac{1}{p_i}) \tag{8}$$

For images, p_i is the probability that the current pixel i in image A has a specific gray value. The probability p_i can be calculated from the histogram of the image. MI can be interpreted as the decrease in uncertainty of image B, when another image A is presented. In other words, if the MI is high, the images are similar.

To register two images using MI, one of the images is transformed to maximize the MI measure. The GPU texture memory has hardware support for interpolation, which is often needed for the image transformations. Different optimization techniques such as gradient descent and Powell's method can be used to find the transformation needed to maximize MI. For a detailed review of registration of medical images using MI see Pluim et al. (2003). The calculation of the MI measure requires summation, which can be done in parallel using the prefix sum scan method. The histogram may be calculated in parallel using sort and count. The number of threads is high, but global synchronization is needed, as this is an iterative method. The optimization techniques gradient descent and Powell's method are not ideal for parallel execution because of their sequential nature (Fluck et al. (2011)). Thus, several GPU-based registration methods run the optimization on the CPU, and the similarity measure on the GPU. Global optimization techniques such as evolutionary algorithms (EAs) are highly amenable to parallelization. However, EAs are generally more computationally expensive, and may be slow even when run in parallel. Lin and Medioni (2008) and Shams and Barnes (2007) presented GPU implementations of the MI computation using CUDA. Shams et al. (2010b) improved their previous implementation by optimizing the histogram computations. This was done using a parallel bitonic sort and count method to avoid performing expensive synchronization and use of atomic counters. With these improvements they reported real-time registration of 3D images, and a 50 times speedup over a CPU version of MI.

Feature-based registration - Iterative closest point

Iterative closest point (ICP) is an algorithm for minimizing the difference between two sets of points. This algorithm was first used for registration by Besl and McKay (1992).

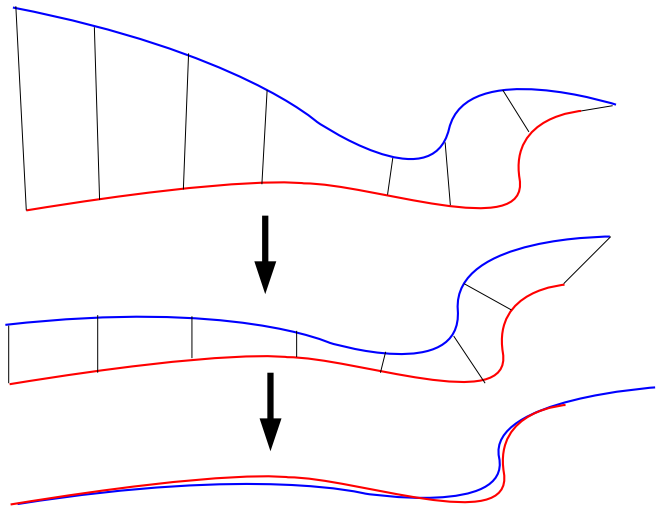

Figure 9: Illustration of the iterative closest point method to align two lines. A set of points is chosen along each line. One of the point sets is iteratively moved and transformed to minimize the distance between each point set.

In order to use this algorithm for registration, corresponding physical points have to be identified in both images. This can be done either manually or by using image processing techniques. The algorithm starts by finding the closest point in the second point set for each point in the first point set. The corresponding points are then used to calculate a transformation, which transforms one of the point sets closer to the other. Transformation parameters are usually estimated using a mean square cost function. This procedure is repeated as long as necessary, and is depicted in Figure 9 for two lines.

Finding the closest points and transforming the corresponding points are both data parallel operations. The thread count is equal to the number of points, which is typically significantly lower than the number of pixels in the image. The memory usage is low, and there is no branch divergence. However, global synchronization is needed at the end of each iteration.

Langis et al. (2001) described a parallel implementation of ICP for clusters where the points were distributed on several nodes. The rigid transformation was computed in parallel using a quaternion-based least squares method. This resulted in an improved speedup due to increased parallelization and reduced communication among the nodes. Qiu et al. (2009) presented a GPU implementation of the ICP algorithm with 88 times speedup over a sequential CPU version.

3.8 Statistical shape models

Several organs in the human body have similar shapes for different individuals. The shape of these organs may be modeled and segmented using a statistical shape model (SSM).

This method creates a statistical model of an organ based on a set of pre-segmented images from several individuals. Segmentation is done by fitting the model to the new image data. The difference between SSMs and atlas models is that SSMs model the shape, while an atlas models the tissue distribution and location of each segmentation class in an image. Nevertheless, one type of SSMs called active appearance models also use intensity information in the image.

Heimann and Meinzer (2009) presented a review on image segmentation using SSMs. They argued that this method is more complex than other methods, but more robust to local image artifacts and noise. An SSM consists of a mean shape and modes of variations. Generally, shapes are represented as a set of landmark points called a point distribution model (PDM). These points have to be present in each training sample, and be located at the same anatomical positions. Setting the landmarks in the training samples can be done manually by an expert. However, this is time consuming, and not practical for large 3D shapes. Thus, automatic methods are often used instead.

After the landmarks have been identified, the shapes of the training samples are aligned using translation, rotation and scaling. The generalized procrustes analysis algorithm (GPA) (Gower (1975)) is often used for this. This algorithm iteratively aligns the shapes to their unknown mean. This entails a series of summations and vertex transformations. All of these calculations are data parallel, and can be performed on the GPU with a thread count equal to the number of landmarks. Next, a shape correspondence algorithm is used to perform registration of all the shapes. The ICP and MI registration algorithms can be used for this (see previous section). Other methods parameterize all shapes to a common base domain, such as a circle for 2D and a sphere for 3D. Corresponding landmarks are then identified as those that are located at the same locations in the base domain. Nevertheless, the initial parameterization of the shapes may not be optimal, and re-parameterization may be needed. Minimum description length (MDL) (Davies et al. (2002)) is an objective function that tries to create optimal landmarks on each shape. This can be used to guide the re-parameterization and give an optimal set of landmarks. Generally, establishing shape correspondence is one of the most challenging tasks of SSMs and one of the major factors influencing the overall result (Heimann and Meinzer (2009)).

After the landmarks have been identified and placed in the same coordinate space, the mean shape and modes of variation can be computed. Assuming that the landmark points are arranged as a single vector $\vec{x}_i = \{(x_1, y_1, z_1), ..., (x_N, y_N, z_N)\}$ of coordinates for each training sample i, the mean shape, $\vec{x}_{\text{mean}}$, can be calculated as the average location of each landmark:

$$\vec{x}_{\text{mean}} = \frac{1}{M} \sum_{i=1}^{M} \vec{x}_i \tag{9}$$

In addition to the mean, a small set of modes which describes the shape variations is calculated. This is usually done with principal component analysis (PCA). Andrecut (2009) and Jošth et al. (2011) both presented a GPU implementation of PCA using CUDA. The amount of speedup depends on the number of landmark points, and they argued that more than a thousand landmark points are necessary. For large organs such as the liver, several

Algorithm 6 Parallel PCA

Input: Matrix of landmarks for each shape: $\mathbf{X} = [\vec{x}_1, \vec{x}_2, ...\vec{x}_M]$
Output: First c eigenvalues: $\phi_1, \phi_2, ...\phi_c$
$\mathbf{R} = \mathbf{X}$
for $k = 1$ to c **do**
    $\phi_k \leftarrow 0$
    for a maximum number of iterations **do**
        Do several matrix operations in parallel
        which result in a new ϕ'_k
        (see Andrecut (2009) for details)
        if $|\phi_k - \phi'_k| < \epsilon$ **then**
            break
        end if
        $\phi_k \leftarrow \phi'_k$
    end for
    Update residual matrix $\mathbf{R}$
end for

thousand landmarks are often employed (Heimann et al. (2009)). However, there might not be any benefit of GPU execution for small organs, where only a few hundred landmarks are used. Algorithm 6 describes a crude parallel PCA implementation. More details can be found in Andrecut (2009). The implementation is iterative and test for convergence by comparing the absolute difference of the new and old eigenvalue ϕ to a parameter ϵ. The actual computations consist of several matrix operations such as multiplication, addition and transpose, all of which can be executed in parallel on the GPU. There are several GPU libraries that can be used to accelerate these matrix operations. A few examples are ViennaCL, MAGMA, cuBLAS and clBLAS.

After PCA has been performed it is possible to approximate each valid shape using the first c modes

$$\vec{x} = \vec{x}_{\text{mean}} + \sum_{i=1}^{c} \vec{b}_i \vec{\phi}_i \tag{10}$$

where $\vec{b}_i$ is the ith shape parameter and $\vec{\phi}_i$ is the ith of the c eigenvalues obtained by PCA.

The calculations of the mean shape $\vec{x}_{\text{mean}}$ and a specific shape $\vec{x}$ can also be run on the GPU. However, the achievable speedup depends on the number of landmark points, which as discussed above can be low. Nevertheless, the creation of the statistical shape model is ideally done only once in a training phase and is not performed for each new segmentation. It can therefore be done offline, and one can argue that the acceleration of the training phase is not as important as the actual segmentation step in the SSM method.

After the SSM is built, an image is segmented using a search algorithm that tries to match the SSM to the image.

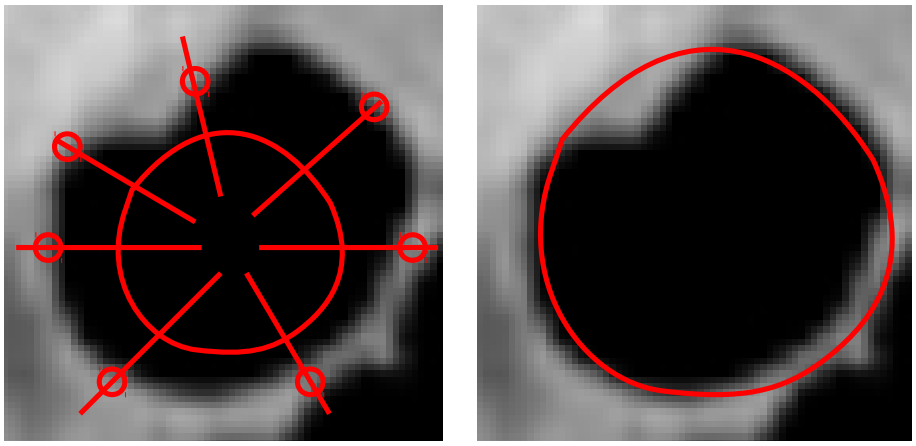

Figure 10: The active shape model algorithm locates borders in a line search from each landmark point on the statistical shape model. A displacement is calculated and the shape is moved, scaled, rotated and deformed to best fit the identified border points. This is repeated until convergence.

Khallaghi et al. (2011) used a registration method based on the linear correlation of linear combination similarity metric. They implemented the registration part on the GPU, while the rest of the SSM method was implemented on the CPU. The registration process entailed simulation of an ultrasound image based on a CT image, and a B-spline deformable registration. They reported a speedup of 350 times in comparison to a CPU implementation. However, they provided few details on the implementation.

Active shape models (ASMs) (Cootes et al. (1995)) is a local search algorithm that searches for contour points along the normal of each landmark point. This is depicted in Figure 10. After a displacement for each landmark point has been calculated, the shape is moved, rotated and scaled. Finally, the shape parameters $\vec{b}_i$ are estimated. This is repeated until the shape change falls below a threshold, which requires global synchronization. ASM is a data parallel method with the thread count equal to the number of landmark points. The memory usage is low, as only the SSM has to be stored.

Another search algorithm for SSMs is active appearance models (AAMs) (Cootes et al. (2001)). AAMs use appearance models to drive the search. These appearance models are able to generate a synthetic image from the current shape. This synthetic image is superimposed on the input image, and used to calculate how well the current shape matches the input image. Finally, this measure is used to estimate the orientation, scale and shape parameters. As with ASM, this is done iteratively, and requires global synchronization. The synthesis of images is done by texture transformation, a task which GPUs excel at due to its data parallel nature and high thread count. Nevertheless, Heimann and Meinzer (2009) argue that AAM is rarely used on 3D images as the memory requirement of AAM is very high.

ASM and AAM have been popular for tracking faces in video. Ahlberg (2002) and Song et al. (2010) presented GPU implementations of AAM and ASM respectively for face tracking. Ahlberg (2002) used OpenGL for the texture mapping in the AAM search.

Song et al. (2010) used the GPU for pre-processing operations such as edge enhancement and tone mapping, and for the ASM search.

3.9 Markov random fields and graph cuts

Markov random field (MRF) segmentation (Wang et al. (2013a)) considers all the pixels in the image as nodes in a graph. All nodes are connected and each pixel has an edge to its neighbor pixels. Each node has a probability distribution associated with it, which consists of the probability of the pixel belonging to each class. These nodes have the Markov property, which states that the probability distribution of a node only depends on its closest neighbors.

MRF segmentation is to find the segmentation S that maximizes the probability $P(S|I)$, where I is the observed image to be segmented. S can express several different segmentation classes for each pixel. This makes MRF segmentation ideal for multi-label segmentation. Using Bayes formula this becomes:

$$P(S|I) = \frac{P(I|S)P(S)}{P(I)} \tag{11}$$

In this formula, $P(I|S)$ is the probability of observing an image I given a segmentation S. $P(S)$ is the probability of a segmentation, and can be used to model how a segmentation result should look like. $P(I)$ is considered to be a normalization constant, and is therefore ignored in the calculations. Structures of interest can be segmented by creating different expressions for $P(I|S)$ and $P(S)$.

There are several methods for maximizing the a posteriori distribution. One method is iterative conditional modes (ICM), which was introduced by Besag (1986). ICM starts with an initial segmentation S, and optimizes the local energy of each pixel deterministically. Thus, each pixel can be processed in parallel. This is repeated until convergence, which requires global synchronization. However, ICM is prone to getting stuck in local minima. Simulated annealing (SA) (Kirkpatrick et al. (1983)) is another optimization method, which can avoid local minima. However, SA generally need a lot more iterations to reach convergence. SA select the class of each pixel stochastically based on a temperature parameter. This temperature is first initialized to a high value, and gradually lowered. This has the effect of allowing the segmentation S to reach many states in the beginning. As the temperature is lowered, the segmentation is gradually restricted to minima states. Both ICM and SA are iterative, and have a medium memory usage as double buffering is required. The thread count is equal to the number of pixels in the image. The branch divergence is low, as the number of instructions in the branches are low.

Griesser et al. (2005) presented a shader implementation of MRF segmentation, but provided few details of their implementation. Valero et al. (2011) implemented a GPU version of the ICM method in the ITK library. They achieved significant speedups, and mention optimizations such as using shared memory and loop unrolling. Jodoin (2006)

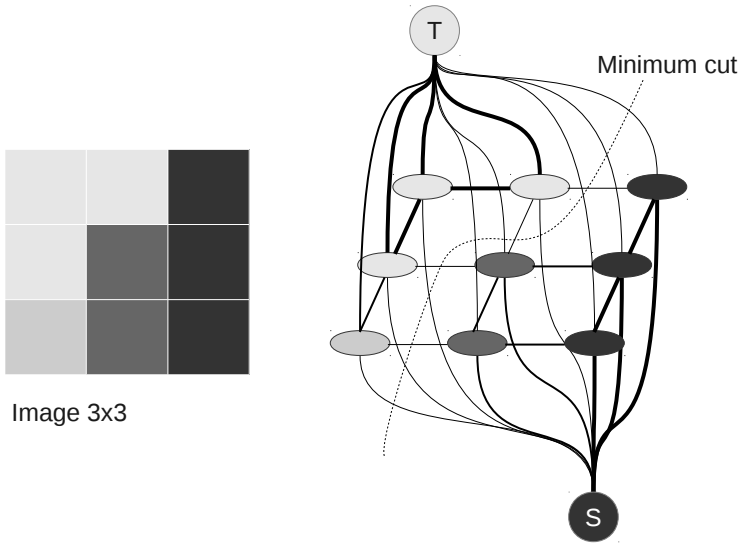

Figure 11: Illustration of graph cut segmentation of a 3x3 image. The image to be segmented is shown to the left, its graph representation on the right. The thickness of the edges indicates their weight.

presented an implementation using NVIDIA's Cg shader language of both SA and ICM. In both cases there is ample parallelism, as there is one thread for each pixel. The result from one iteration is stored in texture memory, so that the neighborhoods of each pixel can be read more efficiently during the next iteration. Walters et al. (2009) presented liver segmentation using ICM and CUDA. They used coalesced reads from global memory to increase performance, and experimented with different thread grouping configurations. Another GPU implementation of ICM based MRF segmentation was presented by Sui et al. (2012). As opposed to the other implementations mentioned here, they did not process pixels with overlapping neighborhoods in parallel. Multiple passes are therefore required for each iteration, and larger images are required for sufficient parallelism.

Modelling $P(S)$ and $P(I|S)$ can require several unknown parameters. These parameters can be estimated using the expectation-maximization (EM) algorithm. This algorithm is an iterative maximum-likelihood method. It requires calculation of the expectation of the conditional distribution $P(S|I)$, which is extremely complex (Zhang (1992)). However, a mean-field approximation can be used to make this calculation feasible Zhang (1992). Saito et al. (2012) presented a GPU implementation of MRF segmentation using the mean-field approximation and CUDA. However, they provided no details on the GPU implementation.

Graph cut (Boykov and Veksler (2006)) is another MRF segmentation method. This method also uses a graph where all the pixels in the image are nodes, and each pixel has an edge to its neighbor pixels. However, all pixels have an additional edge to two special nodes, called a source (S) and sink (T) node. This is depicted in Figure 11. The edges are assigned a weight, so that background pixels have a large weight to one of these nodes, a small weight to the other, and vice versa for the foreground pixels. The weights

of the edges between the pixels are designed to be large between similar pixels, and small between different.

The segmentation is determined using a minimum cut graph algorithm. These algorithms partition the nodes of a graph into two sets. The graph is cut so that the sum of the weights of the cut edges is minimized. The result is a binary segmentation that is optimal in terms of the weights assigned to the edges.

There are several algorithms for finding the minimum cut, and its dual problem maximum flow, where the graph is considered to be a flow network. Two examples are the push-relabel and Ford-Fulkerson algorithms.

The push-relabel method uses two operations, which both are executed for every node in the graph. With one thread for each node, the total number of threads is high. However, there is significant branch divergence, as these operations are only performed for a subset of the nodes during each iteration. The memory usage of this method is high because it has to store several attributes for each edge.

Dixit et al. (2005) presented a GPU implementation of the push-relabel algorithm using shader programming. However, in their comparison with a serial implementation, the GPU implementation was slower except if some approximations were used. Hussein et al. (2007) presented an optimized GPU implementation using CUDA, which was faster than two different serial implementations. Vineet and Narayanan (2008) presented a similar implementation where they improved the performance by using shared and texture memory to speed up memory access. The two previous implementations restrict the graph to a lattice. Garrett and Saito (2009) showed how a GPU implementation of push-relabel could be extended to arbitrary graphs by representing the vertices and edges in a linear array.

An augmenting path is a path in the graph which has available capacity. The Ford-Fulkerson method solves the minimum cut and maximum flow problem by iteratively finding an augmenting path from the source to the sink node. Flow is sent through this path, and this is repeated until no more flow can be sent. This method is not as well suited for data parallel computation as the push-relabel algorithm. However, it is possible to run the method in parallel by splitting the graph and solving each sub-graph in parallel as done by Liu and Sun (2010) and Strandmark and Kahl (2010).

3.10 Centerline extraction and segmentation of tubular structures

Blood vessels, airways, bones, neural pathways and intestines are all examples of important tubular structures in the human body. In addition to the segmentation, the extraction of the centerline of these structures is also important. The centerline is a line that goes through the center and provides a structural representation of the tubular structures (see Figure 12). It is important in several applications such as registration of pre- and intraoperative data, which is a key component in image guided surgery.

There are several methods for extracting tubular structures from medical images. A recent and extensive review on blood vessel extraction was done by Lesage et al. (2009), and an older one was done by Kirbas and Quek (2004). Two reviews on the segmentation of airways were done by Lo et al. (2009) and Sluimer et al. (2006).

A common method for extracting tubular structures is to grow the segmentation iteratively from an initial point or area. For instance using methods such as region growing, active contours and level sets.

A centerline can be extracted from a binary segmentation using iterative morphological thinning, also called skeletonization. With this method, voxels are removed from the segmentation in a particular order until the object can not be thinned anymore. This is an iterative data parallel method with a thread count equal to the size of the volume. The method has branch divergence, because only a subset of the voxels need to be examined at each iteration. Jiménez and Miras (2012) presented a GPU and multi-core CPU implementation of the thinning method by Palágyi and Kuba (1999) using CUDA and OpenCL.

Another approach is to use a distance transform or gradient vector flow (GVF) as done by Hassouna and Farag (2007). As explained previously, computation of GVF can be accelerated on the GPU (Eidheim et al. (2005); He and Kuester (2006); Zheng and Zhang (2012); Smistad et al. (2012b)).

Direct centerline extraction without a prior segmentation is also possible using methods such as shortest path and ridge traversal. Aylward and Bullitt (2002) presented a review of different centerline extraction methods. They proposed an improved ridge traversal method based on a set of ridge criteria, and different methods for handling noise. Bauer and Bischof (2008) showed how this method could be used together with GVF. However, ridge traversal is not a data parallel algorithm and therefore not suited for GPU acceleration.

These methods usually need an initial estimation of candidate centerpoints or the direction of the tubular structure. Tube detection filters (TDFs) are used to detect tubular structures by calculating a probability of each voxel being inside a tubular structure. Most TDFs use gradient information, often in the form of the eigenanalysis of the Hessian matrix. Frangi et al. (1998) presented an enhancement and detection method for tubular structures based on the eigenvalues of this matrix. A similar vessel enhancement method was implemented on the GPU by Wang et al. (2013b) using CUDA. Krissian et al. (2000) created a model-based detection filter that fits a circle to the cross-sectional plane of the tubular structure. These TDFs are data parallel, and are computed for each voxel in the volume. No synchronization is needed, and the memory usage is low, as only one likelihood value has to be stored per voxel.

Erdt et al. (2008) performed the TDF and a region growing segmentation on the GPU and reported a 15 times faster computation of the gradients and up to 100 times faster TDF. Narayanaswamy et al. (2010) did vessel laminae segmentation with region growing and a hypothesis detection on the GPU and reported an 8 times speedup. Bauer et al. used GPU acceleration for the GVF computation in Bauer et al. (2009a), and the TDF calculation in

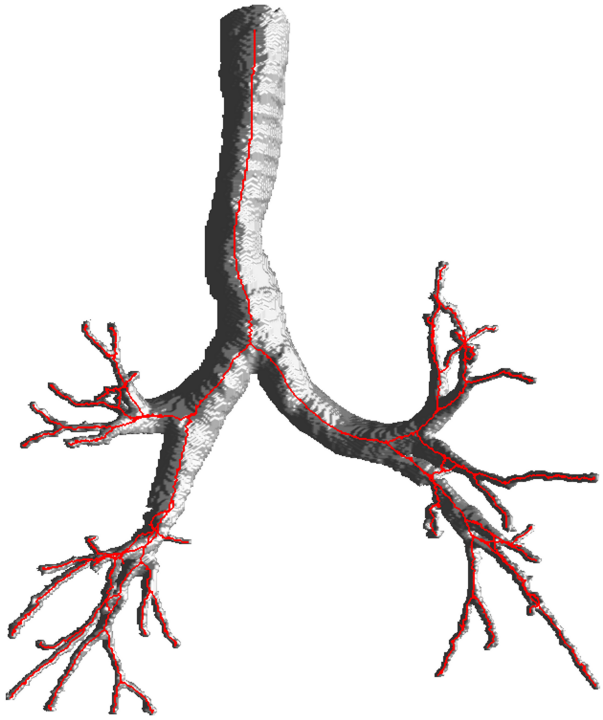

Figure 12: Centerline, displayed in red, of the airway tree. The centerline was extracted using tube detection filters from computed tomography data and the segmentation was created using a region growing algorithm with the centerline as seeds. All the processing was done on the GPU as explained in Smistad et al. (2013).

Bauer et al. (2009b). However, they provided no description of the GPU implementations. Smistad et al. (2012a) presented an implementation of airway segmentation and centerline extraction. In this implementation, dataset cropping, GVF and TDF were executed on the GPU using OpenCL. This implementation was further developed in Smistad et al. (2013) to run completely on the GPU, and process other types of tubular structures such as blood vessels from different organs and modalities.

3.11 Segmentation of dynamic images - Tracking

So far, only segmentation of single images, acquired at one specific time, has been discussed. However, medical image data acquired over time also exist. For instance ultrasound devices captures several images per second. Real-time processing of such data requires streaming of the data directly to the GPU. The segmentation of structures in dynamic image data is often referred to as tracking. One way to do segmentation of dynamic images, is to apply one of the segmentation methods discussed so far on each frame. However, this may not satisfy real-time constraints. Another approach is to use the segmentation of the previous frame to segment the next frame. The segmentation of the previous frame can be used for initialization, or to create some a priori knowledge for

the next frame. Or more advanced statistical state estimation methods can be used, such as Kalman and particle filters. In this section, these two methods will be discussed further. An open source library for tracking called Open Tracking Library (OpenTL) (Panin (2011)) supports GPU processing, and implements both of these methods and others.

3.11.1 Kalman filter

The Kalman filter (Kalman (1960)) is an algorithm that tries to estimate a state using a series of noisy measurements over time. In image segmentation, the state may be a set of parameters describing the transformation of a shape, such as translation, rotation, scaling and deformation. Several types of measurements can be conducted. One type of measurement for object tracking is the offset from each point on the shape to the object's edges in the current image frame. These offsets are found by a line search along the normal in each point, similar to active shape models (ASMs). The measurement process is data parallel, and the thread count is equal to the number of line searches.

The algorithm itself consists of a set of matrix operations, and most of the matrices have sizes dependent on the number of state variables and measurements. Matrix operations such as multiplication, addition and inversion are all data parallel operations, and the thread count is dependent on the matrix size. There exist several linear algebra libraries for the GPU that can be used for acceleration of such operations. A few examples are ViennaCL, MAGMA, cuBLAS and clBLAS.

Thus, segmentation of dynamic images using the Kalman filter is a data parallel operation, and the thread count is dependent on the number of measurements and state variables. These numbers can vary a lot from one application to another. However, they are a lot smaller than the number of voxels. Thus, the thread count is medium. The memory usage is low, as only a few small matrices have to be stored. Some branch divergence may occur on the line searches. For instance if some of the points on the shape are outside of the image. However, the actual algorithm has no or little branch divergence.

Huang et al. (2011) presented a GPU implementation of the Kalman filter written in CUDA. They observed a very large speedup compared to a serial implementation. The number of state variables ranged from 250 to 4500 and measurements from 1000 to 7000.

3.11.2 Particle filter

The particle filter method (Arulampalam et al. (2002)) tries to estimate the posterior density of the state variables given the measurements. This is done by performing a Monte Carlo simulation with a large number of samples, also called particles. Each particle is a possible state for the next time step. The particles are assigned a weight, which determines how well it describes the posterior density. This is done by evaluating how well each particle matches the object in the next image. With a large number of particles this process can be computationally expensive. However, each particle can be processed in

parallel, and an estimate of the next state can be determined by calculating a weighted sum of these particles. Thus, the method is highly data parallel. The thread count is equal to the number of particles. A high particle count generally gives better results, and a couple of thousand particles seems to be common (Montemayor et al. (2006); Brown and Capson (2012)). The memory usage is dependent on how the weight calculation is implemented. For instance, Brown and Capson (2012) generated an image for each particle, and compared each of these synthetic images to the next image, which gave a high memory usage. The rest of the method uses little memory. The same applies for the branch divergence.

Several GPU implementations of particle filtering have been reported, and have primarily focused on accelerating the expensive weight calculation step. Montemayor et al. (2006) used Cg and achieved real-time speeds with up to 2048 particles on a stream of 2D images with the size 320x240. Mateo Lozano and Otsuka (2008) and Lozano and Otsuka (2008) implemented face tracking on a stream of images with size 1024x768 using CUDA. Murphy-Chutorian and Trivedi (2008) and Lenz et al. (2008) did face tracking using GLSL. Brown and Capson (2012) created a GPU framework written in CUDA for tracking 3D models in a stream of 2D images. They used shared memory to accelerate the weight calculation process.

4 Discussion

In the preceding sections, GPU acceleration for medical image segmentation has been reviewed. To conclude the survey, a discussion on the main findings and some predictions regarding the future of image segmentation on GPUs are presented.

4.1 Current state of the art

The main findings of this review are summarized in Table 2. In this table, all the segmentation methods discussed in this paper are listed, and rated using the framework introduced in Section 2.

In general, most segmentation and image processing methods process each pixel using the same instructions, and data from a small neighborhood around the pixel. Thus, the thread count is usually high. Typical sizes of medical datasets are 512×512 for images, and 512^3 for volumes, which amount to over 262 thousand pixels and more than 134 million voxels respectively. However, as seen in this review, some segmentation methods do not process each pixel. Examples include active contours, which move a contour consisting of a set of points, and statistical shape models, that model shapes using a set of landmark points. For these methods, it may only be beneficial to use GPUs when the number of points is in the thousands.

Method	Data parallelism	Thread count	Branch div.	Memory usage	Synch.	GPU suit.
Thresholding	High	High	None	Low	None	High
	Trivial to implement					
Region growing	High	High	High	Low	High	Medium
	Schenke et al. (2005); Pan et al. (2008); Sherbondy et al. (2003); Chen et al. (2006); Harish and Narayanan (2007)					
Morphology	High	High	High	Low	None-High	High
	Eidheim et al. (2005); Thurley and Danell (2012); Karas (2011)					
Watershed	High	High	High	Medium	High	Medium
	Roerdink and Meijster (2001); Kauffmann and Piche (2008); Pan et al. (2008); Vitor et al. (2009); Körbes et al. (2009); Körbes and Vitor (2011); Wagner et al. (2010)					
Active contours - External energy	High	High	None	Low	None	High
	Podlozhnyuk et al. (2007)					
- GVF	High	High	None	High	High	High
	Eidheim et al. (2005); He and Kuester (2006); Zheng and Zhang (2012); Smistad et al. (2012b); Alvarado et al. (2013)					
- Contour evolution	High	Medium	None	Low	High	Medium
	He and Kuester (2006); Zheng and Zhang (2012); Eidheim et al. (2005); Perrot et al. (2011); Schmid et al. (2010); Li et al. (2011); Kamalakannan et al. (2009)					
Level sets - Default	High	High	High	Medium	High	High
	Rumpf and Strzodka (2001); Hong and Wang (2004)					
- Narrow-band	High	Dynamic	High	Medium	High	High
	Cates et al. (2004); Lefohn et al. (2004); Jeong et al. (2009)					
- Sparse-field	High	Dynamic	High	Medium	High	High
	Roberts et al. (2010)					
Atlas-based - Mutual Information	High	High	None	Medium	High	High
	Lin and Medioni (2008); Shams and Barnes (2007); Shams et al. (2010b)					
- Iterative closest point	High	Low-Medium	None	Low	Medium	Medium
	Langis et al. (2001); Qiu et al. (2009)					
Statistical shape mod. - Active shape model	High	Low-Medium	None	Low	Medium	Medium
	Song et al. (2010)					
- Active appearance model	High	High	None	High	Medium	Medium
	Ahlberg (2002)					
Markov random field - Iterative conditional modes	High	High	Low	Medium	High	High
	Griesser et al. (2005); Valero et al. (2011); Jodoin (2006); Walters et al. (2009); Sui et al. (2012)					
- Mean-field	High	High	Low	Medium	High	High
	Saito et al. (2012)					
- Graph cut: Push-relabel	High	High	High	High	High	Medium
	Dixit et al. (2005); Hussein et al. (2007); Vineet and Narayanan (2008); Garrett and Saito (2009)					
- Graph cut: Ford-Fulkerson	Low	-	-	-	-	Low
	Liu and Sun (2010); Strandmark and Kahl (2010)					
Centerline extr. & seg. of tubular structures - 3D thinning	High	High	High	Low	High	High
	Jiménez and Miras (2012)					
- Ridge traversal	Low	-	-	-	-	Low
	Non found					
- Tube Detection Filters	High	High	High	Medium	None	High
	Wang et al. (2013b); Erdt et al. (2008); Narayanaswamy et al. (2010); Bauer et al. (2009b); Smistad et al. (2012a, 2013)					
Dynamic image seg. Kalman filter	High	Medium	Low	Low	High	Medium
	Huang et al. (2011); Panin (2011)					
Particle filter	High	Medium	None-High	Low-High	High	High
	Montemayor et al. (2006); Lenz et al. (2008); Mateo Lozano and Otsuka (2008); Lozano and Otsuka (2008); Murphy-Chutorian and Trivedi (2008); Brown and Capson (2012); Panin (2011)					

Table 2: Comparison of how well the segmentation methods are suited for GPU computation. See section 2 for details on how each method is rated for each criteria. The ratings are based on the most common parallel implementations, parameters and input.

Most segmentation methods are also iterative because they run the same kernel several times. This requires global synchronization, which at present time is not possible to do efficiently from inside a kernel. The iterative processing often require double buffering, because global memory writes are not coherent within one kernel execution. When using textures, double buffering is currently required, as a texture can only be read or written to in a thread. Double buffering doubles the amount of memory used, which can be problematic for some methods such as 3D gradient vector flow.

Branch divergence is also a challenge for several methods, as not all pixels need to be processed. This is the case in segmentation methods such as region growing and narrow-band level sets. The performance loss due to branch divergence can be reduced using stream compaction. However, this comes at a cost, and will not improve performance if it has to be used for each iteration, which is the case for region growing.

Some GPU implementations may not provide a large speedup over an optimized serial version because the implementation implies performing more work. This is true for methods such as region growing and watershed. With region growing, the total number of pixels processed in each iteration is much higher in the data parallel GPU implementation than the serial one.

Hadwiger et al. (2004) presented a report on the state of the art of GPU-based segmentation in 2004. In contrast, there were very few GPU-based segmentation implementations at this time, with level set (Rumpf and Strzodka (2001); Lefohn et al. (2004) being one of the exceptions. They concluded that branch divergence and memory management present challenges for GPU implementations.

4.2 Software predictions

General purpose GPU frameworks such as OpenCL and CUDA have attracted a lot of users in recent years. Their popularity is likely to increase, as they ease the programming of GPUs compared to shader programming.

OpenCL enables efficient use of both GPUs and CPUs. It is likely that more hybrid solutions that use GPUs for the massively data parallel parts, and the CPU for the less parallel parts will appear. The challenge with these hybrid solutions is efficient sharing of data. At the time of writing, sharing data has to be done explicitly by memory transfer over the PCI express bus. However, this seems to be an issue that both major GPU manufacturers want to improve. This will be discussed in more detail in the next section.

It is also likely that there will be an increase in GPU libraries with commonly used data structures and algorithms such as heaps, sort, stream compaction and reduction. Libraries and frameworks that aid in writing image processing algorithms as well as scheduling, memory management and streaming of dynamic image data will probably become more important as more algorithms and image data are processed on the GPU. One framework

that aims to aid the design of image processing algorithms for different GPUs is the Heterogeneous Image Processing Acceleration Framework (HIPAcc).

4.3 Hardware predictions

The two main GPU manufacturers, NVIDIA and AMD, provide some details of the future development of their GPUs. However, these details are subject to change.

In general, the trend in GPU development has been increasing the number of thread processors, the clock speed and the amount of on-board memory. This allows more data to be processed faster in parallel.

NVIDIA recently launched their new Kepler architecture, which provide dynamic parallelism that allow threads to schedule new threads. However, the nesting depth is currently limited to 24 (NVIDIA (2012)). Dynamic parallelism might prove to be useful in segmentation methods that solve PDEs, such as level sets and GVF, by enabling fine grid computations on some image areas and coarse grid computations on other parts. Their current roadmap (NVIDIA (2013b)) suggests that their focus for the two next milestones (Maxwell and Volta) will be on memory. Unified virtual memory will allow CPUs and GPUs to share memory more seamlessly. Further down the road they plan to pile memory modules atop one another, and place them on the same silicon substrate as the GPU core itself. This technology is called *stacked DRAM*, and can supposedly give GPUs access to up to one terabyte per second of bandwidth.

AMD plan to focus on heterogeneous computing through their Heterogeneous System Architecture (HSA) initiative (Advanced Micro Devices (2013)). They state that current CPUs and GPUs have been designed as separate processing elements, and do not work together efficiently. Their plans is to rethink processor design to unify these two processors types, and give applications a unified address space.

Intel recently released another type of processor called the Intel Xeon Phi Coprocessor (Intel (2014)). These processors have a large number of cores ($\sim$60), large cache ($\sim$30MB) and a lot of on-board memory ($\sim$16GB). However, in contrast to GPUs, they have fewer thread processors ($\sim$240). Still, the large cache, memory bandwidth and size may make these processors interesting also for medical image segmentation.

5 Conclusions

In this review, the most common medical image segmentation algorithms have been discussed, and rated according to how suited they are for graphic processing units (GPUs). Through this comparison, it is shown that most segmentation methods are data parallel with a high amount of threads, which makes them well suited for GPU acceleration. However, factors such as synchronization, branch divergence and memory usage can limit

the speedup over serial execution. To reduce the impact of these limiting factors, several GPU optimization techniques are discussed.

References

Adams, R., Bischof, L., 1994. Seeded region growing. IEEE Transactions on Pattern Analysis and Machine Intelligence 16, 641–647. doi:10.1109/34.295913.

Advanced Micro Devices, 2012. AMD Accelerated Parallel Processing OpenCL Programming Guide. Technical Report July.

Advanced Micro Devices, 2013. Heterogeneous System Architecture. URL: http://developer.amd.com/resources/\\heterogeneous-computing/what-is-heterogeneous\\-system-architecture-hsa/. last accessed 12 apr 2013.

Ahlberg, J., 2002. An active model for facial feature tracking. EURASIP Journal on applied signal processing , 566–571.

Alvarado, R., Tapia, J.J., Rolón, J.C., 2013. Medical image segmentation with deformable models on graphics processing units. The Journal of Supercomputing doi:10.1007/s11227-013-1042-4.

Amdahl, G.M., 1967. Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of AFIPS '67 (Spring), ACM Press, New York, New York, USA. pp. 483–485. doi:10.1145/1465482.1465560.

Andrecut, M., 2009. Parallel GPU Implementation of Iterative PCA Algorithms. Journal of Computational Biology 16, 1593–1599. doi:10.1089/cmb.2008.0221.

Arulampalam, M., Maskell, S., Gordon, N., Clapp, T., 2002. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. IEEE Transactions on Signal Processing 50, 174–188. doi:10.1109/78.978374.

Aylward, S.R., Bullitt, E., 2002. Initialization, noise, singularities, and scale in height ridge traversal for tubular object centerline extraction. IEEE transactions on medical imaging 21, 61–75. doi:10.1109/42.993126.

Bauer, C., Bischof, H., 2008. Extracting curve skeletons from gray value images for virtual endoscopy, in: Proceedings of the 4th International Workshop on Medical Imaging and Augmented Reality, Springer. pp. 393–402.

Bauer, C., Bischof, H., Beichel, R., 2009a. Segmentation of airways based on gradient vector flow, in: Proceedings of the 2nd International Workshop on Pulmonary Image Analysis. MICCAI, Citeseer. pp. 191–201.

Bauer, C., Pock, T., Bischof, H., Beichel, R., 2009b. Airway tree reconstruction based on tube detection, in: Proceedings of the 2nd International Workshop on Pulmonary Image Analysis. MICCAI, Citeseer. pp. 203–214.

Besag, J., 1986. On the statistical analysis of dirty pictures. Journal of the Royal Statistical Society. Series B 48, 259–302.

Besl, P.J., McKay, N.D., 1992. A method for registration of 3-D shapes. IEEE Transactions on pattern analysis and machine intelligence .

Billeter, M., Olsson, O., Assarsson, U., 2009. Efficient stream compaction on wide SIMD many-core architectures, in: Proceedings of the Conference on High Performance Graphics, pp. 159–166.

Boykov, Y., Veksler, O., 2006. Graph cuts in vision and graphics: Theories and applications, in: Handbook of mathematical models in computer vision. Springer, pp. 79–96.

Brown, J.A., Capson, D.W., 2012. A Framework for 3D Model-Based Visual Tracking Using a GPU-Accelerated Particle Filter. IEEE transactions on visualization and computer graphics 18, 68–80. doi:10.1109/TVCG.2011.34.

Cates, J.E., Lefohn, A.E., Whitaker, R.T., 2004. GIST: an interactive, GPU-based level set segmentation tool for 3D medical images. Medical image analysis 8, 217–31. doi:10.1016/j.media.2004.06.022.

Chen, H.L.J., Samavati, F.F., Sousa, M.C., Mitchell, J.R., 2006. Sketch-based Volumetric Seeded Region Growing, in: Eurographics Workshop on Sketch-Based Interfaces and Modeling, pp. 123–130.

Cootes, T., Edwards, G., Taylor, C., 2001. Active appearance models. IEEE Transactions on Pattern Analysis and Machine Intelligence 23, 681–685.

Cootes, T.F., Taylor, C., Cooper, D., Graham, K., 1995. Active Shape Models - Their Training and Application. Computer Vision and Image Understanding 61, 38–59.

Davies, R.H., Twining, C.J., Cootes, T.F., Waterton, J.C., Taylor, C.J., 2002. A minimum description length approach to statistical shape modeling. IEEE transactions on medical imaging 21, 525–37. doi:10.1109/TMI.2002.1009388.

Dixit, N., Keriven, R., Paragios, N., 2005. GPU-Cuts : Combinatorial Optimisation , Graphic Processing Units and Adaptive Object Extraction. Technical Report March. Citeseer.

Eidheim, O., Skjermo, J., Aurdal, L., 2005. Real-time analysis of ultrasound images using GPU. International Congress Series 1281, 284–289. doi:10.1016/j.ics.2005.03.187.

Eklund, A., Dufort, P., Forsberg, D., Laconte, S.M., 2013. Medical image processing on the GPU - Past, present and future. Medical image analysis 17, 1073–1094. doi:10.1016/j.media.2013.05.008.

Erdt, M., Raspe, M., Suehling, M., 2008. Automatic hepatic vessel segmentation using graphics hardware, in: Proceedings of the 4th international workshop on Medical Imaging and Augmented Reality, pp. 403–412.

Fluck, O., Vetter, C., Wein, W., Kamen, a., Preim, B., Westermann, R., 2011. A survey of medical image registration on graphics hardware. Computer methods and programs in biomedicine 104, e45–57. doi:10.1016/j.cmpb.2010.10.009.

Frangi, A., Niessen, W., Vincken, K., Viergever, M., 1998. Multiscale vessel enhancement filtering. Medical Image Computing and Computer-Assisted Interventation 1496, 130–137.

Garrett, Z., Saito, H., 2009. Real-time online video object silhouette extraction using graph cuts on the GPU, in: Image Analysis and Processing–ICIAP 2009, pp. 985–994.

Gil, J., Werman, M., 1993. Computing 2-D min, median, and max filters. IEEE Transactions on Pattern Analysis and Machine Intelligence 15, 504–507.

Gower, J., 1975. Generalized procrustes analysis. Psychometrika 40, 33–51.

Griesser, A., Roeck, S., Neubeck, A., Gool, L., 2005. GPU-based foreground-background segmentation using an extended colinearity criterion, in: Vision, Modeling, and Visualization (VMV).

Hadwiger, M., Langer, C., Scharsach, H., Katja, B., 2004. State of the Art Report 2004 on GPU-Based Segmentation. Technical Report.

Harish, P., Narayanan, P., 2007. Accelerating large graph algorithms on the GPU using CUDA. High Performance Computing , 197–208.

Hassouna, M., Farag, A., 2007. On the extraction of curve skeletons using gradient vector flow, in: IEEE 11th International Conference on Computer Vision, IEEE. pp. 1–8. doi:10.1109/ICCV.2007.4409112.

He, Z., Kuester, F., 2006. GPU-Based Active Contour Segmentation Using Gradient Vector Flow, in: Advances in Visual Computing, pp. 191–201.

Heimann, T., van Ginneken, B., Styner, M.a., Arzhaeva, Y., Aurich, V., Bauer, C., Beck, A., Becker, C., Beichel, R., Bekes, G., Bello, F., Binnig, G., Bischof, H., Bornik, A., Cashman, P.M.M., Chi, Y., Cordova, A., Dawant, B.M., Fidrich, M., Furst, J.D., Furukawa, D., Grenacher, L., Hornegger, J., Kainmüller, D., Kitney, R.I., Kobatake, H., Lamecker, H., Lange, T., Lee, J., Lennon, B., Li, R., Li, S., Meinzer, H.P., Nemeth, G., Raicu, D.S., Rau, A.M., van Rikxoort, E.M., Rousson, M., Rusko, L., Saddi, K.a., Schmidt, G., Seghers, D., Shimizu, A., Slagmolen, P., Sorantin, E., Soza, G., Susomboon, R., Waite, J.M., Wimmer, A., Wolf, I., 2009. Comparison and evaluation of methods for liver segmentation from CT datasets. IEEE transactions on medical imaging 28, 1251–65. doi:10.1109/TMI.2009.2013851.

Heimann, T., Meinzer, H.P., 2009. Statistical shape models for 3D medical image segmentation: a review. Medical image analysis 13, 543–63. doi:10.1016/j.media.2009.05.004.

Herk, M.V., 1992. A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. Pattern Recognition Letters 13, 517–521.

Holewinski, J., Pouchet, L.N., Sadayappan, P., 2012. High-performance code generation for stencil computations on GPU architectures, in: Proceedings of the 26th ACM international conference on Supercomputing - ICS '12, ACM Press, New York, New York, USA. pp. 311–320. doi:10.1145/2304576.2304619.

Hong, J.J.Y., Wang, M.D.M., 2004. High speed processing of biomedical images using programmable GPU, in: International Conference on Image Processing, pp. 2455–2458.

Huang, M.Y., Wei, S.C., Huang, B., Chang, Y.L., 2011. Accelerating the Kalman Filter on a

GPU. 2011 IEEE 17th International Conference on Parallel and Distributed Systems , 1016–1020doi:10.1109/ICPADS.2011.153.

Hussein, M., Varshney, A., Davis, L., 2007. On Implementing Graph Cuts on CUDA, in: First Workshop on General Purpose Processing on Graphics Processing Units.

Intel, 2014. Intel xeon phi coprocessor. URL: http://ark.intel.com/products/family/71840/\\Intel-Xeon-Phi-Coprocessors. last accessed 3 jul 2014.

Jeong, W.K., Beyer, J., Hadwiger, M., Vazquez, A., Pfister, H., Whitaker, R.T., 2009. Scalable and interactive segmentation and visualization of neural processes in EM datasets. IEEE transactions on visualization and computer graphics 15, 1505–14. doi:10.1109/TVCG.2009.178.

Jiménez, J., Miras, J.R.D., 2012. Three-dimensional thinning algorithms on graphics processing units and multicore CPUs. Concurrency and Computation: Practive and experience 24, 1551–1571. doi:10.1002/cpe.

Jodoin, P.M., 2006. Markovian segmentation and parameter estimation on graphics hardware. Journal of Electronic Imaging 15, 033005. doi:10.1117/1.2238881.

Jošth, R., Antikainen, J., Havel, J., Herout, A., Zemčík, P., Hauta-Kasari, M., 2011. Real-time PCA calculation for spectral imaging (using SIMD and GP-GPU). Journal of Real-Time Image Processing 7, 95–103. doi:10.1007/s11554-010-0190-5.

Kalman, R., 1960. A new approach to linear filtering and prediction problems. Journal of Fluids Engineering 82, 35–45.

Kamalakannan, S., Gururajan, A., Shahriar, M., Hill, M.M., Anderson, J., Sari-Sarraf, H., Hequet, E.F., 2009. Assessing Fabric Stain Release using a GPU Implementation of Statistical Snakes, in: Niel, K.S., Fofi, D. (Eds.), Proceedings of SPIE, the International Society for Optical Engineering. doi:10.1117/12.806370.

Karas, P., 2011. Efficient Computation of Morphological Greyscale Reconstruction, in: Sixth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, pp. 54–61.

Kass, M., Witkin, A., Terzopoulos, D., 1988. Snakes: Active contour models. International Journal of Computer Vision 1, 321–331. doi:10.1007/BF00133570.

Kauffmann, C., Piche, N., 2008. Cellular automaton for ultra-fast watershed transform on GPU. 2008 19th International Conference on Pattern Recognition , 1–4doi:10.1109/ICPR.2008.4761628.

Khallaghi, S., Abolmaesumi, P., Gong, R.H., Chen, E., Gill, S., Boisvert, J., Pichora, D., Borschneck, D., Fichtinger, G., Mousavi, P., 2011. GPU Accelerated Registration of a Statistical Shape Model of the Lumbar Spine to 3D Ultrasound Images, in: Wong, K.H., Holmes III, D.R. (Eds.), SPIE Medical Imaging. doi:10.1117/12.878377.

Kirbas, C., Quek, F., 2004. A review of vessel extraction techniques and algorithms. ACM Computing Surveys 36, 81–121. doi:10.1145/1031120.1031121.

Kirkpatrick, S., Gelatt, C., Vecchi, M., 1983. Optimization by Simulated Annealing. Science 220, 671–680.

Körbes, A., Lotufo, R., Vitor, G., Ferreira, J., 2009. A Proposal for a Parallel Watershed Transform Algorithm for Real-Time Segmentation, in: Proceedings of Workshop de Visao Computacional WVC.

Körbes, A., Vitor, G., 2011. Advances on watershed processing on GPU architecture. Mathematical Morphology and Its Applications to Image and Signal Processing 6671, 260–271.

Krissian, K., Malandain, G., Ayache, N., 2000. Model-Based Detection of Tubular Structures in 3D Images. Computer Vision and Image Understanding 80, 130–171. doi:10.1006/cviu.2000.0866.

Langis, C., Greenspan, M., Godin, G., 2001. The parallel iterative closest point algorithm, in: 3-D Digital Imaging and Modeling, Published by the IEEE Computer Society. pp. 195–202. doi:10.1109/IM.2001.924434.

Lee, V., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P., 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, in: Proceedings of the 37th annual international symposium on Computer architecture, pp. 451–460.

Lefohn, A.E., Kniss, J.M., Hansen, C.D., Whitaker, R.T., 2004. A streaming narrow-band algorithm: interactive computation and visualization of level sets. IEEE transactions on visualization and computer graphics 10, 422–33. doi:10.1109/TVCG.2004.2.

Lenz, C., Panin, G., Knoll, A., 2008. A GPU-accelerated particle filter with pixel-level likelihood. VMV .

Lesage, D., Angelini, E.D., Bloch, I., Funka-Lea, G., 2009. A review of 3D vessel lumen segmentation techniques: models, features and extraction schemes. Medical image analysis 13, 819–845. doi:10.1016/j.media.2009.07.011.

Li, T., Krupa, A., Collewet, C., 2011. A robust parametric active contour based on fourier descriptors, in: IEEE International Conference on Image Processing, pp. 1037–1040.

Lin, Y., Medioni, G., 2008. Mutual information computation and maximization using GPU. 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops , 1–6doi:10.1109/CVPRW.2008.4563101.

Liu, J., Sun, J., 2010. Parallel graph-cuts by adaptive bottom-up merging, in: 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Ieee. pp. 2181–2188. doi:10.1109/CVPR.2010.5539898.

Lo, P., Ginneken, B.V., Reinhardt, J.M., de Bruijne, M., 2009. Extraction of Airways from CT (EXACT'09), in: Second International Workshop on Pulmonary Image Analysis, pp. 175–189.

Lorensen, W., Cline, H., 1987. Marching cubes: A high resolution 3D surface construction algorithm, in: Proceedings of the 14th annual conference on Computer graphics and interactive techniques, ACM. pp. 163–169.

Lozano, O., Otsuka, K., 2008. Simultaneous and fast 3D tracking of multiple faces in video by GPU-based stream processing, in: Acoustics, Speech and Signal Processing, pp. 713–716.

Mateo Lozano, O., Otsuka, K., 2008. Real-time Visual Tracker by Stream Processing. Journal of Signal Processing Systems 57, 285–295. doi:10.1007/s11265-008-0250-2.

McCool, M.D., 2008. Scalable Programming Models for Massively Multicore Processors. Proceedings of the IEEE 96, 816–831. doi:10.1109/JPROC.2008.917731.

Montemayor, A., Pantrigo, J., Cabido, R., Payne, B., 2006. Bandwidth improved GPU particle filter for visual tracking, in: Ibero-American Symposium on Computer Graphics SIACG.

Murphy-Chutorian, E., Trivedi, M.M., 2008. Particle filtering with rendered models: A two pass approach to multi-object 3D tracking with the GPU, in: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, Ieee. pp. 1–8. doi:10.1109/CVPRW.2008.4563102.

Narayanaswamy, A., Dwarakapuram, S., Bjornsson, C.S., Cutler, B.M., Shain, W., Roysam, B., 2010. Robust adaptive 3-D segmentation of vessel laminae from fluorescence confocal microscope images and parallel GPU implementation. IEEE transactions on medical imaging 29, 583–597. doi:10.1109/TMI.2009.2022086.

NVIDIA, 2010. OpenCL Best Practices Guide. Technical Report.

NVIDIA, 2012. CUDA Dynamic parallelism programming guide. Technical Report.

NVIDIA, 2013a. CUDA C Programming guide. Technical Report July.

NVIDIA, 2013b. NVIDIA CEO Updates NVIDIA's Roadmap. URL: http://blogs.nvidia.com/2013/03/\\nvidia-ceo-updates-nvidias-roadmap/. last accessed 16 apr 2014.

Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J., 2008. GPU computing. Proceedings of the IEEE 96, 879–899. doi:10.1109/JPROC.2008.917757.

Palágyi, K., Kuba, A., 1999. A Parallel 3D 12-Subiteration Thinning Algorithm. Graphical Models and Image Processing 61, 199–221. doi:10.1006/gmip.1999.0498.

Pan, L., Gu, L., Xu, J., 2008. Implementation of medical image segmentation in CUDA, in: International Conference on Technology and Applications in Biomedicine, Ieee. pp. 82–85. doi:10.1109/ITAB.2008.4570542.

Panin, G., 2011. Model-based visual tracking: the OpenTL framework. John Wiley & Sons.

Perrot, G., Domas, S., Couturier, R., Bertaux, N., 2011. GPU Implementation of a Region Based Algorithm for Large Images Segmentation, in: IEEE 11th International Conference on Computer and Information Technology, Ieee. pp. 291–298. doi:10.1109/CIT.2011.60.

Pham, D.L., Xu, C., Prince, J.L., 2000. Current Methods in Medical Image Segmentation. Biomedical Engineering 2, 315–337.

Pluim, J.P.W., Maintz, J.B.A., Viergever, M.a., 2003. Mutual-information-based registration of medical images: a survey. IEEE transactions on medical imaging 22, 986–1004. doi:10.1109/TMI.2003.815867.

Podlozhnyuk, V., Howes, L., Young, E., 2007. Image Convolution with CUDA. Technical Report June. NVIDIA.

Pratx, G., Xing, L., 2011. GPU computing in medical physics: A review. Medical Physics 38, 2685. doi:10.1118/1.3578605.

Qiu, D., May, S., Nüchter, A., 2009. GPU-accelerated nearest neighbor search for 3D registration. Computer Vision Systems , 194–203.

Roberts, M., Packer, J., Sousa, M.C., Mitchell, J.R., 2010. A Work-Efficient GPU Algorithm for Level Set Segmentation, in: Proceedings of the Conference on High Performance Graphics, pp. 123–132.

Roerdink, J.B.T.M., Meijster, A., 2001. The Watershed Transform : Definitions , Algorithms and Parallelization Strategies. Fundamenta Informaticae 41, 187–228.

Rumpf, M., Strzodka, R., 2001. Level set segmentation in graphics hardware, in: Proceedings International Conference on Image Processing, Ieee. pp. 1103–1106. doi:10.1109/ICIP.2001.958320.

Saito, M., Okatani, T., Deguchi, K., 2012. Application of the mean field methods to MRF optimization in computer vision, in: IEEE Conference on Computer Vision and Pattern Recognition, Ieee. pp. 1680–1687. doi:10.1109/CVPR.2012.6247862.

Schenke, S., Burkhard, C.W., Denzler, J., 2005. GPU-Based Volume Segmentation, in: Proceedings of IVCNZ, Dunedin, New Zealand. pp. 171–176.

Schmid, J., Iglesias Guitián, J.a., Gobbetti, E., Magnenat-Thalmann, N., 2010. A GPU framework for parallel segmentation of volumetric images using discrete deformable models. The Visual Computer 27, 85–95. doi:10.1007/s00371-010-0532-0.

Scholl, I., Aach, T., Deserno, T.M., Kuhlen, T., 2010. Challenges of medical image processing. Computer Science - Research and Development 26, 5–13. doi:10.1007/s00450-010-0146-9.

Serra, J., 1986. Introduction to mathematical morphology. Computer Vision, Graphics, and Image Processing 35, 283–305. doi:10.1016/0734-189X(86)90002-2.

Sethian, J., 1999. Level Set Methods and Fast Marching Methods. Second ed., Cambridge University Press.

Shams, R., Barnes, N., 2007. Speeding up mutual information computation using NVIDIA CUDA hardware, in: 9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications, IEEE. pp. 555–560. doi:10.1109/DICTA.2007.23.

Shams, R., Sadeghi, P., Kennedy, R., Hartley, R., 2010a. A survey of medical image registration on multicore and the GPU. IEEE Signal Processing Magazine 27, 50–60. doi:10.1109/MSP.2009.935387.

Shams, R., Sadeghi, P., Kennedy, R., Hartley, R., 2010b. Parallel computation of mutual information on the GPU with application to real-time registration of 3D medical images. Computer methods and programs in biomedicine 99, 133–46. doi:10.1016/j.cmpb.2009.11.004.

Sherbondy, A., Houston, M., Napel, S., 2003. Fast volume segmentation with simultaneous visual-

ization using programmable graphics hardware, in: Visualization, VIS 2003, Ieee. pp. 171–176. doi:10.1109/VISUAL.2003.1250369.

Shi, L., Liu, W., Zhang, H., Xie, Y., Wang, D., 2012. A survey of GPU-based medical image computing techniques. Quantitative Imaging in Medicine and Surgery 2, 188–206. doi:10.3978/j.issn.2223-4292.2012.08.02.

Sluimer, I., Schilham, A., Prokop, M., van Ginneken, B., 2006. Computer Analysis of Computed Tomography Scans of the Lung: A Survey. IEEE transactions on medical imaging 25, 385–405. doi:10.1109/TMI.2005.862753.

Smistad, E., Elster, A.C., Lindseth, F., 2012a. GPU-Based Airway Segmentation and Centerline Extraction for Image Guided Bronchoscopy, in: Norsk informatikkonferanse, Akademika forlag. pp. 129–140.

Smistad, E., Elster, A.C., Lindseth, F., 2012b. Real-time gradient vector flow on GPUs using OpenCL. Journal of Real-Time Image Processing .

Smistad, E., Elster, A.C., Lindseth, F., 2013. GPU accelerated segmentation and centerline extraction of tubular structures from medical images. International Journal of Computer Assisted Radiology and Surgery .

Song, M., Tao, D., Liu, Z., 2010. Image ratio features for facial expression recognition application. IEEE Transactions on Systems, Man, and Cybernetics 40, 779–788.

Strandmark, P., Kahl, F., 2010. Parallel and distributed graph cuts by dual decomposition, in: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Ieee. pp. 2085–2092. doi:10.1109/CVPR.2010.5539886.

Sui, H., Peng, F., Xu, C., Sun, K., Gong, J., 2012. GPU-accelerated MRF segmentation algorithm for SAR images. Computers & Geosciences 43, 159–166. doi:10.1016/j.cageo.2011.10.001.

Thurley, M.J., Danell, V., 2012. Fast Morphological Image Processing Open-Source Extensions for GPU Processing With CUDA. IEEE Journal of Selected Topics in Signal Processing 6, 849–855. doi:10.1109/JSTSP.2012.2204857.

Valero, P., Sánchez, J.L., Cazorla, D., Arias, E., 2011. A GPU-based implementation of the MRF algorithm in ITK package. The Journal of Supercomputing 58, 403–410. doi:10.1007/s11227-011-0597-1.

Vincent, L., Soille, P., 1991. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. IEEE Transactions on Pattern Analysis and Machine Intelligence 13, 583–598.

Vineet, V., Narayanan, P.J., 2008. CUDA cuts: Fast graph cuts on the GPU, in: 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, Ieee. pp. 1–8. doi:10.1109/CVPRW.2008.4563095.

Vitor, G., Ferreira, J., Körbes, A., 2009. Fast image segmentation by watershed transform on graphical hardware, in: Proceedings of 30th CILAMCE.

Wagner, B., Müller, P., Haase, G., 2010. A Parallel Watershed-Transformation Algorithm for the GPU, in: Workshop on Applications of Discrete Geometry and Mathematical Morphology, pp. 111–115.

Walters, J.P., Balu, V., Kompalli, S., Chaudhary, V., 2009. Evaluating the use of GPUs in liver image segmentation and HMMER database searches, in: IEEE International Symposium on Parallel & Distributed Processing, Ieee. pp. 1–12. doi:10.1109/IPDPS.2009.5161073.

Wang, C., Komodakis, N., Paragios, N., 2013a. Markov Random Field modeling, inference & learning in computer vision & image understanding: A survey. Computer Vision and Image Understanding 117, 1610–1627. doi:10.1016/j.cviu.2013.07.004.

Wang, N., Chen, W.f., Feng, Q.j., 2013b. Angiogram Images Enhancement Method Based on GPU, in: World Congress on Medical Physics and Biomedical Engineering, pp. 868–871.

Xu, C., Prince, J., 1998. Snakes, shapes, and gradient vector flow. IEEE Transactions on Image Processing 7, 359–369.

Zhang, J., 1992. The mean field theory in EM procedures for Markov random fields. Signal Processing, IEEE Transactions on 40, 2570–2583.

Zheng, Z., Zhang, R., 2012. A Fast GVF Snake Algorithm on the GPU. Research Journal of Applied Sciences, Engineering and Technology 4, 5565–5571.

Ziegler, G., Tevs, A., Theobalt, C., Seidel, H., 2006. On-the-fly point clouds through histogram pyramids, in: Vision, modeling, and visualization 2006: proceedings, IOS Press. p. 137.

B

FAST: framework for heterogeneous medical image computing and visualization

Authors

Erik Smistad, Mohammadmehdi Bozorgi and Frank Lindseth

Published in

International Journal of Computer Assisted Radiology and Surgery, 2015. Springer.

FAST: framework for heterogeneous medical image computing and visualization

Erik Smistad[1,2], Mohammadmehdi Bozorgi[1] and Frank Lindseth[2,1]

1) Norwegian University of Science and Technology

2) SINTEF Medical Technology. Trondheim, Norway

Abstract

Purpose Computer systems are becoming increasingly heterogeneous in the sense that they consist of different processors, such as multi-core CPUs and graphic processing units (GPUs). As the amount of medical image data increases, it is crucial to exploit the computational power of these processors. However, this is currently difficult due to several factors, such as driver errors, processor differences, and the need for low level memory handling. This paper presents a novel FrAmework for heterogeneouS medical image compuTing and visualization (FAST). The framework aims to make it easier to simultaneously process and visualize medical images efficiently on heterogeneous systems.

Methods FAST uses common image processing programming paradigms, and hides the details of memory handling from the user, while enabling the use of all processors and cores on a system. The framework is open-source, cross-platform and available online.

Results Code examples and performance measurements are presented to show the simplicity and efficiency of FAST. The results are compared to the insight toolkit (ITK) and the visualization toolkit (VTK), and show that the presented framework is faster with up to 20 times speedup on several common medical imaging algorithms.

Conclusions FAST enables efficient medical image computing and visualization on heterogeneous systems. Code examples and performance evaluations have demonstrated that the toolkit is both easy to use, and performs better than existing frameworks, such as ITK and VTK.

1 Introduction

An increasing amount of medical image data is becoming available for any given patient today. Modern image analysis techniques make it possible to extract and visualize more information from the images. The race for using the increasing amount of image data

more effectively is paramount for better diagnostics and therapy in the future. Still, concurrent medical image computing and visualization of both static and dynamic real-time data is computationally expensive. In the efforts toward improving computer assisted radiology and surgery, this may entail that computational demanding research methods that have been assessed to be quantitatively better then existing methods (in terms of accuracy for example) can not be used in a routine clinical setting due to time constraints.

Most modern computer systems are heterogeneous in the sense that they consist of several different processors, such as multi-core CPUs and graphic processing units (GPUs). These processors enable parallel processing, which can accelerate many medical image computing tasks significantly [7, 25]. The programming of this hardware is, however, still difficult due to several factors. One factor is that the software needed to use the hardware, such as GPU drivers and compilers, may contain errors which are hard to debug. Also, the different manufacturers may have interpreted the standards differently. This forces programmers to do more debugging and testing. Since the programmer can not change proprietary software such as GPU drivers, the programmer may even have to write separate code for different hardware manufacturers and software versions. The result is increased software development overhead and fragmented source code.

GPUs were originally programmed using shaders intended for graphics rendering. Newer frameworks, such as CUDA [18], enable general-purpose programming of GPUs. The open computing language (OpenCL) [29] is an open standard for parallel programming of heterogeneous systems. OpenCL enables parallel programming of different processors such as multi-core CPUs and GPUs. These GPU programming tools expose the programmer to several hardware details. For instance, most GPUs have their own memory that is separate from the computer's main memory. This memory is often divided into several different memory spaces such as global, texture and constant memory [19]. Thus, the programmer has to explicitly move data between the different memories during execution.

In this article, we propose a framework called FAST (FrAmework for heterogeneouS medical image compuTing and visualization). This framework aims to make it easier to do efficient processing and visualization of medical images on heterogeneous systems. The framework is open-source and available online[1]. FAST is also cross-platform, supporting Windows, Mac OS X and Linux. The authors believe that in order to achieve satisfactory performance in the more computational demanding medical applications, the framework has to cover the entire pipeline from reading and streaming data to visualizing the result on the screen. Thus, the framework currently includes methods for:

- Reading, writing and streaming image data in different formats.
- Image processing algorithms such as filtering, segmentation and registration.
- Surface mesh extraction and rendering.
- Multi-volume and slice rendering.

[1] `http://github.com/smistad/FAST/`

The framework aims to be easy to use by utilizing common programming paradigms from popular toolkits, such as the insight toolkit (ITK) and the visualization toolkit (VTK), and hiding the details of memory handling from the user. Also, the framework has many tests and benchmarks which enable the user to make sure that all the hardware and software are working properly, and gives the performance and accuracy necessary for a whole range of medical image processing applications. We acknowledge that there exist many medical image computing algorithms created using ITK and VTK. The framework therefore supports interoperability with these frameworks, such that image data can be shared and pipelines from FAST, ITK and VTK can be linked. This may ease the integration of FAST into existing applications.

1.1 Related work

ITK [9, 10] and VTK [21, 12] are two of the most commonly used frameworks for medical image analysis and visualization. ITK contains several image processing algorithms used in the medical domain, while VTK is mostly used for visualization. Several of the image processing filters in ITK and VTK support multi-threading for execution on multi-core CPUs. In this multi-threading model, the input image is split among a set of threads. Each subimage is processed individually and the result is stitched together. These frameworks were not initially created with support for GPU acceleration, except GPU-based rendering. However, extensions have been proposed to enable such support [2, 11]. The current version of ITK (4.6) includes GPU implementations of some algorithms such as thresholding, smoothing and optical flow registration. However, these are implemented as separate modules which are only available if compiled with a specific flag.

The open computer vision library (OpenCV) [20] is another popular image processing and visualization framework. However, this framework focus primarily on 2D image processing and lack several features that are important in the medical imaging domain such as 3D image processing, medical image formats and surface extraction. Still, OpenCV was designed for computational efficiency and with a strong focus on real-time applications. Several algorithms in OpenCV are implemented for the GPU using OpenCL.

While these frameworks provide accelerated processing more as an extension and as an optional feature, the FAST framework presented in this article has been designed with heterogeneous accelerated processing in mind from the start and it is part of the core of the framework. We believe this will result in a framework that is faster and easier to use.

MeVisLab [13, 15] is a software which focus on rapid prototyping of medical image software using a visual programming interface. It also supports integration with ITK and VTK and has support for multi-threading. FAST on the other hand, focus on high performance heterogeneous medical image computing and visualization and has currently no visual programming interface.

One framework that aims to aid the development of image processing algorithms for different GPUs is the Heterogeneous Image Processing Acceleration Framework (HIPAcc)

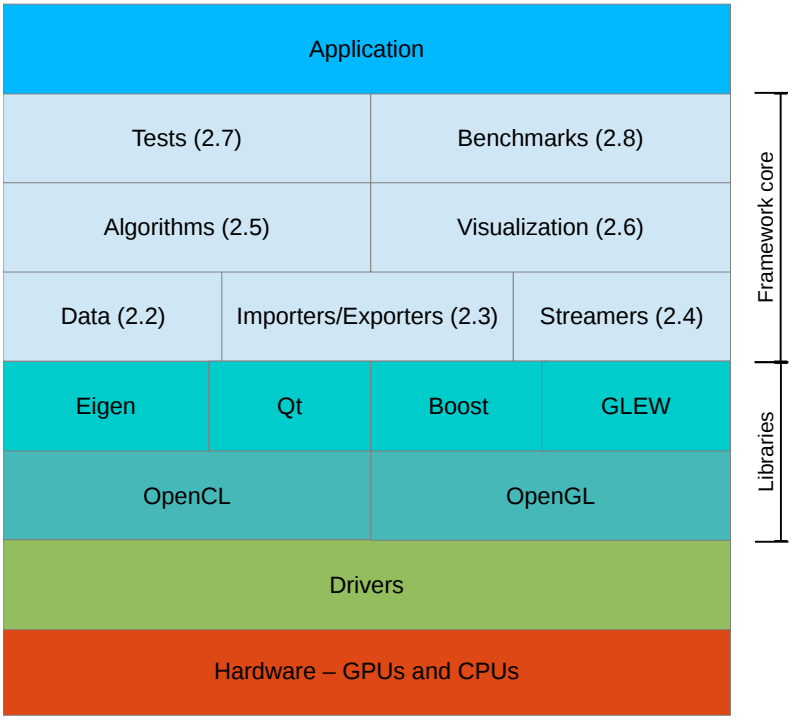

Figure 1: Block diagram of the framework. The numbers indicate which section describes the different parts of the framework.

[14]. However, HIPAcc focus on the design of image processing algorithms and does not include visualization and registration.

1.2 Outline

The next section describes the details of the framework. The result section presents code examples and performance benchmarks of common medical image computing pipelines on different systems. Finally, a discussion and conclusion is presented.

2 Methodology

The FAST framework consists of five main layers, as illustrated in Fig. 1. The bottom layer is the actual hardware, i.e. the CPUs and GPUs. The second layer are the drivers for this hardware, which are provided by the hardware manufacturers. Next is the library layer, which consists of several libraries that are needed in the framework. The libraries in this layer are:

- **Open Computing Library (OpenCL)** - An open standard for parallel programming on heterogeneous systems, including multi-core CPUs, GPUs, and FPGAs. It

is supported by most processor manufacturers including AMD, NVIDIA and Intel.

- **Open Graphics Library (OpenGL)** - A cross-platform library for visualization.
- **GL Extension Wrangler (GLEW)** - A library for handling OpenGL extensions.
- **Eigen** - A fast cross-platform linear algebra library.
- **Qt** - A cross-platform graphical user interface (GUI) toolkit.
- **Boost** - A C++ utility library.

The next layer is the core of the framework, which is split into several groups:

- **Data (2.2)** - Objects for data (both static and dynamic) such as images and meshes, which enables the synchronized processing of such data on a set of heterogeneous devices.
- **Importers/Exporters (2.3)** - Data import and export objects for different formats such as MetaImage (.mhd), raw, ITK and VTK.
- **Streamers (2.4)** - Objects that enable streaming of data.
- **Algorithms (2.5)** - A set of commonly used filtering, segmentation and registration algorithms.
- **Visualization (2.6)** - A set of renderers such as image, volume, slice and mesh renderers.
- **Tests (2.7)** - A set of tests for the framework which ensures that all parts of the framework are working properly.
- **Benchmarks (2.8)** - Mechanisms for measuring, assimilating and reporting the performance of all operations in the framework.

The last layer is the application layer. The framework may be both a stand-alone application, which enables benchmarking and tests of a heterogeneous system, and an external library for other medical image computing applications.

The rest of this section will describe each part of the framework in more detail, but first the execution pipeline of the framework is described.

2.1 The execution pipeline

FAST uses a demand-driven execution pipeline similar to what is used in ITK and VTK. This entails that each processing step is first linked together to form a pipeline, that is not executed until some object calls the update method. This can be done in two ways:

- Explicitly by calling the update method on an object in the pipeline.

- Implicitly by a renderer which calls update on its input connections several times per second.

The pipeline consists of process objects, which extend the abstract base class *ProcessObject*. A process object is an object that performs processing and may have zero, one or several parent process objects. Most process objects produce data objects which extend the abstract base class *DataObject*. Similar to the newest version of VTK (version 6), FAST uses a pipeline where the data objects are not explicitly part of the pipeline. Fig. 2 illustrates a simple pipeline with these two types of objects and how they are connected.

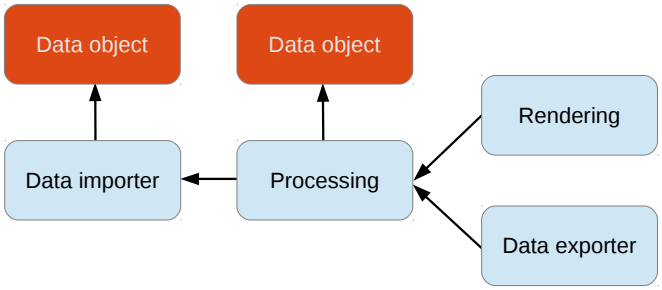

Figure 2: A simple pipeline with process (blue/bright) and data objects (orange/dark). The arrows indicate how the objects are connected.

Data objects have an internal timestamp. The timestamp is always updated when the data is changed. Each process object has a list of timestamps for each connection. These timestamps represent which version of the data objects were used the last time the process object was executed. In addition, each process object has a flag indicating whether it has been modified or not. This could be a parameter or input change.

When the update method is called on a process object, it will first call update on all its parent objects. Thus update will be called on all objects backwards in the pipeline until a process object with no input connections is encountered (e.g. an importer object). A process object will re-execute by calling its execute method, if it is modified or one of its input connections have changed timestamps. Thus each process object will implement its own execute method while the update method is the same for each process object.

2.2 Data management

Throughout this article, we will use the OpenCL terminology and refer to a processor with its memory as a *device*, and the main CPU as the *host*.

Data organization and synchronization is one of the key components in the proposed framework. Image data is represented by an object called *Image* which is used for both 2D and 3D image data. These image objects represent an image on all devices, and its data is guaranteed to be coherent on any devices after being altered. Thus, if an image

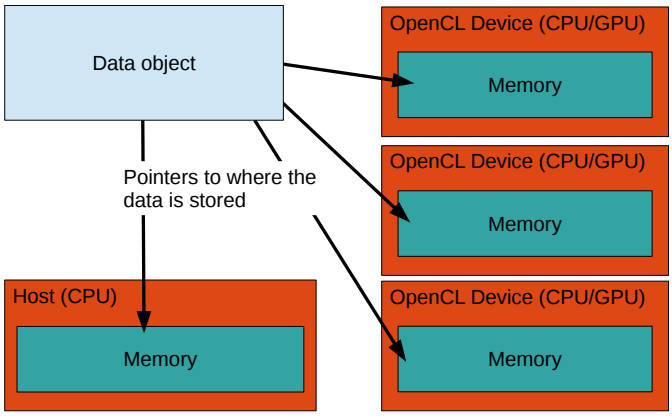

Figure 3: A data object (e.g. an image) has pointers to all the devices where the data is stored. Using the OpenCL terminology, a device is a processor with its memory, and the host is the main CPU.

is changed on one device it will also be changed on the other devices before the data is required on those devices. The same applies for other types of data like meshes where an object called *Mesh* is used to represent a mesh on all devices (see Fig. 3). Dynamic data, such as temporal 2D and 3D image data, is also supported. This is discussed in further detail in section 2.4 on data streaming.

2.2.1 Data access

Two forms of data access are possible in the framework: 1) Read-only and 2) Read and write. The general rule is that several devices can perform read operations on a data object at the same time. However, if a device needs to write data, only that device can have access to the data object at that time. This policy ensures data coherency across devices. Thus, if a device wants to write to an image, it has to wait for all other operations on that image to finish. When a device is writing to an image, no other devices can read or write to that image.

To enforce this policy, several *DataAccess* objects are introduced for each data object. For instance, in OpenCL, an image can be represented either as a buffer (i.e. a regular array) or as an image/texture. Thus there exist one *OpenCLBufferAccess* object and one *OpenCLImageAccess* object to facilitate such access to image data. From these objects an OpenCL *Image* or *Buffer* object can be retrieved, which is needed to perform OpenCL computations on the image. Access to the image from the main memory can also be requested for doing processing on the CPU using C++. The *DataAccess* objects also have methods for releasing the access, thus enabling other devices to perform write operations on the image. The access will also be released in the destructor of this object to avoid deadlocks. When the access is released, the OpenCL *Image/Buffer* object pointer is invalidated to ensure that the program can no longer manipulate the data. However, this does not delete the actual data on the device. When write access to an object is requested, the

framework will check that any previous access objects have been released.

2.2.2 Data change

Every time data is changed on a device, the change should be reflected on the other devices as well. However, this doesn't have to be done immediately. Updating the data can be done the next time the data is requested on another device. This is often referred to as lazy loading. The benefit of lazy loading is that the number of data transfers can be reduced. However, the drawback is that there will be a transfer cost the next time the data is requested on a device which doesn't have the updated data.

Thus, each data object has a set of flags indicating whether the data (in the form of OpenCL buffers, images and C++ pointers) is up to date for each device. When one device has changed some data, these are set to false for all devices except the device in which the change was performed. Next time the data is requested on a device, the flag is checked and if it is false, a data transfer will start and the flag will be set to true for that device.

2.2.3 Data removal

The amount of memory available on a system as well as on graphic cards are limited, and may not be enough when working on large datasets. Thus it is crucial to remove data that is not needed anymore. Data may be deleted explicitly by the programmer, however, this is a burden for the programmer and may easily be forgotten. After the entire pipeline has been defined by the programmer it is known which process objects need which data objects as input. Thus it is possible to delete a data object after all the process objects that use this data object have finished execution. This requires each process object to retain and release the data objects when they are defined as input and when the process object is finished using it. To facilitate this, each data object has a reference counter and when it reaches zero, the data is deleted.

2.2.4 Data types

Medical images are represented in different formats. Some common examples are: Ultrasound (unsigned 8 bit integer), CT (signed/unsigned 16 bit integer) and MR (unsigned 16 bit integer). The framework currently supports the following data formats for images:

- TYPE_FLOAT - 32 bit floating point number
- TYPE_UINT8 - 8 bit unsigned integer
- TYPE_INT8 - 8 bit signed integer
- TYPE_UINT16 - 16 bit unsigned integer

- TYPE_INT16 - 16 bit signed integer

An image can also have multiple channels, or components, and currently 1-4 channels are supported.

2.3 Data import and export

Data can be imported to and exported from the framework in several different forms such as:

- MetaImage file (.mhd, .raw and .zraw)
- Image file (.jpg, .png etc.)
- ITK image object
- VTK image object
- VTK file (.vtk)

In the future, the framework will also support common data formats such as DICOM [16] and NIfTI [17].

2.4 Data streaming

Streamers are process objects that provide access to dynamic data. This can for instance be real-time images from an ultrasound probe or a series of images stored on disk. The output of streamer objects is a *DynamicData* object, which has a method for retrieving the current frame in the stream. The *DynamicData* objects can contain one of several types of data such as images or meshes. The streamers read data into the *DynamicData* object in a separate thread so that processing and data streaming can be performed concurrently. Streamers can use one of three different streaming modes:

- STREAMING_MODE_NEWEST_FRAME_ONLY
  This will only keep the newest frame in the *DynamicData* object.
- STREAMING_MODE_PROCESS_ALL_FRAMES
  This will keep all frames in the *DynamicData* object, but will remove the frame from the object after it has been processed.
- STREAMING_MODE_STORE_ALL_FRAMES
  This will store all frames in the *DynamicData* object.

For the second of these streaming modes it is also important to limit the size of the dynamic data buffer so that the streaming does not use up all memory. With this mode it is therefore possible to set the maximum size of the dynamic data buffer. A producer-consumer model is used to synchronize the use of the data.

The data and process objects are designed so that it is easy to accept both static and dynamic data as input and output to an algorithm.

2.5 Algorithms

Algorithms are implemented in the framework as process objects and thus have to override the execute method. Currently only a few filtering, segmentation and registration algorithms have been implemented such as Gaussian smoothing, seeded region growing [1], thresholding, skeletonization [8], iterative closest point [3] and surface extraction (marching cubes) [23]. All algorithms support parallel processing on CPUs and GPUs. In the near future, we plan to implement and integrate several other algorithms such as level set segmentation, Kalman filter object tracking [28], gradient vector flow [22, 27] and tube detection filters [24, 26].

2.6 Visualization

2.6.1 Graphical user interface and rendering

Qt is used in FAST as the graphical user interface. Qt is cross-platform, supports multithreading, direct rendering from OpenGL and event handling of keyboard and mouse input. A visualization window in the FAST framework can have multiple views, and each view can have multiple renderers. Windows are implemented using Qt's *QWidget* class, while the *View* extends the *QGLWidget*, which is a widget that may be rendered to by OpenGL. The FAST renderers do the actual rendering. These renderers and the event handling is executed in one thread, while the pipeline is run in another thread. This enables concurrent visualization, camera movement and pipeline execution. Five different types of renderers are currently available in FAST:

- Image renderer - For displaying 2D images.
- Slice renderer - Extracts and displays an image from a volume in an arbitrary plane using trilinear interpolation.
- Mesh renderer - Renders a mesh.
- Volume renderer - Creates an image of a volume using ray casting [4].
- Point renderer - Renders a list of points.

2.6.2 Scene graph

Correct placement of images and geometry in the visualization scene is important. FAST uses a scene graph for this purpose. In this directed graph, each data object has a node. All data nodes are connected to a parent node, which can be another data node or a root

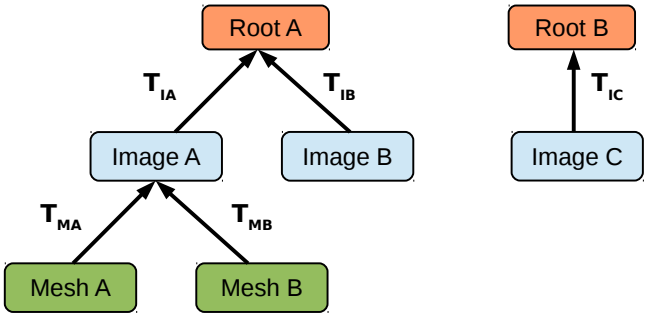

Figure 4: An example of a scene graph. Images A and B are registered because they share a root node. Image C is not registered to any other data. Each edge between the nodes has a transformation object. This transformation determines how data is positioned relative to other nodes. Meshes A and B are dependent on image A. Thus, moving image A will also move these meshes.

node. Each edge between the nodes has a transformation object. This transformation determines how data is positioned relative to other nodes. Fig. 4 shows an example of a scene graph with three images and two meshes. The images A and B share a root node, and are therefore registered. Image C is not registered to any other data. Image A is placed in the visualization scene by applying the transformation T_{IA} to the image. Similarly, image B uses the transformation T_{IB}. Since these images are registered, the corresponding voxel position in image B of a voxel position in image A can be determined by first applying the transformation T_{IA}, and then the inverse transformation of image B T_{IB}^{-1}. The meshes A and B are related to image A. These meshes may for instance be the result of a segmentation of image A. Mesh A is placed in the visualization scene by first applying the transformation from the mesh to image A T_{MA}, and then the transformation from image A to the root node T_{IA}. Thus, if image A is moved in the scene, the meshes A and B are also moved.

When an image or mesh is created, a corresponding data node is created in the scene graph and connected to a root node. However, if the data is created from another data object, it is connected to the data node of that data object instead. For instance, the surface extraction algorithm will connect the resulting mesh to the image used to create the mesh. A visualization of an image and a segmented surface mesh using the scene graph is illustrated in Fig. 6. When importing a MetaImage, any transformation information such as translation and rotation is read from the MetaImage file (.mhd) and put in the scene graph.

2.7 Tests

As much as possible of the framework should be covered by unit and system tests. This enables a user to ensure that the framework is working correctly on the user's current software and hardware configuration. The authors know by experience that new drivers,

compilers and libraries can introduce errors that may stop the framework from working properly. These tests enable a user to quickly detect these problems. The tests are written using the Catch C++ testing framework [5]. Realistic test data is needed to test the framework properly, and is therefore provided for download[2].

The framework is available on the open-source community website GitHub. Each time a user contributes to the project, three different computers will execute all tests with the new code and verify that everything is working. These machines use all the supported operating systems Windows, Mac OS X and Ubuntu Linux and processors from Intel, AMD and NVIDIA. Thus, the source code of the framework is tested continuously on several hardware and software configurations. We believe this is needed in order to ensure the stability of FAST.

2.8 Benchmarks

Users may also want to test how well their current setup performs and see how performance changes when software and hardware changes are introduced. For this purpose benchmarks are provided, which are tests of different pipelines in which performance is measured and reported.

3 Results

This section first presents some examples of how the framework can be used. These examples are provided to show how easy it is to set up pipelines in FAST. Next, the performance of the framework is measured and compared to that of ITK and VTK.

3.1 Code examples

The first example is a simple pipeline of four steps: import 3D image from disk, Gaussian smoothing, surface extraction and rendering. The result is shown in Fig. 5. The steps of the pipeline are linked together using the *getOutputPort* and *setInputConnection* methods of the process objects. This is the same method used by ITK and VTK. The *::pointer* types are smart pointers which are created with the *New* method. These pointers reduce memory problems such as memory leakage.

[2] `http://github.com/smistad/FAST/wiki/Test-data`

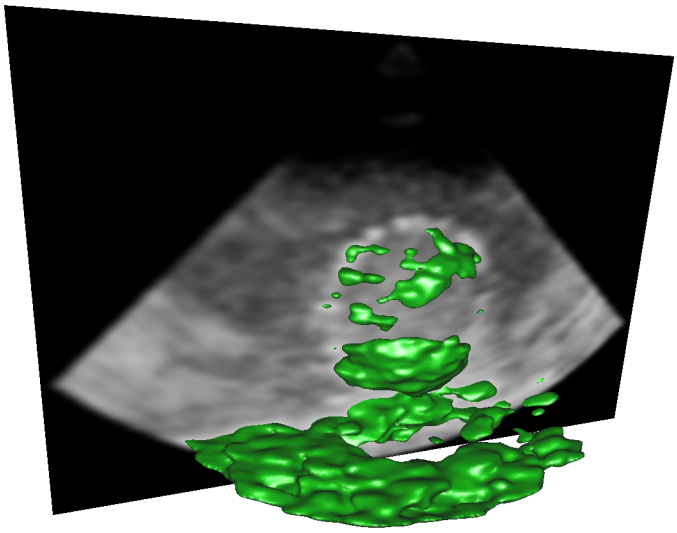

Figure 5: Result of pipeline A in Example 1. A 3D ultrasound image is first smoothed. Then, surface extraction is used to extract a surface mesh from the smoothed image. Finally, a slice of the smoothed 3D image is rendered together with the surface mesh.

Example 1: Pipeline A

```
// Import image
ImageFileImporter::pointer importer = ImageFileImporter::New();
importer->setFilename("image.mhd");

// Blur image with Gaussian smoothing
GaussianSmoothing::pointer smoothing = GaussianSmoothing::New();
smoothing->setInputConnection( importer->getOutputPort());
smoothing->setStandardDeviation(1.0);

// Extract surface mesh with marching cubes
SurfaceExtraction::pointer extraction = SurfaceExtraction::New();
extraction->setInputConnection( smoothing->getOutputPort());

// Render surface mesh
MeshRenderer::pointer meshRenderer = MeshRenderer::New();
meshRenderer->addInputConnection( extraction->getOutputPort());

// Render slice
SliceRenderer::pointer sliceRenderer = SliceRenderer::New();
sliceRenderer->addInputConnection( smoothing->getOutputPort());
sliceRenderer->setSlicePlane(PLANE_X);

// Create a window, attach the renderers and start pipeline
SimpleWindow::pointer window = SimpleWindow::New();
window->addRenderer(meshRenderer);
window->addRenderer(sliceRenderer);
window->start();
```

This pipeline can easily be changed from using a single static image as input to a stream of images by only substituting the *Importer* object with a *Streamer* object. The rest of the pipeline is the same. Example 2 shows how a *ImageFileStreamer* object is created to stream a series of MetaImages from disk. The streamer object uses a filename format to find files. The hash sign (#) is replaced by an integer index which changes for each

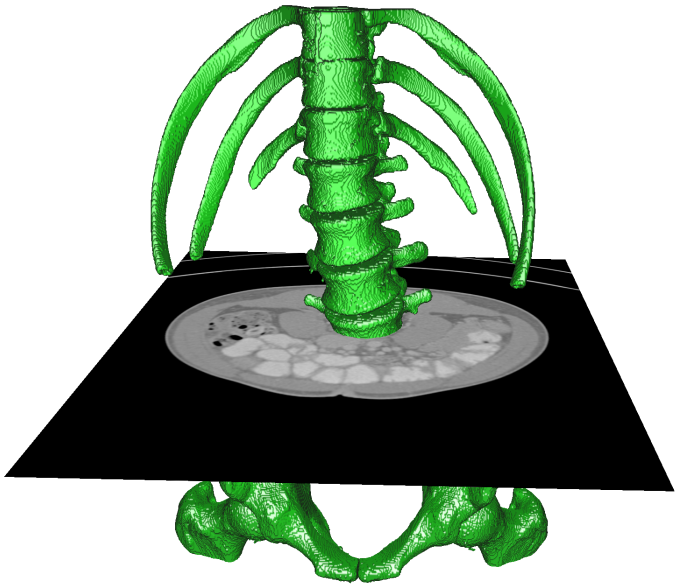

Figure 6: Result of pipeline B in Example 4. Region growing is used to segment the bone structure from a CT scan. A surface mesh is extracted from the segmentation and rendered together with a slice of the CT scan. The scene graph is used to correctly position the two data objects.

image that is loaded. It is possible to change the start index and step which are 0 and 1 respectively by default. The streamer stops when no more images with the format are found.

Example 2: Streaming images

```
ImageFileStreamer::pointer streamer = ImageFileStreamer::New();
streamer->setFilenameFormat("image_frame_#.mhd");
```

The user may want to specify which device should be used as the default device. This is done using the *DeviceManager* object as shown in Example 3. However, each process object may override this if desired.

Example 3: Set the default device to be a GPU

```
DeviceManager::setDefaultDevice( DeviceManager::getOneGPUDevice());
```

The next example shows another pipeline. This pipeline performs region growing segmentation on an image, extracts the surface mesh of the segmentation, and finally renders the mesh and a slice of the input image. The result can be seen in Fig. 6.

Example 4: Pipeline B

```
// Import image
ImageFileImporter::pointer importer = ImageFileImporter::New();
importer->setFilename("CT-Abdomen.mhd");

// Segment image with region growing
SeededRegionGrowing::pointer segmentation = SeededRegionGrowing::New();
segmentation->setInputConnection( importer->getOutputPort());
segmentation->addSeedPoint(261,284,208);
```

```
segmentation->setIntensityRange(150, 5000);

// Extract surface mesh with marching cubes
SurfaceExtraction::pointer extraction = SurfaceExtraction::New();
extraction->setInputConnection( segmentation->getOutputPort());

// Render slice plane
SliceRenderer::pointer sliceRenderer = SliceRenderer::New();
sliceRenderer->setPlaneToRender(PLANE_Z);
sliceRenderer->setIntensityWindow(1000);
sliceRenderer->setIntensityLevel(0)
sliceRenderer->setInputConnection( importer->getOutputPort());

// Render surface mesh
MeshRenderer::pointer meshRenderer = MeshRenderer::New();
meshRenderer->addInputConnection( extraction->getOutputPort());

// Create a window, attach the renderers and start pipeline
SimpleWindow::pointer window = SimpleWindow::New();
window->addRenderer(sliceRenderer);
window->addRenderer(meshRenderer);
window->start();
```

Pipeline C imports a 2D image and performs binary threshold segmentation. The segmentation is skeletonized and finally rendered using the ImageRenderer as shown in Fig. 7.

Example 5: Pipeline C

```
// Import image
ImageFileImporter::pointer importer = ImageFileImporter::New();
importer->setFilename("image.png");

// Segment image with thresholding
BinaryThresholding::pointer thresholding = BinaryThresholding::New();
thresholding->setInputConnection( importer->getOutputPort());
thresholding->setLowerThreshold(0.5);

// Skeletonize the segmentation
Skeletonization::pointer skeletonization = Skeletonization::New();
skeletonization->setInputConnection( thresholding->getOutputPort());

// Render image
ImageRenderer::pointer renderer = ImageRenderer::New();
renderer->addInputConnection( skeletonization->getOutputPort());

// Create a window, attach the renderers and start pipeline
SimpleWindow::pointer window = SimpleWindow::New();
window->addRenderer(renderer);
window->start();
```

The next pipeline first imports two point sets from VTK files (.vtk). The *PointSet* object is a data object, which only contains a set of points. These point sets are then registered using the iterative closest point (ICP) algorithm [3]. Finally, the point sets are rendered using the *PointRenderer* (see Fig. 8).

Example 6: Pipeline D

```
// Import two point sets
PointSetImporter::pointer importerA = PointSetImporter::New();
importerA->setFilename("pointsA.vtk");
PointSet::pointer pointsA = importerA->getOutputPort();
PointSetImporter::pointer importerB = PointSetImporter::New();
importerB->setFilename("pointsB.vtk");
```

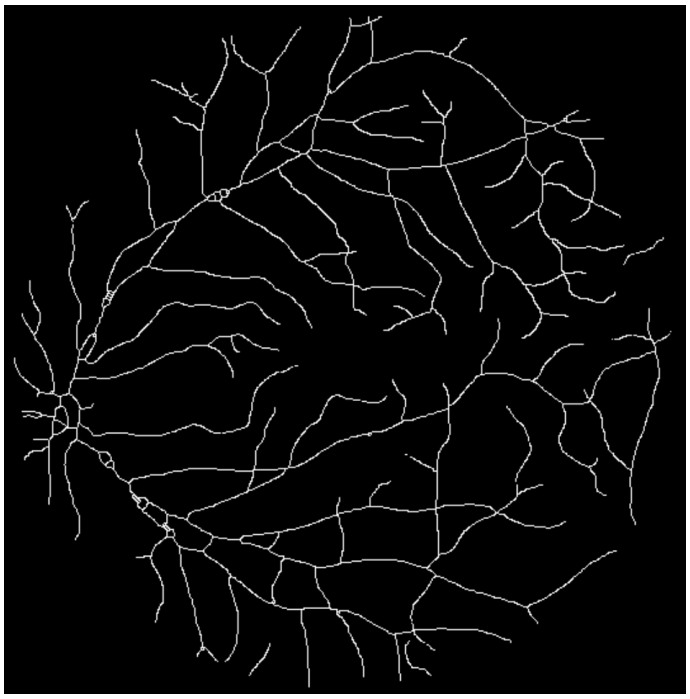

Figure 7: Result of pipeline C in Example 5 where an image of the retina blood vessels is thresholded, and skeletonized using iterative thinning.

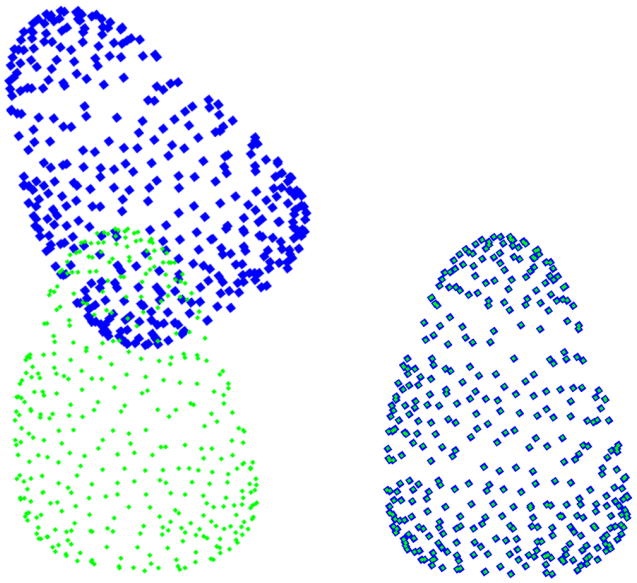

Figure 8: The two point sets of pipeline D in Example 6 before and after the iterative closest point (ICP) algorithm is used to register the two sets.

Pipeline	Framework	System (CPU/GPU/OS)	Runtime (ms)	Memory usage
Pipeline A Data import, Gaussian smoothing, surface extraction and rendering.	FAST	Intel/NVIDIA/Windows	86 (0.2, 34, 52, 0.6)	92 MB
		AMD/AMD/Linux	52 (0.2, 23, 29, 0.6)	53 MB
		Intel/NVIDIA/Mac	135 (0.1, 71, 63, 0.5)	34 MB
	ITK and VTK	Intel/NVIDIA/Windows	696 (9, 97, 511, 78)	260 MB
		AMD/AMD/Linux	1133 (6, 262, 568, 295)	255 MB
		Intel/NVIDIA/Mac	704 (29, 68, 441, 165)	140 MB
Pipeline B Data import, region growing, surface extraction and rendering.	FAST	Intel/NVIDIA/Windows	2293 (230, 1954, 108, 0.7)	398 MB
		AMD/AMD/Linux	2270 (262, 1848, 158, 0.8)	402 MB
		Intel/NVIDIA/Mac	5103 (273, 4588, 242, 0.7)	355 MB
	ITK and VTK	Intel/NVIDIA/Windows	3041 (346, 732, 1673, 290)	1.5 GB
		AMD/AMD/Linux	4615 (301, 906, 2309, 1099)	1.4 GB
		Intel/NVIDIA/Mac	3473 (765, 623, 1616, 467)	1.1 GB
Pipeline C Data import, thresholding, skeletonization and rendering.	FAST	Intel/NVIDIA/Windows	50 (39, 4, 3, 3)	79 MB
		AMD/AMD/Linux	20 (9, 2, 6, 3)	55 MB
		Intel/NVIDIA/Mac	26 (11, 3, 9, 3)	42 MB
	ITK and VTK	Intel/NVIDIA/Windows	856 (3, 1, 819, 33)	28 MB
		AMD/AMD/Linux	489 (5, 0.8, 384, 99)	23 MB
		Intel/NVIDIA/Mac	721 (10, 0.4, 682, 28)	50 MB
Pipeline D Data import, iterative closest point and rendering.	FAST	Intel/NVIDIA/Windows	25 (2, 23, 0.2)	80 MB
		AMD/AMD/Linux	21 (4, 17, 0.2)	52 MB
		Intel/NVIDIA/Mac	9 (1, 8, 0.2)	16 MB
	ITK and VTK	Intel/NVIDIA/Windows	293 (31, 84, 178)	22 MB
		AMD/AMD/Linux	241 (4, 109, 128)	20 MB
		Intel/NVIDIA/Mac	152 (6, 61, 85)	14 MB

Table 1: Performance measurements of the four pipelines in examples 1, 4, 5 and 6. The same pipelines were implemented in ITK and VTK for comparison. Three systems with different operating system and hardware were used. The runtime of each step in the pipeline is listed in parentheses.

```
PointSet::pointer pointsB = importerB->getOutputPort();

// Run iterative closest point
IterativeClosestPoint::pointer icp = IterativeClosestPoint::New();
icp->setMovingSet(pointsA);
icp->setFixedSet(pointsB);

// Render the two point sets
PointRenderer::pointer renderer = PointRenderer::New();
renderer->addInput(pointsA, Color::Blue(), 10);
renderer->addInput(pointsB, Color::Green(), 5);

// Create a window, attach the renderers and start pipeline
SimpleWindow::pointer window = SimpleWindow::New();
window->addRenderer(renderer);
window->start();
```

3.2 Performance

The runtime and memory usage of pipeline A, B, C and D (see examples 1, 4, 5 and 6) were measured and collected in Table 1. The runtime is the average of 10 runs on each system. The memory usage is measured using the system monitor, and includes only the

system memory and not the GPU memory usage. The same pipelines were implemented and measured in ITK and VTK for comparison. Several of the ITK and VTK image processing filters used in these pipelines including smoothing, thresholding, thinning, region growing and iterative closest point use multi-threading. Three different computer systems, all with solid-state drives (SSD), were used for the measurements:

- Intel i5 3.4 GHz CPU with 16 GB RAM, NVIDIA Geforce GTX 970 4 GB running Windows 8.1.
- AMD A10 CPU with 16 GB RAM, AMD Radeon R9 290 GPU 4 GB running Ubuntu 14.04 Linux.
- Intel i5 3.4 GHz CPU with 16 GB RAM, NVIDIA Geforce GTX 780M 4 GB running Mac OS X 10.9.

The following datasets were used for the different pipelines, and informed consent was obtained from all patients for being included in the study:

- **Pipeline A:** 3D ultrasound image, unsigned 8 bit integer, 276x249x200 voxels ≈ 14 MB.
- **Pipeline B:** CT image, signed 16 bit integer, 512x512x426 voxels ≈ 223 MB.
- **Pipeline C:** 2D image, 565x584 pixels ≈ 11 kB.
- **Pipeline D:** Two point set files of the left ventricle with 386 3D points each of ≈ 31kB.

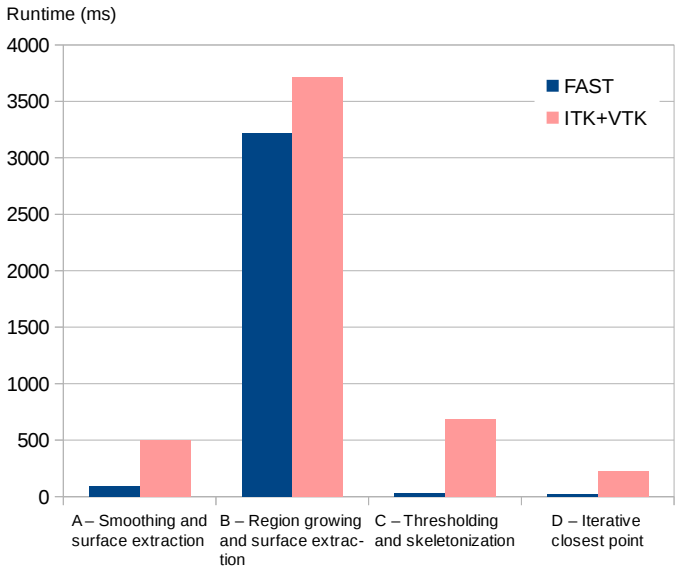

Figure 9: Average runtime performance for all computer system in section 3.2 of the four pipelines in examples 1, 4, 5 and 6 using FAST, ITK and VTK.

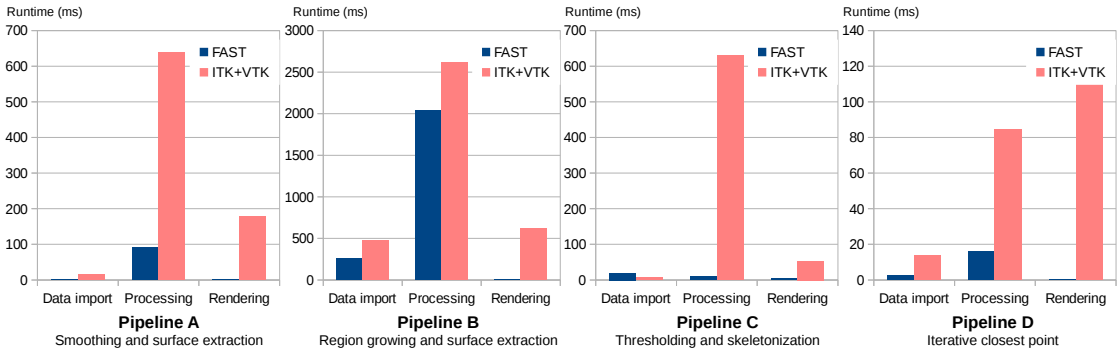

Figure 10: Detailed average runtime performance for all computer system in section 3.2 of the four pipelines in examples 1, 4, 5 and 6 using FAST, ITK and VTK.

4 Discussion

As more medical imaging data becomes available, a framework that exploits the increasingly heterogeneous and parallel computers is needed. These heterogeneous systems are hard to program due to several factors such as such as driver errors, processor differences, and the need for low level memory handling. The FAST framework makes medical image processing and visualization easier by using familiar programming paradigms, and hiding the details of memory handling from the user, while still enabling the use of all processors and cores on a system. Errors and differences in proprietary software and hardware (e.g. GPU drivers) can not be fixed by the medical imaging community, as the development of these are dependent on the manufacturers. FAST aims to provide a large set of tests and benchmarks to detect and report these problems. This enables an easy way for a user to check if there are any problems for a specific setup.

Although ITK has a couple of algorithms that support OpenCL as an optional extension, we believe that it is necessary to support heterogeneous processing in the entire framework to achieve the best performance. This include all steps in a pipeline from data import, to processing and visualization. Enabling this kind of support in ITK, VTK or MeVisLab would most likely mean rewriting the entire core of these toolkits.

The code examples in the previous section show how easy it is to set up different pipelines consisting of data import, streaming, processing and rendering. Implementation of the same pipelines in ITK and VTK required more lines of code and code complexity mainly due to the need for exporting data from ITK to VTK and templating in ITK. One may argue that this is because ITK and VTK have more features. However, we believe that common operations, such as these four pipelines, should require little code.

The performance measurements in Table 1 and figures 9 and 10 show that FAST is faster for the four pipelines with speedups of up to 20 times compared to ITK and VTK. This speedup is mainly due to the fact that FAST is able to use the GPU for processing and rendering, while ITK and VTK rely on multi-threading for acceleration. All steps in the pipeline, including data import, processing and rendering, are faster with the proposed

framework, with region growing, 2D thresholding and data import in pipeline C as the only exceptions. As shown in Fig. 10, the largest difference in runtime is with rendering, where FAST uses from 0.2 to 3 ms, while VTK uses 28 to 1099 ms.

ITK and VTK use more system memory than FAST on pipelines A and B which use large 3D images as input with sizes of 14 and 223 MB. However, FAST uses the GPU for most of this processing and will therefore also use more GPU memory. The overall goal is to significantly reduce the total memory usage in medical image computing and visualization applications. This will be done by avoiding data duplications, and keeping tracking of the CPU and GPU memory used in future versions of FAST.

For FAST to be accepted by the medical imaging community, it needs commonly used algorithms to speed up the development of high-level image processing algorithms. We plan to implement more algorithms in FAST, and hope that others will contribute to this open-source framework through the collaboration platform GitHub. Still, developers can use algorithms that already exist in ITK, as FAST supports interoperability with ITK pipelines. Documentation and examples are also vital for the framework's success. Thus, an online open-source wiki has been created[3].

The most important application for high performance medical image processing and visualization is image guided surgery, where preoperative data is combined with intraoperative data which needs to be acquired, processed and visualized in the operating room. Optical and electromagnetic tracking are important in this context. Our future work will be on incorporating tracking in the framework to enable surgical navigation.

The authors strongly believe in open-source code for medical image computing and visualization as a mean for advancing the state of the art, as well as open data for evaluating algorithms. Each year a vast amount of new methods and modifications of existing ones are proposed (e.g. registration and segmentation algorithms). In order to increase the amount of methods that are actually used clinically it's crucial that new algorithms are benchmarked thoroughly, both in terms of accuracy and computational performance. In the last decade, several challenges have been arranged [6], and open databases with an established ground truth are probably the best tool we have today to achieve this. As the proposed framework becomes more mature the authors hope that FAST can contribute to this effort and make more algorithms clinically ready faster.

5 Conclusion

A novel framework for efficient medical image computing and visualization has been presented. The framework was built from ground up with optimal performance on heterogeneous systems in mind. Code examples and performance evaluations have demonstrated that the toolkit is both easy to use, and performs better than existing frameworks, such as

[3] `http://github.com/smistad/FAST/wiki/`

ITK and VTK. Built-in benchmarking support will make additional fine-tuning a lot easier, and produce new insight about heterogeneous computing in the medical domain. As more quality and performance benchmarked functionality is added to the framework, the authors hope that FAST will be a valid tool for bringing more medical imaging software into clinical practice in the years to come.

Acknowledgements

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 610425. The hardware used in this project was funded by the MedIm (Norwegian Research School in Medical Imaging) Travel and Research Grant.

Conflict of interest Erik Smistad, Mohammadmehdi Bozorgi and Frank Lindseth declare that they have no conflict of interest.

References

[1] R. Adams and L. Bischof. Seeded region growing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(6):641–647, June 1994.

[2] R. Beare, D. Micevski, C. Share, L. Parkinson, P. Ward, W. Goscinski, and M. Kuiper. CITK - an architecture and examples of CUDA enabled ITK filters. pages 1–8, 2011.

[3] P. J. Besl and N. D. McKay. A method for registration of 3-D shapes. *IEEE Transactions on pattern analysis and machine intelligence*, 1992.

[4] M. Bozorgi and F. Lindseth. GPU-based multi-volume ray casting within VTK for medical applications. *International journal of computer assisted radiology and surgery*, May 2014.

[5] Catch. C++ Automated Test Cases in Headers. `https://github.com/philsquared/Catch/` - Last accessed 10. Oct 2014.

[6] Consortium for Open Medical Image Computing. Grand Challenges in Biomedical Image Analysis. `http://grand-challenge.org/` - Last accessed 25. Nov 2014.

[7] A. Eklund, P. Dufort, D. Forsberg, and S. M. Laconte. Medical image processing on the GPU - Past, present and future. *Medical image analysis*, 17(8):1073–1094, June 2013.

[8] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Pearson Prentice Hall, third edition, 2008.

[9] L. Ibanez and W. Schroeder. *The ITK Software Guide*. Kitware, 2.4 edition, 2004.

[10] Kitware. Insight toolkit (ITK). `http://itk.org/` - Last accessed 18. Aug 2014.

[11] Kitware. ITK Release 4 GPU Acceleration. http://www.itk.org/Wiki/ITK/Release_4/GPU_Acceleration/ - Last accessed 10. Oct 2014.

[12] Kitware. Visualization toolkit (VTK). http://www.vtk.org/ - Last accessed 18. Aug 2014.

[13] M. Koenig, W. Spindler, J. Rexilius, J. Jomier, F. Link, and H.-O. Peitgen. Embedding VTK and ITK into a visual programming and rapid prototyping platform. In *Proceedings of SPIE*, volume 6141, pages 61412O–61412O–11, 2006.

[14] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Generating Device-specific GPU code for Local Operators in Medical Imaging. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, number Section III, 2012.

[15] MeVis Medical Solutions AG. MeVisLab. http://www.mevislab.de - Last accessed 26. Jan 2015.

[16] P. Mildenberger, M. Eichelberg, and E. Martin. Introduction to the DICOM standard. *European Radiology*, 12:920–927, 2002.

[17] Neuroimaging Informatics Technology Initiative. NIfTI-1 Data Format. http://nifti.nimh.nih.gov/ - Last accessed 26. Jan 2015.

[18] NVIDIA Corporation. CUDA. http://developer.nvidia.com/cuda-zone/ - Last accessed 26. Jan 2015.

[19] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[20] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov. Real-time computer vision with OpenCV. *Communications of the ACM*, 55(6):61, June 2012.

[21] W. Schroeder, K. Martin, and B. Lorensen. *Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Kitware, 4th edition, 2006.

[22] E. Smistad, A. C. Elster, and F. Lindseth. Real-time gradient vector flow on GPUs using OpenCL. *Journal of Real-Time Image Processing*, pages 1–8, 2012.

[23] E. Smistad, A. C. Elster, and F. Lindseth. Real-Time Surface Extraction and Visualization of Medical Images using OpenCL and GPUs. In *Norsk informatikkonferanse*, pages 141–152. Akademika forlag, 2012.

[24] E. Smistad, A. C. Elster, and F. Lindseth. GPU accelerated segmentation and centerline extraction of tubular structures from medical images. *International Journal of Computer Assisted Radiology and Surgery*, 9(4):561–575, 2014.

[25] E. Smistad, T. L. Falch, M. Bozorgi, A. C. Elster, and F. Lindseth. Medical image segmentation on GPUs – A comprehensive review. *Medical Image Analysis*, 20(1):1–18, 2015.

[26] E. Smistad and F. Lindseth. A New Tube Detection Filter for Abdominal Aortic Aneurysms. In *Proceedings of MICCAI 2014 Workshop on Abdominal Imaging: Computational and Clinical Applications*, 2014.

[27] E. Smistad and F. Lindseth. Multigrid gradient vector flow computation on the GPU. 2014.

[28] E. Smistad and F. Lindseth. Real-time Tracking of the Left Ventricle in 3D Ultrasound Using Kalman Filter and Mean Value Coordinates. In *Proceedings MICCAI Challenge on Echocardiographic Three-Dimensional Ultrasound Segmentation (CETUS)*, pages 65–72, Boston, 2014.

[29] The Khronos Group. OpenCL. `http://www.khronos.org/opencl/` - Last accessed 26. Jan 2015.

C

GPU accelerated segmentation and centerline extraction of tubular structures in medical images

Authors

Erik Smistad, Anne C. Elster and Frank Lindseth

Published in

International Journal of Computer Assisted Radiology and Surgery, volume 9, issue 4, July 2014, pages 561-575.

GPU accelerated segmentation and centerline extraction of tubular structures from medical images

Erik Smistad[1], Anne C. Elster[1], Frank Lindseth[1,2]

1) Norwegian University of Science and Technology

2) SINTEF Medical Technology. Trondheim, Norway

Abstract

Purpose To create a fast and generic method with sufficient quality for extracting tubular structures such as blood vessels and airways from different modalities (CT, MR and US) and organs (brain, lungs and liver) by utilizing the computational power of graphic processing units (GPUs).
Methods A cropping algorithm is used to remove unnecessary data from the datasets on the GPU. A model-based tube detection filter combined with a new parallel centerline extraction algorithm and a parallelized region growing segmentation algorithm is used to extract the tubular structures completely on the GPU. Accuracy of the proposed GPU method and centerline algorithm is compared to the ridge traversal and skeletonization/thinning methods using synthetic vascular datasets.
Results The implementation is tested on several datasets from three different modalities: airways from CT, blood vessels from MR and 3D Doppler Ultrasound. The results show that the method is able to extract airways and vessels in 3-5 seconds on a modern GPU and is less sensitive to noise than other centerline extraction methods.
Conclusions Tubular structures such as blood vessels and airways can be extracted from various organs imaged by different modalities in a matter of seconds, even for large datasets.

1 Introduction

Blood vessels and airways are both examples of important tubular structures in the human body. The extraction of these structures can be essential for planning and guidance of several surgical procedures such as bronchoscopy, laparoscopy and neurosurgery.

Registration is to create a mapping between two domains, for instance between an image and the patient or between different image modalities [34]. Registration is an important

step in image guided surgery as it enables us to accurately plot the location of surgical tools inside the body onto images of the patient using optical or magnetic tracking technology. Tubular structures extracted from preoperative images can be matched to similar intraoperative structures, e.g. airways generated by a tracked bronchoscope or brain vessels extracted from power Doppler based 3D ultrasound, and consequently create the mapping between preoperative images and the patient. Also, extracted tubular structures from pre- and intraoperative image data can be used to reduce registration errors when a corresponding point (anatomical landmarks or fiducials) patient registration method is used.

Furthermore, during surgical procedures, anatomical structures have a tendency to move and deform inside the body due to respiration, pulsation, external pressure and resection. This is called anatomical shift and is a major challenge as it reduces the surgical navigation accuracy. However, it has been shown that registration of blood vessels from pre- and intraoperative image data can be used to detect and correct organshift such as brainshift [38].

The automatic extraction of tubular structures can be very time consuming. As time during surgery is very crucial, long-lasting processing should be avoided. Preoperative data is often acquired just before the procedure and thus it is desirable to process these data as fast as possible as well. The purpose of this work is to create a fast and generic method with sufficient quality for extracting tubular structures such as blood vessels and airways from different modalities (CT, MR and US) and organs (brain, lungs, liver) by utilizing the computational power of graphic processing units (GPUs).

The rest of the introduction discuss GPU computing and provides a brief survey of existing methods for extracting tubular structures from medical images. An overview of the contributions in this paper is also given. The methodology section provides a detailed description of each part of the implementation including how it is optimized for the GPU and evaluated. In the result section, performance is measured in terms of speed and quality. Finally, the results are discussed and conclusions are given.

1.1 GPU computing

Several image processing techniques are data parallel because each pixel can be processed in parallel using the same instructions. Graphic Processing Units (GPUs) allow many pixels/voxels to be processed in the same clock cycle, enabling substantial speedups. The GPU is a type of single instruction, multiple data (SIMD) processor. It can perform the same instruction on each element in a dataset in parallel. This is achieved by having many functional units like arithmetic-logic units (ALUs) that share a control unit. Fig. 1 depicts the general layout of a GPU and its memory hierarchy. The GPU originally had a fixed pipeline that was created for fast rendering of 3D graphics. The introduction of programmable shaders in the pipeline made it possible to run programs on the GPU. However, the task of programming shaders to solve arbitrary problems requires knowledge

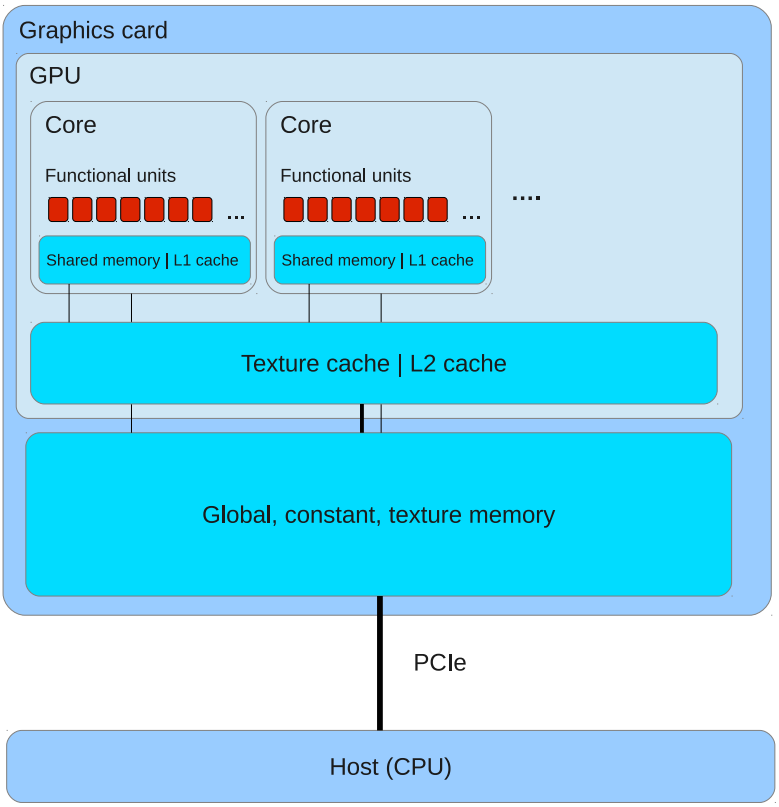

Figure 1: General architecture of a GPU and its memory hierarchy. Note however, that the actual architecture is much more complex and differ for each GPU. This diagram only shows the general features.

about the GPU pipeline as the problem at hand needs to be transformed into a rendering problem. General purpose GPU (GPGPU) programming languages and frameworks such as CUDA and OpenCL were created to make GPU programming easier. The field of GPU computing is still young. However, a brief survey of medical image processing and visualization on the GPU was recently provided by Shi et al. [39].

1.2 Methods for extracting tubular structures

Tubular structures are usually extracted from volumes in two different ways:

- As a **segmentation**, either as a binary classification where each voxel in the volume is given a non-zero value if it belongs to the tubular structure or as a surface model of the structure.
- As a **centerline**, i.e. a line that goes through the center of the tubular structures.

Both representations are useful in different applications. For instance, the centerline is very useful for registration while the segmentation is useful for volume estimation and

visualization of the structures' surface.

There exist several methods for extracting tubular structures from medical images. A recent and extensive review on blood vessel extraction was done by Lesage et al. [29] and an older one by Kirbas and Quek [25]. Two reviews on the segmentation of airways were done by Lo et al. [31] and Sluimer et al. [40].

A common method for extracting tubular structures is to grow the segmentation iteratively from an initial point or area using methods such as region growing [27, 45, 16], active contours and wave front propagation (e.g. snakes and level sets) [24, 35, 46, 32]. A centerline can then be extracted from the segmentation using skeletonization and 3D thinning methods [28, 18, 22].

Growing a segmentation using only a model of desired intensity values has shown to give limited result in several applications such as airway segmentation where the thin airway walls may cause severe segmentation leakage [32]. Thus in many applications it may be necessary to use a model of the shape of the tubular structures as well. Also, these growing methods are very sensitive to initialization.

Tube Detection Filters (TDFs) are used to detect tubular structures and calculates a probability that a specific voxel is inside a tubular structure. Most TDFs use gradient information, often in the form of an eigenanalysis of the Hessian matrix. Frangi et al. [15] presented an enhancement and detection method for tubular structures based on the eigenvalues of this matrix. Krissian et al. [26] created a model-based detection filter that fits a circle to the cross-sectional plane of the tubular structure defined by the eigenvectors of the Hessian.

A centerline can be extracted directly from the TDF result without a segmentation using methods such as ridge traversal. Aylward et al. [2] provides a review of different centerline extraction methods and proposed an improved ridge traversal algorithm based on a set of ridge criteria and different methods for handling noise. Bauer et al. [6] showed how this method could be used together with Gradient Vector Flow. For applications where only the centerline is needed, segmentation can be skipped using this method and thus reduce processing time.

Some related work on accelerating the extraction of tubular structures on the GPU exist. Erdt et al. [14] performed the TDF and a region growing segmentation on the GPU and reported a 15 times faster computation of the gradients and up to 100 times faster TDF. Narayanaswamy et al. [36] did vessel luminae region growing segmentation on the GPU and reported a speedup of 8. Bauer et al. presented a GPU acceleration for airway segmentation by doing the Gradient Vector Flow computation on the GPU in [7] and the TDF calculation on the GPU in [8]. However, they only provide a limited description of the GPU implementations. Helmberger et al. performed region growing for airway segmentation on the GPU and a lung vessel segmentation on the GPU using a TDF [21]. They reported a runtime of 5-10 minutes using a modern GPU and CUDA compared to a runtime of up to an hour using only the CPU.

1.3 Contributions

The methodology in this paper is inspired by the works of Bauer et al. [7, 3, 8] and Krissian et al. [26] and is a continuation of our previous paper on GPU accelerated airway segmentation [41].

The main contributions in this paper are:

- A fast and generic method that can extract tubular structures like blood vessels and airways from different modalities (e.g. CT, MR and Ultrasound) and organs (e.g. lung, brain and liver) entirely on the GPU.
- A new parallel GPU algorithm for extracting centerlines directly from the TDF result.
- A generic parallel cropping algorithm for reducing memory usage on the GPU.

2 Methodology

The implementation is written in C++ and OpenCL and is available online[1]. OpenCL is a framework for running parallel programs on heterogeneous platforms such as CPU and GPU. The implementation consists of five main steps that are all executed on the GPU (see Fig. 2).

The first step is to crop the volume in order to reduce the total memory usage. The second step involves a few pre-processing steps such as Gaussian smoothing and Gradient Vector Flow which are necessary to make the results less sensitive to noise and differences in tube contrast and size. After pre-processing, the model-based TDF by Krissian et al. [26] is used. From the TDF result, the centerlines are extracted using a new parallel algorithm. Finally, a segmentation is performed using the centerlines as seeds for a region growing procedure. However, if only the centerlines are needed for a given application, the segmentation step can be skipped. The rest of this section will describe each of the five steps in further detail.

2.1 Cropping

Memory on the GPU is limited and may not be enough for processing large datasets. However, most medical datasets contain a lot of data that is not part of the structures of interest. Usually these areas are located at the borders of the image. For instance, in the thorax CT image in Fig. 3, the actual lungs where the airways and blood vessels are located, constitutes only about 50% of the image. The rest consist of space outside the body, body fat and the bench that the patient is resting on. As several of the methods

[1] http://github.com/smistad/Tube-Segmentation-Framework/

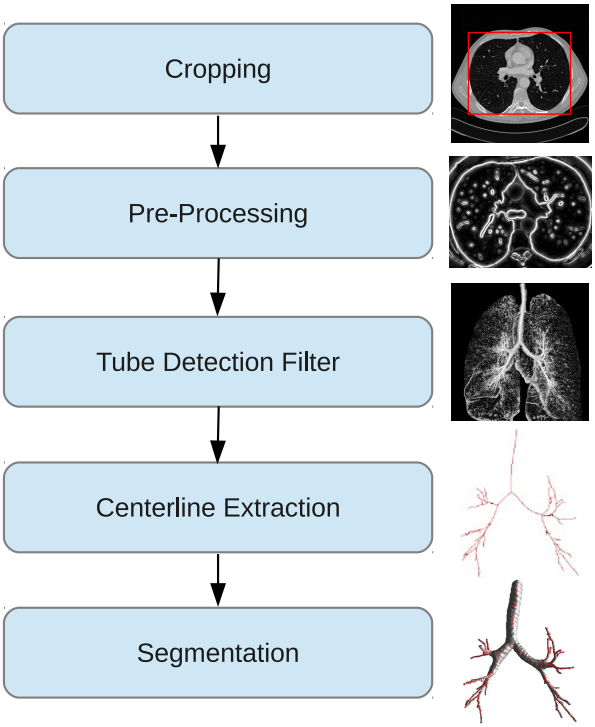

Figure 2: Block diagram of the implementation

used to perform segmentation and centerline extraction process each voxel in the entire volume, removing the unnecessary data will not only reduce memory usage, but also execution time.

In our previous work [41], we presented a novel cropping algorithm for airway segmentation that could be run in parallel on the GPU using less than half a second for large CT volumes of the lungs. In this paper, this algorithm is extended to crop other medical datasets, such as MR and 3D Doppler Ultrasound. The cropping method works by considering slices in all three orthogonal directions x, y and z. For each slice s, the method determines if the slice intersects the region of interest (ROI). This is done by counting the number of rows in the slice that intersects the ROI for each slice and storing it as L_s. If $L_s > L_{\text{min}}$, the slice is considered to have intersected the ROI. The cropping borders are found by traversing through L_s twice from $s = 0$ and $s =$ size and finding the first slice that has a value above a specific threshold L_{min}. These slices are then selected as cropping borders c_1 and c_2. This is done for each direction and results in 3 pairs of cropping borders which is all that is needed to crop the volume. An example of how this cropping procedure works is shown in Fig. 3. For some applications and directions it may be necessary to start the search from the middle $s = \frac{\text{size}}{2}$ to the end instead. This was the case for the axial direction of CT airway datasets.

Algorithm 1 provides pseudocode for the cropping method. The function CALCULATEL is used for estimating L for each slice in a given direction and the function FINDCROPBORDERS is used to find the cropping borders for a specific direction given L and using

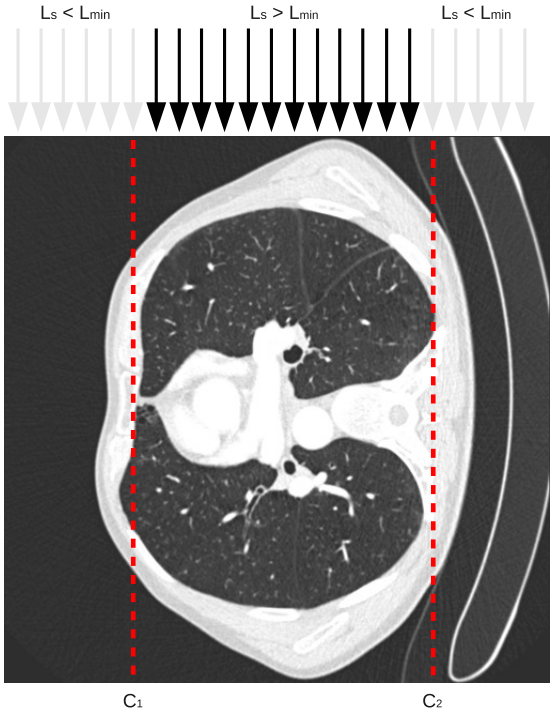

Figure 3: Example of the cropping procedure. The black arrows indicate slices that have $L_s > L_{\min}$ and thus intersected the ROI while the grey arrows are the opposite. This can be used to find the cropping borders, marked with dotted red lines in the figure. This is done in all three directions and each slice is processed in parallel.

the threshold $L_{\min}$. Each direction and slice can be processed in parallel on the GPU. For a dataset of size 512x512x512 this results in 3*512 individual threads that can be processed using the same instructions.

The parts of this cropping method that is application dependent, aside from the parameter $L_{\min}$, is the estimation of L_s and whether the search for cropping borders starts from the middle or at the ends of the dataset in a given direction.

For MR and 3D Doppler Ultrasound images it is sufficient to remove the background from the dataset. For this purpose L_s can be estimated by counting the number of voxels n_v on a scan line that is above a certain threshold I_T. L_s is then set to the number of rows in slice s where $n_v > I_T$. A threshold value of 100 was found to be enough for the MR and Ultrasound datasets.

For CT images of the lungs, fat and other tissue that are not part of the lungs can also be discarded by counting the number of areas that are above and below a certain threshold. Details on this estimation of L_s can be found in our previous work [41].

Algorithm 1 Cropping

```
function CROP(volume)
    L ← CALCULATEL(volume, x)
    x1, x2 ← FINDCROPBORDERS(L, x)
    L ← CALCULATEL(volume, y)
    y1, y2 ← FINDCROPBORDERS(L, y)
    L ← CALCULATEL(volume, z)
    z1, z2 ← FINDCROPBORDERS(L, z)
    crop volume according to x1,x2,y1,y2,z1 and z2
     return volume
end function

function CALCULATEL(volume, direction)
    for each slice s in direction in parallel do
        Estimate Ls
    end for
     return L
end function

function FINDCROPBORDERS(L, direction)
    size ← volume.direction.size
    c1 ← −1, c2 ← −1, s ← 0
    while (c1 = −1 or c2 = −1) and s < size do
        if Ls > Lmin and c1 = −1 then
            c1 ← s
        end if
        if Lsize−1−s > Lmin and c2 = −1 then
            c2 ← size −1 − s
        end if
        s ← s + 1
    end while
     return c1, c2
end function
```

2.2 Pre-processing and Gradient Vector Flow

Before the actual tube extraction, some pre-processing is necessary. First, an optional thresholding is performed on the dataset using a lower and upper threshold (I_{min} and I_{max}). Thresholding may be necessary for datasets which have a large range of intensity values such as CT images. The thresholding is done to remove unnecessary gradient information in the image which may lead to unwanted tubular structures being detected. For instance, when extracting airways all intensities above -500 HU can be converted to -500 as no airways have intensity above this threshold. Second, some noise suppression is

performed. This is done by blurring the dataset using Gaussian smoothing with standard deviation σ. Afterwards, the gradient vector field $\vec{V}$ is created and normalized using a parameter called $V_{\max}$. All gradients with a length above this parameter will be set to unit length and the others will be scaled accordingly. The gradient normalization is necessary for contrast invariance. $V_{\max}$ should be adapted to the expected level of contrast and noise. Also, if black tubular structures are to be extracted (e.g. airways), the gradients have to inverted $\vec{V} = -\nabla I$. All of these pre-processing parameters ($I_{\min}$, $I_{\max}$, σ, $V_{\max}$) are modality dependent and the values used in this paper for each modality is collected in Table 1.

Filters that use the Hessian matrix to detect tubular structures require gradient information to be present in the center of the tube. For large tubes, such as *trachea* and the main *bronchi*, the gradient information will not exist in the center. Thus, it is necessary to propagate the gradient information from the tube edge to the center. There exist two main methods of doing this: Gaussian scale space and Gradient Vector Flow (GVF). Xu et al. [47] originally introduced GVF as an external force field for active contours. Bauer et al. [5, 3] were the first to show that GVF could be used to create scale-invariance of TDFs. The GVF method has the advantage that it is feature-preserving and thus can avoid the problem of several structures diffusing into each other to create the illusion of a tubular structure at a higher scale. Also, GVF is only calculated using one scale. However, it has the disadvantage that it is very computationally expensive. Nevertheless, it has been shown that GVF can be accelerated using GPUs. Eidheim et al. [13], He and Kuester [20] and Zheng and Zhang [48] all presented a GPU implementation of GVF and Active Contours using shader languages. However, their implementation was for 2D images only. In this paper, a highly optimized 3D GPU implementation of GVF from Smistad et al. [42] was used with a predefined number of 250 iterations. This implementation allows GVF to be calculated for large volumes in only a few seconds.

2.3 Tube Detection Filter

Krissian et al. [26] created a TDF that assumes that the cross-section of the tubular structure is circular. Their TDF calculates how well a circle match the gradient information in the cross-sectional plane defined by the eigenvectors of the Hessian matrix. The TDF starts by creating a circle with a small radius in the cross-sectional plane. $N = 32$ evenly spaced points on the circle is sampled from the vector field. Each point, i, is found by calculating its angle α from the center and then calculating a vector $\vec{d_i}$ which lies in the plane and has angle α.

$$\alpha = \frac{2\pi i}{N} \tag{1}$$

$$\vec{d_i} = \vec{e}_2 \sin\alpha + \vec{e}_3 \cos\alpha \tag{2}$$

The position of point i on a circle with radius r and center $\vec{v}$ is then given as $\vec{v} + r\vec{d_i}$. How well the circle match the gradient information is calculated as the average dot product of

the gradient at position i and the inward normal of the circle at point i which is equal to $-\vec{d_i}$. The TDF of Krissian et al. [26] is shown in equation 3. The radius of the circle is increased with 0.5 voxels as long as the average dot product also increases.

$$T(\vec{v},r,N) = \frac{1}{N}\sum_{i=0}^{N-1} \vec{V}(\vec{v}+r\vec{d_i}) \cdot -\vec{d_i} \tag{3}$$

As noted by Bauer et al. [7, 3], the GVF method may eliminate the gradient information for small low-contrast tubular structures. Thus to detect these tubular structures it is necessary to run the TDF two times. Once with a small radius on the initial vector field to detect the small low-contrast structures and once with the GVF vector field to detect the rest. Different amounts of Gaussian blur can be used for the tube detection of large and small structures (σ_{small} and σ_{large} as seen in Table 1). The TDF response from each of these are combined by selecting the largest TDF value for each voxel.

2.4 Centerline Extraction

Centerline extraction from TDF results has primarily been done by ridge traversal [2, 4, 6, 5]. One problem with the ridge traversal procedure is that it can't be run in parallel. Thus, the GVF vector field and the TDF result has to be transferred to the CPU. Nevertheless, the serial ridge traversal algorithm can be used together with the rest of the GPU algorithms presented in this paper (e.g. cropping, pre-processing, tube detection and segmentation).

In this section, a new parallel centerline extraction (PCE) algorithm is presented. This centerline algorithm, unlike ridge traversal, can be run efficiently in parallel on a GPU. The method has 4 main steps: Identifying centerpoints, filtering centerpoints, link centerpoints and centerline selection.

2.4.1 Identify candidate centerpoints

The method for extracting centerlines starts by identifying all possible centerpoints. This is done by creating a 3D structure with the same size as the dataset. This structure is initialized to 0 for each voxel and all voxels with a TDF value above the threshold $T_c = 0.5$ is set to 1.

2.4.2 Filter centerpoints

The next step removes centerpoints that are either not in the center of a tube or too close to other centerpoints. Whether a centerpoint is in the center of a tube or not can be determined by the magnitude of the GVF vector field $|\vec{V}|$, because $|\vec{V}|$ is smallest in the center of the tube.

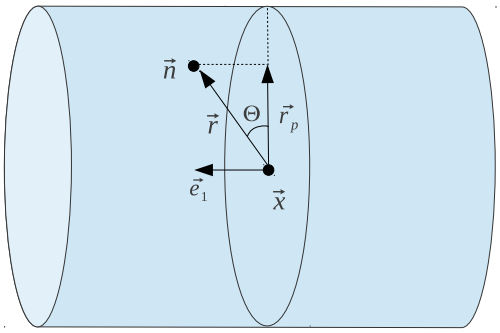

Figure 4: Determining the angle θ from a centerpoint $\vec{x}$ to its neighbor $\vec{n}$.

First, a vector from the centerpoint $\vec{x}$ to a neighbor voxel $\vec{n}$ is calculated:

$$\vec{r} = \vec{n} - \vec{x} \tag{4}$$

Second, this vector is projected onto the cross-sectional plane of the tube (see Fig. 4). The plane's normal $\vec{e}_1$ is the eigenvector of the Hessian matrix associated with the eigenvalue of smallest magnitude. This vector points in the direction of the tube.

$$\vec{r}_p = \vec{r} - \vec{e}_1(\vec{e}_1 \cdot \vec{r}) \tag{5}$$

Finally, the angle θ from the plane to the vector $\vec{r}$ can be calculated using the projected vector $\vec{r}_p$:

$$\theta = \cos^{-1}\left(\frac{\vec{r} \cdot \vec{r}_p}{|\vec{r}||\vec{r}_p|}\right) \tag{6}$$

Let $\vec{N}$ be the set of all neighbor voxels that are close ($|\vec{r}| < r$, where r is from Eq. 3) and the angle is $\theta < 30^\circ$. For each of these $\vec{n}$, the magnitude of the GVF vector field $|\vec{V}|$ is compared to the centerpoint $\vec{x}$. The centerpoint is only valid if the magnitude for the centerpoint $\vec{x}$ is lower than all $\vec{n} \in \vec{N}$:

$$C(\vec{x}) = \begin{cases} 1 & \text{if} \quad \forall \vec{n} \in \vec{N} \quad |\vec{V}(\vec{n})| > |\vec{V}(\vec{x})| \\ 0 & \text{else} \end{cases} \tag{7}$$

This has the effect that it removes centerpoints that are not in the center of a tubular structure.

The next step is to remove centerpoints that are too close to each other. The reason for doing this is that it reduces the total number of centerpoints and thus makes the next step, linking the centerpoints, much more efficient. Removing points that are too close to each other is done by dividing the entire dataset into a grid with each grid element spanning 4x4x4 voxels. For each cube in the grid, the best centerpoint is selected and the rest of the centerpoints in that cube is removed. The centerpoint with the highest TDF value is selected as the best centerpoint in a cube.

2.4.3 Link centerpoints

For each centerpoint, the method establishes links between the centerpoints to create centerlines. This is done by connecting each centerpoint to the two centerpoints that are closest and fulfills the following criteria:

- The angle between them is above 120 degrees.
- The average TDF value along the line is higher than $T_{\text{mean}} = 0.5$.

2.4.4 Centerline selection

Due to noise and other image artifacts invalid centerpoints and centerlines may be created. However, these are usually short and not connected to the actual tubular structures. Thus invalid centerlines can often be discarded based on their length.

In this step, all centerpoints that are connected with centerlines from the previous section are assigned the same label. Those that are not connected get different labels. Graph component labeling is the problem of finding and labeling nodes in a graph that are connected. Hawick et al. [19] presented several GPU implementations of algorithms for graph component labeling. In our implementation, an iterative method using atomic operations was used. Assuming N labels, N counters are created and initialized to 0. A kernel is executed for each centerpoint and the length of each centerline, identified with a label, is determined by using an atomic increment operation on the counter identified by the centerpoints' labels. After the execution of this kernel, the counters will contain the total length of each centerline. When the length of all connected centerlines have been calculated, the largest centerline or all centerlines with a specified minimum length can be extracted.

2.5 Segmentation

Bauer et al. [7] proposed a method for generating 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, but in the inverse direction. This segmentation method is a type of seeded region growing, where the centerlines are the seeds and the direction and magnitude of the vectors from the GVF vector field is used to determine if the segmentation is allowed to continue to grow.

In this paper, a data parallel version of this algorithm is presented (see Algorithm 2). First, the centerlines, C, are dilated in parallel on the GPU and added to the segmentation S. Next, the neighboring voxels of S is added to a queue Q. For each iteration, the GROW function runs a kernel on each voxel $\vec{x}$ in the entire volume. If the voxel $\vec{x}$ is part of Q, the gradients of all unsegmented neighbors are checked to see if they point to $\vec{x}$ and

has a larger magnitude than $\vec{x}$. If such a neighbor voxel $\vec{y}$ is found, $\vec{x}$ is added to S, its neighbor $\vec{y}$ is added to Q and the stopGrowing variable is set to false. Since this variable is initialized to true for every iteration, the growing procedure will stop when no more voxels are added.

For the 3D Ultrasound Doppler modality another segmentation method than the inverse gradient tracking method is used. The reason for this, is that this data can be quite noisy. This alternative segmentation method starts by calculating an average radius based on the circle fitting method for each link. For each discrete point on the centerline, all voxels within a sphere with the same radius is marked as part of the segmentation.

Algorithm 2 Parallel Inverse Gradient Flow Tracking

```
S ← DILATE(C)
Q ← DILATE(S) - S
stopGrowing ← false
while !stopGrowing do
    stopGrowing ← true
    GROW(S, Q, stopGrowing)
end while
return S

function GROW(S, Q, stopGrowing)
    for each voxel x⃗ in parallel do
        if x⃗ ∈ Q then
            for each voxel y⃗ ∈ Adj26(x⃗) do
                if y⃗ ∉ S and |V⃗(y⃗)| > |V⃗(x⃗)| and
                  argmax_{z⃗∈Adj26(y⃗)} ( ((z⃗−y⃗)·V⃗(y⃗)) / (|(z⃗−y⃗)||V⃗(y⃗)|) ) = x⃗ then
                    S ← S ∪ {x⃗}
                    Q ← Q ∪ {y⃗}
                    stopGrowing ← false
                end if
            end for
        end if
    end for
end function
```

2.6 GPU Optimization

2.6.1 Texture system

The GPU has a specialized memory system for images, called the texture system. The GPU has this because the GPU is primarily made and used for fast rendering which involves mapping images, often called textures, onto 3D objects.

The texture system is optimized for fetching and caching data from 2D and 3D textures [37, 1] (see Fig. 1 for an overview of the memory hierarchy on the GPU). The fetch unit of the texture system is also able to perform interpolation and data type conversion in hardware.

Since most of the calculations in this implementation involves the processing of voxels, the implementation can be accelerated considerably by storing the volumes as 3D textures and using the texture system. This increases the speed of fetching data and trilinear interpolation which is used in the TDF calculation when sampling arbitrary points on a circle.

In this implementation, textures has been used for almost all 3D and 2D structures, such as the vector field $\vec{V}$, TDF and segmentation.

NVIDIA's OpenCL implementation does not support writing to 3D textures in a kernel. Thus for NVIDIA GPUs, the results has to be written to a regular buffer first and then copied to a texture. Still, writing to 3D textures is possible with CUDA.

Memory access latency can also be improved by reducing the number of bytes transferred from global memory to the chip. The most common way to store a floating point number on a computer is by using 32 bits with the IEEE 754 standard. However, most GPUs also support a texture storage format called 16-bit normalized integer. With this format, the data is stored as 16-bit integers (shorts) in textures. However, when it is requested, the texture fetch unit converts the 16-bit integer to a 32-bit floating point number with a normalized range from -1.0 to 1.0 or 0.0 to 1.0. This reduces accuracy, and may not be sufficient for all applications. However, it was found to be sufficient for this application (see result section). This storage format also halves the global memory usage, thus allowing much larger volumes to fit in the limited GPU memory. In our recent work on optimizing GVF for GPU execution [42], it was discovered that using textures and the 16-bit format could make the parallel execution a lot faster, depending on the size of the dataset being processed. In this implementation, the 16-bit normalized integer format is used for the dataset, vector fields and TDF result.

2.6.2 Stream compaction

After finding the candidate centerpoints, we only want to process these points in the next centerpoint filtering step. This can be done by launching a kernel for every voxel in the volume and have an if statement checking whether the voxel is a candidate centerpoint. However, this can be very inefficient on a GPU. As explained in the introduction, the functional units on the GPU are grouped together and share a control unit. This means that the functional units in a group have to execute the same instructions in each clock cycle. To ensure that the correct result is generated by if statements, the GPU will use masking techniques. Nevertheless, such an if statement may not reduce the processing time as it would if it was executed sequentially on a CPU. On a GPU, it might even increase the processing time due to the need of masking techniques to ensure correct results. This

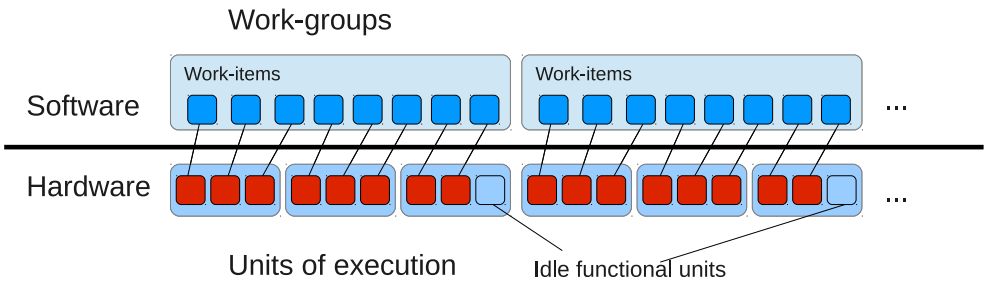

Figure 5: Grouping with work-group size of 8 work-items and unit of execution size of 3. As 8 is not a multiple of 3, there will be idle functional units for each work-group that is scheduled. This leads to an inefficient use of the GPU.

is a common problem in GPU computing and one solution is a method called stream compaction. Stream compaction removes voxels that should not be processed from the volume so that the kernel is only run for the valid voxels, thus no if statement is needed. Stream compaction can be done on the GPU with logarithmic time complexity. Two methods for performing stream compaction is parallel prefix sum (see Billeter et al. [11] for an overview) and Histogram Pyramids by Ziegler et al. [49]. In this work, Histogram Pyramids has been used due to the fact that this data structure has shown to be better in some applications by exploiting the GPU's texture system for faster memory access. The original implementation by Ziegler et al. [49] was for 2D. However, in our previous work [43], we presented a 3D version of this stream compaction algorithm which also reduced the memory usage for this data structure.

The Histogram Pyramid stream compaction method has been used in three places of this implementation. All in the centerline extraction step. The 3D Histogram Pyramid is used after the candidate centerpoint step and filter centerpoints step. A 2D Histogram Pyramid is used after the link centerpoints step, where each link is stored in an adjacency matrix on the GPU.

2.6.3 Work-group size

Work-items, also called threads, 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 GPUs respectively. The work-items are also grouped together at a higher level in software. These groups are called work-groups in the OpenCL terminology (in CUDA they are referred to as thread blocks). If the number of work-items in a work-group is not a multiple of the unit of execution size, some of the GPUs' functional units will be idle for each work-group that is executed as shown in Fig. 5. Thus, the work-group sizes can greatly affect performance and optimal size can vary a lot from device to device. There is a maximum number of work-items that can exists in one work-group. This limit is on AMD GPUs currently 256 and on most NVIDIA GPUs

it is 1024. 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, for 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.

For most of the GPUs used on this implementation a work-group size of 4x4x4 was used. One exception is the new Kepler GPUs from NVIDIA where a work-group of 16x8x8 was found to be much better. The 4x4x4 work-group size gives a total of 64 work-items in each work-group. 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.

2.7 Evaluation

In this section, the evaluation of the proposed GPU method is described.

2.7.1 Comparison with other methods

The method in this paper was compared in terms of speed and quality with other commonly used segmentation and centerline extraction algorithms. Blood vessels from the MR Angio, Doppler Ultrasound and synthetic datasets were segmented using thresholding after performing Gaussian blur. As thresholding is unsuitable for segmenting airways, an implementation of region growing, similar to the conservative region growing used in Graham et al. [16], was used instead. This region growing methods starts by automatically finding a seed point inside *trachea*. This is done by looking for a dark circular region in the middle of one of the upper slices. After a seed has been found, the dataset is filtered with a Gaussian mask with $\sigma = 0.5$ voxels and the intensities are capped at -500 HU as no airways have intensities above this threshold. Next, a region growing procedure with segmentation leakage detection is used. The region growing is performed several times with increasing threshold starting with the intensity of the seed. For each iteration, the volume size is measured. If the volume size increases with more than 20 000 voxels in one iteration a segmentation leakage has most likely occurred and the previous threshold is used. Finally, a morphological closing is performed to remove any holes inside the segmentation.

The proposed GPU implementation can be used together with both the PCE algorithm and the ridge traversal algorithm for the centerline extraction step. Thus, with the serial ridge traversal algorithm a hybrid solution is used where all steps except the centerline extraction step is run on the GPU.

For the centerline extraction, the proposed GPU method is evaluated with both the proposed PCE centerline algorithm and the ridge traversal algorithm and compared to an ITK filter by Homann [22] based on the skeletonization algorithm by Lee et al. [28]. This skeletonization method performs iterative thinning of a segmented volume. Note that the implementation by Homann [22] does not exploit parallelism.

2.7.2 Qualitative analysis

To show the general applicability of the method, clinical images from three different modalities and two different organs were used:

1. Computer Tomography scans of the lungs (Airways, 12 datasets)
2. Magnetic Resonance images of the brain (Blood vessels, 4 datasets)
3. 3D Ultrasound Doppler images of the brain (Blood vessels, 7 datasets)

The study was approved by the local ethics committee, and the patients gave informed consent prior to the procedure. For each modality, several datasets were processed using the proposed GPU implementation together with the PCE and the ridge traversal centerline algorithms and region growing / thresholding together with skeletonization.

Note that for each modality the same parameters were used, except for a small set of modality dependent parameters such as blur and radius (see Table 1).

2.7.3 Speed and memory usage

The speed of the method was measured on all the clinical datasets using three different GPUs from both AMD and NVIDIA. Two high-end GPUs with a peak performance of around 4 tera floating point operations per second (TFLOPS) (AMD HD7970 and NVIDIA Tesla K20). And one GPU of the previous generation with a peak performance of about 1 TFLOPS (NVIDIA Tesla C2070). The implementation was run using both 16-bit normalized integers and 32-bit floating point vectors to see how the two different data types affected the speed. The proposed method was also run on an Intel i7-3770 CPU (4 cores, 3.4 GHz) with 16 GB memory to show the speedup of using a GPU versus a multi-core CPU. This was also done to demonstrate that the proposed implementation can be run in parallel on a multi-core CPU with no modification.

For comparison, runtime measurements for region growing, thresholding and skeletonization were performed for each modality using an Intel i7-3770 CPU with 4 cores running at 3.4 GHz. Parts of the region growing and thresholding methods were parallelized using OpenMP.

As explained earlier, the memory available on GPUs is limited. Thus it is important to keep the memory usage as low as possible. In this paper, a cropping procedure and a 16-bit normalized integer data format was used to reduce the memory usage on the GPU. To show the effect of the cropping procedure, the average dataset size and peak memory usage before and after cropping was measured on several datasets from different modalities. Peak memory usage occurs in the Gradient Vector Flow step. In this step, 3 vector fields with 3 components, each of the same size as the dataset are needed. For an uncropped volume of size 512x512x800 and 32-bit floats this amounts to $3*3*4*512*512*800$ bytes = 7200 MB. When using 16-bit normalized integers the memory usage is halved.

2.7.4 Quantitative analysis

The quality of the extracted centerlines and the segmentation were measured using realistic synthetic vascular tree volumes and their ground truth segmentation and centerlines. These synthetic volumes and their ground truth data were created using the VascuSynth software by Hamarneh and Jassi [17, 23]. One of these synthetic volumes is depicted in Fig. 6. Three generated datasets were used. Each with a different amount of Gaussian additive noise. This was done to show how well the different methods performs with increasing amounts of noise.

Each discrete point of the centerline is called a centerpoint. The accuracy of the centerline was measured using the Hausdorff distance measure which is the average distance from each centerpoint of the extracted centerline to the closest point on the ground truth centerline. To estimate how much of the vascular tree was extracted, each extracted point marks all ground truth centerpoints within a radius of 4 voxels as detected. The total percentage extracted is then calculated as the number of detected points divided by the total number of ground truth centerpoints. Any extracted centerpoint that was farther away than 4 voxels from a grouth truth centerpoint was marked as invalid. The parameters for the amount of Gaussian blur and $V_{\max}$ were adjusted for each dataset and centerline method so that no extracted centerpoints were marked as invalid. Precision and recall for the segmentation is calculated by comparing each voxel of the segmentation result to the ground truth.

The quantitative analysis was performed using the proposed GPU implementation with both PCE and ridge traversal and thresholding+skeletonization together with 16-bit normalized integers and 32-bit floating point numbers.

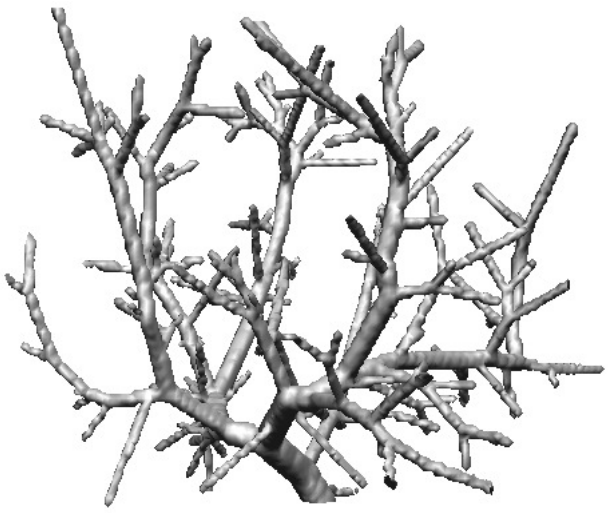

Figure 6: Synthetic vascular image created using the VascuSynth software by Hamarneh and Jassi [17, 23].

Parameter	CT Airways	MR Vessels	US Vessels
I_{min}	-1024	100	50
I_{max}	-400	300	200
σ_{small}	0.5	1.0	2.0
σ_{large}	1.0	1.0	3.0
V_{max}	0.3	0.1	0.1
r_{min}	0.5	0.5	1.5
r_{max}	25	8	7
L_{min}	128	10	0

Table 1: A list of modality dependent parameters and the values used for each of the datasets.

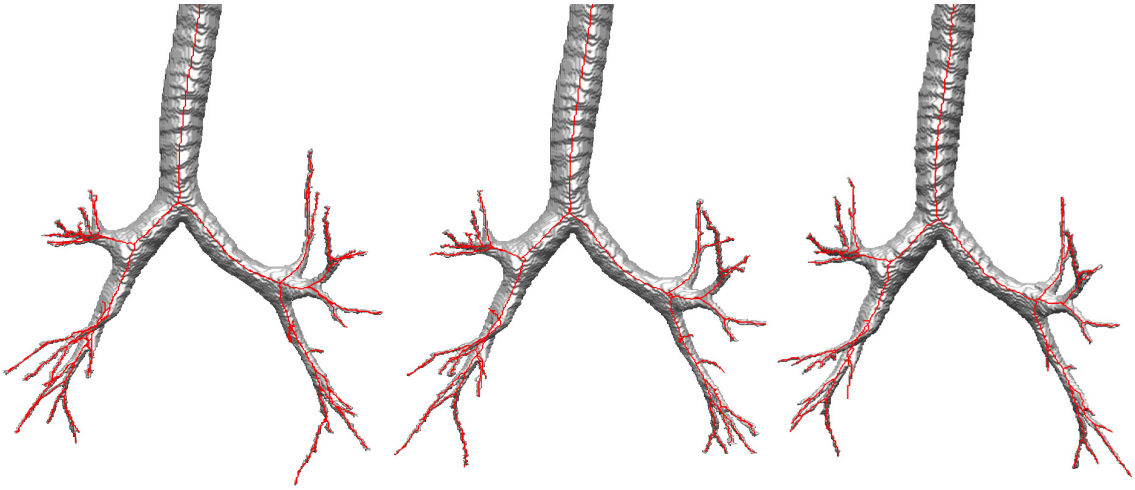

Figure 7: Results for a CT image of the lungs. **Left:** Proposed GPU method + proposed PCE algorithm. **Middle:** Proposed GPU method + ridge traversal algorithm. **Right:** Region growing with skeletonization

3 Results

3.1 Qualitative analysis

Figures 7, 8 and 9 show results for each method on each modality. Also, to further show the general applicability of the method, extracted vessels from liver and lung is included in Fig. 10. These results indicate that the method is able to extract tubular structures from several modalities and organs with comparable quality by changing only a few parameters (see Table 1).

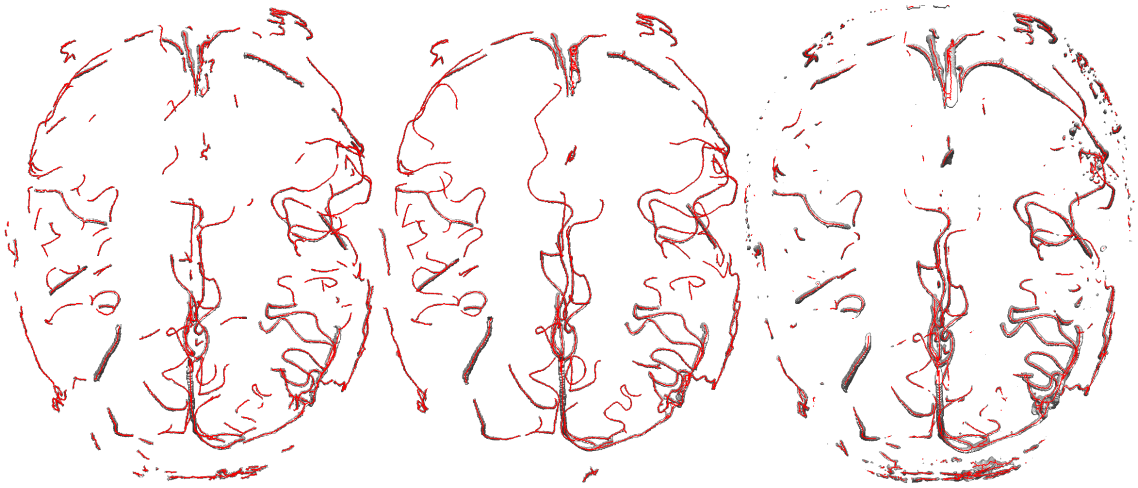

Figure 8: Results for an MR Angio image of the brain. **Left:** Proposed GPU method + proposed PCE algorithm. **Middle:** Proposed GPU method + ridge traversal algorithm. **Right:** Thresholding with skeletonization

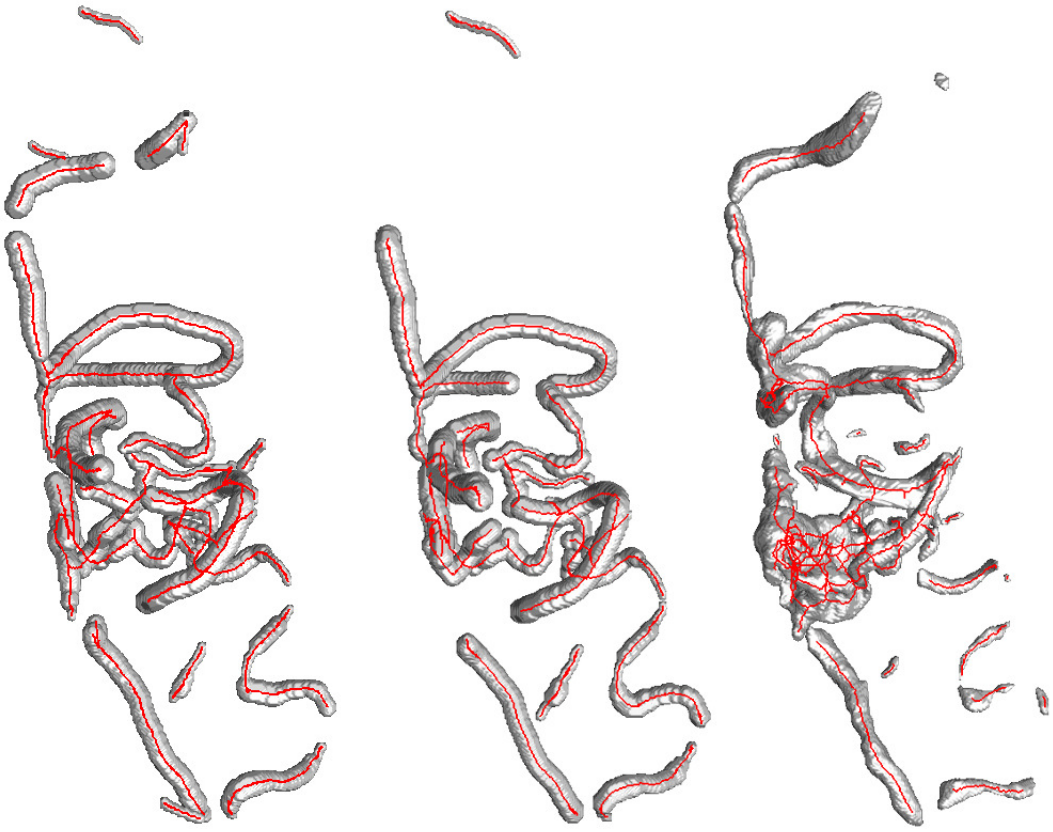

Figure 9: Results for a 3D Ultrasound Doppler image of vessels in the brain. **Left:** Proposed GPU method + proposed PCE algorithm. **Middle:** Proposed GPU method + ridge traversal algorithm. **Right:** Thresholding with skeletonization

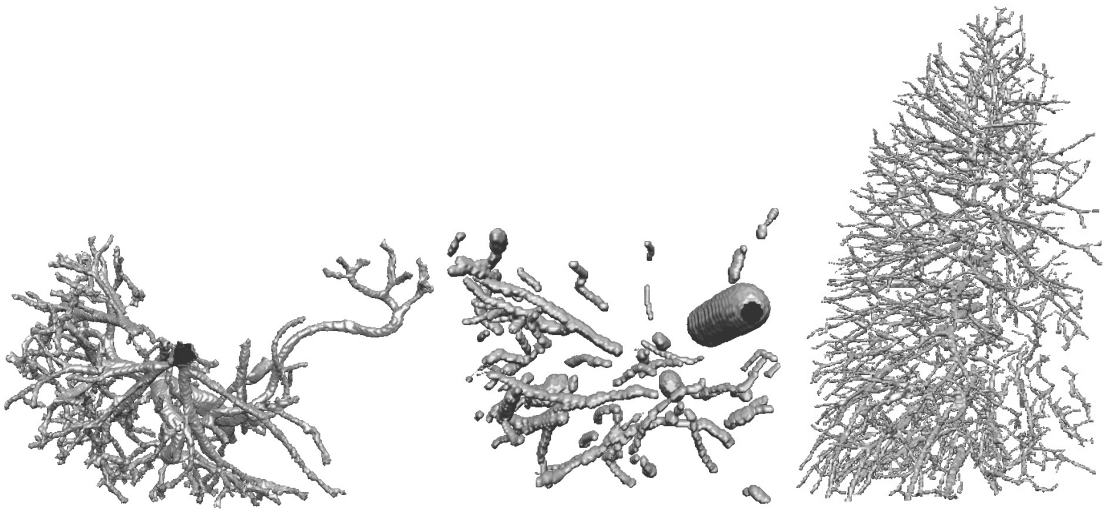

Figure 10: Segmentation result from other organs using proposed GPU method. From left to right: Vessels of liver from CT, vessels of liver from MR and vessels of one lung from CT.

Method	**Datasets**	**AMD HD7970** 16-bit / 32-bit (secs)	**NVIDIA Tesla K20** 16-bit / 32-bit (secs)	**NVIDIA Tesla C2070** 16-bit / 32-bit (secs)	**Intel i7-3770 CPU** 32-bit (secs)
Proposed GPU	CT Airways (12)	4.7 / 6.9	21.9 / 13.4	40.9 / 19.2	177.1
implementation	MR Vessels (4)	4.6 / 6.6	28.7 / 16.4	44.9 / 26.6	200.7
+ Proposed PCE	US Vessels (7)	2.7 / 3.8	13.0 / 7.1	24.1 / 14.8	134.4
Proposed GPU	CT Airways (12)	5.8 / 8.3	22.3 / 13.9	37.3 / 19.3	175.9
implementation	MR Vessels (4)	6.3 / 8.5	29.7 / 17.5	45.3 / 27.3	200.5
+ Ridge traversal	US Vessels (7)	3.4 / 4.7	13.3 / 7.4	24.1 / 15.0	141.5

Table 2: Average runtime of 10 runs using the proposed GPU implementation together with the proposed parallel centerline algorithm and the ridge traversal centerline algorithm on different datasets and devices. The first three devices (HD7970, K20, C2070) are GPUs while the last device (i7-3770) is a multi-core CPU.

3.2 Speed and memory usage

The speed measurements of our GPU implementation with the proposed centerline extraction method and the ridge traversal algorithm is collected in Table 2. These results show that using 16-bit normalized integers is faster than 32-bit on AMD GPUs, and opposite on NVIDIA GPUs.

Table 3 contains speed measurements of the non-GPU methods: region growing, thresholding and skeletonization. Comparing the runtime of Table 2 and 3 reveals that the GPU methods are much faster than the simple serial segmentation and skeletonization methods.

Table 4 shows the average memory usage for all the clinical datasets, both with and without cropping and the 16-bit data type. From these results it is evident that the memory usage is significantly reduced when cropping and 16-bit normalized integers are used.

Segmentation and centerline method	Datasets	Avg. runtime (seconds)
Region Growing + Skeletonization	CT Airways (12)	158
Thresholding + Skeletonization	MR Vessels (4)	77
Thresholding + Skeletonization	US Vessels (7)	33

Table 3: Average runtime of 10 runs using region growing, thresholding and skeletonization/thinning on different modalities.

Datasets	Avg. original size	Avg. percentage removed	Avg. peak memory usage without cropping (MB) 16-bit / 32-bit	Avg. peak memory usage with cropping (MB) 16-bit / 32-bit
CT Airways (12)	512x512x704	76%	3169 / 6339	762 / 1524
MR Vessels (4)	628x640x132	23%	2826 / 5652	793 / 1586
US Vessels (7)	272x288x437	31%	1223 / 2445	417 / 834

Table 4: Memory usage and effect of cropping

3.3 Quantitative analysis

Table 5 contains the results of the quantitative analysis described in 2.7.4. From these results it is clear that using the 16-bit normalized integer format does not affect the quality compared to using the standard 32-bit floating point numbers. The same applies to the clinical datasets.

Furthermore, thresholding is able to extract more from the synthetic datasets for noise levels 0.1 and 0.2. However, for noise level 0.3, the proposed PCE algorithm is able to extract almost 10% more than the thresholding and skeletonization technique and the ridge traversal algorithm.

4 Discussion

4.1 Qualitative analysis

The results of the clinical datasets (Fig. 7, 8 and 9) indicate that the quality of the segmentation and centerlines are quite comparable with some small differences. However, if the segmented tubular structure is very irregular or has holes, skeletonization will create poor centerlines as can be seen in Fig. 9. The PCE and ridge traversal algorithms however, do not suffer from this problem as the centerline extraction is not based on the segmentation result.

There are several examples in the literature of methods that claim to be robust enough to segment and extract centerlines of tubular structures of different types (e.g. vessels and airways), organs and modalities. Some examples are Bauer et al. [3, 4, 5, 6, 7, 8], Krissian et al. [26], Aylward et al. [2], Benmansour et al. [10], Li et al. [30], Behrens et al. [9], Cohen et al. [12], Lorigo and Faugeras [33] and Spuhler et al. [44]. However, most of

Dataset	Noise(σ)	Method	Avg. centerline error (voxels) 16-bit / 32-bit	Extracted centerpoints (%) 16-bit / 32-bit	Segmentation recall 16-bit / 32-bit	Segmentation precision 16-bit / 32-bit
Dataset 1	0.1	Proposed GPU method + PCE	0.57 / 0.58	95.6 / 95.6	0.79 / 0.79	0.84 / 0.84
		Proposed GPU method + Ridge traversal	0.35 / 0.35	92.9 / 92.9	0.78 / 0.78	0.84 / 0.84
		Thresholding + Skeletonization	- / 0.34	- / 98.8	- / 0.70	- / 0.99
Dataset 2	0.2	Proposed GPU method + PCE	0.60 / 0.59	80.9 / 80.8	0.57 / 0.57	0.83 / 0.83
		Proposed GPU method + Ridge traversal	0.31 / 0.31	76.1 / 76.1	0.56 / 0.56	0.86 / 0.86
		Thresholding + Skeletonization	- / 0.36	- / 82.1	- / 0.67	- / 0.89
Dataset 3	0.3	Proposed GPU method + PCE	0.65 / 0.65	54.4 / 54.4	0.36 / 0.36	0.79 / 0.79
		Proposed GPU method + Ridge traversal	0.31 / 0.31	42.4 / 42.4	0.28 / 0.28	0.90 / 0.90
		Thresholding + Skeletonization	- / 0.47	- / 45.6	- / 0.47	- / 0.74

Table 5: Performance on three synthetic dataset created with the VascuSynth software (Hamarneh and Jassi [17, 23]). For each line, the first value is acquired using 16-bit normalized integers and the second using 32-bit floats.

these present results only for a few datasets of one or two organs/modalities. The PhD thesis of Bauer and related articles [3, 4, 5, 6, 7, 8] is one exception that present results for several different organs (e.g. lung, heart and liver), however only from CT. Although their approach is similar to the approach in this paper, Bauer et al. use different methods to perform the major steps (tube detection, centerline extraction and segmentation) for each organ. In this paper, results from several organs (e.g. lung, brain and liver), modalities (e.g. CT, MR and Ultrasound) and structures (e.g. vessels and airways) are presented and use the same method for all the major steps. In addition, the method presented in this paper is open source and very fast.

4.2 Speed and memory usage

The proposed GPU implementation is slightly slower using 1-2 seconds more when used with the ridge traversal centerline extraction method than PCE on the two fastest GPUs, the AMD HD7970 and the NVIDIA Tesla K20. However, for the slower GPU, the proposed GPU implementation with ridge traversal is just as fast or even faster. Since this GPU have a peak performance of about one fourth to that of the HD7970 and K20 GPUs, the parallel computation cost of PCE on this slower device is most likely higher than the ridge traversal computation plus the data transfer time.

It is clear from the results that using 16-bit normalized integers instead of 32-bit floats for the vector fields is faster on AMD GPUs, and slower on NVIDIA GPUs. This is due to the fact that NVIDIA's OpenCL implementation does not support writing directly to 3D textures. Because of this restriction, buffers have to be used in the most computationally expensive step, Gradient Vector Flow. This means no 3D cache optimization and hardware

data type conversion. Both of which can increase performance.

The runtime of the proposed GPU implementation on a multi-core Intel CPU is several minutes compared to a few seconds on the high-end GPUs. This illustrates the huge speedup gained from running tube detection and segmentation on the GPU.

Skeletonization is the most time-consuming step of the serial methods and is mainly dependent on the thickness of the tubular structures. This is evident in the long execution time of over 2 minutes when processing the airway datasets. Nevertheless, the skeletonization implementation used in this comparison does not exploit parallelism.

Helmberger et al. [21] noted that it is difficult to process a large CT scan due to the limited memory on the GPU. They solved this challenge by decomposing the volume into overlapping sub-volumes that are processed sequentially on the GPU. However, this takes more time and they reported runtime of several minutes. In this paper, the memory limit is avoided by performing cropping and using a 16-bit normalized integer data format. Table 4 shows that the cropping algorithm is able to discard a large portion of the total input volume. This reduces memory usage significantly and without it, no GPU at the present time would have enough memory to perform the entire calculation in one step for large medical images. Using 16-bit for storage also halves the memory usage allowing larger volumes to be processed entirely on the GPU. On average, the peak memory usage is below 1 GB when cropping and 16-bit data types are used, which is below the memory limit of most modern GPUs.

4.3 Quantitative analysis

The average centerline error is worse for the proposed PCE algorithm than the ridge traversal and skeletonization methods. This increased centerline error is due to the fact that the PCE algorithm creates straight lines between centerpoints. However, it is below 0.7 voxels which we argue is not problematic for most applications and this approximation enables the proposed PCE algorithm to extract over 10% more of the synthetic vascular tree compared to the ridge traversal algorithm for large noise levels (0.3).

Thresholding assumes that all voxels with an intensity above some threshold is part of the tubular structures. This assumption is correct for these synthetic datasets and is thus able to extract more for noise levels 0.1 and 0.2. However, this assumption is usually never correct for a clinical dataset and especially not if the noise level is high. This is evident with noise level 0.3 and in the MR Angio modality in Fig. 8 where the segmentation contains some noise and parts of the cranium.

5 Conclusion

In this article, a fast and generic method that can extract tubular structures such as blood vessels and airways from images of different modalities (CT, MR and US) and organs (brain, lungs and liver) was presented. This was achieved by utilizing the computational power of modern Graphic Processing Units. The method was compared to other methods such as region growing, thresholding, skeletonization by thinning and ridge traversal. Results from both synthetic and clinical datasets from three different modalities (CT, MR and US) was presented. The results show that the method is able to extract airways and vessels in 3-5 seconds on a modern GPU. These near real-time speeds can be beneficial in reducing processing time in image guided surgery applications such as bronchoscopy, laparoscopy and neurosurgery. Although faster and more general than other methods, the quality of the centerline and segmentation was found to be comparable for all the methods.

Acknowledgements

Thank you to the people of the Heterogeneous and Parallel 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 and NVIDIA's CUDA Research Center Program for their hardware contributions to the HPC Lab. Without their continued support this project would not have been possible.

Conflict of interest Erik Smistad, Anne C. Elster and Frank Lindseth declare that they have no conflict of interest.

References

[1] AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide. Technical Report December, 2012. `http://developer.amd.com/download/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf` - accessed 4. July 2013.

[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. In *Proceedings of the 30th DAGM Symposium on Pattern Recognition*, pages 163–172. Springer, 2008.

[5] C. Bauer and H. Bischof. Edge based tube detection for coronary artery centerline extraction. *The Insight Journal*, 2008.

[6] C. Bauer and H. Bischof. Extracting curve skeletons from gray value images for virtual endoscopy. In *Proceedings of the 4th International Workshop on Medical Imaging and Augmented Reality*, pages 393–402. Springer, 2008.

[7] C. Bauer, H. Bischof, and R. Beichel. Segmentation of airways based on gradient vector flow. In *Proceedings of the 2nd International Workshop on Pulmonary Image Analysis. MICCAI*, pages 191–201. Citeseer, 2009.

[8] 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. MICCAI*, pages 203–214. Citeseer, 2009.

[9] T. Behrens, K. Rohr, and H. S. Stiehl. Robust segmentation of tubular structures in 3-D medical images by parametric object detection and tracking. *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*, 33(4):554–61, Jan. 2003.

[10] F. Benmansour and L. D. Cohen. Tubular Structure Segmentation Based on Minimal Path Method and Anisotropic Enhancement. *International Journal of Computer Vision*, 92(2):192–210, Mar. 2010.

[11] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the Conference on High Performance Graphics*, pages 159–166, 2009.

[12] L. D. Cohen and T. Deschamps. Segmentation of 3D tubular objects with adaptive front propagation and minimal tree extraction for 3D medical imaging. *Computer methods in biomechanics and biomedical engineering*, 10(4):289–305, Aug. 2007.

[13] O. Eidheim, J. Skjermo, and L. Aurdal. Real-time analysis of ultrasound images using GPU. *International Congress Series*, 1281:284–289, May 2005.

[14] M. Erdt, M. Raspe, and M. Suehling. Automatic hepatic vessel segmentation using graphics hardware. In *Proceedings of the 4th international workshop on Medical Imaging and Augmented Reality*, pages 403–412, 2008.

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

[16] M. W. Graham, J. D. Gibbs, D. C. Cornish, and W. E. Higgins. Robust 3-D airway tree segmentation for image-guided peripheral bronchoscopy. *IEEE transactions on medical imaging*, 29(4):982–97, Apr. 2010.

[17] G. Hamarneh and P. Jassi. VascuSynth: simulating vascular trees for generating volumetric image data with ground-truth segmentation and tree analysis. *Computerized medical imaging and graphics*, 34(8):605–616, Dec. 2010.

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

[19] K. Hawick, a. Leist, and D. Playne. Parallel graph component labelling with GPUs and CUDA. *Parallel Computing*, 36(12):655–678, Dec. 2010.

[20] Z. He and F. Kuester. GPU-Based Active Contour Segmentation Using Gradient Vector Flow. In *Advances in Visual Computing*, pages 191–201, 2006.

[21] M. Helmberger, M. Urschler, M. Pienn, Z. Bálint, A. Olschewski, and H. Bischof. Pulmonary Vascular Tree Segmentation from Contrast-Enhanced CT Images. In *Proceedings of the 37th Annual Workshop of the Austrian Association for Pattern Recognition*, pages 1–10, Apr. 2013.

[22] H. Homann. Implementation of a 3D thinning algorithm. *The Insight Journal*, 2007.

[23] P. Jassi and G. Hamarneh. VascuSynth: Vascular Tree Synthesis Software. *The Insight Journal*, 2011.

[24] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, Jan. 1988.

[25] C. Kirbas and F. Quek. A review of vessel extraction techniques and algorithms. *ACM Computing Surveys*, 36(2):81–121, June 2004.

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

[27] T.-Y. Law and P. A. Heng. Automated extraction of bronchus from 3D CT images of lung based on genetic algorithm and 3D region growing. *Proceedings of SPIE*, 3979:906–916, 2000.

[28] T. Lee, R. Kashyap, and C. Chu. Building skeleton models via 3-D medial surface/axis thinning algorithms. *CVGIP: Graphical Model and Image Processing*, 56(6):462–478, 1994.

[29] 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):819–845, Dec. 2009.

[30] H. Li and A. Yezzi. Vessels as 4-D curves: global minimal 4-D paths to extract 3-D tubular surfaces and centerlines. *IEEE transactions on medical imaging*, 26(9):1213–23, Sept. 2007.

[31] 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.

[32] P. Lo, J. Sporring, H. Ashraf, J. J. H. Pedersen, and M. de Bruijne. Vessel-guided airway tree segmentation: A voxel classification approach. *Medical image analysis*, 14(4):527–538, Mar. 2010.

[33] L. Lorigo and O. Faugeras. Codimension-two geodesic active contours for the segmentation of tubular structures. In *Computer Vision and Pattern Recognition*, pages 444–451, 2000.

[34] J. B. A. Maintz and M. A. Viergever. A survey of medical image registration. *Medical Image Analysis*, 2(1):1–36, 1998.

[35] R. Malladi, J. Sethian, and B. Vemuri. Shape Modeling with Front Propagation: A Level Set Approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(2):158–175, 1995.

[36] A. Narayanaswamy, S. Dwarakapuram, C. S. Bjornsson, B. M. Cutler, W. Shain, and B. Roysam. Robust adaptive 3-D segmentation of vessel laminae from fluorescence confocal microscope images and parallel GPU implementation. *IEEE transactions on medical imaging*, 29(3):583–597, Mar. 2010.

[37] NVIDIA. OpenCL Best Practices Guide. Technical report, 2010. `http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA\_OpenCL\_BestPracticesGuide.pdf` - accessed 4. July 2013.

[38] I. Reinertsen, F. Lindseth, G. Unsgaard, and D. L. Collins. Clinical validation of vessel-based registration for correction of brain-shift. *Medical image analysis*, 11(6):673–684, Dec. 2007.

[39] L. Shi, W. Liu, H. Zhang, Y. Xie, and D. Wang. A survey of GPU-based medical image computing techniques. *Quantitative Imaging in Medicine and Surgery*, 2(3):188–206, 2012.

[40] 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.

[41] E. Smistad, A. C. Elster, and F. Lindseth. GPU-Based Airway Segmentation and Centerline Extraction for Image Guided Bronchoscopy. In *Norsk informatikkonferanse*, pages 129–140. Akademika forlag, 2012.

[42] E. Smistad, A. C. Elster, and F. Lindseth. Real-time gradient vector flow on GPUs using OpenCL. *Journal of Real-Time Image Processing*, 2012.

[43] E. Smistad, A. C. Elster, and F. Lindseth. Real-Time Surface Extraction and Visualization of Medical Images using OpenCL and GPUs. In *Norsk informatikkonferanse*, pages 141–152. Akademika forlag, 2012.

[44] C. Spuhler, M. Harders, and G. Székely. Fast and Robust Extraction of Centerlines in 3D Tubular Structures Using a Scattered-Snakelet Approach. *Proc. SPIE*, 6144, Mar. 2006.

[45] B. van Ginneken, W. Baggerman, and E. M. van Rikxoort. Robust segmentation and anatomical labeling of the airway tree from thoracic CT scans. *International Conference on Medical Image Computing and Computer-Assisted Intervention*, 11:219–26, Jan. 2008.

[46] A. Vasilevskiy and K. Siddiqi. Flux maximizing geometric flows. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(12):1565–1578, Dec. 2002.

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

[48] Z. Zheng and R. Zhang. A Fast GVF Snake Algorithm on the GPU. *Research Journal of Applied Sciences, Engineering and Technology*, 4(24):5565–5571, 2012.

[49] G. Ziegler, A. Tevs, C. Theobalt, and H. Seidel. On-the-fly point clouds through histogram pyramids. In *Vision, modeling, and visualization 2006: proceedings, November 22-24, 2006, Aachen, Germany*, page 137. IOS Press, 2006.

D

A new tube detection filter for abdominal aortic aneurysms

Authors

Erik Smistad, Reidar Brekken and Frank Lindseth

Published in

Proceedings of MICCAI 2014 Workshop on Abdominal Imaging: Computational and Clinical Applications. Lecture Notes in Computer Science, volume 8676, September 2014, pages 229-238.

A New Tube Detection Filter for Abdominal Aortic Aneurysms

Erik Smistad[1,2], Reidar Brekken[2] and Frank Lindseth[2,1]

1) Norwegian University of Science and Technology

2) SINTEF Medical Technology. Trondheim, Norway

Abstract

Tube detection filters (TDFs) are useful for segmentation and centerline extraction of tubular structures such as blood vessels and airways in medical images. Most TDFs assume that the cross-sectional profile of the tubular structure is circular. This assumption is not always correct, for instance in the case of abdominal aortic aneurysms (AAAs). Another problem with several TDFs is that they give a false response at strong edges. In this paper, a new TDF is proposed and compared to other TDFs on synthetic and clinical datasets. The results show that the proposed TDF is able to detect large non-circular tubular structures such as AAAs and avoid false positives.

1 Introduction

Tube detection filters (TDFs) are used to detect tubular structures in 3D images. They perform a shape analysis on each voxel and return a value indicating the likelihood of the voxel belonging to a tubular structure. The likelihood can be used for segmentation and centerline extraction of tubular structures such as abdominal aortic aneurysms from medical images. The segmentation and centerline of these structures are useful for visualization, volume estimation, registration and planning and guidance of vascular interventions.

Many TDFs use second order derivative information to perform the shape analysis like the eigenanalysis of the Hessian matrix. The eigenvalues of this matrix can be used to determine the shape of the local structure and the eigenvectors can be used to find the shape's orientation. To calculate the Hessian matrix at a voxel inside a tubular structure, the gradient information from the edges has to be present. For small tubular structures this is not a problem, but for large ones the gradients have to be propagated from the edges to the center. One way to do this is to compute the Hessian matrix in a Gaussian scale space by convolution with a Gaussian of different standard deviations. The final

TDF measure is calculated as the maximum response over all scales. One problem with using Gaussian scale space is that on larger scales objects diffuse into each other and small tubular structures that are close to one another can diffuse together and give the impression that a larger tubular structure is present. Bauer and Bischof [2] suggested to replace the gradient vector field from the Gaussian scale space with an edge-preserving diffusion process called gradient vector flow (GVF), originally introduced by Xu and Prince [13] as an external force field to guide active contours. With the GVF, only one scale is needed and the problem of objects diffusing into each other is avoided.

Frangi et al. [6] introduced a TDF called a vesselness filter. This filter uses the eigenvalues (λ) of the Hessian matrix to determine whether the current voxel $\vec{x}$ is part of a tubular structure. With the three measures $R_a = |\lambda_2|/|\lambda_3|$, $R_b = |\lambda_1|/\sqrt{|\lambda_2\lambda_3|}$ and $S = \sqrt{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}$ the vesselness filter is defined in (1).

$$T_v(\vec{x}) = \begin{cases} 0 & \text{if} \quad \lambda_2 > 0 \quad \text{or} \quad \lambda_3 > 0 \\ (1 - e^{-\frac{R_a^2}{2\alpha^2}}) e^{-\frac{R_b^2}{2\beta^2}} (1 - e^{-\frac{S^2}{2c^2}}) & \text{else} \end{cases} \tag{1}$$

Frangi et al. used Gaussian scale space methods to do the multi-scale filtering, however Bauer et al. [2, 3, 4] later used the vesselness TDF successfully with the GVF.

The circle fitting TDF introduced by Krissian et al. [8] uses the eigenvectors of the Hessian matrix to identify the tubes cross-sectional plane. In this plane a circle is fitted to the underlying edge information. The fitting procedure samples N points on a circle with radius r and calculates the average dot product (2) of the edge direction ($\vec{V}$) and the circle's inward normal ($-\vec{d}_i$). The radius is gradually increased and the radius with the highest average is selected. The TDF response is then equal to the average with the select radius r as in (2).

$$T_{cf}(\vec{x}) = \frac{1}{N} \sum_{i=0}^{N-1} \vec{V}(\vec{x} + r\vec{d}_i) \cdot (-\vec{d}_i) \tag{2}$$

As a measure of edge direction, Krissian et al. [8] used the gradient calculated at the scale corresponding to the current radius. Bauer et al. [5] used the GVF field instead. Since this TDF assumes that the cross-sectional profile of the tubular structure is circular, it produces a lower response for non-circular tubular structures. Also, the cross-section of a tube is estimated using the eigenvectors of the Hessian matrix which are not accurate, hence even if the tubes are circular the cross-section may often appear as ellipses instead. Furthermore, the circle fitting TDF can give response in voxels where there is not a tubular structure. A semi-circle with a very high contrast can be enough to give a medium response. Pock et al. [9] proposed a symmetry measure to reduce the false response at such edges. This measure reduces the TDF response where the gradient's magnitude, i.e. the contrast, differs along the circle. However, this also reduces the response for tubular structures with a non-circular cross-section. Bauer [1] concluded in his thesis that several TDFs, including the vesselness and circle fitting TDF, have the problem that the response decreases significantly when the cross-section of the tubular structure deviates

from a circle, which makes these tubular structures hard to distinguish from noise in the TDF response.

In this paper, a new TDF is proposed that uses GVF and is able to properly detect non-circular irregular tubular structures and reduce the amount of false responses. In addition, it is demonstrated that a multigrid method is necessary for calculating the GVF for large tubular structures such as abdominal aortic aneurysms (AAAs).

2 Methods

Previously, we have developed a framework for extracting airways and blood vessels from different image modalities (e.g. CT, MR and US) using tube detection filters [10]. The framework consists of five main steps that are all executed on the graphic processing unit (GPU) (see Fig. 1) . The first step is to crop the volume in order to reduce the total memory usage. The second step involves some pre-processing, such as Gaussian smoothing and gradient vector flow, which are necessary to make the results less sensitive to noise and differences in tube contrast and size. After pre-processing, the TDF is performed. From the TDF result, the centerlines are extracted and finally, a segmentation is performed with a region growing procedure using the centerlines as seeds. The entire implementation is available online[1]. Previously, the circle fitting TDF by Krissian et al. [8] was used in this framework. In this paper, a new TDF is proposed as a replacement for this filter to improve detection of large non-circular tubular structures and avoid detection of false tubular structures.

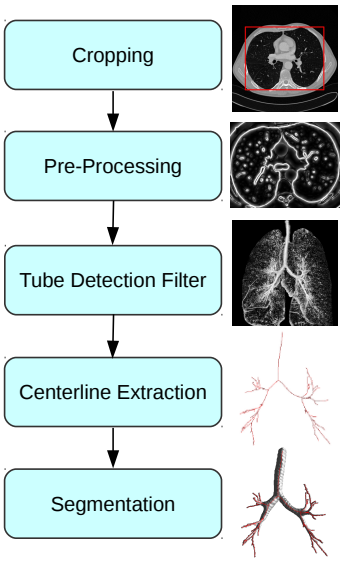

Figure 1: Block diagram of the implementation

2.1 Large Tubular Structures and Gradient Vector Flow

The most common way to calculate GVF is to use Euler's method as demonstrated by Xu and Prince [13]. However, this method is very slow to converge [7]. And for large tubular structures where the gradients at the edges have to diffuse a long way to the center, this becomes a problem (see Fig. 3). To solve this problem, Han et al. [7] used multigrid methods to calculate GVF and achieved a much better convergence rate. In this paper, a GPU implementation of this multigrid method was used [11].

[1] http://github.com/smistad/Tube-Segmentation-Framework/

2.2 A New TDF for Non-Circular Tubular Structures

Like the circle fitting TDF, the proposed TDF uses the eigenvectors of the Hessian matrix to identify the orientation of the tubular structures. The two eigenvectors associated with the eigenvalues of the largest magnitude $\vec{e}_2$ and $\vec{e}_3$ span the cross-sectional plane of the tubular structure. In this plane, N line searches are performed from the current voxel $\vec{x}$ at different angles. For each line search i, a phasor is used to create vectors $\vec{d}_i$ that define the search direction θ.

$$\theta_i = \frac{2\pi i}{N} \qquad \vec{d}_i = \vec{e}_2 \sin\theta_i + \vec{e}_3 \cos\theta_i \tag{3}$$

Each line search continues until the edge of the tubular structure is encountered and the distance from the center to the edge for line search i is r_i. The edges are detected as the first peak in the vector field's magnitude above the fixed threshold 0.01. This threshold states the minimum gradient magnitude of an edge of a tubular structure. Thus, the value of 0.01 will allow most edges, but it is necessary to eliminate noise. If a dataset has noise with a higher contrast, this threshold may be increased. The problem of detecting false tubular structures is reduced by limiting the length of the line searches with a parameter r_{max}. However, when detecting very large tubular structures, such as AAAs, r_{max} has to be set high and thus might not reduce the number of false positives. Also, if only large tubular structures are to be detected, a parameter, r_{min}, can be set which sets the lower bound for the radius of the tubular structures to be detected. Using these distances, a measure $C(\vec{x})$ is created of how likely it is that the voxel $\vec{x}$ is in the center of the tubular structure (4). This measure enables the proposed TDF to be used for extracting centerlines and was also used by Wink et al. [12].

$$C(\vec{r}) = \frac{2}{N} \sum_{i=0}^{N/2-1} \frac{\min(r_i, r_{N/2+i})}{\max(r_i, r_{N/2+i})} \tag{4}$$

Finally, the TDF measure T is defined as the product of the center likelihood measure C and a measure M of how well the gradient vectors at the border correspond to the direction of the tubular structure $\vec{e}_1$.

$$M(\vec{x}) = \frac{1}{N} \sum_{i=0}^{N-1} \left(1 - |\vec{V}^n(\vec{x} + r_i\vec{d}_i) \cdot \vec{e}_1|\right) \tag{5}$$

$$T(\vec{x}) = \begin{cases} 0 & \text{if} \quad \exists i \vec{V}^n(\vec{x} + r_i\vec{d}_i) \cdot (-\vec{d}_i) < 0 \\ C(\vec{r})M(\vec{x}) & \text{else} \end{cases} \tag{6}$$

Ideally, the gradient vectors $\vec{V}$ should be perpendicular to the direction of the tubular structure. This can be checked by taking the dot product of the normalized vectors $\vec{V}^n$ and $\vec{e}_1$. The closer the dot product is to zero, the closer the two vectors are to being perpendicular. At the borders of large tubular structures, the data will, locally, resemble more a plate structure than a tubular structure which may lead to an incorrect tube direction $\vec{e}_1$.

The measure M thus reduces the response in the borders of the tubular structure where the tube direction $\vec{e}_1$ may be incorrect. This greatly improves the centerline extraction which uses the tube direction $\vec{e}_1$ [10]. Also, if there exist a vector that is more than $90°$ from the direction to the center $-\vec{d}_i$, the TDF measure is set to 0. This is done to further reduce the amount of false responses in which the edge gradient has another direction than towards the center and is similar to the circularity measure used by Pock et al. [9].

3 Results

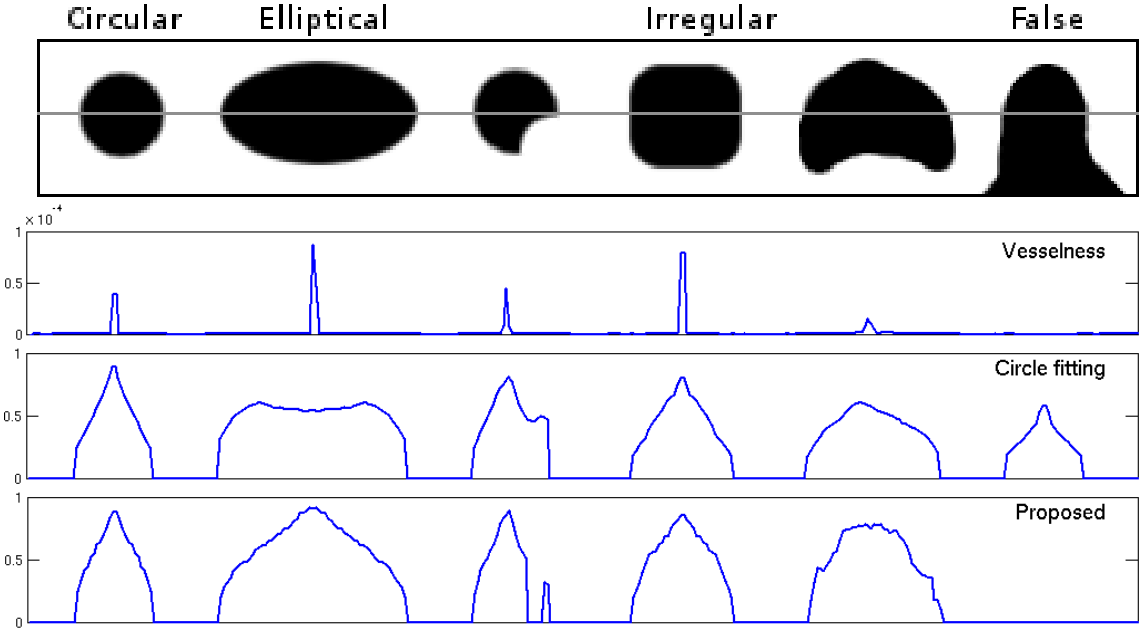

Figure 2: The top row shows the cross-section of five different tubular structures and one false tubular structure. The three graphs below are the responses from the vesselness, circle fitting and proposed TDFs respectively, measured in a line that goes through the middle of all the cross-sections (the grey line in the top row).

In this section, results of the proposed TDF are presented for both synthetic and clinical data and compared to the vesselness and circle fitting TDF in conjunction with GVF. The parameters used for the GVF are $\mu = 0.1$ with 6 iterations. The vesselness TDF was run with the parameters $\alpha = 0.5$, $\beta = 0.5$ and $c = 100$. And the circle fitting TDF used 32 sample points and the proposed TDF used $N = 12$ line searches.

Synthetic Data: A dataset containing tubular structures with different types of cross-sectional profiles was created. The profiles are displayed in the top of Fig. 2. This dataset contains tubular structures with circular, elliptical, several irregular profiles and one false tubular structure. The vesselness, circle fitting and proposed TDF were performed on this dataset. The responses along a line going through the middle of all of these tubular structures were recorded and are displayed as graphs in Fig. 2. The figure shows that

the response of the circle fitting TDF is considerably reduced when performed on tubular structures with a non-circular cross-section, while the proposed TDF detects these almost as well as the circular structure. The circle fitting TDF also has a high response at the false tubular structure to the far right.

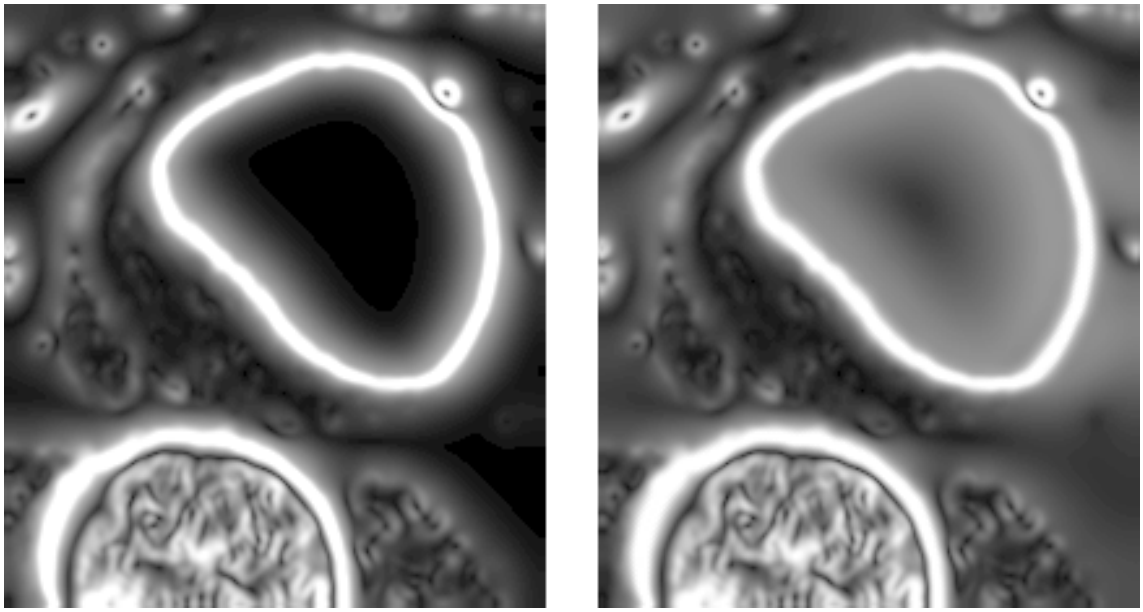

Figure 3: Magnitude of the vector field after running gradient vector flow (GVF) on a AAA CT dataset. **Left:** Euler's method with 1000 iterations. **Right:** Multigrid method with 6 iterations. The image to the left shows that GVF with Euler's method has problems with diffusing the gradients on the edge of the aneurysm to the center which is necessary for the TDFs.

Clinical Data: The TDFs were also executed on clinical CT datasets of abdominal aortic aneurysms (AAAs). Figure 3 illustrates the need for the multigrid method when calculating the GVF on large tubular structures such as an AAA. The figure shows the magnitude of the vector field after running GVF using Euler's method with 1000 iterations (about 6 seconds) and the multigrid method with 6 iterations (about 1 second). From this figure, it is evident that GVF with Euler's method has problems with diffusing the gradients on the edge of the aneurysm to the center, which is necessary for the TDFs. Over 10 times more iterations would be needed to reach the center with Euler's method which would reduce performance considerably. However, with the multigrid method the gradients are diffused to the center in about 1 second.

Figure 4 shows a maximum intensity projection of the response for each TDF on a CT image of an AAA. The TDFs were all executed on the same GVF vector field thus requiring only one scale. The same window and level were used on the circle fitting and the proposed TDF as both of these TDF have responses from 0 to 1. Also, the minimum radius (r_{min}) and maximum radius (r_{max}) used were 7 and 45 mm. This enables visual comparison of the two TDFs and it is clear that the circle fitting TDF creates a weaker response in the aneurysm than the proposed TDF. Furthermore, the amount of noise, especially from the spine, is higher with the circle fitting TDF. A different level and window

were used for the vesselness TDF as its range is exponential. However, the AAA was not detected with this filter.

Figures 5 and 6 depicts the results using three different algorithms on four different AAA CT images. For comparison, the first column in the figures shows the segmentation result using the seeded region growing segmentation method. However, as this method leads to segmentation leakage into the spine on all datasets, the centerline was not possible to extract. The middle and right column show the segmentation surface and centerlines obtained with the circle fitting and the proposed TDF using the framework from [10] and the multigrid GVF method [11]. Here, r_{min} and r_{max} were set to 2 and 45 mm respectively. The vesselness TDF was not able to detect the AAAs and was therefore not included. The datasets consisted of 388-420 slices with size 512x512. The runtime of the entire implementation (see Fig. 1) including the TDF, centerline extraction and segmentation for these datasets was 4-10 seconds using a modern AMD Radeon HD7970 GPU.

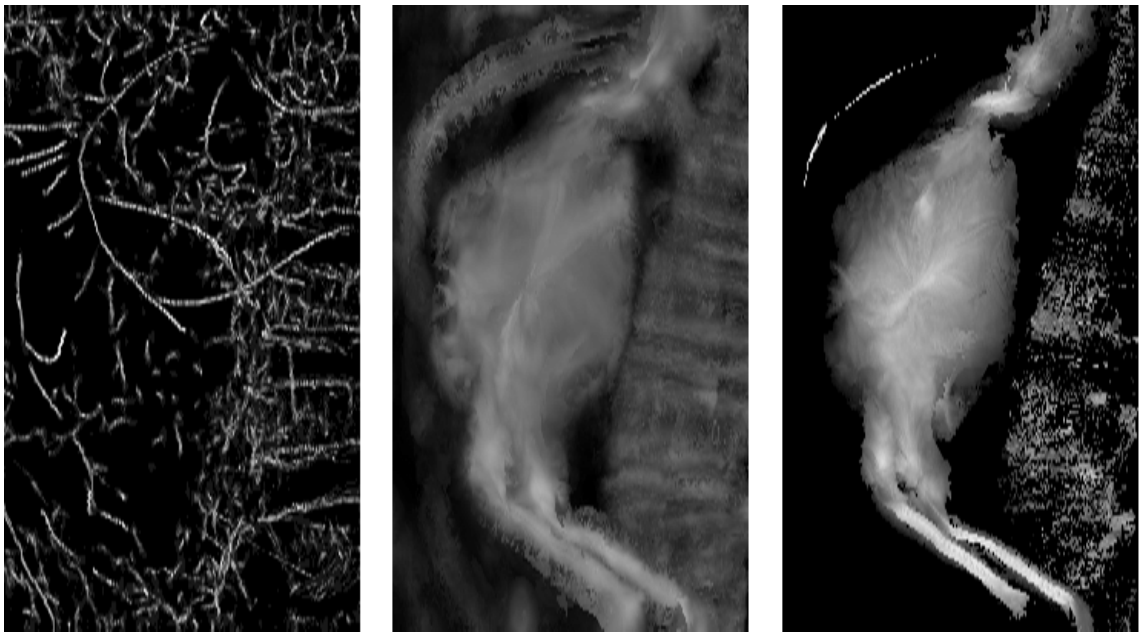

Figure 4: Maximum intensity projection of TDF responses on a CT image of an abdominal aortic aneurysm (AAA) using the same GVF vector field. **Left:** Vesselness TDF. **Middle:** Circle fitting TDF. **Right:** Proposed TDF. The same level and window were used on the circle fitting and proposed TDF. A different level and window were used for the vesselness TDF as its range is exponential.

4 Discussion

The results shows that the proposed TDF is able to properly detect large non-circular tubular structures such as AAAs in CT images. Figures 5 and 6 show that seeded region growing fails to segment the AAAs due to leakage to the spine and the circle fitting

Figure 5: Left: Region growing. **Middle:** Circle fitting TDF. **Right:** Proposed TDF.

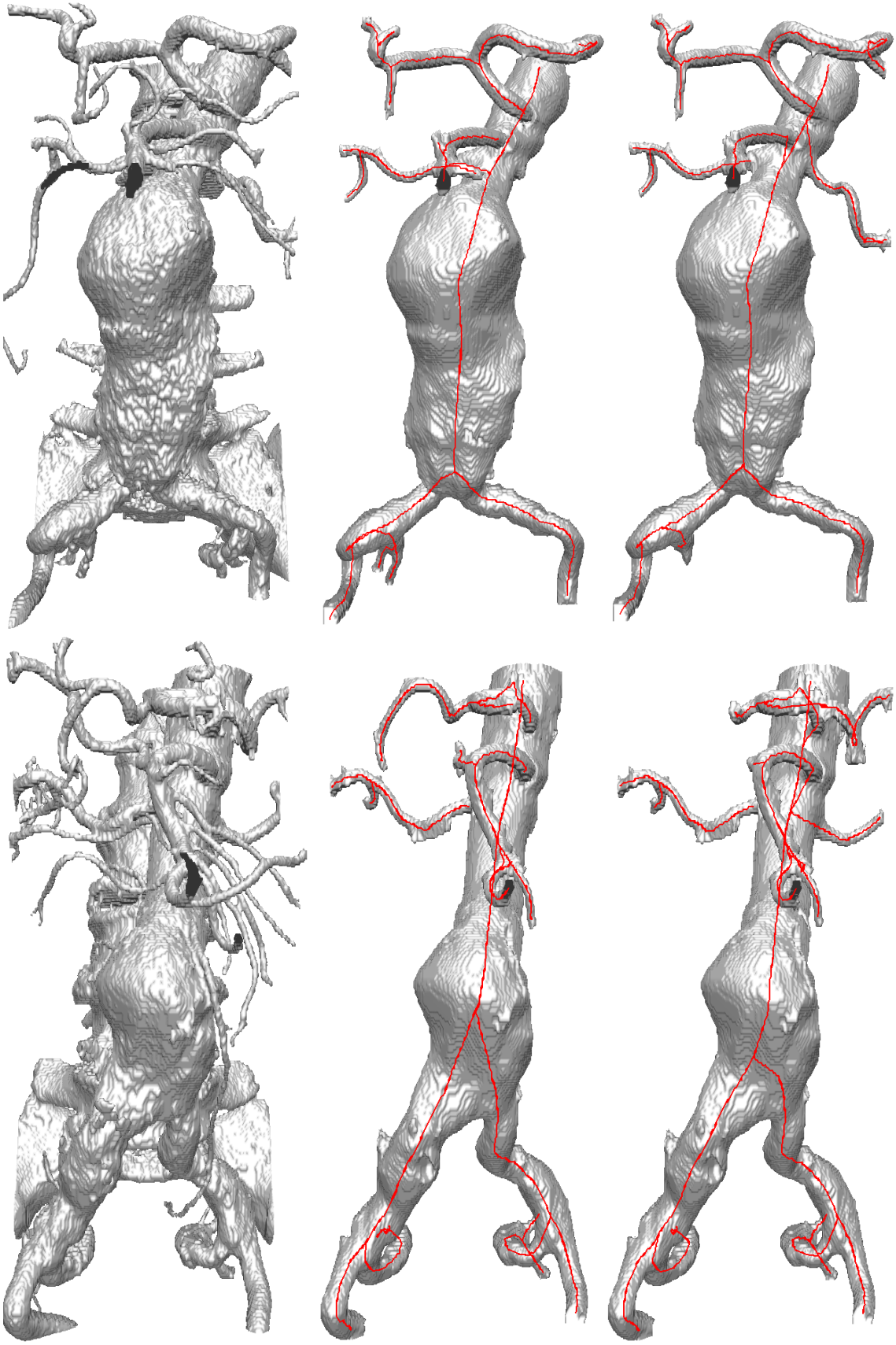

Figure 6: Left: Region growing. **Middle:** Circle fitting TDF. **Right:** Proposed TDF.

TDF is not able to properly detect some of the AAAs that deviate most from a circular cross-sectional profile.

The response of the vesselness and circle fitting TDF is dependent on the contrast due to the use of eigenvalues (Eq. 1) and gradient (Eq. 2). However, the response of the proposed TDF is invariant to the contrast due to the use of the normalized gradient $\vec{V}^n$ in (5). Nevertheless, Bauer and Bischof [4] proposed a solution to this by adding a parameter $F_{\max}$ for the maximum contrast. Any gradient with a magnitude above this parameter would be normalized and any below, divided by this parameter. But this has also the effect of amplifying the effect of noise. The proposed TDF eliminates the need for this parameter.

5 Conclusions

A new tube detection filter using gradient vector flow was proposed and compared with two other commonly used filters. It was shown that the proposed filter is able to properly detect non-circular tubular structures such as abdominal aortic aneurysms and thus enable segmentation and centerline extraction of these structures.

References

[1] Bauer, C.: Segmentation of 3D Tubular Tree Structures in Medical Images. Ph.D. thesis, Graz University of Technology (2010)

[2] Bauer, C., Bischof, H.: A novel approach for detection of tubular objects and its application to medical image analysis. In: Proceedings of the 30th DAGM Symposium on Pattern Recognition. pp. 163–172. Springer (2008)

[3] Bauer, C., Bischof, H.: Edge based tube detection for coronary artery centerline extraction. The Insight Journal (2008)

[4] Bauer, C., Bischof, H.: Extracting curve skeletons from gray value images for virtual endoscopy. In: Proceedings of the 4th International Workshop on Medical Imaging and Augmented Reality. pp. 393–402. Springer (2008)

[5] Bauer, C., Bischof, H., Beichel, R.: Segmentation of airways based on gradient vector flow. In: Proceedings of the 2nd International Workshop on Pulmonary Image Analysis. MICCAI. pp. 191–201. Citeseer (2009)

[6] Frangi, A., Niessen, W., Vincken, K., Viergever, M.: Multiscale vessel enhancement filtering. Medical Image Computing and Computer-Assisted Interventation 1496, 130–137 (1998)

[7] Han, X., Xu, C., Prince, J.: Fast numerical scheme for gradient vector flow computation using a multigrid method. Image Processing, IET (1), 48–55 (2007)

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

[9] Pock, T., Beichel, R., Bischof, H.: A novel robust tube detection filter for 3D centerline extraction. Image Analysis pp. 481–490 (2005)

[10] Smistad, E., Elster, A.C., Lindseth, F.: GPU accelerated segmentation and centerline extraction of tubular structures from medical images. International Journal of Computer Assisted Radiology and Surgery 9(4), 561–575 (2014)

[11] Smistad, E., Lindseth, F.: Multigrid gradient vector flow computation on the GPU (2014), manuscript submitted for publication.

[12] Wink, O., Niessen, W.J., Viergever, M.a.: Fast delineation and visualization of vessels in 3-D angiographic images. IEEE transactions on medical imaging 19(4), 337–46 (Apr 2000)

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

E

Multigrid gradient vector flow computation on the GPU

Authors

Erik Smistad and Frank Lindseth

Published in

Journal of Real-Time Image Processing, 2014.

Multigrid gradient vector flow computation on the GPU

Erik Smistad[1,2] and Frank Lindseth[2,1]

1) Norwegian University of Science and Technology

2) SINTEF Medical Technology. Trondheim, Norway

Abstract

Gradient vector flow (GVF) is a feature-preserving spatial diffusion of image gradients. It was introduced to overcome the limited capture range in traditional active contour segmentation. However, the original iterative solver for GVF, using Euler's method, converges very slowly. Thus many iterations are needed to achieve the desired capture range. Several groups have investigated the use of graphic processing units (GPUs) to accelerate the GVF computation. Still, this does not reduce the number of iterations needed. Multigrid methods, on the other hand, have been shown to provide a much better capture range using considerable less iterations. However, non-GPU implementations of the multigrid method are not as fast as the Euler method when executed on the GPU. In this paper, a novel GPU implementation of a multigrid solver for GVF written in OpenCL is presented. The results show that this implementation converges and provides a better capture range about 2-5 times faster than the conventional iterative GVF solver on the GPU.

1 Introduction

Gradient vector flow (GVF) is a feature-preserving spatial diffusion of image gradients. The GVF field is defined as the vector field $\vec{V}$, that minimizes the energy function E:

$$E(\vec{V}) = \int \mu |\nabla \vec{V}(\vec{x})|^2 + |\vec{V}(\vec{x}) - \vec{V}_0(\vec{x})|^2 |\vec{V}_0(\vec{x})|^2 d\vec{x} \quad (1)$$

where $\vec{V}_0$ is the initial vector field. The first part of this integrand $|\nabla \vec{V}(\vec{x})|$, is the diffusion part that favors a vector field that is smooth. The second part $|\vec{V}(\vec{x}) - \vec{V}_0(\vec{x})|$, on the other hand, is the feature-preserving part that pushes the vector field to be similar to the initial vector field. The last part $|\vec{V}_0(\vec{x})|$ reduces the feature preservation for weak edges so that they are smoothed out instead. The parameter μ governs how much the vector field should be smoothed. Thus μ should be increased if there is a lot of noise. Also, note that the gradient operator ∇ is applied separately for each vector component.

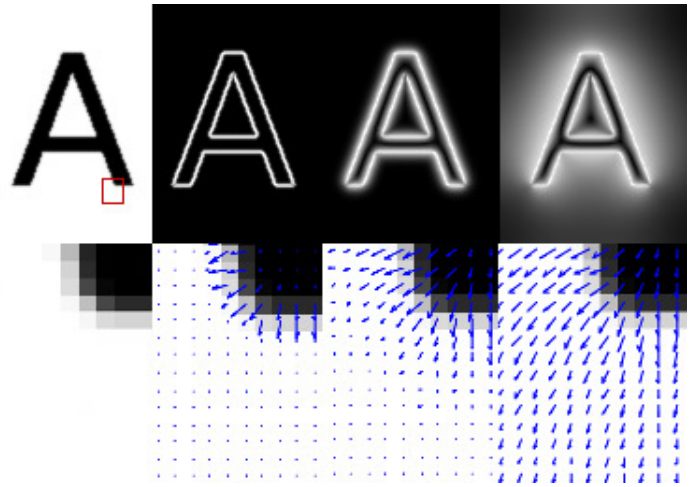

Figure 1: Example of GVF execution using Euler's method. The image to the left is the input image and the three next images show the GVF vector field after 0, 10 and 400 iterations. The top row shows the magnitude of the vector field and the bottom row shows the vectors superimposed on a zoomed area of the input image.

Figure 1 depicts the process of the GVF algorithm. The image to the left is the input image. Next, is the initial vector field $\vec{V}_0$ and the next two images show the vector field $\vec{V}$ after 10 and 400 iterations. The top row shows the magnitude of the vectors fields while the bottom row shows the vectors superimposed on a zoomed area of the input image. The initial image shown top-left is an image smoothed by convolution with a Gaussian.

The GVF algorithm was introduced by Xu and Prince [19] as a new external force field for active contours (AC). Also known as snakes or deformable models, AC are curves that move in an image while trying to minimize their energy and are used extensively for boundary detection and segmentation. The original snake, introduced by Kass et al. [12], has the problem of getting stuck in boundary concavities and low capture range. The capture range is how far from the object's border a snake can be initialized and still converge to the border. The GVF method is able to overcome both these problems.

After its introduction, the GVF algorithm has been applied for several other image processing applications. Bauer and Bischof [3] developed a novel approach to use the GVF as a replacement for the scale-space framework in Hessian based tube detection. Hassouna and Farag [10] and Bauer and Bischof [4] used the GVF to extract skeletons from objects. Ray and Acton [14] used GVF to track leukocytes from intravital video microscopy. Guo and Lu [8] argued that GVF combined with mutual information can improve multi-modal image registration.

Xu and Prince [19] showed that the GVF field can be found by solving the Euler equation:

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

This can be done by treating the vector field $\vec{V}$ as a function of time and using Euler's

method:

$$\begin{aligned}\vec{V}(\vec{x},t+1) =& \vec{V}(\vec{x},t) + \mu\nabla^2\vec{V}(\vec{x},t) - \\ & (\vec{V}(\vec{x},t) - \vec{V}(\vec{x},0))|\vec{V}(\vec{x},0)|^2\end{aligned} \tag{3}$$

Algorithm 1 shows how this is done numerically.

Algorithm 1 3D Gradient vector flow using Euler's method

Input: Initial vector field $\vec{V}_0$ and the constant μ.
$\vec{V} \leftarrow \vec{V}_0$
for a number of iterations **do**
  for all voxels $\vec{x}$ **do**
    $L \leftarrow -6\vec{V}(\vec{x}) + \vec{V}(x+1,y,z) + \vec{V}(x-1,y,z) + \vec{V}(x,y+1,z) + \vec{V}(x,y-1,z) + \vec{V}(x,y,z+1) + \vec{V}(x,y,z-1)$
    $\vec{V}_n(\vec{x}) \leftarrow \vec{V}(\vec{x}) + \mu L - (\vec{V}(\vec{x}) - \vec{V}_0(\vec{x}))|\vec{V}_0(\vec{x})|^2$
  end for
  $\vec{V} \leftarrow \vec{V}_n$
end for

Calculating the GVF field serially using this numerical approach is slow due to the need for many iterations to converge. However, since each pixel is calculated independently of the other pixels, each pixel can be processed in parallel with the same instructions for each iteration. This data parallelism makes the GVF ideal for execution on graphic processing units (GPUs). The GPU is a type of single instruction, multiple data (SIMD) processor. It can perform the same instruction on each element in a dataset in parallel. GPUs achieve this with many functional units (e.g. ALUs) that share control units.

Because of the simplicity and data parallelism of Euler's method for solving GVF (see Algorithm 1), there exist several GPU implementations of this method. Eidheim et al. [6], He and Kuester [11] and Zheng and Zhang [20] all presented GPU implementations of GVF and active contours for 2D images using shader languages. A GPU implementation of 2D GVF written in CUDA was done by Alvarado et al. [2]. In our previous work [16], we presented a highly optimized GPU implementation of GVF for both 2D and 3D images using OpenCL. This implementation uses both texture memory and a 16-bit storage format to reduce memory latency and has been used for fast segmentation and centerline extraction of tubular structures in medical images [15, 17]

Han et al. [9] proposed an alternative numerical scheme to Euler's method using a multigrid method. Their results showed significant improvement in speed and quality.

There exist several implementations of multigrid methods on the GPU. Some examples are Bolz et al. [5] who implemented a sparse matrix multigrid solver on the GPU and Grossauer and Thomas [7] who implemented a denoising filter and a solver for optical flow on the GPU using multigrid methods. However, to our knowledge, there are no published implementations on multigrid methods for GVF on the GPU.

In this paper, we present a parallel GPU implementation of GVF for 3D images using the numerical multigrid scheme presented by Han et al. [9]. The implementation is available online[1].

The next section describes the multigrid solver for GVF and how it was implemented and optimized for the GPU. The implementation was evaluated on several large medical 3D datasets and execution time and average error are reported in the result section. Finally, a discussion of the results and conclusions are presented.

2 Methods

2.1 Multigrid gradient vector flow

Throughout this article, a *computational grid* refers to the current vector field $\vec{V}$ with a specific resolution. While Euler's method only work on one computational grid with one specific resolution, multigrid (MG) solvers work on several computational grids with different resolutions. Thus MG methods are a type of multiresolution methods. The general idea of MG methods is to accelerate the convergence by solving the same problem only on a coarser computational grid and then use this solution when solving the finer grid. Thus this is a recursive method and for each recursive call there are five steps:

1. Pre-smoothing: Smooth the current grid to remove high frequency errors.
2. Restriction: Create a coarser grid of the current grid.
3. Run this method recursively on the coarser grid from the previous step.
4. Correction: Prolongate/Interpolate the solution of the previous step to the same resolution as the current grid and use it to correct the current solution.
5. Post-smoothing: Smooth the current grid again.

This is called the V-cycle and is depicted in Figure 2. In the next sections, these steps are explained in more detail. Note that for each of these steps there are several choices of methods and parameters. Han et al. [9] investigated which of these choices gave the best convergence rate for GVF. Thus in this study, the same methods and parameters have been used.

2.1.1 Smoothing

The purpose of the pre- and post-smoothing is to reduce high frequency errors. This is done using the red-black Gauss-Seidel (RBGS) relaxation method. The advantage of using the RBGS method versus the default lexiographic Gauss-Seidel method is that RBGS

[1] `http://github.com/smistad/GPU-Multigrid-Gradient-Vector-Flow/`

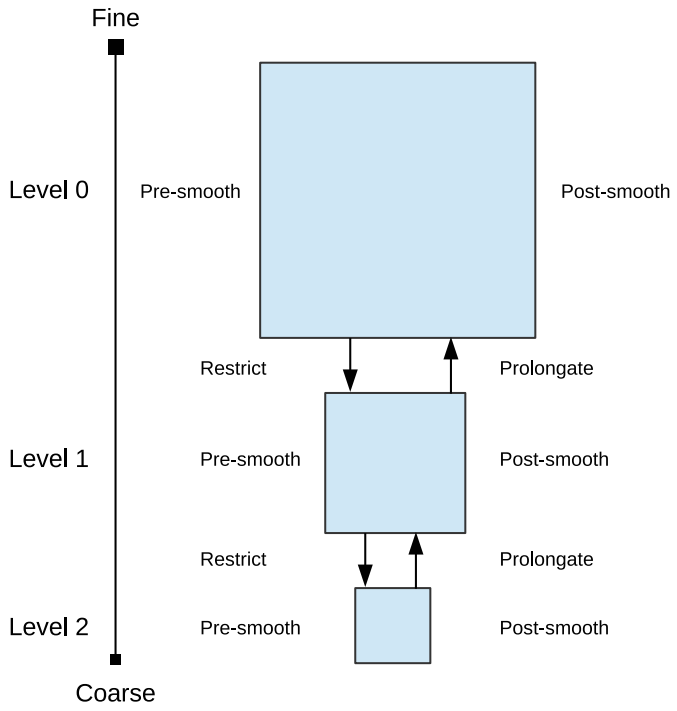

Figure 2: The multigrid V-cycle with 1 levels.

allows half of the voxels in the grid to be computed in parallel. Which is crucial for the GPU implementation. One important parameter in this step, is how many iterations of smoothing will be performed.

In the rest of the article, the following notation will be used. v_l is the current solution for one component in the GVF vector field $\vec{V}$ at resolution level l. r_l is the residual of the current solution v_l at resolution level l. The vector $\vec{x} = [x, y, z]$, is the voxel position. The squared magnitude of the initial vector field $\vec{V}_0$ is constant and simplified to $S_l(\vec{x}) = |\vec{V}_0(\vec{x})|_l^2$. Assuming isotropic spacing h in the computational grid, the update equation for the Gauss-Seidel method is as follows [9]:

$$\begin{aligned} L(x,y,z) &= v_l(x+1,y,z) + v_l(x-1,y,z) + v_l(x,y+1,z) \\ &\quad + v_l(x,y-1,z) + v_l(x,y,z+1) + v_l(x,y,z-1) \\ v_l(x,y,z) &= \frac{2\mu L(x,y,z) - 2h_l^2 r_l(x,y,z)}{12\mu + h_l^2 S_l(\vec{x})} \end{aligned} \tag{4}$$

The update equation is executed on the entire dataset at each level and is done with two kernels as shown in Algorithm 2. To accomplish the checkerboard (red-black) pattern as shown in Figure 3, the Manhatten distance from origo $(x+y+z)$ is first calculated. If the Manhatten distance is even the voxel is red, and if it is odd the voxel is black. The first kernel, GAUSSSEIDELRED, calculates the red voxels using Equation 4. Thus this kernel only works on half the voxels in the dataset. The second kernel, GAUSSSEIDELBLACK, copies the red voxels from the previous kernel and computes the black voxels.

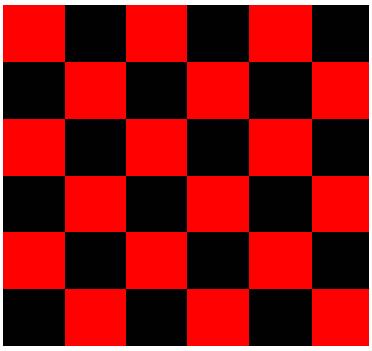

Figure 3: Checkerboard pattern used in the red-black Gauss-Seidel method to process half the voxels in parallel on the GPU.

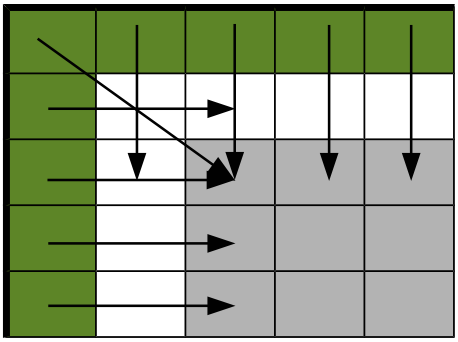

Figure 4: Illustration of boundary conditions in the top left corner of an image. Boundary pixels (green/dark) get the same value as the pixels two steps inside the image (arrows). This will create zero gradients at the white pixels because a central difference scheme is used for the Laplace operator.

A double buffering mechanism is used here with the datasets v_{read} and v_{write}. This is necessary because the data is stored in textures which can only be read or written to in a kernel. More details about the use of textures can be found in the optimization section.

At the boundary of the image, the neighboring voxels needed to calculate L in Equation 4 does not exist. It is desirable to have a zero gradient at the boundary, because a gradient larger than zero at the boundary would diffuse into the rest of the image giving an impression of an edge at the boundary. This can have the effect of forcing the active contours towards the image boundary. There are several ways to implement a zero gradient at the boundary. In this implementation, any voxel that is on the boundary of the image will change its value to the same as the voxel two steps inside the image as shown in Figure 4. For example a voxel with coordinate $x = 0$ uses the value of the voxel with coordinate $x = 2$. Also, a voxel with $x = N - 1$ uses the value of the voxel with coordinate $x = N - 3$. The same applies for the y and z coordinates. The reason for doing it this way is that the calculations are simple.

2.1.2 Restriction

The restriction step downsamples the residual r at level l to a coarser grid (level $l + 1$).

The residual is calculated using the current solution v_l and residual r_l as [9]:

$$r_l(\vec{x}) = r_l(\vec{x}) - \left(\frac{\mu L(\vec{x}) - 6v_l(\vec{x})}{h_l^2} - v_l(\vec{x}) S_l(\vec{x}) \right) \tag{5}$$

The restriction operator used in this implementation takes the average of each 2x2x2 voxel cell and creates a grid which is half the size in each dimension as shown in Equation 6.

Algorithm 2 Parallel red-black Gauss-Seidel

```
function GAUSSSEIDEL(r, v, i, S, h)
    for i times do
        GAUSSSEIDELRED(r, v, v_t, S, h)
        GAUSSSEIDELBLACK(r, v_t, v, S, h)
    end for
    return v
end function

function GAUSSSEIDELRED(r, v_read, v_write, S, h)
    for each voxel (x, y, z) in parallel do
        if x + y + z is even then
            Use Equation 4 to calculate v for voxel x, y, z
        end if
    end for
end function

function GAUSSSEIDELBLACK(r, v_read, v_write, S, h)
    for each voxel (x, y, z) in parallel do
        if x + y + z is even then
            Copy the red voxel from v_read to v_write
        else
            Use Equation 4 to calculate v for voxel x, y, z
        end if
    end for
end function
```

The same operator is used when creating the different levels of the squared magnitude of the initial vector field S_l.

$$\begin{aligned} r_{l+1}(x,y,z) = \frac{1}{8}(&r_l(2x,2y,2z) + r_l(2x+1,2y,2z) \\ &+ r_l(2x,2y+1,2z) + r_l(2x,2y,2z+1) \\ &+ r_l(2x+1,2y+1,2z) + r_l(2x,2y+1,2z+1) \\ &+ r_l(2x+1,2y,2z+1) + r_l(2x+1,2y+1,2z+1)) \end{aligned} \tag{6}$$

If the grid size is 256x256x256 for level l, the next level will have size 128x128x128. Usually, the size of the finest grid (level $l = 0$), the actual image size, is not equal in every dimension or a power of two for that matter. By taking the largest dimension and rounding up towards the closest number that is a power of 2 (see Equation 7), the size of the next level can be determined.

$$A = 2^{\lceil log2(max(M,N,O)) \rceil - 1} \tag{7}$$

Then the size of level 1 would be AxAxA. Thus for an input vector field of size 460x390x120 (level 0), level 1 would have a size of 256x256x256, and level 2 would have a size of

128x128x128. This method leads to some waste of space and processing, but gives a much simpler implementation. The spacing of each level is calculated as $h_{l+1} = 2h_l$. The multigrid method will process grids from level 0 to the coarsest level with the smallest possible size, 2x2x2.

2.1.3 Prolongation

Prolongation is the opposite of restriction. Prolongation resamples and increases the size of the grid. It is used when correcting the current solution with a solution of a coarser grid such that CORRECT$(v_l, v_{l+1}) \leftarrow v_l +$
PROLONGATE(v_{l+1}). Bi- or trilinear interpolation may be used as a prolongation operator, but according to Han et al. [9] a simple nearest voxel method (Equation 8) give a better convergence rate.

$$v_l(x, y, z) = v_{l+1}\left(\left\lfloor \frac{x}{2} \right\rfloor, \left\lfloor \frac{y}{2} \right\rfloor, \left\lfloor \frac{z}{2} \right\rfloor\right) \tag{8}$$

2.1.4 The V-cycle

Putting all of this together we end up with Algorithm 3. This algorithm needs 6 separate GPU kernels: GAUSSSEIDELRED, GAUSSSEIDELBLACK, RESIDUAL, RESTRICT, CORRECT and INITIALIZETOZERO which simply initializes a solution to zero.

The two constants b and c determines how many times pre- and post-smoothing will be performed.

Algorithm 3 The V-cycle

function VCYCLE(r_l, v_l, l, S_l, h_l, b, c)
  $v_l \leftarrow$ GAUSSSEIDEL(r_l, v_l, b, S_l, h_l)
  if l is NOT the coarsest grid level **then**
    % Calculate residual of current solution v_l
    $r_l \leftarrow$ RESIDUAL(r_l, v_l)
    $r_{l+1} \leftarrow$ RESTRICT(r_l)
    % Initialize coarse solution to 0
    $v_{l+1} \leftarrow$ INITIALIZETOZERO
    $v_{l+1} \leftarrow$ VCYCLE$(r_{l+1}, v_{l+1}, l+1, S_{l+1}, h_{l+1}, b, c)$
    % Correction of the v_l *using the coarse solution*
    $v_l \leftarrow$ CORRECT(v_l, v_{l+1})
  end if
  $v_l \leftarrow$ GAUSSSEIDEL(r_l, v_l, c, S_l, h_l)
  return v_l
end function

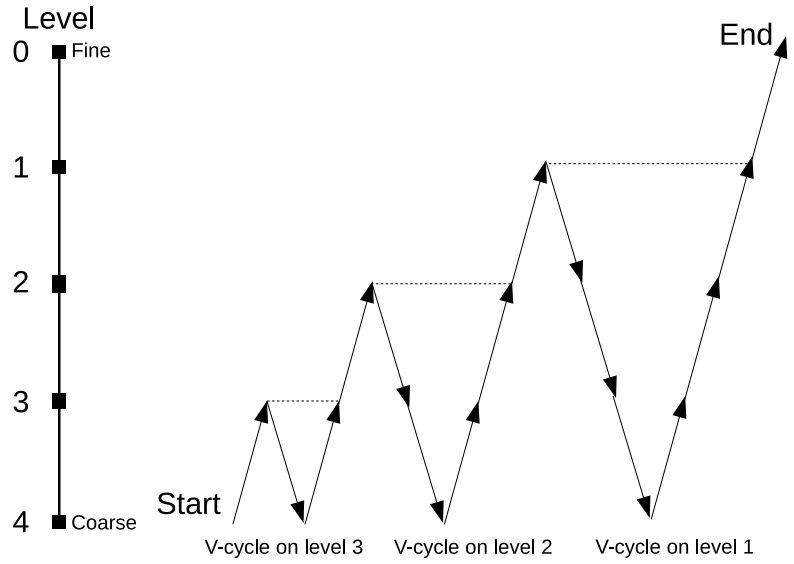

Figure 5: The full multigrid algorithm with 4 levels.

2.1.5 The full multigrid algorithm

One MG scheme is to repeat the V-cycle until convergence. However, faster convergence can be achieved with the full MG algorithm (FMG) [9]. The FMG algorithm is based on the MG V-cycle. However, instead of performing a set of similar V-cycles, the FMG algorithm starts with the coarsest grid and uses the solution for this grid to get a good initialization of the next finer grid (see Figure 5). This is done recursively using the function RECURSIVEFULLMULTIGRID for all computational grids as shown in Algorithm 4. The FMG algorithm can also be repeated until convergence, the constant a determines how many times the FMG algorithm will be repeated. The function FULLMULTIGRID is the entry point of the entire method and takes in the parameters a, b and c and the initial vector field $\vec{V}_0$. The FMG algorithm needs one more additional GPU kernel and that is the PROLONGATION kernel which implements Equation 8. The other kernels, RESTRICT, RESIDUAL and INITIALIZETOZERO, are the same as in the V-cycle algorithm.

2.2 GPU optimization

Accessing the off-chip global memory on a GPU is a very time-consuming operation [1]. Since all of the kernels in this implementation require many memory access operations and few arithmetic operations, the performance of these kernels are memory-bound. Thus, optimization of these kernels should focus on optimizing the memory access. In this section, GPU memory optimization techniques such as using the texture memory system and a 16-bit storage format are described.

Algorithm 4 The full multigrid algorithm

function RECURSIVEFULLMULTIGRID(r_l, l, b, c)
  if l is the coarsest grid **then**
    $v_l \leftarrow$ INITIALIZETOZERO
  else
    $r_{l+1} \leftarrow$ RESTRICT(r_l)
    $v_{l+1} \leftarrow$ RECURSIVEFULLMULTIGRID($r_{l+1}, l+1, b, c$)
    $v_l \leftarrow$ PROLONGATE(v_{l+1})
  end if
  $v_l \leftarrow$ VCYCLE($r_l, v_l, l, S_l, h_l, b, c$)
  return v_l
end function

function FULLMULTIGRID($\vec{V}_0$, a, b, c)
  $\vec{V} \leftarrow$ INITIALIZETOZERO
  for a times **do**
    for each component $C \in [x, y, z]$ **do**
      % *Calculate initial residual*
      $r \leftarrow$ RESIDUAL($-\vec{V}_{0,C}|\vec{V}_{0,C}|^2, \vec{V}_C$)
      $\vec{V}_C \leftarrow$ RECURSIVEFULLMULTIGRID($r, 0, b, c$)
    end for
  end for
  return $\vec{V}$
end function

2.2.1 Texture memory

The GPU has a specialized memory system for images, called the texture system. It has this system because the GPU is primarily made and used for fast rendering which involves mapping images, often called textures, onto 3D objects. The texture system specializes in fetching and caching data from 2D and 3D textures [13, 1]. The fetch unit of the texture system is also able to perform interpolation and data type conversion in hardware. When working with images and volumes, using the texture system to store these structures can greatly improve performance as shown in our previous work on GVF [16]. All of the vector fields, residuals and squared magnitudes at different levels are stored in textures.

2.2.2 16-bit storage format

Memory access can also be improved by reducing the number of bytes transferred from global memory to the chip. Floating point numbers are usually represented using 32 bits and the IEEE 754 standard. However, if the floating point numbers are normalized between 0.0 and 1.0 or -1.0 and 1.0 a different format can be used. Most GPU's texture system supports normalized 8- and 16-bit integers. With this format, the data is stored as 8- or 16-bit integers in the textures. However, when the data is requested, the texture fetch unit converts the integer to a 32-bit floating point number with a normalized range. This reduces accuracy, and may not be sufficient for all applications. In our previous work on GPU-based GVF using Euler's method, the results showed that 8-bit was too inaccurate for any practical use [16]. Also, our previous work on applications such as segmentation and centerline extraction of airways and blood vessels using GVF has shown that 16-bit gave just as good results as 32-bit and increased the speed considerably on large images [17, 18]. The 16-bit storage format also halves the global memory usage, thus allowing much larger volumes to reside completely in the GPU memory.

2.2.3 Work-group size

Threads are executed on the GPU in groups. AMD calls these units of execution *wavefronts* while NVIDIA calls them *warps* [1, 13]. The units are executed atomically and have at the time of writing the size of 64 (AMD) or 32 (NVIDIA) threads. The threads are also grouped in software. In OpenCL these groups are called work-groups, and in CUDA they are called thread-blocks. If the work-group sizes are not a multiple of the wavefront/warp size, some of the GPUs thread processors will be idle for each work-group that is executed. Also, there is a maximum number of threads that can reside in a work-group. On AMD GPUs, this limit is currently 256 and on NVIDIA up to 1024.

When a kernel is scheduled on the GPU using OpenCL, the kernel is executed on a global grid. The grid size has to be dividable by the work-group size. Thus if an image of size 512x512x256 is to be processed with one kernel per voxel, a 3D global execution grid is used with the same size of the image. A possible work-group size is then 4x4x4

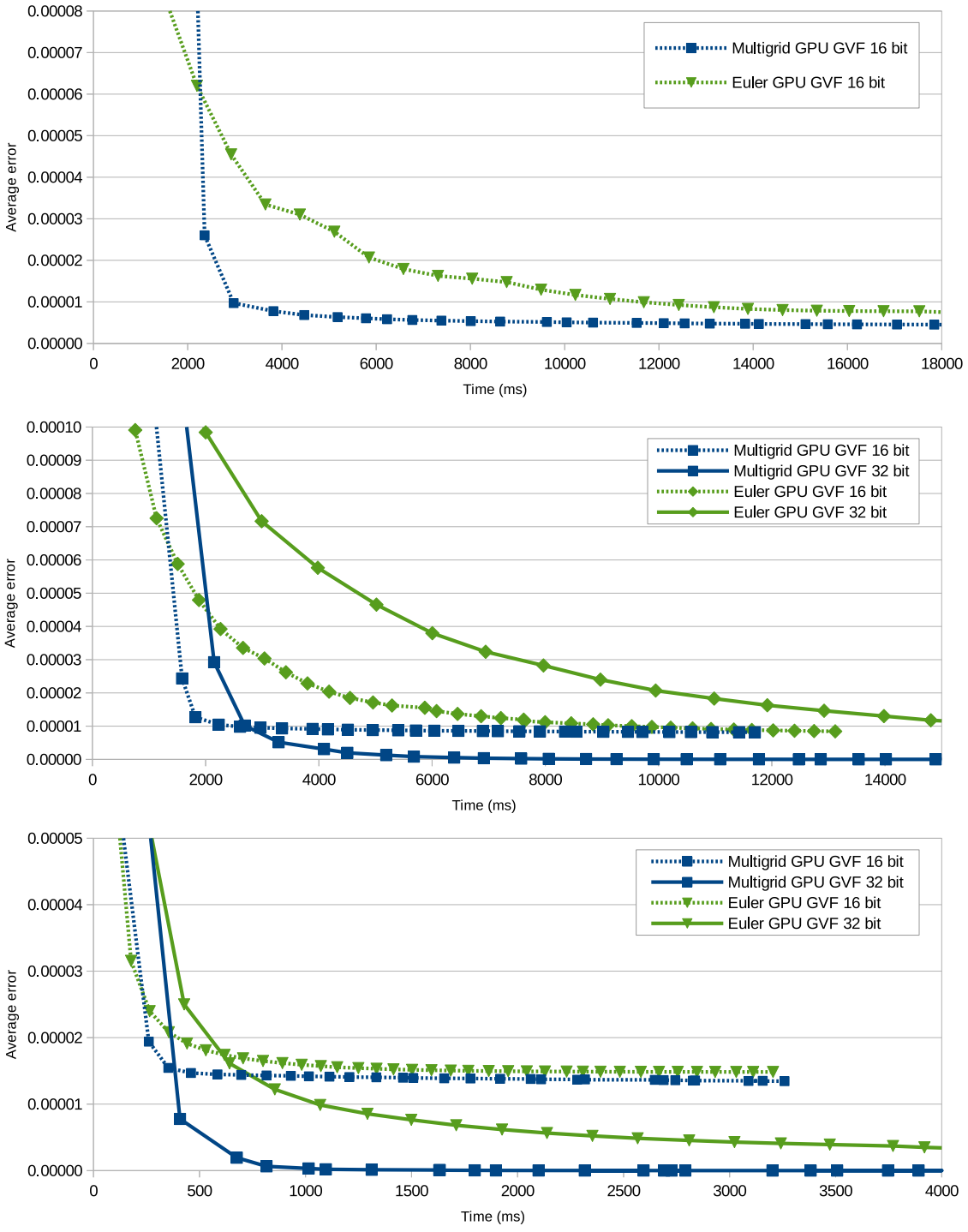

Figure 6: Average error ϵ over time in ms for both the Euler and multigrid GPU implementations with 32-and 16-bit floating point storage formats and datasets of different sizes. **Top:** 512x512x512. **Middle:** 512x512x256. **Bottom:** 256x256x256.

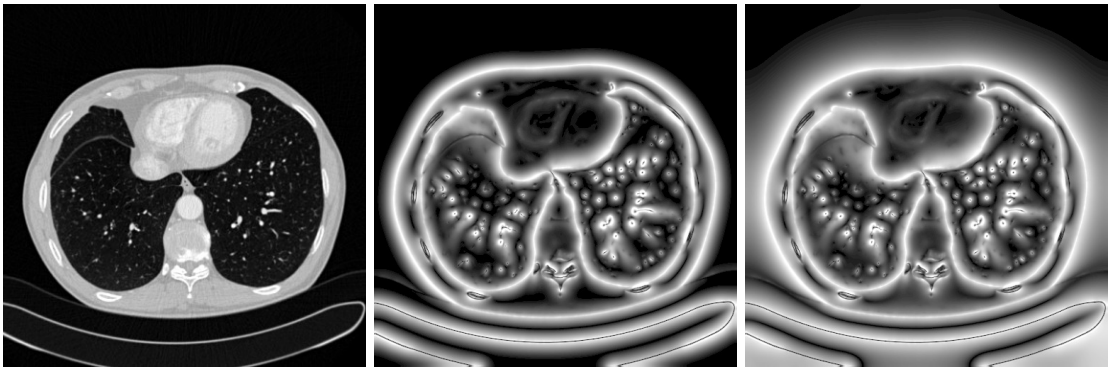

Figure 7: Magnitude of the GVF vector field after the same amount of execution time displayed using the same intensity transformation. **Left:** Input image (Thorax CT). **Middle:** Euler GPU GVF [16] (512 iterations). **Right:** Multigrid GPU GVF (15 iterations). Note the larger capture range with the multigrid method.

because 512 and 256 is dividable by 4, and 4x4x4 = 64 threads which is a multiple of the wavefront/warp sizes and is below the maximum limit.

In this implementation a work-group size of 4x4x4 was used. However, the optimal work-group size can vary from different GPUs. Volumes that have a dimension size that is not dividable by 4 are cropped.

3 Results

The overall goal of the proposed GVF implementation is to achieve a low error as fast as possible. Thus the GVF error and execution time were measured at different number of iterations. This was done on three different datasets using a modern AMD Radeon HD7970 GPU with 3GB memory. The setup was running Ubuntu 12.04, AMD Catalyst 12.11 graphic drivers and AMD APP SDK 2.9. The parameters used in all experiments are $\mu = 0.1$, $a = 1$, $b = 2$ and $c = 1$. Recall that the constant a is the number of times the FMG algorithm is repeated, and b and c are the number of pre- and post-smoothing iterations. These constants were determined through experimentation. The gradient of the input image smoothed with a Gaussian filter with standard deviation $\sigma = 0.5$ was used as the input vector field $\vec{V}_0$ in all experiments.

The graphs in Figure 6 show the average error ϵ versus time for both the Euler [16] and multigrid method on the GPU. These measurements were done on three different datasets with varying sizes using both 16- and 32-bit storage. One large volume of size 512x512x512, one medium volume of size 512x512x256 and one small volume of size 256x256x256. All of the volumes are clinical computed tomography (CT) volumes. The

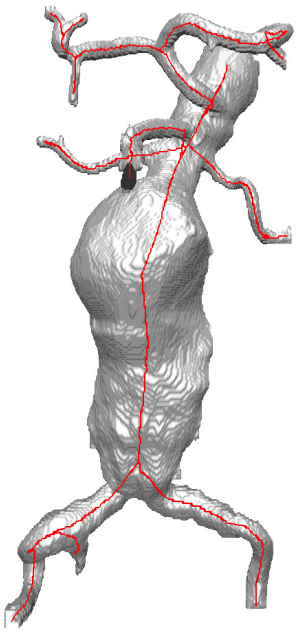

Figure 8: Segmentation and centerline extraction of an abdominal aortic aneurysm using the proposed multigrid GVF implementation [18].

average error ϵ is calculated using Equation 2 over all N voxels:

$$\epsilon = \frac{1}{N} \sum_{\vec{x}} \left| \mu \nabla^2 \vec{V}(\vec{x}) - (\vec{V}(\vec{x}) - \vec{V}_0(\vec{x})) |\vec{V}_0(\vec{x})|^2 \right| \tag{9}$$

From these graphs, it is evident that the MG GPU method converges faster than the Euler GPU method for all three datasets. However, it was not possible to process the largest volume (512x512x512) with either method using 32-bit storage as there was not enough memory on the GPU to do this. Thus only results for 16-bit storage are included in the graph for this volume.

Figure 7 shows the increased capture range with the MG method versus the Euler method [16] when run on a CT thorax image using the same amount of execution time. The figure shows images of the magnitude of the GVF vector field $|\vec{V}|$ using the same intensity transformation for visual comparison.

Table 1 shows the average runtime on different datasets for the two GPU implementations and a serial C++ CPU implementation of Euler's method[2]. The datasets were processed 10 times with Euler's method first using a fixed number of iterations and then the multigrid GPU implementation was executed for as many iterations as needed to reach the same error ϵ or lower as the Euler method. The results show that the multigrid GPU implementation is several times (1.9-5.1) faster than the Euler GPU implementation.

[2] `http://github.com/smistad/Gradient-Vector-Flow/`

Dataset size	**Euler iterations**	**Euler CPU** 32-bit	**Euler GPU** 16-bit / 32-bit	**Multigrid GPU** 16-bit / 32-bit	**Multigrid iterations needed**	**Speedup** 16-bit / 32-bit
512x512x512	512	3085 secs	7.80 / N/A secs	2.17 / N/A secs	4	3.6 / N/A
512x512x256	512	1531 secs	3.88 / 10.16 secs	1.4 / 1.99 secs	4	2.8 / 5.1
256x256x256	256	188 secs	0.46 / 1.10 secs	0.24 / 0.33 secs	3	1.9 / 3.3
256x256x128	256	97 secs	0.23 / 0.54 secs	0.12 / 0.17 secs	2	1.9 / 3.2

Table 1: Average runtime for three different GVF implementations: One serial CPU and one GPU implementation of the Euler method, and the proposed multigrid GPU implementation. The Euler method is run with a specific number of iterations for each dataset, and the multigrid method is run for as many iterations needed to reach the same error ϵ or lower.

4 Discussion

Defining N as the size of the largest dimension of an image, the Euler method need at least N iterations to diffuse gradients to all voxels of the image. Han et al. [9] defined this as a rule of thumb of how many iterations should be used with this method. This is due to the discrete Laplace operator used which only uses neighbor voxels. Thus, the gradients can only diffuse one voxel at a time. Multigrid methods, on the other hand, can diffuse gradients across the image in a single iteration. This is why multigrid methods for GVF achieve a greater capture range and thus lower error faster. In the experiment depicted in Figure 7, both methods were run for the same amount of time and the result is that the multigrid method ends up with a higher capture range. Although multigrid methods need fewer iterations, the multigrid iterations are much more time consuming as they do a lot more work in each iteration.

In our previous work [18], the proposed MG GVF implementation was used in the segmentation and centerline extraction of abdominal aortic aneurysms (AAAs) (see Figure 8). In this work, GVF is used to diffuse the image gradients from the edge of the blood vessels to the center. Because AAAs often involve very large blood vessels, the gradients have to diffuse a long way and thus benefit a lot from the MG GPU implementation. Using the proposed implementation, 6 iterations and 1-2 seconds of processing were sufficient. While over 10,000 iterations and several minutes of processing were needed to achieve the same result with the Euler GPU implementation using the same GPU.

From the graphs in Figure 6 it is clear that the amount of speedup depends on what the target average error is and the size of the dataset. The speedup may also vary a lot for different types of GPUs. These graphs also show that it is possible to get a lower average error with 32-bit than with 16-bit storage format.

Note that the multigrid method processes one component of the vector field at a time. This is less efficient than processing all components at the same time as with the Euler method in Algorithm 1. The reason for processing one component at a time in this multigrid implementation is to reduce memory usage. Thus, if a GPU had more memory, all components could be processed in parallel and the method would probably be even faster. As GPUs get more and more memory every year, this will most likely be possible in the near future.

5 Conclusions

In this paper, a GPU implementation of a multigrid solver for gradient vector flow was presented. The results showed that this multigrid implementation was able to achieve a higher capture range with a lower average error faster than a highly optimized GPU implementation of the traditional Euler's method for calculating the gradient vector flow.

References

[1] Advanced Micro Devices. AMD Accelerated Parallel Processing OpenCL Programming Guide. Technical Report November, 2013. `http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf` - Last accessed 8. August 2013.

[2] Rigo Alvarado, Juan J. Tapia, and Julio C. Rolón. Medical image segmentation with deformable models on graphics processing units. *The Journal of Supercomputing*, 68(1):339–364, December 2013.

[3] Christian Bauer and Horst Bischof. A novel approach for detection of tubular objects and its application to medical image analysis. In *Proceedings of the 30th DAGM Symposium on Pattern Recognition*, pages 163–172. Springer, 2008.

[4] Christian Bauer and Horst Bischof. Extracting curve skeletons from gray value images for virtual endoscopy. In *Proceedings of the 4th International Workshop on Medical Imaging and Augmented Reality*, pages 393–402. Springer, 2008.

[5] Jeff Bolz, I Farmer, E Grinspun, and P Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2003*, 22(3):917–924, 2003.

[6] O.C. Eidheim, J. Skjermo, and L. Aurdal. Real-time analysis of ultrasound images using GPU. *International Congress Series*, 1281:284–289, May 2005.

[7] Harald Grossauer and Peter Thoman. GPU-based multigrid: Real-time performance in high resolution nonlinear image processing. *Computer Vision Systems*, 5008:141–150, 2008.

[8] Yujun Guo and Cheng-chang Lu. Multi-modality Image Registration Using Mutual Information Based on Gradient Vector Flow. In *18th International Conference on Pattern Recognition (ICPR'06)*, pages 697–700. Ieee, 2006.

[9] X Han, C Xu, and J.L. Prince. Fast numerical scheme for gradient vector flow computation using a multigrid method. *Image Processing, IET*, 1(1):48–55, 2007.

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

[11] Zhiyu He and Falko Kuester. GPU-Based Active Contour Segmentation Using Gradient Vector Flow. In *Advances in Visual Computing*, pages 191–201, 2006.

[12] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, January 1988.

[13] NVIDIA. OpenCL Best Practices Guide. Technical report, 2010. `http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf` - Last accessed 8. August 2013.

[14] Nilanjan Ray and Scott T Acton. Motion gradient vector flow: an external force for tracking rolling leukocytes with shape and size constrained active contours. *IEEE transactions on medical imaging*, 23(12):1466–78, December 2004.

[15] Erik Smistad, Anne C. Elster, and Frank Lindseth. GPU-Based Airway Segmentation and Centerline Extraction for Image Guided Bronchoscopy. In *Norsk informatikkonferanse*, pages 129–140. Akademika forlag, 2012.

[16] Erik Smistad, Anne C. Elster, and Frank Lindseth. Real-time gradient vector flow on GPUs using OpenCL. *Journal of Real-Time Image Processing*, pages 1–8, 2012.

[17] Erik Smistad, Anne C. Elster, and Frank Lindseth. GPU accelerated segmentation and centerline extraction of tubular structures from medical images. *International Journal of Computer Assisted Radiology and Surgery*, 9(4):561–575, 2014.

[18] Erik Smistad and Frank Lindseth. A New Tube Detection Filter for Abdominal Aortic Aneurysms. In *Proceedings of MICCAI 2014 Workshop on Abdominal Imaging: Computational and Clinical Applications*, 2014.

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

[20] Zuoyong Zheng and Ruixia Zhang. A Fast GVF Snake Algorithm on the GPU. *Research Journal of Applied Sciences, Engineering and Technology*, 4(24):5565–5571, 2012.

Real-time Tracking of the Left Ventricle in 3D Ultrasound Using Kalman Filter and Mean Value Coordinates

Authors

Erik Smistad and Frank Lindseth

Published in

Proceedings MICCAI Challenge on Echocardiographic Three-Dimensional Ultrasound Segmentation (CETUS), September 2014, pages 65-72. Midas Journal.

Real-time Tracking of the Left Ventricle in 3D Ultrasound Using Kalman Filter and Mean Value Coordinates

Erik Smistad[1,2] and Frank Lindseth[2,1]

1) Norwegian University of Science and Technology

2) SINTEF Medical Technology. Trondheim, Norway

Abstract

A method for real-time automatic tracking of the left ventricle (LV) in 3D ultrasound is presented. A mesh model of the LV is deformed using mean value coordinates enabling large variations. Kalman filtering and edge detection is used to track the mesh in each frame. The method is evaluated using the framework of the Challenge on Endocardial Three-dimensional Ultrasound Segmentation (CETUS). The results show that the method is able to robustly track the LV in all sequences with a mean mesh difference of about 2.5 mm.

1 Introduction

Ultrasound is a real-time, flexible and affordable medical image modality which makes it ideal for intraoperative imaging. However, 3D ultrasound images can be hard to interpret and visualize due to noise and other imaging artifacts. Tracking of structures in ultrasound images can be a way to make this easier and provide quantitative measures such as volume size.

The Kalman filter [1] is a method for estimating a state using a series of noisy measurements over time. This method can be used to track meshes in ultrasound images by using a state consisting of translation, rotation, scaling and deformation parameters of a mesh model in addition to edge detection measurements. Jacob et al. [2] developed a tracking method for myocardial borders using a Kalman filter and active contours in 2D ultrasound. Orderud [3] presented a method for tracking the left ventricle (LV) in 3D ultrasound using a deformable contour model. This was later extended to incorporate local deformation using a B-spline surface in [4] and using a subdivision surface in [5]. B-spline and subdivision surfaces can model mesh deformations using a set of control points.

If the surface is simple, only a few control points are needed which results in faster computation of the Kalman filter. However, creating such a model can involve a lot of manual work. In this paper, we present a fully automatic method for tracking a closed surface with free-form surface deformation based on the Kalman filter approach by Orderud et al. [4]. The presented method uses a mesh model with mean value coordinates which is able to deform a complex shape with few control points. It also makes the shape modelling easier as only a surface consisting of a set of points is required, and some calculations such as generating surface points needed for edge detection are avoided. The new method is tested on 3D ultrasound images of the LV of the heart, and evaluated as part of the Challenge on Endocardial Three-dimensional Ultrasound Segmentation (CETUS).

2 Methods

In this section, the different methods used are described. First, the mesh model for the LV is presented together with the method for deforming it. Next, the Kalman filter used to track the mesh in the images is described and finally, a pseudo-code of the complete implementation is provided.

2.1 Mesh model

The LV is modelled as a set of points $\mathbf{p}$. The initial mesh is defined by the points $\mathbf{p}_0$ and is first transformed using a local transformation $\mathbf{p}_l = T_l(\mathbf{p}_0, \mathbf{x}_l)$ and then a global transformation $\mathbf{p} = T_g(\mathbf{p}_l, \mathbf{x}_g)$. $\mathbf{x}_l$ and $\mathbf{x}_g$ are the local and global transformation parameters. $\mathbf{p}_0$ was created from the end diastolic reference mesh of the first patient and resampled down to $M = 386$ vertices using the surface simplification method of Garland and Heckbert [6]. The reason for reducing the number of vertices is to increase the speed of the implementation.

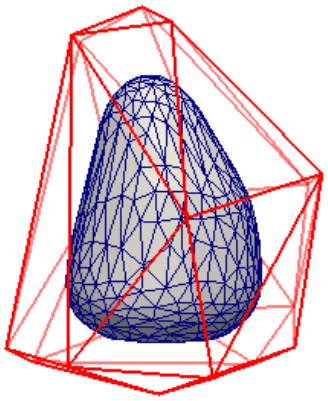

Figure 1: Mesh model of the LV and the control mesh around (red).

Mean value coordinates are used to perform the local deformation of the mesh. This is done using a control mesh $\mathbf{c}$ which has a lot less vertices than the mesh. The control mesh was created from $\mathbf{p}_0$ by resampling it to $N = 18$ vertices using [6] and finally scaling it by 1.5. The mesh is deformed by moving the vertices in the control mesh. Initially, a weight $w_{i,j}$ is calculated between each vertex j in each triangle of the control mesh and vertex i in the mesh $\mathbf{p}_0$. The mean value coordinate weights are calculated as described by Ju et al. [7] using equation (1) where Ψ and θ are the dihedral angles and arc lengths as depicted in Fig. 2.

$$w_{i,a} = \frac{\theta_a - \cos(\Psi_b)\theta_c - \cos(\Psi_c)\theta_b}{2\sin(\Psi_b)\sin(\Psi_c)|\vec{c}_a - \vec{p}_i|} \tag{1}$$

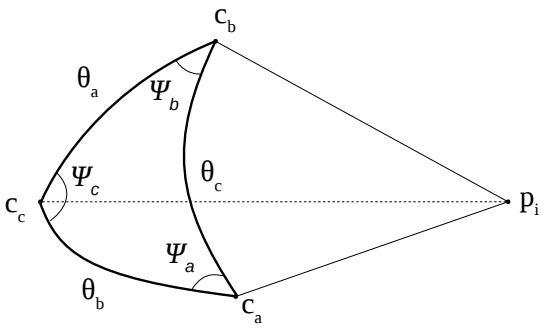

Figure 2: Notation used for the calculation of mean value coordinates. $\vec{p}_i$ is a point on the model mesh. $\vec{c}_a, \vec{c}_b, \vec{c}_c$ are control points for one triangle in the control mesh. Ψ and θ are the angles for the spherical triangle formed by these control points.

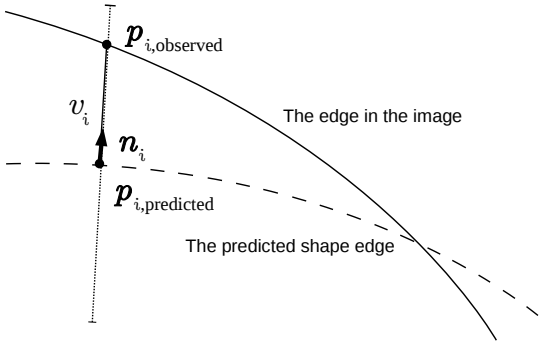

Figure 3: Edge detection for a vertex i on the predicted mesh $\mathbf{p}$. A line is created with the center at the vertex position and in the direction of its normal $\vec{n}_i$. v_i is then the normal displacement between the vertex and the detected edge on this line.

The weights for the other vertices in the triangle ($\vec{c}_b$ and $\vec{c}_c$) are calculated using the same formula by swapping a with b or c. After all the weights have been calculated, they are normalized as $w'_{i,j} = w_{i,j} / \sum_{k,l} w_{k,l}$. The calculation of the normalized weights are only performed once for the mesh model and is not repeated for every dataset. The local state vector $\vec{x}_l$ consist of a displacement vector $\vec{d}_j$ for each of the control mesh' vertices. The local deformation of the mesh is calculated by using the normalized weights and the displacement vectors:

$$\vec{p}_{l,i} = T_l(\vec{p}_0, \vec{x}_l)_i = \sum_{j=0}^{N} w'_{i,j}(\vec{c}_j + \vec{d}_j) \tag{2}$$

The global transformation is performed using equation (3) below. The mesh is first moved so that its centroid $\vec{C}$ is placed in the origin. Next, rotation around each axis is performed as well as scaling. Finally, translation is performed using $\vec{T}$. As translation, rotation and scaling is used in all three directions, the global state vector $\vec{x}_g$ consist of 9 values.

$$\vec{p}_i = T_g(\vec{p}_l, \vec{x}_g)_i = \mathbf{R}_z\mathbf{R}_y\mathbf{R}_x\mathbf{S}(\vec{p}_{l,i} - \vec{C}(\vec{p}_l)) + \vec{C}(\vec{p}_l) + \vec{T} \tag{3}$$

Initially, the mesh $\mathbf{p}_0$ is placed automatically in the center of the image and scaled to 0.8 of its original size, effectively placing the mesh model inside the LV. Although it is slightly counterintuitive to do the local transformation first, this greatly simplifies the calculations. For instance, performing the global transformation first would require calculating the weights $w_{i,j}$ in every iteration.

2.2 Mesh tracking using a Kalman filter

As proposed by Orderud et al. [4], the state of a mesh $\mathbf{p}$ is described using both the local and global transformation parameters $\mathbf{x} = [\vec{x}_l, \vec{x}_g]$. A Kalman filter with a motion model

(4) is used to predict the mesh's state $\bar{\mathbf{x}}_{k+1}$, and the corresponding covariance error matrix $\bar{\mathbf{P}}_{k+1}$ in (5). For the state transition model, diagonal matrices were used with values 1.5 for $\mathbf{A}_1$ and -0.5 for $\mathbf{A}_2$. A diagonal matrix was also used for the process error matrix $\mathbf{Q}$ with values 1.5 for the global and 0.001 for the local transformation parameters.

$$\bar{\mathbf{x}}_{k+1} = \mathbf{A}_1\hat{\mathbf{x}}_k + \mathbf{A}_2\hat{\mathbf{x}}_{k-1} \tag{4}$$

$$\bar{\mathbf{P}}_{k+1} = \mathbf{A}_1\hat{\mathbf{P}}_k\mathbf{A}_1^T + \mathbf{A}_2\hat{\mathbf{P}}_{k-1}\mathbf{A}_2^T + \mathbf{A}_1\hat{\mathbf{P}}_k\mathbf{A}_2^T + \mathbf{A}_2\hat{\mathbf{P}}_{k-1}\mathbf{A}_1^T + \mathbf{Q} \tag{5}$$

The edge detection finds the normal displacement ($v_i = \vec{n}_i^T(\vec{p}_{i,\text{observed}} - \vec{p}_{i,\text{predicted}})$) for each vertex i in the predicted mesh in a 25 mm long line centered at the vertex and in the direction of the normal $\vec{n}$ as shown in Fig. 3. The edge is detected using the STEP model [8] which entails finding a k that maximizes the following measure where $l(t)$ is the image intensity at step t along the line with step size 0.6 mm (see Fig. 4).

$$\sum_{t=0}^{k}\left|\frac{1}{k+1}\sum_{j=0}^{k}(l(j)) - l(t)\right| + \sum_{t=k+1}^{L-1}\left|\frac{1}{L-k}\sum_{j=k+1}^{L-1}(l(j)) - l(t)\right| \tag{6}$$

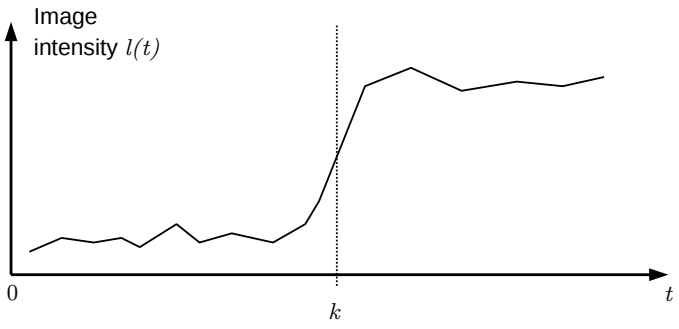

Figure 4: Edge detection using the STEP model.

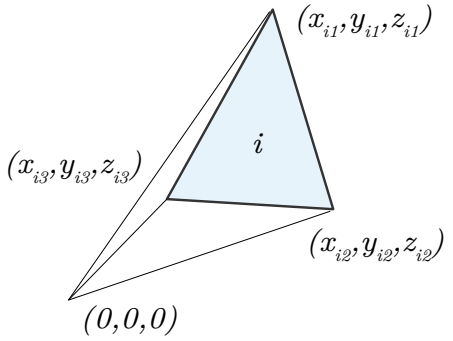

Figure 5: Tetrahedron formed by a triangle i and the origin.

A measurement noise value r_i is also recorded for each vertex and is calculated based on the edge strength:

$$r_i = \frac{1}{\frac{1}{k+1}\sum_{j=0}^{k} l(j) - \frac{1}{L-k}\sum_{j=k+1}^{L-1} l(j)} \tag{7}$$

However, these edge measurements are nonlinear and thus an extended Kalman filter has to be used in which the observation model is linearized. This is done by calculating Jacobi matrices that relate changes in each vertex i to changes in the mesh state $\mathbf{x}$. The final measurement vector $\vec{h}_i^T$ is the normal projection of these Jacobi matrices:

$$\vec{h}_i^T = \vec{n}_i^T\frac{\partial T_g(\mathbf{p}_l, \vec{x})_i}{\partial \vec{x}} = \vec{n}_i^T\left[\frac{\partial T_g(\mathbf{p}_l, \vec{x}_g)_i}{\partial \vec{x}_g}, \frac{\partial T_g(\mathbf{p}_l, \vec{x}_g)_i}{\partial \mathbf{p}_l}\frac{\partial T_l(\mathbf{p}_0, \vec{x}_l)_i}{\partial \vec{x}_l}\right] \tag{8}$$

By assuming that the measurements are independent, the measurement noise covariance matrix $\mathbf{R}$ becomes a diagonal matrix of the measurement noise values r_i. The multiplications of $\mathbf{R}$, the measurement-to-state transition matrix $\mathbf{H}$ and the measurements $\vec{v}$ becomes a simple summation as shown in equation (9) [3]. This makes the Kalman update equations (10) invariant to the number of measurements which improves speed significantly, as matrix inversion of large matrices is avoided.

$$\mathbf{H}^T\mathbf{R}^{-1}\mathbf{v} = \sum_{i=0}^{M} \vec{h}_i^T r_i^{-1} v_i \qquad \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H} = \sum_{i=0}^{M} \vec{h}_i^T r_i^{-1} \vec{h}_i \tag{9}$$

Using $\hat{\mathbf{P}}_k\mathbf{H}^T\mathbf{R}^{-1}$ as the Kalman gain, the updated state and error covariance estimate becomes:

$$\hat{\mathbf{x}}_k = \bar{\mathbf{x}}_k + \hat{\mathbf{P}}_k\mathbf{H}^T\mathbf{R}^{-1}\mathbf{v} \qquad \hat{\mathbf{P}}_k = (\bar{\mathbf{P}}_k^{-1} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})^{-1} \tag{10}$$

2.3 End-systolic and end-diastolic volumes

The volume of the mesh is calculated for every frame using equation (11) [9]. This equation calculates the signed volume of tetrahedrons formed by each triangle in the mesh and the origin as depicted in Fig. 5. The end-systolic (ES) and end-diastolic (ED) phases are identified using the minimum and maximum volumes respectively from the volumes of the meshes in all image frames.

$$V = \left| \sum_i \frac{1}{6}(x_{i2}y_{i3}z_{i1} - x_{i3}y_{i2}z_{i1} + x_{i3}y_{i1}z_{i2} - x_{i1}y_{i3}z_{i2} - x_{i2}y_{i1}z_{i3} + x_{i1}y_{i2}z_{i3}) \right| \tag{11}$$

The pseudo-code below describes the complete implementation which is written in C++. The entire Kalman filter procedure is repeated 10 times per image frame.

3 Results

The method was evaluated as part of the Challenge on Endocardial Three-dimensional Ultrasound Segmentation (CETUS). In this challenge, a dataset of 30 sequences of 3D ultrasound volumes of one cardiac cycle was provided. The sequences were collected from both healthy subjects and subjects with a history of myocardial infarction and dilated cardiomyopathy using three different ultrasound probes. The same parameters were used for all subjects. Results for the training and test dataset are gathered in tables 1 and 2 respectively. The tables contain measures such as mean absolute difference (MAD), hausdorff distance (HD) and min and max error all expressed in millimeters. The correlation of the ejection fraction (EF) and stroke volume (SV) was 0.91 and 0.92 for the training dataset and 0.91 and 0.55 for the test dataset. A Bland-Altman analysis of the EF and SV gave the

Algorithm 1 Implementation

Set initial state x_0 and $x_1 = x_0$
$k \leftarrow 1$
for each image frame **do**
  for a number of iterations **do**
    Predict state and error for the current frame using Eq. (4) and (5).
    Perform transformations to create the predicted shape $\mathbf{p} = T_g(T_l(\mathbf{p}_0, \vec{x}_l), \vec{x}_g)$
    Perform edge detection for each vertex in the mesh $\mathbf{p}$
    Assimilate the measurements using Eq. (8) and (9).
    Estimate the state and error for the current frame using Eq. (10).
    $k \leftarrow k + 1$
  end for
  Calculate volume size V of the current mesh model defined by $\vec{x}_k$ using Eq. (11).
  if $V < V_{\text{min}}$ **then**
    $V_{\text{min}} \leftarrow V$
    $\vec{x}_{ES} \leftarrow \vec{x}_k$
  end if
  if $V > V_{\text{max}}$ **then**
    $V_{\text{max}} \leftarrow V$
    $\vec{x}_{ED} \leftarrow \vec{x}_k$
  end if
end for
Save the meshes defined by $\vec{x}_{ES}$ and $\vec{x}_{ED}$ to disk

95% limits of agreements intervals 1.52±12.37 and 0.34±19.14 for the training dataset and 3.87±8.15 and 0.75±20.31 for the test dataset. Fig. 6 show two ultrasound images where the border of the result mesh of the proposed method and the ground truth is illustrated. The average runtime per subject was measured to be 2.1 seconds with a standard deviation of 0.6 seconds. This includes everything from reading data, processing and storing the result meshes to disk. The average runtime per image frame was measured to be 65 ms which enables real-time tracking of the LV. The runtime was measured on a system with an Intel i7-3770 CPU running at 3.4 GHz, 16 GB RAM and a solid-state drive.

4 Discussion

The results show that the presented method is able to automatically and robustly track the LV in 3D ultrasound with a mean mesh difference at about 2.5 mm. However, the results for the training set is slightly better than for the test set which may indicate that the parameters have been overtuned for the training set. Also, the max error was high (∼10mm) on some sequences. The image to the right in Fig. 6 illustrates such a case. The experts have included bright parts of the heart's apex in the image, while the proposed method

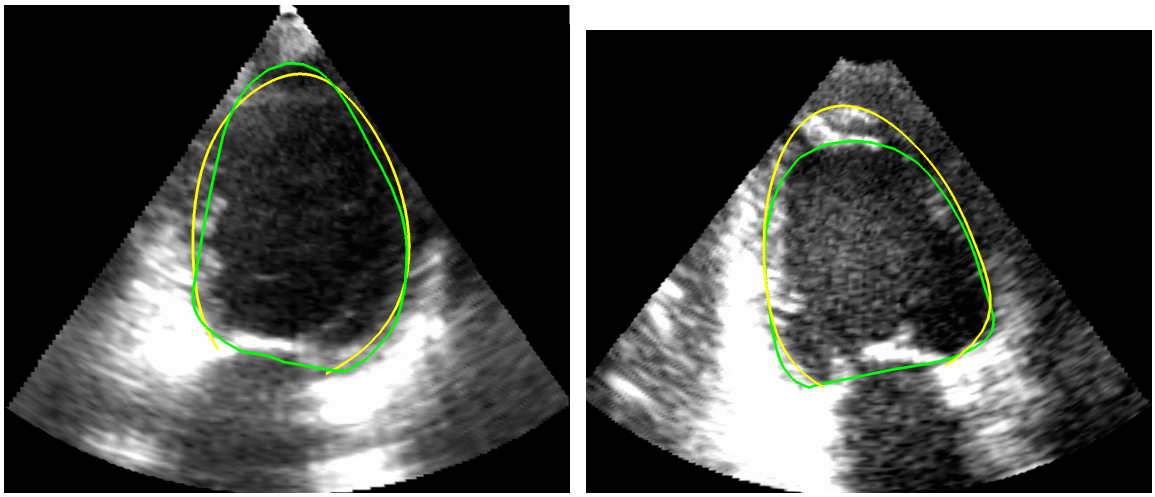

Figure 6: Result of subject 5 ES to the left and subject 10 ED to the right. The yellow line is the mesh border of the ground truth given by the CETUS organizers and the green line is the mesh border of the proposed method.

track the inside edge. Thus, to deal with this problem, different edge detection methods may be needed for different parts of the mesh model. The implementation achieved speeds that enable real-time tracking and it is mainly the number of model and control mesh vertices (M and N) that affect the speed. The proposed mesh model which use mean value coordinates is able to model a complex shape with few control points. This may prove useful when tracking more complex shapes in which traditional methods such as B-spline and subdivision surfaces will have to use many control points which reduces speed significantly. Thus, our future work will focus on applying this method to other applications such as tracking the ventricle of the brain in 3D ultrasound for guidance of ventricular drainage procedures.

5 Conclusion

A method for fully automatic real-time tracking of the LV in 3D+t ultrasound was presented. The method was able to track the LV in all 30 sequences and achieved a mean mesh difference of about 2.5 mm. However, the max error was high (~ 10 mm) on some of the sequences due to failure to detect the LV border in some areas.

References

[1] Kalman, R.: A new approach to linear filtering and prediction problems. Journal of Fluids Engineering **82**(1) (1960) 35–45

Subject	MAD	HD	Modified dice	Min error	Max error	EF	Reference EF	SV	Reference SV
1 ES	2.35	6.82	0.1	0	6.82	-	-	-	-
1 ED	1.39	4.01	0.05	0.01	4.01	41.4	44.2	122.1	126.6
2 ES	1.66	6.22	0.08	0.01	6.05	-	-	-	-
2 ED	1.33	4.73	0.06	0	4.59	22.9	19.3	46.3	39.6
3 ES	4.47	8.88	0.26	0.01	8.88	-	-	-	-
3 ED	3.53	7.42	0.16	0.02	7.42	43.6	55	72.2	67.7
4 ES	2.95	9.69	0.15	0.02	9.34	-	-	-	-
4 ED	2.07	8.06	0.1	0.01	7.61	35	24.1	69	46.8
5 ES	1.31	3.97	0.07	0	3.97	-	-	-	-
5 ED	1.58	5.24	0.08	0	4.98	26.4	25.9	32.2	32.5
6 ES	2.33	7.7	0.12	0.01	7.7	-	-	-	-
6 ED	2.43	7.34	0.1	0.01	7.34	54.9	53.6	72.4	64.9
7 ES	3.96	10.55	0.22	0.03	10.55	-	-	-	-
7 ED	2.08	6.67	0.1	0.01	6.67	33.1	47.2	44.9	56.8
8 ES	3.31	9.27	0.17	0.01	9.27	-	-	-	-
8 ED	2.41	7.52	0.1	0.01	7.52	42	48.8	67.3	72.2
9 ES	1.69	6.26	0.08	0	6.26	-	-	-	-
9 ED	1.44	5.21	0.07	0	5.21	47.5	51	66.9	67.9
10 ES	2.32	10.34	0.11	0	9.86	-	-	-	-
10 ED	1.96	11.36	0.09	0.01	10.68	17	16.9	30.3	32.6
11 ES	1.96	6.66	0.13	0	6.31	-	-	-	-
11 ED	1.51	6.46	0.08	0.01	6.36	38.5	34.9	64.5	65.4
12 ES	3.13	9.67	0.1	0.02	9.67	-	-	-	-
12 ED	2.83	8.32	0.09	0.02	7.87	17.1	23.2	57.5	75.5
13 ES	1.9	6.77	0.06	0	6.28	-	-	-	-
13 ED	1.84	5.79	0.05	0.02	5.79	14.1	13.6	58.9	55.3
14 ES	2.86	7.86	0.11	0	7.86	-	-	-	-
14 ED	3.09	7.82	0.11	0.01	7.82	18.3	15.1	48.9	37.1
15 ES	1.98	6.19	0.06	0	6.19	-	-	-	-
15 ED	2.19	7.28	0.07	0	7.28	22.5	24.3	85.7	93
Mean	2.33	7.34	0.10	0.01	7.21	31.62	33.14	62.60	62.26
Std. Dev.	0.80	1.87	0.05	0.01	1.80	12.81	15.25	22.55	24.88

Table 1: Results for the training dataset.

Subject	MAD	HD	Modified dice	Min error	Max error	EF	Reference EF	SV	Reference SV
16 ES	1.48	6.53	0.08	0	6.53	-	-	-	-
16 ED	1.37	5.19	0.07	0	4.89	43	47.3	49.9	56.1
17 ES	1.71	4.2	0.11	0	4.2	-	-	-	-
17 ED	1.5	4.75	0.08	0.01	4.33	38.7	46.9	44.9	54.3
18 ES	3.25	8.02	0.12	0.01	8.02	-	-	-	-
18 ED	2.96	6.77	0.1	0.02	6.77	22.7	26	59.9	57.3
19 ES	3.27	8.63	0.17	0	8.63	-	-	-	-
19 ED	3.61	9.12	0.15	0.01	9.12	39.6	38.7	66.5	49.2
20 ES	2.53	8.09	0.17	0	7.46	-	-	-	-
20 ED	2.01	9.12	0.1	0	8.83	48	55	53.4	61.8
21 ES	2.51	7.98	0.12	0	7.89	-	-	-	-
21 ED	1.79	4.32	0.07	0.01	4.22	32.3	36.1	63.2	67.1
22 ES	3.86	12.95	0.18	0.02	12.95	-	-	-	-
22 ED	2.79	6.93	0.12	0.02	6.93	29.1	38.2	50.5	53.7
23 ES	3.66	12.97	0.2	0	12.97	-	-	-	-
23 ED	4.54	12.3	0.19	0	12.3	49.8	48.5	64.9	46.9
24 ES	2.76	7.59	0.14	0.05	7.59	-	-	-	-
24 ED	1.79	7.06	0.08	0	6.7	27.1	31.8	39.6	44.3
25 ES	2.21	7.96	0.13	0.01	7.13	-	-	-	-
25 ED	2.37	11.68	0.11	0.04	10.82	51.6	56.8	69.8	75
26 ES	3.35	11.4	0.15	0	10.74	-	-	-	-
26 ED	3.52	13.04	0.14	0.04	12.53	30.8	33.9	65.4	78.3
27 ES	3.03	6.37	0.23	0.04	6.37	-	-	-	-
27 ED	2.54	5.3	0.15	0.02	5.3	40	51.9	40.6	43
28 ES	1.96	6.11	0.12	0	6.11	-	-	-	-
28 ED	2.75	7.71	0.13	0.02	7.71	49	52.6	46.7	42.9
29 ES	2.29	5.69	0.16	0	5.69	-	-	-	-
29 ED	2.06	4.67	0.1	0.01	4.67	56.2	55	50.2	43.9
30 ES	4.47	13.19	0.19	0.01	13.19	-	-	-	-
30 ED	5.28	12.84	0.18	0.01	12.84	34.6	31.9	70.7	51.1
Mean	2.77	8.28	0.13	0.01	8.11	39.50	43.37	55.74	54.99
Std. Dev.	0.97	2.94	0.04	0.01	2.91	10.01	10.01	10.60	11.29

Table 2: Results for the test dataset.

[2] Jacob, G., Noble, J.A., Behrenbruch, C., Kelion, A.D., Banning, A.P.: A shape-space-based approach to tracking myocardial borders and quantifying regional left-ventricular function applied in echocardiography. IEEE transactions on medical imaging **21**(3) (March 2002) 226–38

[3] Orderud, F.: A Framework for Real-Time Left Ventricular Tracking in 3D+T Echocardiography , Using Nonlinear Deformable Contours and Kalman Filter Based Tracking. Computers in Cardiology **33** (2006) 125–128

[4] Orderud, F., Hansgå rd, J.g., Rabben, S.I.: Real-time tracking of the left ventricle in 3D echocardiography using a state estimation approach. Medical Image Computing and Computer-Assisted Intervention (MICCAI) **10**(Pt 1) (January 2007) 858–865

[5] Orderud, F., Rabben, S.I.: Real-time 3D segmentation of the left ventricle using deformable subdivision surfaces. In: Computer Vision and Pattern Recognition, Ieee (June 2008) 1–8

[6] Garland, M., Heckbert, P.S.: Surface simplification using quadric error metrics. In: Proceedings of the 24th annual conference on Computer graphics and interactive techniques, New York, New York, USA, ACM Press (1997) 209–216

[7] Ju, T., Schaefer, S., Warren, J.: Mean Value Coordinates for Closed Triangular Meshes. ACM Transactions on Graphics **24**(3) (2005) 561–566

[8] Rabben, S., Torp, A., Stø ylen, A., Slø rdahl, S., Bjø rnstad, K., Haugen, O., Angelsen, B.: Semiautomatic contour detection in ultrasound M-mode images. Ultrasound in medicine and biology **26**(2) (2000) 287–296

[9] Zhang, C., Chen, T.: Efficient feature extraction for 2D/3D objects in mesh representation. In: Image Processing. (2001) 935–938

G

Real-Time Automatic Vessel Segmentation and Model Registration for Improved Ultrasound-Guided Regional Anaesthesia of the Femoral Nerve

Authors

Erik Smistad and Frank Lindseth

Submitted to

IEEE Transactions on Medical Imaging, April 2015.

Real-Time Automatic Vessel Segmentation and Model Registration for Improved Ultrasound-Guided Regional Anaesthesia of the Femoral Nerve

Erik Smistad[1,2] and Frank Lindseth[2,1]

1) Norwegian University of Science and Technology

2) SINTEF Medical Technology, Trondheim, Norway.

Abstract

The goal is to create an assistant for ultrasound-guided femoral nerve block. By segmenting and visualizing the important structures such as the femoral artery, we hope to improve the success of these procedures. This article is the first step towards this goal and presents novel methods for identifying the femoral artery and registering a model of the surrounding anatomy to the ultrasound images in real-time.

The femoral artery is modelled as a circle compressed by the ultrasound probe. The vessel is first detected by a novel algorithm which initializes the vessel tracking. This algorithm is completely automatic and requires no user interaction. Vessel tracking is achieved with a Kalman filter. A mesh model of the surrounding anatomy was created from a CT dataset. Registration of this model is achieved by first placing the ultrasound image frames at the target site. After this initialization, each ultrasound image frame is registered to the artery model using the detected center-points from the tracking. A bone segmentation method is also used, and if any bone is detected, it is used to register the model in the head-feet direction.

The vessel detection method was able to automatically detect the femoral artery and initialize the tracking in all 12 ultrasound sequences. The accuracy of the tracking algorithm achieved an average dice similarity coefficient of 0.90, mean absolute distance of 0.42 mm, and Hausdorff distance 1.17 mm. The average runtime was measured to be 42, 5, 0.11 and 34 milliseconds for the vessel detection, tracking, registration and bone segmentation methods respectively.

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 610425.

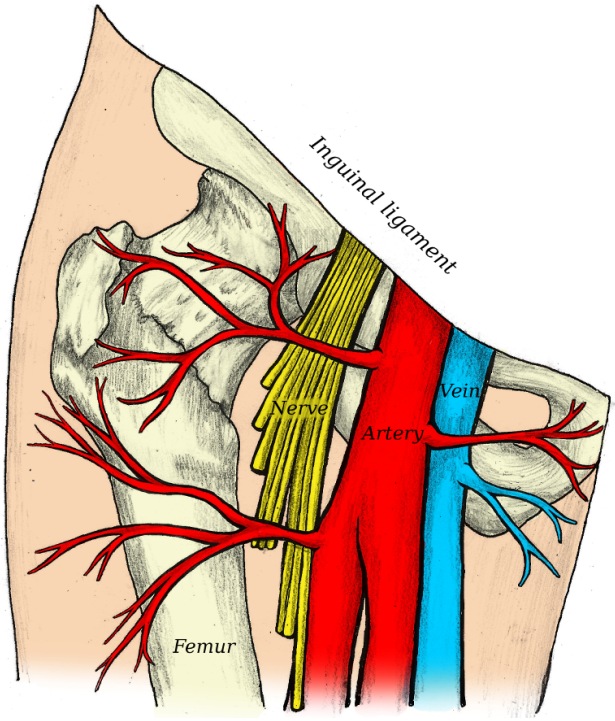

Figure 1: Illustration of the femoral nerve block region showing the femoral artery, vein and nerve along with femur and the pelvic bone. Image courtesy of H. E. Mørk (helemork.com)

1 Introduction

The use of regional anaesthesia (RA) is increasing due to the benefits over general anaesthesia (GA) such as reduced morbidity and mortality [20, 3, 25], reduced postoperative pain, earlier mobility, shorter hospital stay, and lower costs [7]. Despite these clinical benefits, RA remains less popular than GA. One reason for this is that GA is far more successful and reliable than RA. Ultrasound has been employed to increase the success rate of RA [12, 9]. However, ultrasound-guided RA can be a challenging technique, especially for inexperienced physicians and in difficult cases. Good theoretical, practical and non-cognitive skills are needed in order to achieve confidence in performing RA and to keep complications to a minimum. Studies indicate that RA education focusing on illustrations and text alone is not sufficient [26]. The RASimAs project (Regional Anaesthesia Simulator and Assistant) is a European research project which aims at providing a virtual reality simulator to improve the training of doctors performing RA, as well as an assistant to lessen the cognitive burden and help performing RA procedures.

This article focuses on creating an assistant for ultrasound-guided RA to block the femoral nerve. In this application, the femoral artery is an important structure used to identify the location of the femoral nerve as shown in figures 1 and 3. This article presents novel methods for identifying the femoral artery in ultrasound images and registering a model of the surrounding anatomy to the images. The idea is that the registered model together with

the segmented artery will help locate the femoral nerve and fascias. Also, the registered model will be visualized together with the ultrasound probe in a 3D scene thus giving the user an overview of the probe location and the surrounding anatomy. Ultrasound is a real-time imaging modality and delivers several images per second. The segmentation and registration methods must be able to process the ultrasound images in real-time to be useful for the femoral nerve block assistant.

Several methods for segmentation of the cross-section of vessels in 2D ultrasound have been reported, using methods such as level sets [1], fuzzy c-means clustering [2] and evolutionary algorithms [14]. These methods focus on segmenting a single image. However, in this work the goal is to segment the femoral artery in real-time on a sequence of ultrasound images. Guerrero et al. [13] presented a method for vessel segmentation and tracking in ultrasound images using an extended Kalman filter. Their method was fast and accurate, but it had to be manually initialized with a seed point inside the vessel.

The contributions of this article are:

- A real-time automatic vessel detection method. This method eliminates the need for manual initialization such as in the method of Guerrero et al. [13].
- A real-time vessel tracking method of the femoral artery similar to the approach of Guerrero et al. [13]. While their method uses two Kalman filters, one for estimating the position of the vessel and another to estimate the shape, the proposed method uses only one Kalman filter resulting in a simpler method.
- A real-time vessel registration method which registers a model of the surrounding anatomy to the ultrasound images.
- An ultrasound bone segmentation method based on the method of Foroughi et al. [11]. In comparison to their method, the proposed method uses a GPU to achieve real-time bone segmentation.

2 Methods

This section first describes the vessel model used to detect and track the femoral artery. Next, the vessel detection and tracking methods are presented. Finally, the bone segmentation and registration methods are described. To achieve real-time performance, the presented methods are implemented using the framework for heterogeneous medical image computing and visualization (FAST) [21]. This framework enables efficient computation and visualization on heterogeneous systems which include different processors such as CPUs and graphic processing units (GPUs). GPUs have shown to have great potential in accelerating medical image segmentation [24], registration [10] and visualization [22, 6].

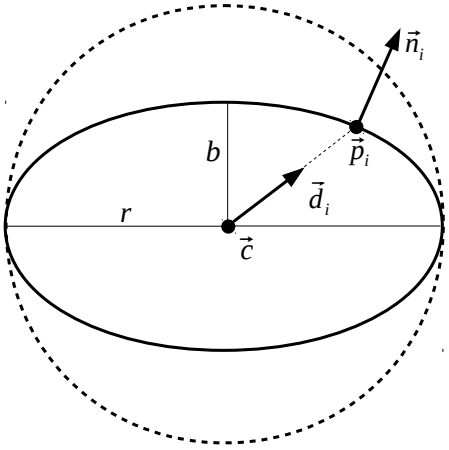

Figure 2: Vessel cross-section modelled as a compressed circle with radius r, center $\vec{c}$ and flattening factor $f = 1 - \frac{r}{b}$.

2.1 Vessel model

The vessel cross-section in the ultrasound images is modelled as a compressed circle of radius r. The circle is compressed with the flattening factor $f = 1 - \frac{b}{r}$, creating an ellipse with major and minor radii r and b as shown in Fig. 2. The point $\vec{p}_i$ and its normal $\vec{n}_i$ on a circle of N points with center $\vec{c}$ can be calculated with the following equations.

$$\alpha_i = \frac{2\pi i}{N} \tag{1}$$

$$\vec{d_i} = [\cos(\alpha_i), (1-f)\sin(\alpha_i)] \tag{2}$$

$$\vec{p_i} = \vec{c} + r\vec{d_i} \tag{3}$$

$$\vec{n}_i = \frac{[(1-f)r\cos(\alpha_i), r\sin(\alpha_i)]}{|(1-f)r\cos(\alpha_i), r\sin(\alpha_i)]|} \tag{4}$$

2.2 Vessel detection

In this section, a novel fully automatic vessel detection method is presented which is used to initialize the vessel tracking algorithm described in the next section. First, the image is blurred using convolution with a Gaussian mask ($\sigma = 0.5mm$) and then the image gradients $\vec{G}$ are calculated. For a given radius r and flattening f, the vessel score S is calculated as the average dot product of the outward normal $\vec{n}_i$ and the corresponding image gradient at N points on the compressed circle, as shown in (5). Before the dot product is calculated, the image gradient is normalized so that it has unit length. This normalization makes this vessel detection method invariant to the contrast of the image,

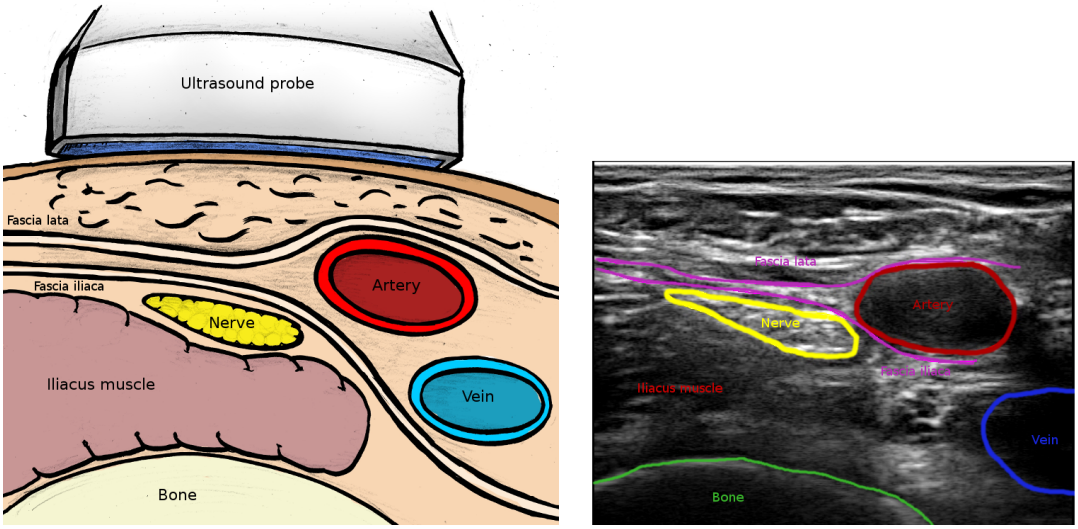

Figure 3: Left: Cross-section illustration of the region of interest (ROI). **Right:** Ultrasound image of the ROI with annotations of the important structures. Image courtesy of H. E. Mørk (helemork.com)

and only the direction of the gradients influence the score.

$$S(\vec{c}, r, f) = \frac{1}{N} \sum_{i=0}^{N-1} \vec{n}_i \cdot \frac{\vec{G}(\vec{p}_i)}{|\vec{G}(\vec{p}_i)|} \tag{5}$$

For each pixel, ellipses of different radii ranging from 3.5 to 6 mm, flattening factor from 0 to 0.5 and $N = 32$ samples were used to calculate the vessel score. The ellipse with the highest score is selected for each pixel. The best score and the values r and f is stored for each pixel. The ellipse with the highest score of all pixels is selected and used to initialize the tracking. Real-time performance of this vessel segmentation method is achieved by using a GPU to compute the vessel score of all pixels in parallel.

For a detected vessel to be accepted it has to have a vessel score S above the threshold $T_s = 0.8$. Also, the centerpoint detected from five consecutive frames has to be within 1.5 mm of each other and the average intensity of the detected vessel border has to be above the threshold $T_b = 40$. These requirements are needed to make the vessel detection robust enough to properly initialize the vessel tracking.

2.3 Vessel tracking

Vessel tracking in the ultrasound images is achieved with a Kalman filter [17]. The Kalman filter estimates a state using a set of noisy measurements over time. The state consists of 4 variables, the vessel center, radius and flattening factor $\mathbf{x} = [c_x, c_y, r, f]$. The state is predicted for the next image frame using a motion model as shown in (6),

along with the covariance error matrix $\mathbf{P}$ in (7).

$$\bar{\mathbf{x}}_{t+1} = \mathbf{A}_1\hat{\mathbf{x}}_t + \mathbf{A}_2\hat{\mathbf{x}}_{t-1} \tag{6}$$

$$\bar{\mathbf{P}}_{t+1} = \mathbf{A}_1\hat{\mathbf{P}}_t\mathbf{A}_1^T + \mathbf{A}_2\hat{\mathbf{P}}_{t-1}\mathbf{A}_2^T + \mathbf{A}_1\hat{\mathbf{P}}_t\mathbf{A}_2^T + \mathbf{A}_2\hat{\mathbf{P}}_{t-1}\mathbf{A}_1^T + \mathbf{Q} \tag{7}$$

For the motion model, the velocity was dampened by a factor of $d = 0.5$ so that $\mathbf{x}_{t+1} = \mathbf{x}_t + (\mathbf{x}_t - \mathbf{x}_{t-1})0.5 = 1.5\mathbf{x}_t - 0.5\mathbf{x}_{t-1}$. This gives diagonal state transition matrices $\mathbf{A}_1$ and $\mathbf{A}_2$ with values 1.5 and -0.5 respectively. The dampening reduces tracking failure when the vessel moves quickly in the image and suddenly stops. A diagonal matrix was also used for the process error matrix $\mathbf{Q}$ with values 0.01. The size of these matrices is equal to the size of the state vector (4×4).

A hybrid edge detection method is used to detect two different types of edges, step edges and ridge edges. The edge detection finds the normal displacement ($v_i = \vec{n}_i^T(\vec{p}_{i,\text{observed}} - \vec{p}_{i,\text{predicted}})$) for each point i in the predicted circle in a line centered at the point and in the direction of the normal. The length of these lines is set to be equal to the radius of the circle. The hybrid edge detection method first looks for a step edge using a step model [19], which entails finding a k that maximizes the following measure where $l(s)$ is the image intensity at step s along the line.

$$\sum_{s=0}^{k}\left|\left[\frac{1}{k+1}\sum_{j=0}^{k} l(j)\right] - l(s)\right| + \sum_{s=k+1}^{L-1}\left|\left[\frac{1}{L-k}\sum_{j=k+1}^{L-1} l(j)\right] - l(s)\right| \tag{8}$$

A measurement noise value r_i is also recorded for each edge and is calculated based on the edge strength:

$$r_i = \frac{1}{\frac{1}{L-k}\sum_{j=k+1}^{L-1} l(j) - \frac{1}{k+1}\sum_{j=0}^{k} l(j)} \tag{9}$$

The step edge is only accepted if the denominator of (9) is above the threshold $T_e = 10$. If the step edge is not accepted, a ridge edge detection method is used. This method looks for the first position s on the line where the gradient is larger than the threshold T_e. The measurement noise value for the ridge edges are set to be $r_i = \frac{1}{l(s)-l(s-1)}$.

These edge measurements are nonlinear because they cannot be expressed as matrix multiplication of the state $\mathbf{x}$. Therefore an extended Kalman filter is used in which the observation model is linearized. This is done by calculating the Jacobi matrix that relate changes in each circle point $\vec{p}_i$ to changes in the state $\mathbf{x}$. The final measurement vector $\vec{h}_i^T$

is the normal projection of these Jacobi matrices:

$$\begin{aligned}\vec{h}_i^T &= \vec{n}_i^T \frac{\partial \vec{p}_i}{\partial \mathbf{x}} \\ &= \vec{n}_i^T \begin{bmatrix} 1 & 0 & \cos(\alpha_i) & 0 \\ 0 & 1 & (1-f)\sin(\alpha_i) & -r\sin(\alpha_i) \end{bmatrix}\end{aligned} \tag{10}$$

By assuming that the measurements are independent, the measurement noise covariance matrix $\mathbf{R}$ becomes a diagonal matrix of the measurement noise values r_i. The multiplications of $\mathbf{R}$, the measurement-to-state transition matrix $\mathbf{H}$ and the measurements $\vec{v}$ becomes a simple summation as shown in equations (11) and (12) [18]. If no edge is found for a measurement point i, it is omitted in the summations.

$$\mathbf{H}^T\mathbf{R}^{-1}\mathbf{v} = \sum_{i=0}^{N-1} \vec{h}_i^T r_i^{-1} v_i \tag{11}$$

$$\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H} = \sum_{i=0}^{N-1} \vec{h}_i^T r_i^{-1} \vec{h}_i \tag{12}$$

This makes the Kalman update equations (13) and (14) invariant to the number of measurements which improves speed as matrix inversion of large matrices is avoided. Using $\hat{\mathbf{P}}_{t+1}\mathbf{H}^T\mathbf{R}^{-1}$ as the Kalman gain, the updated state and error covariance estimate becomes [5]:

$$\hat{\mathbf{P}}_{t+1} = (\bar{\mathbf{P}}_{t+1}^{-1} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})^{-1} \tag{13}$$

$$\hat{\mathbf{x}}_{t+1} = \bar{\mathbf{x}}_{t+1} + \hat{\mathbf{P}}_{t+1}\mathbf{H}^T\mathbf{R}^{-1}\mathbf{v} \tag{14}$$

2.4 Bone segmentation

Although the bone is not an important feature for identification of the nerve in the ultrasound images, it helps with the registration of the model as shown in the next section.

The dynamic programming approach of Foroughi et al. [11] was used to segment the bone in the ultrasound images. This method first calculates the probability of each pixel being the interface between non-bone and bone tissue. Bone creates a specific response in B-mode ultrasound images. High reflection and shadowing effect are two features which are used to calculate this probability. A Laplacian of Gaussian (LoG) filter is used to calculate the amount of reflection. The shadow strength of a pixel is calculated using the weighted sum of the intensity values of all pixels beneath. The bone probability is then the product of the amount of reflection and shadow. The calculation of the bone probability images was done on a GPU to make this method real-time. Dynamic programming is used to find the bone interface in the bone probability image.

2.5 Registration

The identified vessel and bone structures are used to register the model to the ultrasound images. The anatomical model was created from an abdominal CT image. The bone was segmented from this image using region growing. The surface and centerline of the femoral artery was extracted using the tubular extraction method of Smistad et al. [23].

The registration process requires an approximate initialization of the model. Thus an automatic landmark registration is first used to provide a rough registration of the model to the ultrasound images. The ultrasound probe is tracked using an optical tracking system. The position of the four corners of each frame in the ultrasound image stream is collected, and the average position of these four corners is calculated. The corners of frames that are closer than 3 mm from previous frames are not included. This is done to evenly distribute the positions over the scanned area. The rotation and translation to a target frame located at the femoral artery is calculated using procrustes analysis and Kabsch's algorithm [16].

After the initialization, each frame is registered to the model using the centerline and the tracked vessel centerpoint $\vec{x}$ of that frame. First, the point on the centerline c_i that is closest to the image plane is found. The image has four corners ($\vec{I}_0, \vec{I}_1, \vec{I}_2, \vec{I}_3$), which are used to calculate the normal $\vec{n}$ (15). The distance d from a centerline point i to the image plane is calculated using (16).

$$\vec{n} = \frac{(\vec{I}_1 - \vec{I}_0) \times (\vec{I}_2 - \vec{I}_0)}{|(\vec{I}_1 - \vec{I}_0) \times (\vec{I}_2 - \vec{I}_0)|} \tag{15}$$

$$d_i = |(\vec{I}_0 - \vec{c}_i) \cdot \vec{n}| \tag{16}$$

$$c = \text{argmin}_{i=[0,C-1]} d_i \tag{17}$$

Movement is only allowed in the image plane. This is enforced by projecting the closest centerline point $\vec{c}_c$ to the plane, and then calculating the translation $\vec{T}$ using the projected point $\vec{c}_c'$ and the vessel centerpoint $\vec{x}$.

$$\vec{c}_c' = \vec{c}_c + d_c \vec{n} \tag{18}$$

$$\vec{T} = \vec{c}_c' - \vec{x} \tag{19}$$

If bone is detected in the image, it used to register the model in the head-feet direction. The iterative closest point (ICP) algorithm [4] is used for this purpose.

2.6 Evaluation

A total of 12 ultrasound image sequences from 3 subjects were collected. The number of images per sequence ranged from 110 to 524. For each sequence, the vessel was manually segmented in 4 randomly selected frames. The dice similarity coefficient D [8] was calculated to measure the overlapping regions of the segmentation S and the ground

truth G as shown in (20). For the contour of the segmentation, the mean absolute distance A and Hausdorff distance H was calculated in millimeters. These measures are calculated using the shortest distance $\bar{d}(j)$ from contour point j in G to the contour in S as shown in (21) and (22).

$$D = \frac{2|S \cap G|}{|S| + |G|} \tag{20}$$

$$A = \frac{1}{O} \sum_{j=0}^{O-1} \bar{d}(j) \tag{21}$$

$$H = \max_{j \in [0..O-1]} \bar{d}(j) \tag{22}$$

3 Results

The vessel detection initialized the tracking successfully in all 12 sequences. On average, the tracking was successfully initialized after the vessel detection was run on 84 frames. Assuming 25 frames per second, the tracking is initialized in about 3.4 seconds. In 8% of the manually segmented images, a false artery was detected. Due to the requirement that the vessel must be detected at a similar location in five consecutive frames this did not lead to an incorrect initialization. Of the correctly identified arteries, the vessel detection method achieved a dice similarity coefficient of 0.87, mean absolute distance 0.61 mm, and Hausdorff distance of 1.62 mm.

The results of the vessel tracking method for each ultrasound sequence are summarized in Table 1. On average, the tracking method achieved a dice similarity coefficient of 0.90, mean absolute distance of 0.42 mm, and Hausdorff distance 1.17 mm, after being initialized by the proposed detection method. Images of the best and worst segmentation results, according to D, are shown in Fig. 4 for each subject. Fig. 5 show 3D visualizations of one ultrasound frame and the model of the surrounding anatomy after initialization and after registration for each of the three subjects.

The runtime of the vessel detection, tracking, registration and bone segmentation was also measured. Table 2 contains the average speed of processing one frame of each sequence with the different steps. The computer used to measure the runtime was running Ubuntu 14.04 Linux with an AMD A10 CPU with 16 GB RAM, an AMD Radeon R9 290 GPU with 4 GB RAM, and a solid state drive (SSD).

4 Discussion

The vessel detection method was able to automatically detect the femoral artery and initialize the tracking in all 12 ultrasound sequences, while the method of Guerrero et al. has to be manually initialized. This is a great benefit to the femoral nerve block assistant

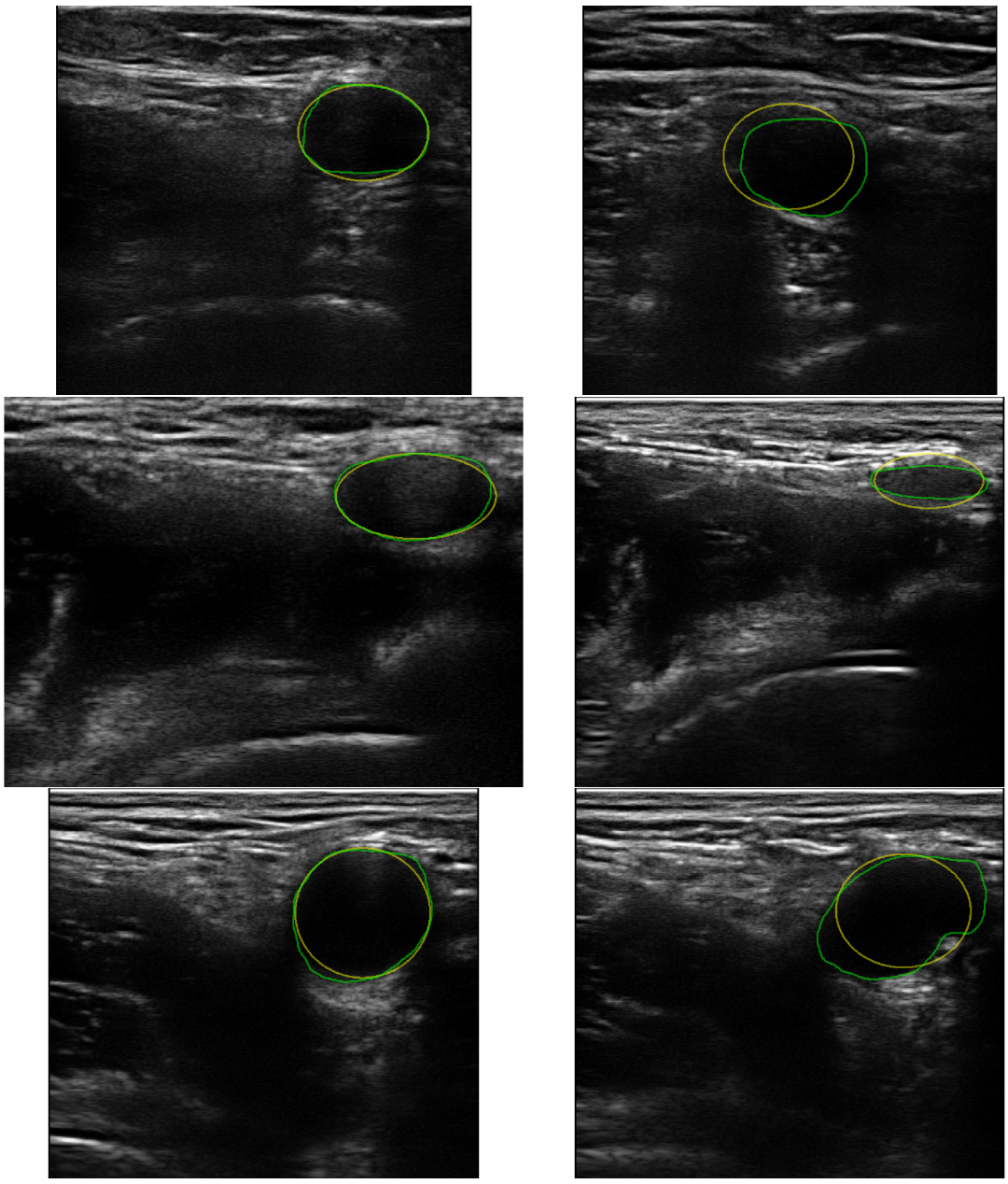

Figure 4: Best (left column) and worst (right column) segmentation results for each of the three subjects determined by the dice similarity coefficient in (20). The green line is the manually segmented artery, while the yellow smooth line is the result of the proposed vessel tracking method.

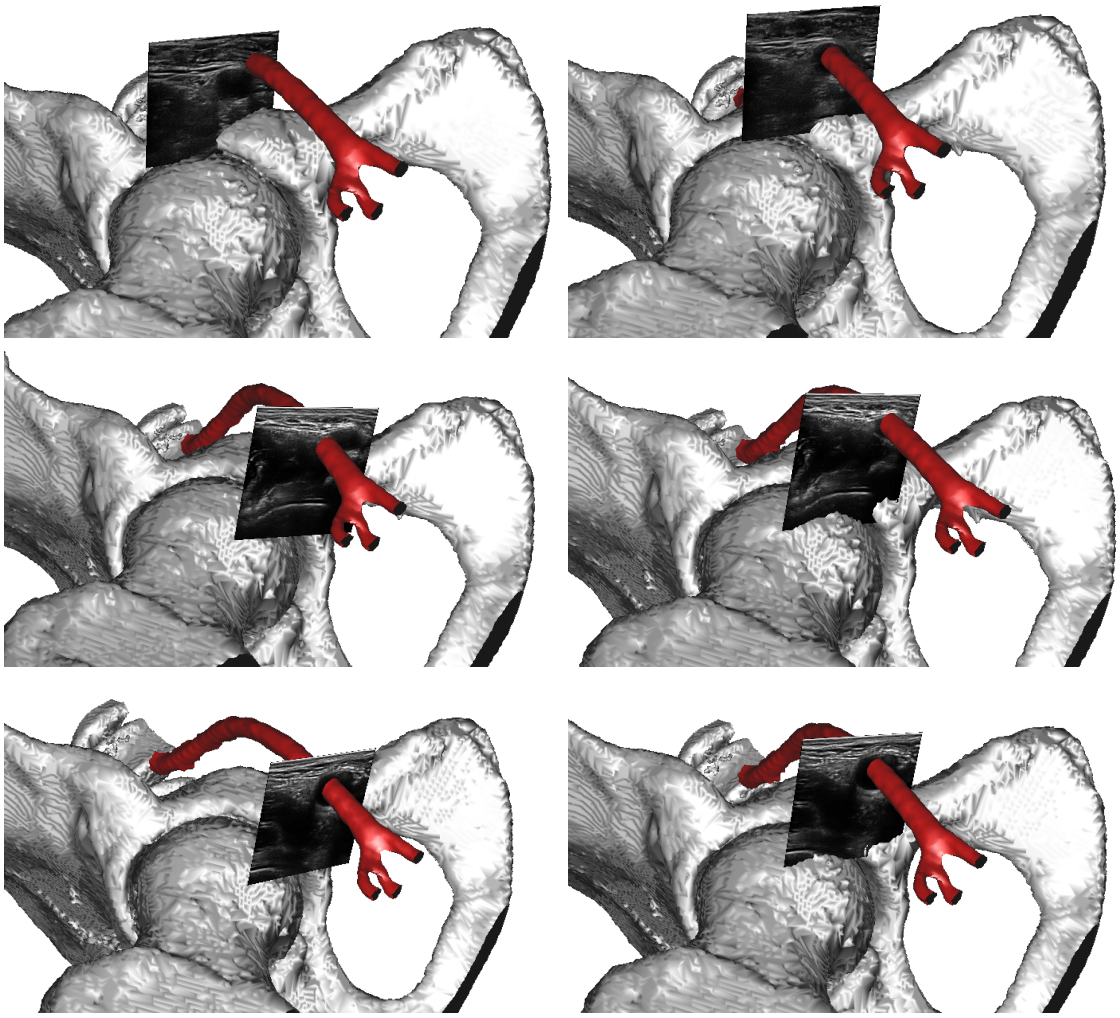

Figure 5: 3D visualization of one ultrasound frame and the model of the surrounding anatomy after initialization (left) and after registration (right) for each of the three subjects.

Subject	Seq.	Dice similarity coeff.	Mean absolute distance (mm)	Hausdorff distance (mm)
1	1	0.93	0.30	0.79
1	2	0.92	0.34	1.00
1	3	0.93	0.30	0.86
1	4	0.91	0.48	1.37
2	1	0.90	0.35	1.12
2	2	0.94	0.22	0.62
2	3	0.90	0.36	1.12
2	4	0.83	0.47	1.03
3	1	0.86	0.81	1.90
3	2	0.89	0.60	1.53
3	3	0.85	0.41	1.28
3	4	0.90	0.41	1.38
Average		0.90	0.42	1.17
Std. dev.		0.04	0.16	0.35

Table 1: Accuracy of the vessel tracking using the measures in (20-22).

Subject	Seq.	Image size	Vessel detection	Vessel tracking	Vessel reg.	Bone seg.
1	1	616×749	48.77	7.59	0.09	35.96
1	2	616×749	43.89	5.86	0.10	37.63
1	3	616×657	34.92	6.59	0.07	35.73
1	4	616×657	39.79	6.57	0.16	39.76
2	1	616×820	57.95	6.76	0.21	36.53
2	2	616×820	50.93	6.73	0.11	29.91
2	3	616×681	39.03	3.51	0.10	31.85
2	4	616×681	38.77	4.24	0.08	36.90
3	1	616×681	32.48	3.19	0.11	31.74
3	2	616×681	34.86	3.88	0.12	32.08
3	3	616×681	40.05	3.23	0.08	31.76
3	4	616×681	38.46	3.49	0.10	28.96
Average			41.66	5.14	0.11	34.07
Std. dev.			7.48	1.68	0.04	3.42

Table 2: Average speed in milliseconds of processing one frame for each of the ultrasound sequences. Note that the vessel detection is only run for the first frames, until the tracking is initialized. The bone segmentation is run concurrently with the other tasks until some bone is found.

Parameter	Description	Value used	Possible range
T_e	Edge detection threshold. For an edge detection measurement to be accepted, the edge must be stronger than this value. Lowering it will allow weaker edges (possibly false edges) to be accepted as measurements for the Kalman filter. The value used is low and allow most edges.	10	0 - 255
T_b	Vessel detection border threshold. The average intensity along the vessel border must be at least this value to be accepted in the vessel detection step. Lowering this value will make the detection algorithm accept dark circles with low intensity borders. Setting it too high, and the detection will not accept any vessels dark circles as vessels.	40	0 - 255
T_s	Vessel detection score threshold. The average fit of the inward normals and the image gradient at each sample point on the circle (5) must be higher than this value for the vessel to be accepted. A vessel score of 1 is a perfect fit.	0.8	-1 - 1
N	Number of samples to be used for the detection and tracking algorithms. Increasing it may increase accuracy, but will also result in slower runtime as more measurements must be acquired.	32	4 - ∞
$\mathbf{Q}$	Covariance matrix of the process noise. This matrix influences how much the measurements affect the state update in the Kalman filter. Lowering it will make the state update less affected by the measurements and vica versa.	0.01	0 - ∞
d	Dampening factor in the motion model (6). This value determines how much the previous state estimate $\mathbf{x}_{t-1}$ influence the state prediction. Lowering it will decrease the velocity from one frame to another and setting it to 0 will remove the motion completely. If set to 1, the motion model will use constant velocity.	0.5	0-1

Table 3: A list of parameters used in the proposed methods along with a description of how their value influence performance.

application as no user interaction is needed. Guerrero et al. [13] reported a mean error of 1.7 pixels using their method on ultrasound images of the common carotid artery, jugular vein and saphenous vein and artery. They do not report the error in millimeters nor the pixel spacing of their ultrasound data. The mean absolute distance using the proposed method on the femoral artery datasets in pixels is 7.78, thus worse than the method by Guerrero et al. However, this may be due the low pixel spacing of the ultrasound images used in this work (0.044-0.058 mm). Nevertheless, we argue that the achieved tracking accuracy (mean absolute difference of 0.42 mm) is good in terms of the application of ultrasound-guided femoral nerve block.

Fig. 4 shows the worst result for each subject in the right column. The worst result of subject 2 (middle) is due to an overcompression of the artery, and the artery was only allowed a compression of 0.5 in this work. The image frame of the worst case result of subject 3 (bottom) was located at the branch of the deep artery of the thigh. The elliptical model was not able to properly describe the cross-section of this branch section.

As long as the artery is properly tracked, the artery model will be placed inside the artery of the ultrasound image. The area of the target site which will be scanned during the femoral nerve block procedure is small (within 5 cm inferior of the inguinal ligament), and the ultrasound images are initialized to this area. If bone is detected in the ultrasound images, it is used to register the model in the head-feet direction. Currently, a static model of the surrounding anatomy is used, which does not incorporate the anatomical variances seen in a population. This may not be accurate enough and remains to be evaluated.

The runtime of the tracking was about 5 ms for each image, thus less than the 23 ms reported by Guerrero et al. [13]. For the vessel detection and bone segmentation the runtime was higher. However, these two methods are executed concurrently and both are within the real-time constraint of 20-25 frames per second of the ultrasound system. Foroughi et al. [11] reported a runtime of 550 ms of their bone segmentation implementation on images of size 378×378. However, in this work GPUs have been employed to reduce the processing time to an average of 34 milliseconds to satisfy the real-time constraints of this application. The pixel height of the images used in this work ranged from 657 to 820, while the width was constant at 616. The runtime of the vessel detection is dependent on the size of the images. Sequences 1 and 2 from subject 2 had the largest height of 820, thus the runtime is larger for these sequences.

The proposed methods contain several parameters, which values have been determined through experimentation. Table 3 provides a list of these parameters along with a description of how their value influence the performance of the proposed methods.

Future work includes segmentation of other structures, such as the femoral nerve, fascia lata and fascia iliaca, needle insertion guidance and enhancement of the local anaesthesia after insertion. The idea is that the registered model together with the segmented artery will help locate the femoral nerve and fascias (see Fig. 3). It may also be necessary to create a model which incorporates the anatomical differences in a population using methods such as statistical shape models [15].

5 Conclusion

The presented methods are able to automatically and accurately track the femoral artery in ultrasound images and use this to register a model of the surrounding anatomy in real-time. This will be part of an assistant for ultrasound-guided regional anaesthesia of the femoral nerve.

References

[1] A. R. Abdel-Dayem. Level set framework for detecting arterial lumen in ultrasound images. In *Proceedings of Computational Vision and Medical Image Processing, VIPIMAGE*, pages 267–272, 2011.

[2] A. R. Abdel-Dayem and M. R. El-Sakka. Fuzzy C-Means Clustering for Segmenting Carotid Artery Ultrasound Images. In *Image Analysis and Recognition*, pages 935–948. 2007.

[3] W. S. Beattie, N. H. Badner, and P. Choi. Epidural analgesia reduces postoperative myocardial infarction: a meta-analysis. *Anesthesia and analgesia*, 93:853–858, 2001.

[4] P. J. Besl and N. D. McKay. A method for registration of 3-D shapes. *IEEE Transactions on pattern analysis and machine intelligence*, 1992.

[5] A. Blake and M. Isard. *Active Contours: The Application of Techniques from Graphics, Vision, Control Theory and Statistics to Visual Tracking of Shapes in Motion*. Springer London, 1998.

[6] M. Bozorgi and F. Lindseth. GPU-based multi-volume ray casting within VTK for medical applications. *International journal of computer assisted radiology and surgery*, May 2014.

[7] V. W. Chan, P. W. Peng, Z. Kaszas, W. J. Middleton, R. Muni, D. G. Anastakis, and B. a. Graham. A comparative study of general anesthesia, intravenous regional anesthesia, and axillary block for outpatient hand surgery: clinical outcome and cost analysis. *Anesthesia and analgesia*, 93:1181–1184, 2001.

[8] L. R. . Dice. Measures of the Amount of Ecologic Association Between Species. *Ecology*, 26(3):297–302, 1945.

[9] J. Dolan, A. Williams, E. Murney, M. Smith, and G. N. C. Kenny. Ultrasound Guided Fascia Iliaca Block: A Comparison With the Loss of Resistance Technique. *Regional Anesthesia and Pain Medicine*, 33(6):526–531, 2008.

[10] O. Fluck, C. Vetter, W. Wein, a. Kamen, B. Preim, and R. Westermann. A survey of medical image registration on graphics hardware. *Computer methods and programs in biomedicine*, 104(3):e45–57, Dec. 2011.

[11] P. Foroughi, E. Boctor, M. J. Swartz, R. H. Taylor, and G. Fichtinger. Ultrasound Bone Segmentation Using Dynamic Programming. *2007 IEEE Ultrasonics Symposium Proceedings*, pages 2523–2526, 2007.

[12] J. Griffin and B. Nicholls. Ultrasound in regional anaesthesia. *Anaesthesia*, 65 Suppl 1:1–12, 2010.

[13] J. Guerrero, S. E. Salcudean, J. a. McEwen, B. a. Masri, and S. Nicolaou. Real-time vessel segmentation and tracking for ultrasound imaging applications. *IEEE Transactions on Medical Imaging*, 26(8):1079–1090, 2007.

[14] P. Guzman, R. Ros, and E. Ros. Artery Segmentation in Ultrasound Images Based on an Evolutionary Scheme. *Informatics*, 1:52–71, 2014.

[15] T. Heimann and H.-P. Meinzer. Statistical shape models for 3D medical image segmentation: a review. *Medical image analysis*, 13(4):543–63, Aug. 2009.

[16] W. Kabsch. A discussion of the solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A*, 34(6):827–828, 1978.

[17] R. Kalman. A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering*, 82(1):35–45, 1960.

[18] F. Orderud. A Framework for Real-Time Left Ventricular Tracking in 3D+T Echocardiography , Using Nonlinear Deformable Contours and Kalman Filter Based Tracking. *Computers in Cardiology*, 33:125–128, 2006.

[19] S. Rabben, A. Torp, A. Stø ylen, S. Slø rdahl, K. Bjø rnstad, O. Haugen, and B. Angelsen. Semiautomatic contour detection in ultrasound M-mode images. *Ultrasound in medicine and biology*, 26(2):287–296, 2000.

[20] A. Rodgers, N. Walker, S. Schug, A. Mckee, H. Kehlet, a. V. Zundert, D. Sage, M. Futter, G. Saville, T. Clark, and S. Macmahon. Reduction of postoperative mortality and morbidity with epidural or spinal anaesthesia: results from overview of randomised trials. *British Medical Journal*, 321:1493–1497, 2000.

[21] E. Smistad, M. Bozorgi, and F. Lindseth. FAST: framework for heterogeneous medical image computing and visualization. *International Journal of Computer Assisted Radiology and Surgery*, 2015.

[22] E. Smistad, A. C. Elster, and F. Lindseth. Real-Time Surface Extraction and Visualization of Medical Images using OpenCL and GPUs. In *Norsk informatikkonferanse*, pages 141–152. Akademika forlag, 2012.

[23] E. Smistad, A. C. Elster, and F. Lindseth. GPU accelerated segmentation and centerline extraction of tubular structures from medical images. *International Journal of Computer Assisted Radiology and Surgery*, 9(4):561–575, 2014.

[24] E. Smistad, T. L. Falch, M. Bozorgi, A. C. Elster, and F. Lindseth. Medical image segmentation on GPUs – A comprehensive review. *Medical Image Analysis*, 20(1):1–18, 2015.

[25] S. C. Urwin, M. J. Parker, and R. Griffiths. General versus regional anaesthesia for hip fracture surgery: a meta-analysis of randomized trials. *British journal of anaesthesia*, 84(4):450–455, 2000.

[26] B. S. d. Worm, M. Krag, and K. Jensen. Ultrasound-Guided Nerve Blocks - Is Documentation and Education Feasible Using Only Text and Pictures? *PLoS ONE*, 9(2), 2014.

Appendix

A1

Real-time surface extraction and visualization of medical images using OpenCL and GPUs

Authors

Erik Smistad, Anne C. Elster and Frank Lindseth

Published in

Norsk informatikkonferanse 2012, pages 141-152.

Real-Time Surface Extraction and Visualization of Medical Images using OpenCL and GPUs

Erik Smistad[1], Anne C. Elster[1], Frank Lindseth[1,2]

1) Norwegian University of Science and Technology

2) SINTEF Medical Technology. Trondheim, Norway

Abstract

Marching Cubes (MC) is an algorithm that extracts surfaces from volumetric scalar data. It is used extensively in visualization and analysis of medical data from modalities like CT and MR, usually after a 3D segmentation of the structures of interest have been performed. Implementations of MC on CPUs are slow, using several seconds (even minutes) to extract the surface before sending it to the Graphics Processing Unit (GPU) for rendering. Fast surface extraction implementations are very beneficial in medical applications, where large datasets are used and time is crucial. Analysis of medical image data often entails changing different parameters, thus real-time implementations are very desirable. MC is a completely data-parallel algorithm, making it ideal for execution on GPUs. GPU processing enables the result to be rendered on screens in a few milliseconds. In this paper, a MC implementation written in OpenCL that runs entirely on the GPU is presented. We show that our implementation uses a more efficient storage scheme than previous GPU implementations, and that this enables real-time processing of large medical datasets. Our implementation also shows that GPU implementations written in OpenCL has the potential of being just as fast and efficient as CUDA or shader implementations.

1 Introduction

Creating 3D visualizations of large medical datasets using serial processing on the Central Processing Unit (CPU) is very time consuming and inefficient. The Marching Cubes (MC) algorithm was introduced by Lorensen and Cline [9], and has become the standard algorithm for generating surfaces from volumetric data. MC divides the 3D dataset into a set of cubes that can be processed independently. The original implementation processed each cube sequentially. Image analysis in medical imaging applications often requires experimentation of parameters before a satisfactory result is achieved. For each trial of parameters the result has to be visualized. The total waiting time for creating the surface needed to visualize the result of many trials can become very long. The waiting time

can be significantly reduced by exploiting the data parallel nature of the MC algorithm and running it on a Graphics Processing Unit (GPU). These processors have several hundred functional units that can each process a cube in parallel. MC is a completely data parallel algorithm as each cube in the grid can be processed independently. A typical medical dataset can have from 2 to 200 million cubes. Thus parallel implementations of the algorithm has the potential of large speedups.

The Open Computing Language (OpenCL) is a new framework for writing programs that can execute on heterogeneous platforms. OpenCL enables execution, data transfer and synchronization on different devices, such as CPUs, GPUs and Cell Broadband Engines, without having to write device or vendor specific code.

In this paper, we present a MC implementation written in OpenCL, that runs entirely on the GPU. It uses the Histogram Pyramid data structure, presented by Ziegler *et al.* [18]. We show that our implementation use a more efficient storage scheme for Histogram Pyramids than previous implementations and that this enables the real-time processing of large medical datasets.

The next section describes the MC algorithm, GPU computing and related work. The methodology section describes our implementation in detail. In the results section, performance measures for our implementation on three different GPUs are presented and compared to the implementation of Dyken *et al.* [5]. The last two sections include discussion and conclusions based on the results.

2 Background

In this section, an introduction to the MC algorithm and GPU computing is given. This is followed up with a discussion on the challenges and related work of running MC on the GPU.

2.1 Marching Cubes

MC was introduced by Lorensen and Cline [9] as an algorithm for creating a 3D surface consisting of triangles from a volumetric dataset of scalars. The algorithm uses a parameter, called the iso-value, to classify points in the dataset as either inside or outside the surface. The dataset is divided into a grid cubes, and each corner in each cube is represented by a data point in the dataset. By knowing which corners are outside and inside, triangles can be placed inside each cube to create the entire surface. In total, there are $2^8 = 256$ unique corner configurations of a cube. However, by considering symmetry this can be reduced to the 15 configurations depicted in Figure 1. It has been shown, that using only these 15 configurations can lead to topologically incorrect surfaces due to ambiguities. Chernyaev [3] showed how to deal with this by extending the number of unique configurations to 33.

Linear interpolation is often used to place the vertices of the triangles and approximate the surface normals, so that the surface becomes more smooth and represents the data better.

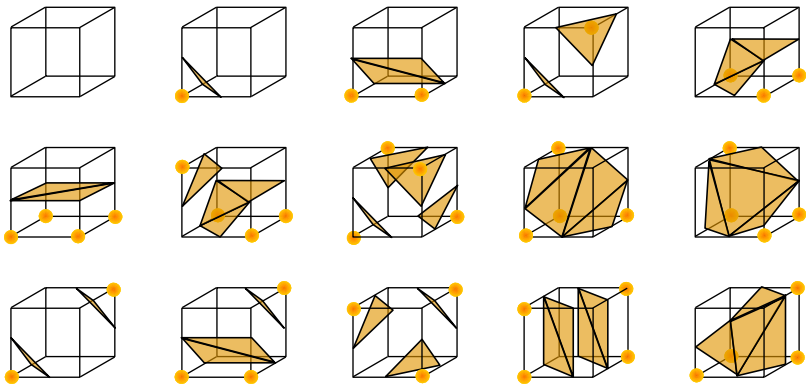

Figure 1: The 15 cube configurations from Lorensen and Cline [9], and the set of triangles that represents the surface. The marked corner points are considered to be inside the surface.

2.2 GPU Computing

GPUs were originally designed to help speed up the memory-intensive rendering calculations in demanding 3D applications. These devices are now increasingly used to accelerate the numerical computations in science and technology [15, 2]. The calculations the original GPUs were targeting was texture mapping, rendering polygons and transformation of coordinates. The GPU is a type of single instruction, multiple data (SIMD) processor. It can perform the same instruction on each element in a dataset in parallel. GPUs achieve this by having several hundred functional units. These are usually not referred to as "cores" in the same sense as the multi-core CPUs. McCool [10] defined a core as a processing element with an independent flow of control. The functional units on a GPU do not have an independent flow of control. They are grouped together in a SIMD manner, so that the functional units in one group has to perform the same instruction in a clock cycle. These SIMD groups can thus be referred to as cores with the above definition. Most current GPUs also allow branching to avoid executing unnecessary instructions. If the code flow is convergent in a SIMD group, no special treatment is needed, and only the instructions needed are executed. However, if the code flow is divergent in a SIMD group, the GPU will run all the instructions, and no time is saved. The GPU use masking techniques to ensure the correct answer is produced by each processing element.

The GPU originally had a fixed pipeline that was created for fast rendering. The introduction of programmable shaders in the pipeline enabled the possibility of running programs on the GPU. Programming shaders to solve arbitrary problems requires deep knowledge about the pipeline of the GPUs to be able to transform the problem into a rendering problem. General-purpose GPU (GPGPU) programming languages and frameworks like CUDA and OpenCL were created to ease the programming of the GPU.

2.3 Marching Cubes on the GPU

Several parallel multi-chip, multi-threaded and multi-core CPU implementation have been proposed in the literature. A survey of these implementations was done by Newman and Yi [12]. Pascucci [13] accelerated a variation of MC, called the Marching Tetrahedra algorithm, by creating a quad per tetrahedra and letting a vertex shader program calculate the vertices on the GPU. Klein *et al.* [8] moved the calculations to the fragment shader by coding the data in textures. Reck *et al.* [14] improved on these methods by removing empty cubes on the CPU using an interval tree. Goetz *et al.* [6] used a vertex shader program on the MC algorithm and Johansson and Carr [7] improved on this by removing empty cubes using a similar method to that of Reck *et al.* [14].

Each cube in the voxel grid can be processed independently of the other cubes. The main challenge with running MC on the GPU is how to store the triangles of each cube in memory in parallel. In the serial implementation, this is simple by using a stack and adding the triangles to the stack as each cube is processed. Two things are needed to store the triangle data in parallel on the GPU: 1) The number of triangles produced, so that the proper amount of memory can be allocated. 2) A unique index for each cube, so that the cubes can store their triangles in separate places.

It is not possible to assume that all the cubes produce triangles, because the device memory is too small for allocating memory for the maximum number of triangles. For most medical datasets only a small amount of the cubes actually produce triangles.

NVIDIA has included a CUDA and an OpenCL GPU implementation of MC in their SDKs. Their implementations use the parallel algorithm prefix sum to calculate the sum of triangles and storage index to each cube. Stream compaction is performed on the prefix sum result to avoid processing cubes that do not produce any triangles. Aksnes and Hesland [1] used this method to run MC on large porous rock datasets. Dyken *et al.* [5] used a data structure called Histogram Pyramid (HP), originally presented by Ziegler *et al.* [18], and implemented a vertex shader, geometry shader and CUDA version of it. This method was shown to be slightly better than NVIDIAs prefix sum scan approach in cases where the dataset was sparse. More recently, Ciznicki *et al.* [4] presented a multi GPU implementation of Marching Tetrahedra using the Histogram Pyramid data structure.

2.4 Our contribution

This paper builds on the work by Dyken *et al.* [5] and is a continuation of our previous workshop paper [16]. Similar to Dyken *et al.* and Ciznicki *et al.* we have used Histogram Pyramids. The main contributions of our paper are:

- OpenCL is used, which enables execution on GPUs from different vendors. This differs from CUDA which is for NVIDIA GPUs only.
- Dyken *et al.* packed the 3D data in 2D textures and used 2D Histogram Pyramids.

In this work, we extend Histogram Pyramids to 3D. This increase cache locality and removes the need for address translations.

- An efficient storage scheme for Histogram Pyramids is presented. This scheme reduces the memory consumption allowing larger volumes to be processed with a single pass.

3 Methodology

This section starts with explaining and extending the data structure Histogram Pyramids to 3D. Finally, a detailed description of our implementation is presented.

3.1 Histogram Pyramids

The Histogram Pyramid (HP) data structure consists of a stack of textures. These textures can be either 2D or 3D. Figure 2 illustrates the construction of a HP in 2D. Let's say we are interested in the white pixels in the 4x4 image to the left. The base level of the HP is created as a 2D texture of the same size as the original image. An element in the base level will have a 0 if the corresponding pixel in the image is black and 1 if it is white. The next level of the HP is created by summing 2x2 cells and storing it in another 2D texture with the size halved in both dimensions. This procedure is repeated until a 1x1 texture is left and no more reduction can be performed. The sum in the top level is the sum of the 1s in the base level. This sum can be used to allocate memory.

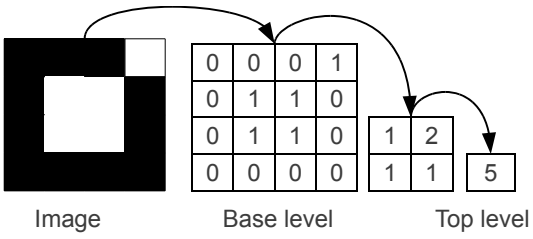

Figure 2: Construction of a HP

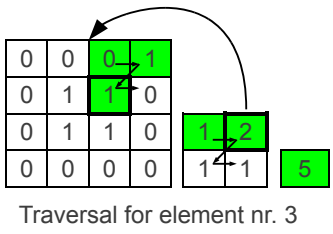

Figure 3: Traversal of a HP

To retrieve a specific white pixel with a given index the HP is traversed as shown in Figure 3. The traversal starts with the second level. The elements are scanned in a Z pattern as shown in the figure. When the sum of all scanned elements + the current element are above or equal to the index of the requested pixel, the procedure jumps to the next level and scans the 2x2 cell that corresponds to the last element in the scan. This process is repeated until the base level is reached. The final element is the one requested.

MC is a 3D algorithm, hence the HP has to be designed so that it can be used for 3D. Dyken *et al.* [5] used a flat 3D layout to pack the volume onto a 2D texture and then used the HP in the same way as in the example above. The drawback of using the flat

3D layout, is that it requires some extra computation for the address translation from 3D to 2D. It is also possible to extend the HP to 3D, as shown in Figure 4, by using 3D textures in OpenCL. This was done in our implementation. Writing to a 3D texture requires an OpenCL extension called *cl_khr_3d_image_writes*. AMD currently supports this extension, but NVIDIA does not. Due to this restriction on NVIDIA GPUs, a separate version was created for NVIDIA devices. This version uses regular buffers instead of textures, and Morton codes [11] to facilitate 3D caching. This is a bit slower than the 3D texture version used on AMD devices, due to a decrease in cache hits and additional processing. In a 3D HP, summing and traversal is performed on 2x2x2 cells instead of 2x2 cells. Also, in the example above, the element in the base level had values of 0 or 1. In MC, each cube can produce between 0 and 5 triangles, where each triangle consists of 3 vertices each. Thus, each element in the base level of the HP for MC will have a number between 0 and 5, depending on how many triangles each cube produces.

Most modern GPUs have support for several texture formats of different data types. These include 8, 16 and 32 bit integer data types. In our 3D Histogram Pyramid implementation, each level is stored as one texture. This enables the use of different texture formats for each HP level. 8 bit storage format for each pixel is sufficient for the base level because each cube can only produce a maximum of 5 triangles. And the maximum for the second level is 8 * 5 = 40, which is also within the 8 bit limit. The next three levels can use 16 bit data types. By using these different texture formats the memory required for the Histogram Pyramid is reduced significantly and this allows faster processing and larger volumes to be processed in a single pass. This differs from the implementation of Dyken *et al.* [5] where all levels are packed in a single texture with the 32 bit format.

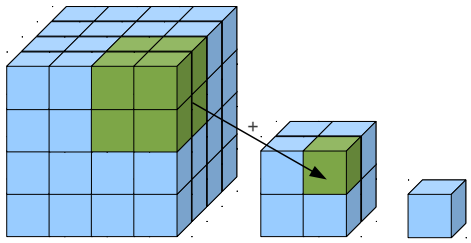

Figure 4: 3D Histogram Pyramid

3.2 Implementation

Our MC implementation consists of 6 main steps as depicted in Figure 5. It is based on the original MC algorithm by Lorensen and Cline [9].

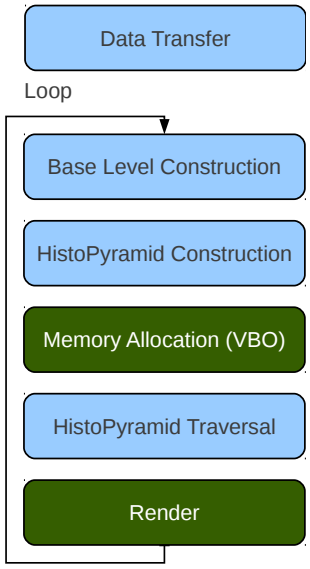

Figure 5: Block diagram of our MC implementation

The bright/blue steps are performed using OpenCL, while the dark/green steps are performed using OpenGL. Synchronization is necessary for each switch between the two APIs. In this section, each step will be explained in detail.

Data Transfer. The first step is to transfer the dataset to the device using the fast PCI express bus. The dataset is stored as a 3D texture on the device. Most GPUs today have a separate texture cache which allow for fast retrieval. This step is only performed once.

Base Level Construction. In this step, the base level of the HP is created. Recall that the base level contains the number of triangles necessary for each cube. In medical imaging the scalar field is usually constant. However, the iso-value can be changed, which can change the number of triangles needed. All levels of the HP are stored in textures on the GPU. This reduces the impact of the HP construction and traversal steps significantly, with cache hits over 90%. A NDRange kernel is run with the size of the dataset and the base level. This kernel creates an 8 bit cube index, where each bit represents a corner in the cube. If the corner has a value in the original dataset which is below the iso-value, that bit is set to 1, and if it is above it is set to 0. With this 8 bit index, we can look up in a table how many triangles are needed for this specific cube and store it in the base level.

Histogram Pyramid Construction. The entire HP can be constructed by a set of NDRange kernel calls in OpenCL. The number of calls needed is log_2 of the size of the base level. If the base level has the size 256x256x256, a NDRange kernel of size 128x128x128 is executed to fill the next level which has the size 128x128x128. In the next step, a NDRange kernel of size 64x64x64 is executed and so on until the 1x1x1 level is reached. This kernel simply sum all the elements in a 2x2x2 cell in the previous level and stores the sum in the current level.

Memory Allocation. When the HP has been created, the sum of triangles is retrieved from the 1x1x1 top level of the HP, and sent to the CPU via the PCI-express. This sum is used to allocate memory on the graphics card for all the vertices and normals needed to store the surface. The memory is allocated in the form of a vertex buffer object (VBO) in OpenGL. After the memory has been allocated, OpenGL has to synchronize and transfer control back to OpenCL.

Histogram Pyramid Traversal. The memory is filled with the output of the MC algo-

rithm by running a NDRange kernel of the same size as the total sum of triangles retrieved in the previous step. This kernel implements the HP Traversal procedure from section 3.1 using the global index as the triangle element index. When the 3D coordinate of the triangle's cube is located, the exact coordinates and normal of each vertex in the triangle can be calculated. The cube index is reused to look up in a table the cube edges that this triangle should have its vertices on. Linear interpolation is performed on each vertex with the data from the original dataset for each corner. The normals are calculated using forward differences as shown by Lorensen and Cline [9]. Finally, the vertices and normals are stored in the VBO made in the previous step.

The total sum of triangles is not necessarily dividable by a multiple of the units of execution (32 on NVIDIA, 64 on AMD). This sum must also be dividable on the number of work-items in each work-group. If the number of work-items in a work-group is not a multiple of the units of execution, several threads in each work-group will be idle on the GPU. To deal with this, we add a set of dummy work-items so that the total number of work-items is dividable by 64 and set the work-group size to 64. This way, only work-items in the last work-group will be idle.

Render. When the traversal step has created all the vertex and normal data, the CPU is notified and the control is transfered to OpenGL, which then renders the contents of the VBO on the screen. If the iso-value or scalar field has changed, all of these steps can be repeated to create a new surface or the program can continue rendering the next frame using the same surface.

4 Results and Discussion

The performance of our implementation was assessed by measuring the average number of frames per second (FPS) and execution time on the graphics device. For comparison, we also tested the OpenGL shader implementation of Dyken *et al.* [5] that also uses Histogram Pyramids (HPs). We call this implementation *HPMC Shader*. The dataset used for the measurements was a rotational angiography scan of a head with an aneurysm taken from [17] and depicted in Figure 7. The algorithm was run with a constant iso-value of 0.2 for 5 different sizes of the original dataset with size 512^3. Each dataset was processed on three different GPUs: AMD Radeon HD 5870 with 1GB memory, NVIDIA GTX 470 with 1280MB memory and NVIDIA Tesla C2070 with 6GB memory. The OpenCL implementations used were AMD APP 2.6 and NVIDIA CUDA 4.2. The FPS and execution time was measured in the rendering loop and gathered in Table 1 and 2. The two implementations were not able to process the largest dataset on all devices. This is indicated with a - in the tables. The execution time for each step in our implementation was also measured and is depicted in Figure 6.

Size	AMD HD5870	NVIDIA GTX 470	NVIDIA Tesla C2070
1024^3	-	-	-
512^3	3324 ms (0.3 FPS)	526 ms (1.9 FPS)	65 ms (15 FPS)
256^3	5 ms (223 FPS)	7 ms (149 FPS)	7 ms (140 FPS)
128^3	3 ms (394 FPS)	2 ms (556 FPS)	3 ms (389 FPS)
64^3	2 ms (519 FPS)	2 ms (524 FPS)	1 ms (1154 FPS)

Table 1: Performance of the shader implementation of Dyken *et al.* [5]. Their implementation was not able to process the largest dataset (1024^3) on any of the devices.

Size	AMD HD5870	NVIDIA GTX 470	NVIDIA Tesla C2070
1024^3	-	-	1279 ms (0.8 FPS)
512^3	34 ms (30 FPS)	127 ms (7.9 FPS)	136 ms (7.3 FPS)
256^3	10 ms (105 FPS)	19 ms (52 FPS)	19 ms (50 FPS)
128^3	4 ms (223 FPS)	4 ms (241 FPS)	3 ms (276 FPS)
64^3	3 ms (319 FPS)	2 ms (524 FPS)	2 ms (498 FPS)

Table 2: Performance of our OpenCL implementation

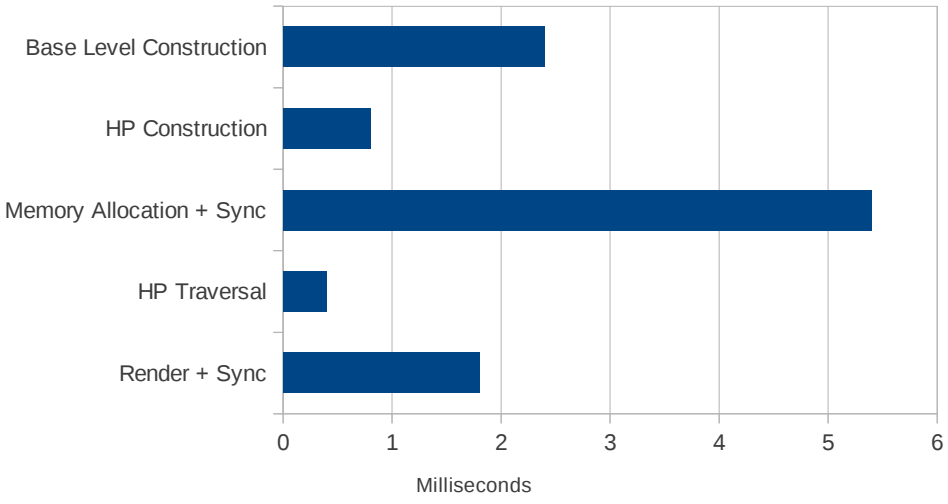

Figure 6: Execution time of each step of our implementation when run on the 256^3 dataset using an AMD Radeon HD5870

4.1 OpenCL-OpenGL Synchronization

Comparing the performances of the HPMC Shader versus our implementation, Tables 1 and 2 show that the HPMC Shader implementation is almost twice as fast for the three smallest datasets. With the profiling tool gDEBugger, it was discovered that the synchronization between OpenCL and OpenGL is very time consuming as shown in Figure 6.

The total time used on synchronizing between these two APIs was measured to be from 2 to 20 ms. This makes the GPU stay idle for the major part of the total execution time for the smallest datasets. It is thus believed that this synchronization cost is the reason for the HPMC Shader implementation being faster than our OpenCL implementation on the smaller datasets.

The synchronization cost is a major problem with the OpenCL-OpenGL interoperability. A possible solution to this problem has been proposed by The Khronos Group through an extension in both APIs, allowing them to share synchronization objects which should enable more efficient synchronization. The extensions are called *GL_ARB_cl_event* and *cl_khr_gl_event*. At the time of writing none of these extensions are implemented by any of the GPU vendors.

4.2 Histogram Pyramid Memory Usage

The HPMC Shader implementation was not able to process the 1024^3 dataset on any of the GPUs. This is due to the fact that this implementation requires over 5GB just to store the Histogram Pyramid. Our implementation, on the other hand, use a compressed storage format that use only a little over 1GB to store the HP for the large 1024^3 dataset. The HPMC Shader implementation is able to process the 512^3 dataset. However, there is not enough memory on the HD5870 and GTX 470 GPUs to store the entire HP. This forces the application to move data back and forth from the host to the GPU, resulting in a large drop in speed as can be seen in Table 1. Our implementation use only 148MB to store the 512^3 dataset and can thus extract and visualize the surface quickly on all three GPUs.

The excess use of memory is due to the way HPMC Shader stores the Histogram Pyramid. HPMC Shader stores the HP in a 2D texture with all the HP levels as Mipmap levels. The disadvantage of this is that the same texture format has to be used for all levels. Our implementation has a single texture for each level, enabling the use of 8 and 16 bit texture formats when it is sufficient.

N is the size of the HP in each dimension. N has to be larger than the dataset size in each dimension for it to fit, and it has to be a power of 2. The total memory requirements of the HP with our method is calculated in bytes as shown in equation 1. The first two levels use only one byte per voxel, while levels 3 to 5 use 2 bytes and the rest use 4 bytes.

$$N^3 + \left(\frac{N}{2}\right)^3 + 2\left(\frac{N}{4}\right)^3 + 2\left(\frac{N}{8}\right)^3 + 2\left(\frac{N}{16}\right)^3 + 4\sum_{i=5}^{log_2(N)}\left(\frac{N}{2^i}\right)^3 \quad (1)$$

HPMC Shader has to use 32 bit storage for all levels and this leads to a much higher memory usage. The HPMC Shader uses 4 channels in a 2D texture which results in $4 * 4 = 16$ bytes per pixel. The size of the texture M^2 is chosen so that the entire dataset fits into the top level, hence $4M^2 >= N^3$. As with N, M also has to be a power of 2.

The memory requirements of the HPMC Shader implementation is given by equation 2, and is always larger than the HP size of our OpenCL implementation.

$$16 \sum_{i=0}^{log_2(M)} \left(\frac{M}{2^i}\right)^2 \tag{2}$$

Table 3 shows the memory requirements for both methods for datasets of different sizes.

N	M	HP Size of HPMC Shader	HP Size of proposed
2048	65536	**87 381**	**9 509**
1024	16384	**5 461**	1 188
512	8192	1 365	148
256	2048	85	18
128	1024	21	2
64	256	1	< 1

Table 3: Storage size in MBs of Histogram Pyramids of different sizes for both methods. Critical HP sizes are marked in bold/red.

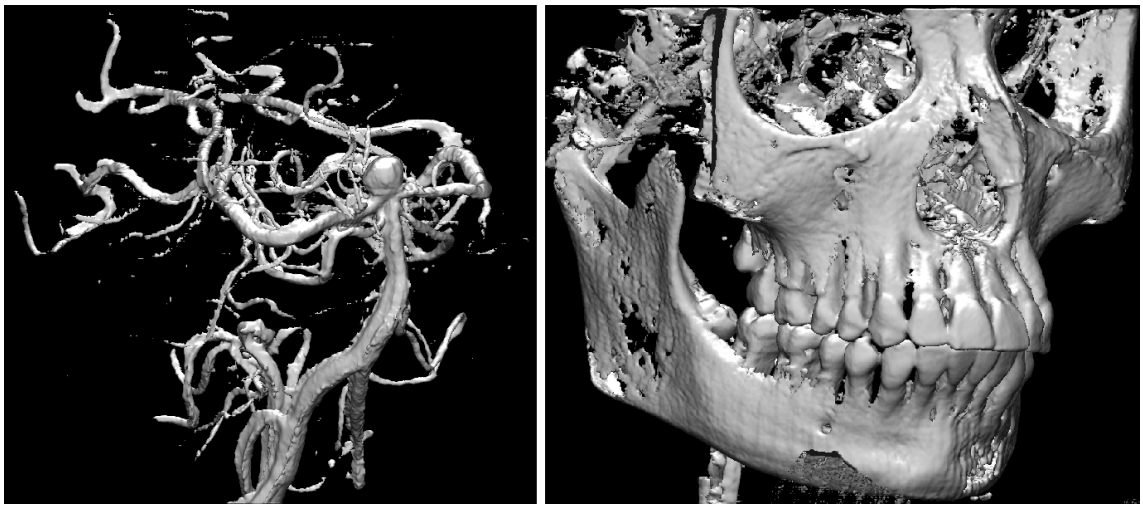

Figure 7: Two rendered results of the MC implementation

4.3 Other GPU Implementations

NVIDIA's CUDA and OpenCL implementations that use prefix sum, were also tested on these datasets using an NVIDIA Geforce GTX460 with 2GB device memory. Their OpenCL implementation was excluded from the comparisons above because the largest dataset it could process was 64^3, running with an average FPS of 452 and memory usage of 23 MB. Also, NVIDIA's CUDA implementation failed for the largest dataset due to memory exhaustion.

4.4 Improvements on other platforms

The improvement of storing data in a more efficient format should be applicable to OpenGL shader and CUDA implementations which would allow larger volumes to be processed with the same speeds as our OpenCL implementation. Due to the expensive OpenCL-OpenGL synchronization, an OpenGL shader implementation using this efficient storage format would probably be faster than our OpenCL implementation.

5 Conclusions

In this paper, an OpenCL implementation of Marching Cubes (MC) that uses Histogram Pyramids was presented. Our novel implementation is able to extract and visualize surfaces from large datasets (512^3 and 1024^3) faster than other implementations. This is achieved by using an efficient storage scheme that significantly reduces the memory usage of the Histogram Pyramid data structure. Our results revealed that the implementation lose some performance due to an expensive synchronization costs in the OpenCL-OpenGL interoperability. This issue will hopefully be overcome in future GPUs.

The source code of this implementation is available online[1].

Acknowledgments

Great thanks goes to the people of Anne C. Elster's HPC-Lab at NTNU (`http://research.idi.ntnu.no/hpc-lab`) for all their assistance. The author would also like to convey thanks to the Dept. of Computer and Information Science at NTNU, NVIDIA and AMD. Without their hardware contributions to the HPC-Lab, this project would not have been possible.

References

[1] E. O. Aksnes, H. Hesland, and A. C. Elster. GPU Techniques for Porous Rock Visualization. Technical report, Norwegian University of Science andTechnology, January 2009. IDI TR no. 02/10, ISSN 1503-416X.

[2] A. Aqrawi and A. Elster. Bandwidth reduction through multithreaded compression of seismic images. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1730 –1739, May 2011.

[1] http://github.com/smistad/GPU-Marching-Cubes

[3] E. V. Chernyaev. Marching Cubes 33: Construction of Topologically Correct Isosurfaces. Technical report, CERN, 1995.

[4] M. CiÅijnicki, M. Kierzynka, K. Kurowski, B. Ludwiczak, K. NapieraÅĆa, and J. PalczyÅĎski. Efficient isosurface extraction using marching tetrahedra and histogram pyramids on multiple gpus. *Parallel Processing and Applied Mathematics*, 7204:343–352, 2012.

[5] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel. High-speed Marching Cubes using HistoPyramids. *Computer Graphics Forum*, 27(8):2028–2039, Dec. 2008.

[6] F. Goetz, T. Junklewitz, and G. Domik. Real-time marching cubes on the vertex shader. In *Proceedings of Eurographics*, volume 2005, page 2, 2005.

[7] G. Johansson and H. Carr. Accelerating marching cubes with graphics hardware. *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research - CASCON '06*, page 39, 2006.

[8] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.*, pages 186–195.

[9] 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.

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

[11] G. M. Morton. A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing. Technical report, IBM Ltd., Ottowa, Canada, 1966.

[12] T. Newman and H. Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30(5):854–879, Oct. 2006.

[13] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2004.

[14] F. Reck, C. Dachsbacher, R. Grosso, G. Greiner, and M. Stamminger. Realtime isosurface extraction with graphics hardware. In *Proc. Eurographics*, pages 1–4, 2004.

[15] E. Smistad, A. C. Elster, and F. Lindseth. Real-time gradient vector flow on gpus using opencl. *Journal of Real-Time Image Processing*, pages 1–8. 10.1007/s11554-012-0257-6.

[16] E. Smistad, A. C. Elster, and F. Lindseth. Fast Surface Extraction and Visualization of Medical Images using OpenCL and GPUs. In *The Joint Workshop*

on High Performance and Distributed Computing for Medical Imaging 2011, 2011. http://idi.ntnu.no/~smistad/papers/Fast_Surface_Extraction_and_Visualization_of_Medical_Images_using_OpenCL_and_GPUs/article.pdf.

[17] Volvis. www.volvis.org. Accessed 20 Feb. 2011.

[18] G. Ziegler, A. Tevs, C. Theobalt, and H. Seidel. On-the-fly point clouds through histogram pyramids. In *Vision, modeling, and visualization 2006: proceedings, November 22-24, 2006, Aachen, Germany*, page 137. IOS Press, 2006.

A2

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

Authors

Erik Smistad, Anne C. Elster and Frank Lindseth

Published in

Norsk informatikkonferanse 2012, pages 129-140.

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

Erik Smistad[1], Anne C. Elster[1], Frank Lindseth[1,2]

1) Norwegian University of Science and Technology

2) SINTEF Medical Technology. Trondheim, Norway

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. 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 exists 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.

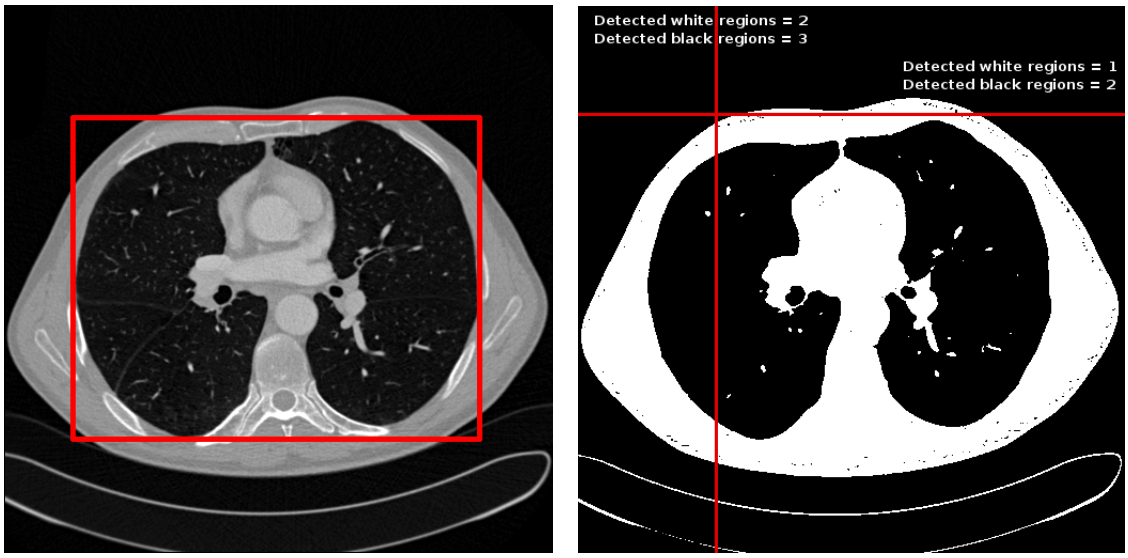

Figure 2: Left: A typical CT image of the lungs. The border indicates the minimal rectangular area that includes the lungs. **Right:** Thresholded CT image and two scan lines in the x and y directions. For each scan line the number of detected white and black regions are listed.

2 Methodology

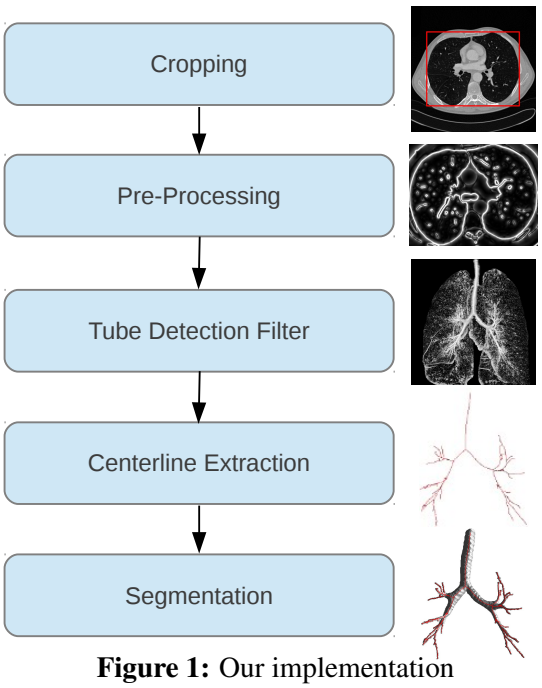

Figure 1: Our implementation

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

2.1 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, is recorded for each slice in the function CALCULATEL. A pixel on the scan line is categorized as black or white based on the intensity in the CT volume using a threshold $T_{\text{HU}} = -150HU$. 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_c. Figure 2 shows a CT image thresholded using T_{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_{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 FINDCROPBORDERS locates the cropping borders (c_1 and 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 CALCULATEL function. This creates many threads and is ideal for GPU execution. The 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 $L_{\text{min}} = 128$ and $T_c = 30$ were chosen through experimentation.

2.2 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

Algorithm 1 Cropping

```
function CROP(volume)
    L ← CALCULATEL(volume, x)
    x1, x2 ← FINDCROPBORDERS(L, x)
    L ← CALCULATEL(volume, y)
    y1, y2 ← FINDCROPBORDERS(L, y)
    L ← CALCULATEL(volume, z)
    z1, z2 ← FINDCROPBORDERS(L, z)
    crop volume according to x1,x2,y1,y2,z1 and z2
return volume
end function

function FINDCROPBORDERS(L, direction)
    size ← volume.direction.size
    c1 ← −1, c2 ← −1
    if direction = z then
        s ← size/2
        a ← −1
    else
        s ← 0
        a ← 1
    end if
    while (c1 = −1 or c2 = −1) and s < size do
        if aLs > aLmin then
            c1 ← s
        end if
        if aLsize−1−s > aLmin then
            c2 ← size −1 − s
        end if
        s ← s + 1
    end while
return c1, c2
end function

function CALCULATEL(volume, direction)
    for each slice s in direction do
        Ls ← 0
        for each scan line do
            for each scan line element do
                if volume[position] > THU then
                    if Wc = Tc then
                        Wd ← Wd + 1
                        Bc ← 0
                    end if
                    Wc ← Wc + 1
                else
                    if Bc = Tc then
                        Bd ← Bd + 1
                        Wc ← 0
                    end if
                    Bc ← Bc + 1
                end if
            end for
        end for
        if Wd > 1 and Bd > 1 then
            Ls ← Ls + 1
        end if
    end for
return L
end function
```

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.

$$\vec{V}^n(\vec{v}) = \begin{cases} \frac{\vec{V}(\vec{v})}{|\vec{V}(\vec{v})|} & \text{if } |\vec{V}(\vec{v})| \geq F_{\text{max}} \\ \frac{\vec{V}(\vec{v})}{F_{\text{max}}} & \text{else} \end{cases} \tag{1}$$

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 exists 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 $E(\vec{V})$:

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

where $\vec{V}_0$ is the initial gradient vector field and μ 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:

$$\mu\nabla^2\vec{V} - (\vec{V} - \vec{V}_0)|\vec{V}_0|^2 = \vec{0} \tag{3}$$

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 Lapacian $\nabla^2\vec{V}(\vec{v})$ is approximated using a 7 point stencil finite difference scheme.

Algorithm 2 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

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.

2.3 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 corresponds 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 $\alpha = \frac{2\pi i}{N}$ 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 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.

$$T(\vec{v}, r, N) = \frac{1}{N} \sum_{i=0}^{N-1} \vec{V}(\vec{v} + r\vec{d_i}) \cdot -\vec{d_i} \tag{4}$$

2.4 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 $\vec{t}_i$ 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.

2.5 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.

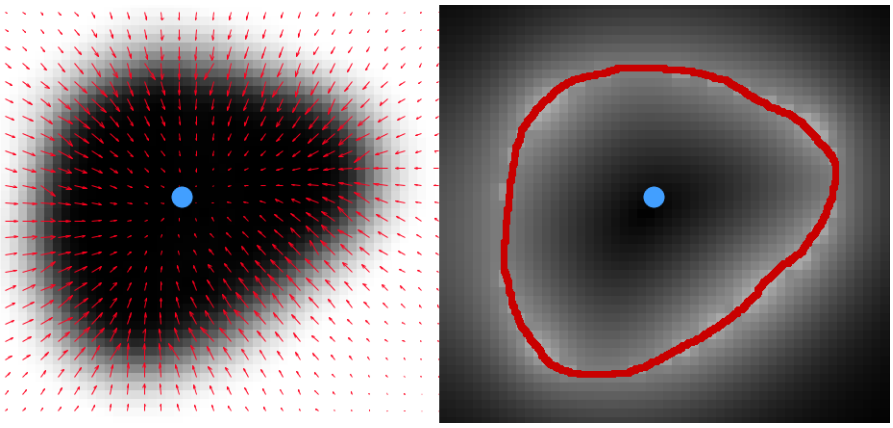

Figure 3: Inverse Gradient Flow Tracking Segmentation. **Left:** The GVF vector field superimposed on the cross section of a tube. The dot in the middle is the dilated centerline. From this centerline, the segmentation is grown in the inverse direction of the vectors as long as the length of the vectors increase. **Right:** The magnitude of the GVF vector field. The border shows the final segmentation.

2.6 Texture Cache Optimizations

Most modern GPUs have a separate texture cache. These texture caches exists 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.

2.7 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 trilin-

ear 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.

2.8 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 Hostpital 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

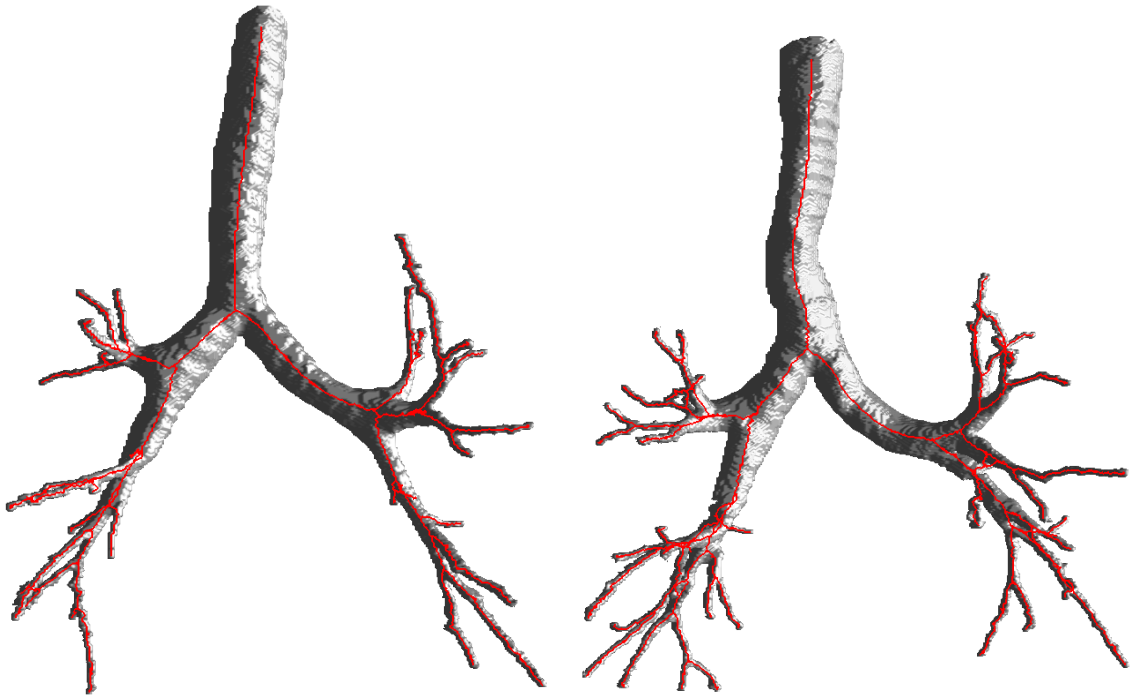

Figure 4: Segmentation and centerline for two of the patients.

measured in MBs for both the original and cropped volume. Peak memory usage occurs in the GVF step.

Dataset	GPU Runtime	Multi-threaded CPU Runtime
Patient 1	31 secs	12 min 52 secs
Patient 2	31 secs	14 min 43 secs
Patient 3	26 secs	10 min 44 secs
Patient 4	27 secs	14 min 4 secs
Patient 5	20 secs	10 min 5 secs
Patient 6	38 secs	17 min 25 secs

Table 1: Speed measurements

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

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.

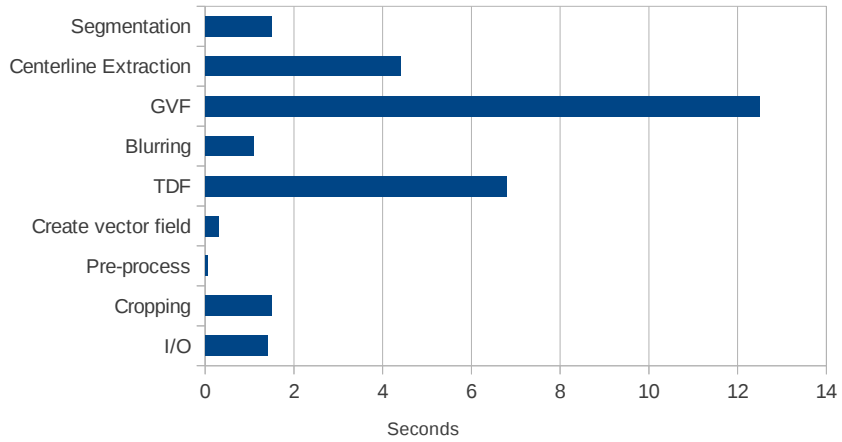

Figure 5: Measured runtime for each step of the implementation in seconds for the first dataset.

3.1 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_{min} variable has a reasonable value. If the value is to high, the upper part of 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 $L_{\text{min}} = 128$.

3.2 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 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.

3.3 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

[1] 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.

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

[3] 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.

[4] 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.

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

[6] D. Deguchi, M. Feuerstein, T. Kitasaka, Y. Suenaga, I. Ide, H. Murase, K. Imaizumi, Y. Hasegawa, and K. Mori. Real-time marker-free patient registration for electromagnetic navigated bronchoscopy: a phantom study. *International journal of computer assisted radiology and surgery*, 7(3):359–69, May 2012.

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

[8] Z. He and F. Kuester. GPU-Based Active Contour Segmentation Using Gradient Vector Flow. *Advances in Visual Computing*, pages 191–201, 2006.

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

[10] 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.

[11] 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.

[12] 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.

[13] E. Smistad, A. C. Elster, and F. Lindseth. Real-time gradient vector flow on GPUs using OpenCL. *Journal of Real-Time Image Processing*, 2012. DOI: 10.1007/s11554-012-0257-6.

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

A3

Real-time gradient vector flow on GPUs using OpenCL

Authors

Erik Smistad, Anne C. Elster and Frank Lindseth

Published in

Journal of Real-Time Image Processing, volume 10, issue 1, March 2015, pages 67-74.

Real-time gradient vector flow on GPUs using OpenCL

Erik Smistad[1], Anne C. Elster[1], Frank Lindseth[1,2]

1) Norwegian University of Science and Technology

2) SINTEF Medical Technology. Trondheim, Norway

Abstract

The Gradient Vector Flow (GVF) is a feature-preserving spatial diffusion of gradients. It is used extensively in several image segmentation and skeletonization algorithms. Calculating the GVF is slow as many iterations are needed to reach convergence. However, each pixel or voxel can be processed in parallel for each iteration. This makes GVF ideal for execution on Graphic Processing Units (GPUs). In this paper, we present a highly optimized parallel GPU implementation of GVF written in OpenCL. We have investigated memory access optimization for GPUs, such as using texture memory, shared memory and a compressed storage format. Our results show that this algorithm really benefits from using the texture memory and the compressed storage format on the GPU. Shared memory, on the other hand, makes the calculations slower with or without the other optimizations because of an increased kernel complexity and synchronization. With these optimizations our implementation can process 2D images of large sizes (512^2) in real-time and 3D images (256^3) using only a few seconds on modern GPUs.

1 Introduction

The Gradient Vector Flow (GVF) is a feature-preserving spatial diffusion of gradients. The GVF field is defined as the vector field $\vec{V}$, that minimizes the energy function E:

$$E(\vec{V}) = \int \mu |\nabla \vec{V}(\vec{x})|^2 + |\vec{V}_0(\vec{x})|^2 |\vec{V}(\vec{x}) - \vec{V}_0(\vec{x})|^2 d\vec{x} \quad (1)$$

where $\vec{V}_0$ is the initial vector field.

The GVF was introduced by Xu and Prince [11] as a new external force field for active contours (AC). Also known as snakes or deformable models, AC are curves that move in an image while trying to minimize its energy and are used extensively for boundary detection and segmentation. The traditional snake introduced by Kass *et al.* [8] has the

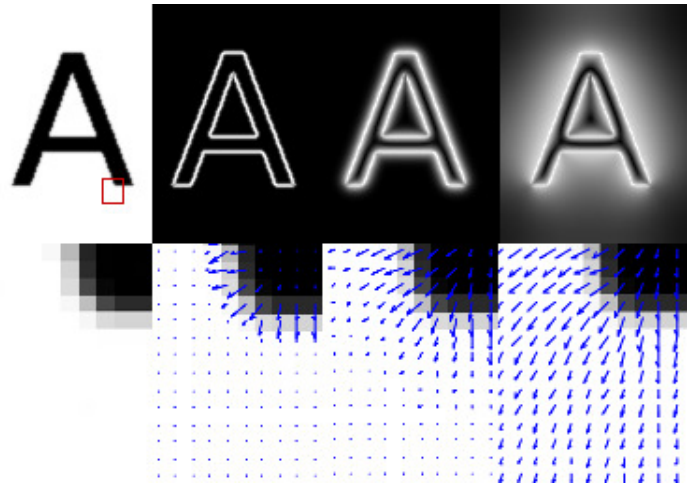

Figure 1: Example of GVF execution. From left to right: **Top:** 1) Smoothed image. 2) Magnitude of image gradients $\vec{V}_0$ 3) Magnitude of GVF after 10 iterations, 4) Magnitude of GVF after 400 iterations. **Bottom:** 1)Zoomed area of smoothed image 2, 3 and 4) Image gradients superimposed on zoomed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

problem of getting stuck in boundary concavities and low capture range. The GVF snake can deal with these problems.

Fig. 1 depicts the GVF when used for Active Contours. The initial image shown top-right is an image smoothed by convolution with a Gaussian. Next is the initial vector field $\vec{V}_0$ displayed using vector magnitude in the top row and the vectors in a zoomed region below. The next column shows the GVF field after 10 iterations of diffusion and the last column 400 iterations.

After its introduction, the GVF has been applied on several other image processing applications. Bauer and Bischof [2] developed a novel approach to use the GVF as a replacement for the scale-space framework in Hessian based tube detection. Hassouna and Farag [6] and Bauer and Bischof [3] used the GVF to extract skeletons from objects. Ray and Acton [10] used GVF to track leukocytes from intravital video microscopy. Guo and Lu [4] argued that GVF combined with Mutual Information can improve multi-modal image registration.

Xu and Prince [11] showed that the GVF field can be found by solving the Euler equation:

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

This is done by treating the vector field $\vec{V}$ as a function of time. Calculating the GVF field serially using this numerical approach is slow due to the need for many iterations to reach convergence. However, since each pixel is calculated independently of the other

pixels, each pixel can be processed in parallel with the exact same instructions for each iteration. This data parallelism makes the GVF ideal for running on Graphic Processing Units (GPUs). GPUs enable execution of the same instructions on many different data elements in parallel.

He and Kuester [7] 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. Performance result for only one NVIDIA GPU was presented. Also, Han *et al.* [5] proposed another serial numerical scheme for GVF using a multigrid method. Their results showed significant improvement in speed.

In this paper, we present an optimized parallel GVF implementation written in OpenCL. OpenCL is a new cross-platform framework for writing applications that can run on heterogeneous systems. In contrast to the work of He and Kuester [7], we investigate three different memory optimization techniques for GPUs instead of just using the texture memory. We also discuss 3-dimensional GVF and show results for both GPUs and multi-core CPUs from different manufacturers.

In the next section, we show how GVF can be implemented in parallel and note that the algorithm is memory intensive. We also present three memory usage optimizations for GPUs: texture memory, shared memory and a 16-bit floating point data type for storage. Section 3 presents performance results for each optimization in terms of both speed and memory usage. An analysis of the accuracy of the 16-bit floating point data type is also conducted. Section 4 provides a discussion of the presented results and the last section conclusions.

2 GPU Implementation

The parallel version of the numerical implementation of GVF by Xu and Prince [11] is given in Alg. 1 and for 3D in Alg. 2. The Laplacian $\nabla^2\vec{V}(\vec{x})$ is calculated using a finite difference method. On the boundaries of the image, some of the neighboring points required to calculate the Laplacian, will not exist. This can be solved by expanding the image with 1 pixel in all directions and have the same vector on the border as the third outermost pixel as depicted in Fig. 2. The gradient at the original border will then be 0. In practice, this is done by swapping the x, y or z components in the read address to 2 if it is 0 and to M-2 if it is M, where M is the size of that dimension.

From these pseudocodes, we can see that calculating the GVF needs 6 global memory accesses for 2D and 8 for 3D and about 20 ALU operations. The GVF computation is memory-bound because global memory access can have a latency of several hundred clock cycles while the ALU operations are only a small fraction of this [1]. Thus, in this project, we have focused on optimizing memory access and storage.

Algorithm 1 Parallel 2D Gradient Vector Flow

for all points $\vec{x}$ in parallel **do**
  laplacian $\leftarrow -4\vec{V}(\vec{x}) + \vec{V}(x+1,y) + \vec{V}(x-1,y) + \vec{V}(x,y+1) + \vec{V}(x,y-1)$
  $\vec{V}(\vec{x}) \leftarrow \vec{V}(\vec{x}) + \mu *$ laplacian $-(\vec{V}(\vec{x}) - \vec{V}_0(\vec{x}))|\vec{V}_0(\vec{x})|^2$
end for

Algorithm 2 Parallel 3D Gradient Vector Flow

for all points $\vec{x}$ in parallel **do**
  laplacian $\leftarrow -6\vec{V}(\vec{x}) + \vec{V}(x+1,y,z) + \vec{V}(x-1,y,z) + \vec{V}(x,y+1,z) + \vec{V}(x,y-1,z) + \vec{V}(x,y,z+1) + \vec{V}(x,y,z-1)$
  $\vec{V}(\vec{x}) \leftarrow \vec{V}(\vec{x}) + \mu *$ laplacian $-(\vec{V}(\vec{x}) - \vec{V}_0(\vec{x}))|\vec{V}_0(\vec{x})|^2$
end for

The unoptimized GPU implementation uses regular global memory with a 32-bit floating point storage format. In this article, we explore using texture memory as an alternative to global memory as well as shared memory in combination with texture and global memory. We also use a compressed 16-bit floating point storage format with each of these 4 memory combinations as an alternative to the default 32-bit format. Thus in total, we test 8 different memory optimization combinations on the GPU.

2.1 Texture memory

The default memory on GPUs is called global memory. This memory is not always cached (for AMD GPUs, global memory caching has to be enabled explicitly). When caching is enabled, it only has linear spatial locality. Most modern GPUs also have a separate texture memory system. Textures are 1D, 2D or 3D structures that can be addressed based on coordinates. GPUs have this texture memory system because GPUs are primarily used for 3D applications where textures are mapped to 3D objects to create a more realistic 3D scene. The textures are stored off-chip, but are cached and have spatial locality in multiple dimensions. When working with images and volumes this cache with 2D/3D

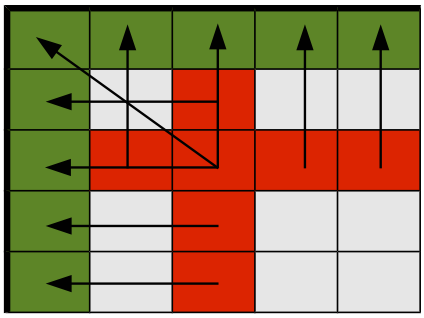

Figure 2: The top left corner of an image. The arrows indicate the values the boundary pixels use.

spatial locality can increase cache hits.

In the GVF calculations, there are two 2D/3D structures: the GVF field $\vec{V}$ and the initial vector field $\vec{V_0}$. We optimize our implementation by putting both of these data structures in textures. In OpenCL, textures are called images, and an image bound to a kernel can only be either read or written to. This is a limitation needed to assure cache coherency. Since the GVF vector field $\vec{V}$ has to be both read and written, we have used a double buffering mechanism.

By creating two textures for the GVF field $\vec{V}$, we use one texture for writing and one for reading, and after each iteration we swap the textures in the arguments to the kernel.

The handling of the boundaries as depicted in Fig. 2 can be handled automatically by the texture system using the addressing flag *ADDRESS_CLAMP_TO_EDGE*. With this flag set, pixels requested outside of the texture will use the pixel value closest to the request pixel.

In OpenCL, writing to a 3D texture is an optional extension called *cl_khr_3d_image_writes*. AMD supports it while NVIDIA does not. To support 3D GVF calculation on NVIDIA GPUs we created a separate kernel for these devices that uses global memory instead of textures for $\vec{V}$. Since global memory only have linear spatial cache locality, this is expected to reduce the number of cache hits.

2.2 Shared Memory

Shared memory is an on-chip memory that is shared among all work-items in a work-group. This memory is reported by GPU manufacturers to be more than 10 times faster than global memory which is off-chip ([1],[9]). It is generally beneficial to use shared memory when several work-items need the same data from global memory as their neighboring work-items.

When calculating the Laplacian, $\nabla^2\vec{V}(\vec{x})$, the data from the 4 (or 6 for 3D) closest neighboring pixels are needed. If N is the total number of pixels, there will be $5N$ global memory accesses to $\vec{V}$ in total because each pixel is requested 5 times. By using shared memory the number of global memory accesses can be reduced significantly.

The input image is divided into a set of work-groups as shown in Fig. 3. Each work-group process one tile of the input image and allocates a block of shared memory with the same size as the work-group. Each work-item in a work-group loads the pixel value from global memory and stores it in shared memory. As the work-items on the edges of the work-group will not have all their neighbor's data in shared memory, these work-items will not do calculations, only load data. These pixels are called the work-group's *frame* and are calculated by their neighboring work-groups. This causes some overhead in terms of redundant global memory accesses and work-items that are idle, but this is very small compared to the overhead of $5N$ global memory accesses to $\vec{V}$.

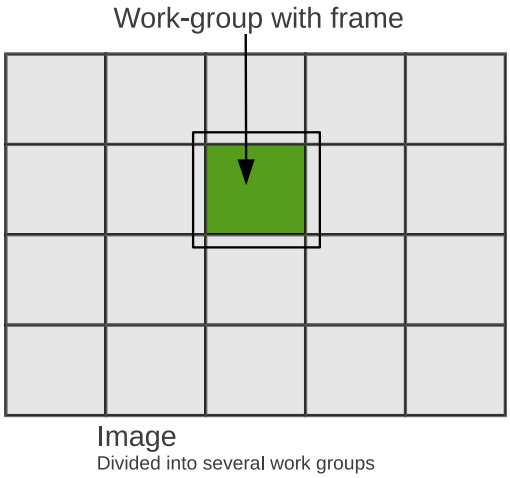

Figure 3: The input image is divided into several work-groups. The green/dark area is the part of the work-group that is calculated and the box around is the frame where only data is loaded.

Synchronization is necessary after writing to the shared memory, because all work-items in a work-group are not executed simultaneously (if a work-group is above a certain size). Work-items in a work-group can synchronize using a barrier in the shared memory.

The shared memory is divided into several banks usually 16 or 32. Memory requests to different banks can be served in parallel while memory requests to the same bank has to be serialized. Requests to the same bank in a clock cycle is called a bank conflict. These bank conflicts can be avoided with a sequential access pattern.

2.3 16-bit float storage format

Memory access can also be improved by reducing the number of bytes transferred from global memory to the chip. The most common way to store a floating point number on a computer, at present time, is by using 32 bits with the IEEE 754 standard. However, most GPUs also support a texture storage format called normalized 16-bit integer. With this format, the data is stored as 16-bit integers (shorts) in textures, but when it is requested, the texture fetch unit converts the 16-bit integer to a 32-bit floating point number with a normalized range from -1.0 to 1.0. This reduces accuracy, and may not be sufficient for all applications. Due to the reduced accuracy, the 16-bit storage format is made optional in our implementation. This storage format also halves the global memory usage, thus allowing much larger 3D volumes to reside completely in the GPU memory.

2.4 Work-group sizes

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. 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 many work-items that can exists in one work-group. On AMD GPUs, this limit is currently 256 and on NVIDIA up to 1024. In conjunction with shared memory, we want to maximize the size of the work-group minus the frame, given this limit. For 2D, this is maximum when the work-group is 16x16 and for 3D, 8x8x4. *E.g.* an image of size 512x512 would give 32x32 work-groups of size 16x16. Also, in OpenCL, each dimension has to be dividable by the work group-size. Thus, we pad the data so that the size is dividable by the highest possible work-group. This avoids idle threads and branch divergence while keeping a large work-group size.

3 Results

3.1 Speed

The speed of our implementation was measured using OpenCL timers. Fig. 4 shows the average execution time of one iteration on an image of size 512x512 with different combinations of global, texture and shared memory as well as 32-bit and 16-bit storage formats. This figure clearly shows that using the texture memory is faster than using regular global memory. Also, it illustrates that utilizing shared memory slows down the computation and that the 16-bit storage format is only beneficial when used together with the texture memory. Fig. 5 shows the average total execution time for images of different sizes for both 32 and 16-bit. In this figure, we notice that as the image size increases, the execution time difference also increases. All of these tests were run on an AMD Radeon HD5870 with 1GB of memory.

Tables 1 and 2 includes the average execution time measured both on 2D and 3D and on several different GPUs and multi-core CPUs. For the GPUs only the texture memory with the 16-bit storage format was used. For the CPUs the same version was used, but with 32-bit instead. From these two tables, we observe two things: 1) Execution on GPUs is much faster than on CPUs. 2) While NVIDIAs GPUs are comparable to AMDs GPUs on the 2D dataset in terms of speed, NVIDIAs GPUs perform much worse on the 3D dataset.

3.2 Memory usage

Global synchronization is needed in each iteration when calculating GVF in parallel. Because global synchronization is not possible inside a kernel, a double buffering mechanism is needed. This means that two copies of the vector field $\vec{V}$ is needed in addition to the initial vector field $\vec{V_0}$. The GPU implementation needs 2 vector components (x and y) * 3 vector fields * 32 bits = 24 bytes per pixel and 36 bytes per voxel for 3D volumes,

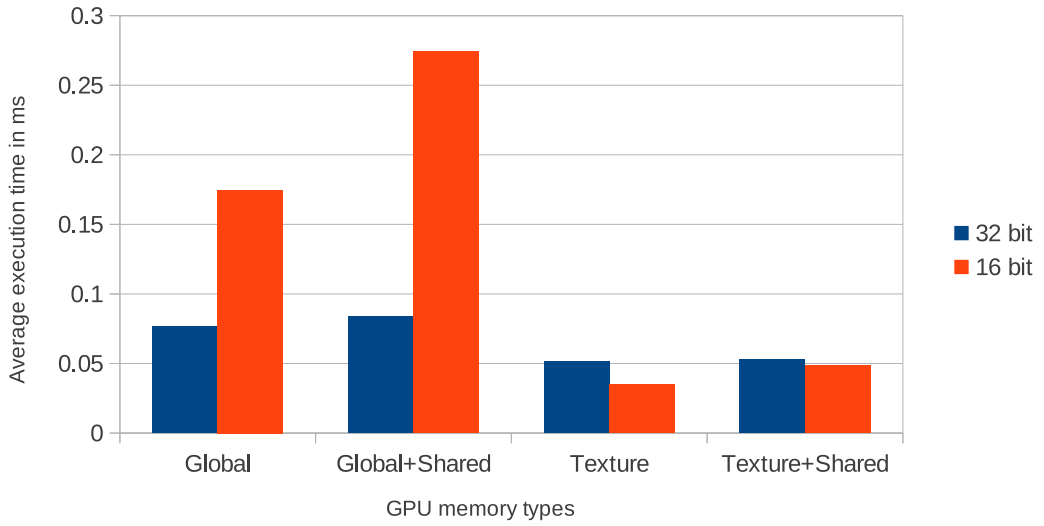

Figure 4: Average execution time for one iteration of a 512x512 image measured in milliseconds using OpenCL timers with both 32-bit and 16-bit storage format and different combinations of using regular global memory, texture memory and shared memory.

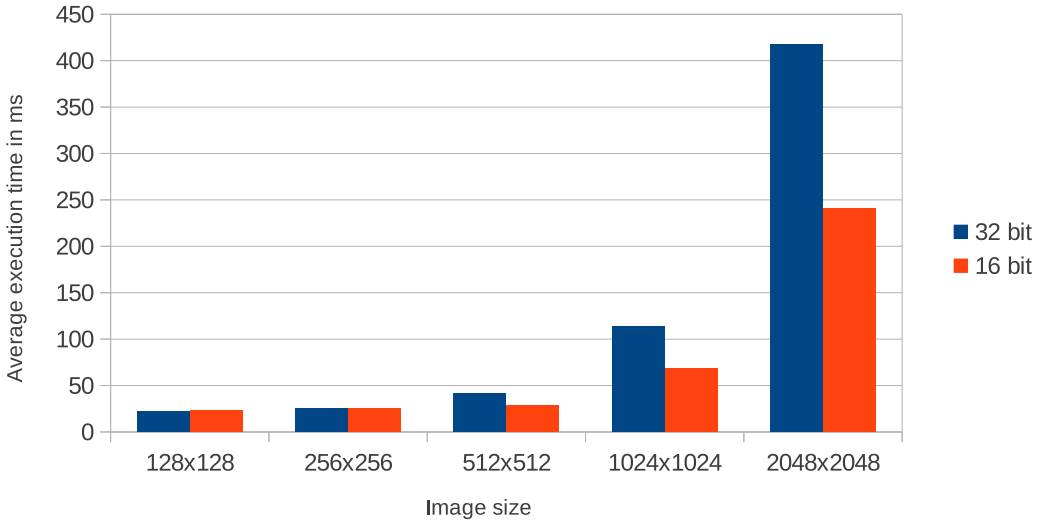

Figure 5: Average execution time for 512 iterations of images of different sizes using OpenCL timers with both 32 and 16-bit storage format. Note: The execution time difference between 32 and 16-bit storage format increases with the size of the images.

Processor	One iteration	All iterations
AMD 5870	0.035 ms	28 ms
AMD Mobile 5830	0.147 ms	77 ms
NVIDIA Quado FX5800	0.104 ms	66 ms
NVIDIA Tesla c2070	0.077 ms	41 ms
Intel i5 750	1.485 ms	851 ms
Intel i7 720	2.344 ms	1550 ms

Table 1: Average execution speeds for a 2D image of size 512x512 run for 512 iterations. The first 4 processors are GPUs, while the rest are multi-core CPUs.

Processor	One iteration	All iterations
AMD 5870	4.501 ms	1124 ms
AMD Mobile 5830	20.739 ms	5129 ms
NVIDIA Quadro FX5800	105.631 ms	27172 ms
NVIDIA Tesla c2070	27.989 ms	7151 ms
Intel i5 750	310.846 ms	92591 ms
Intel i7 720	378.876 ms	106747 ms

Table 2: Average execution speeds for a 3D volume of size 256^3 run for 256 iterations. The first 4 processors are GPUs, while the rest are multi-core CPUs.

because of the additional z component. On the other hand, when using a 16-bit float storage format, the memory usage is halved. As an example a volume of size 512^3 would consume 4.5 GB with the 32-bit data type and only 2.25 GB with the 16-bit data type. Figures 6 and 7 graphs the memory usage for this implementation for images and volumes for both 32- and 16-bit. Both figures depict the fact that the difference in memory usage increases as the dataset size increases.

3.3 Relative accuracy

We measured the relative error between a 32-bit and a 16-bit floating point data type on the final GVF vector field of the 512x512 image shown in Fig. 8. This was done by calculating the GVF for each data type on the same image. Relative error measures for both the magnitude and angle were calculated as shown in Eq. 3 and 4. From these equations, the average, variance, maximum and minimum were calculated for all pixels $\vec{x}$ and collected in table 3.

$$M_{error} = ||\vec{V}_{16bit}(\vec{x})| - |\vec{V}_{32bit}(\vec{x})|| \tag{3}$$

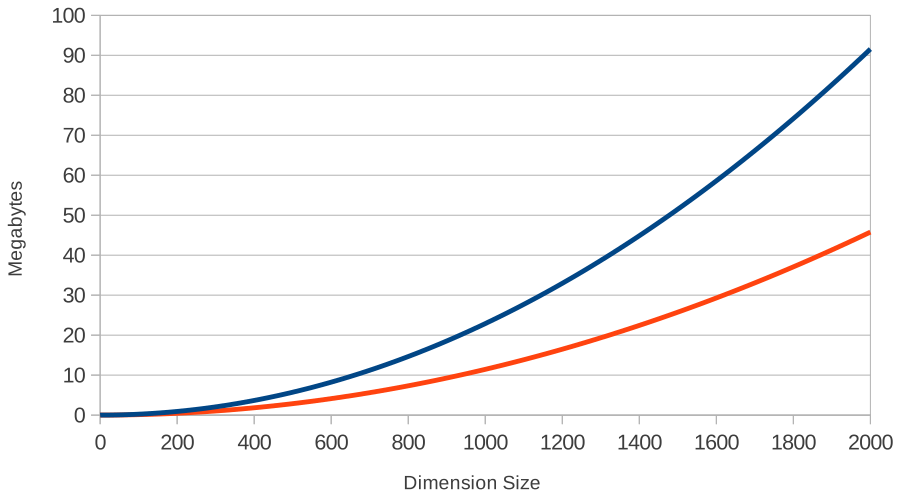

Figure 6: Memory usage in MBs versus size of image. Dimension size x on the x axis is the size of one of the dimensions so that total number of pixels is x^2

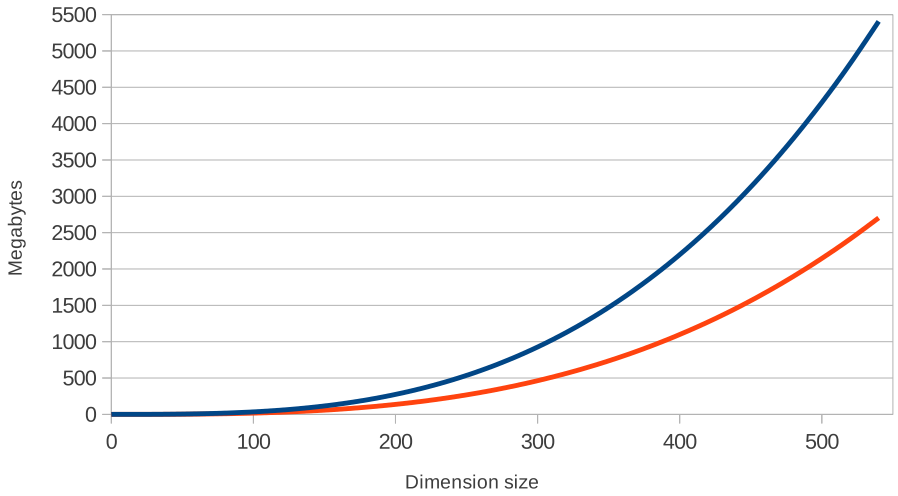

Figure 7: Memory usage in MBs versus size of volume. Dimension size x on the x axis is the size of one of the dimensions so that total number of voxels is x^3

	M_{error}	θ_{error}
Average	0.00078	0.55
Variance	4.29e-7	0.59
Maximum	0.00377	3.14
Minimum	8.92e-10	0

Table 3: Relative error of vector magnitude M and angle θ from 32-bit to 16-bit floating point storage format. Calculated using Eq. 3 and 4 on the image in Fig. 8. Angles are in radians

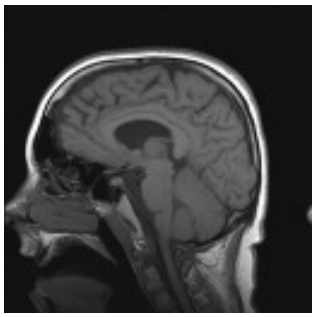

Figure 8: The 512x512 MRI Brain scan image the relative error measurements have been run on.

$$\theta_{error} = cos^{-1}\left(\frac{\vec{V}_{16bit}(\vec{x}) \cdot \vec{V}_{32bit}(\vec{x})}{|\vec{V}_{16bit}(\vec{x})||\vec{V}_{32bit}(\vec{x})|}\right) \tag{4}$$

4 Discussion

4.1 Speed

Fig. 4 shows that introducing shared memory actually makes the calculations slower. The reason for this is threefold: the code is more complex, requires explicit work-group synchronization and more threads/work-items are needed. Also, we notice that using the texture memory on the GPU is much faster than using the global memory, which is due to the 2D/3D caching.

This figure further shows that using the 16-bit storage format without textures is slower than using the 32-bit storage format. When the 16-bit format is used in conjunction with textures on GPUs all the data type conversions are done in hardware in the texture fetch units which is much faster than doing the conversion in the code. With CPUs using 32 bits is faster than 16 bits because although the CPU supports texture structures in OpenCL, the CPU does not have dedicated texture fetch units that can do the data type conversion in hardware as GPUs do.

Also, we noticed from tables 1 and 2 that NVIDIAs GPUs performed much worse on

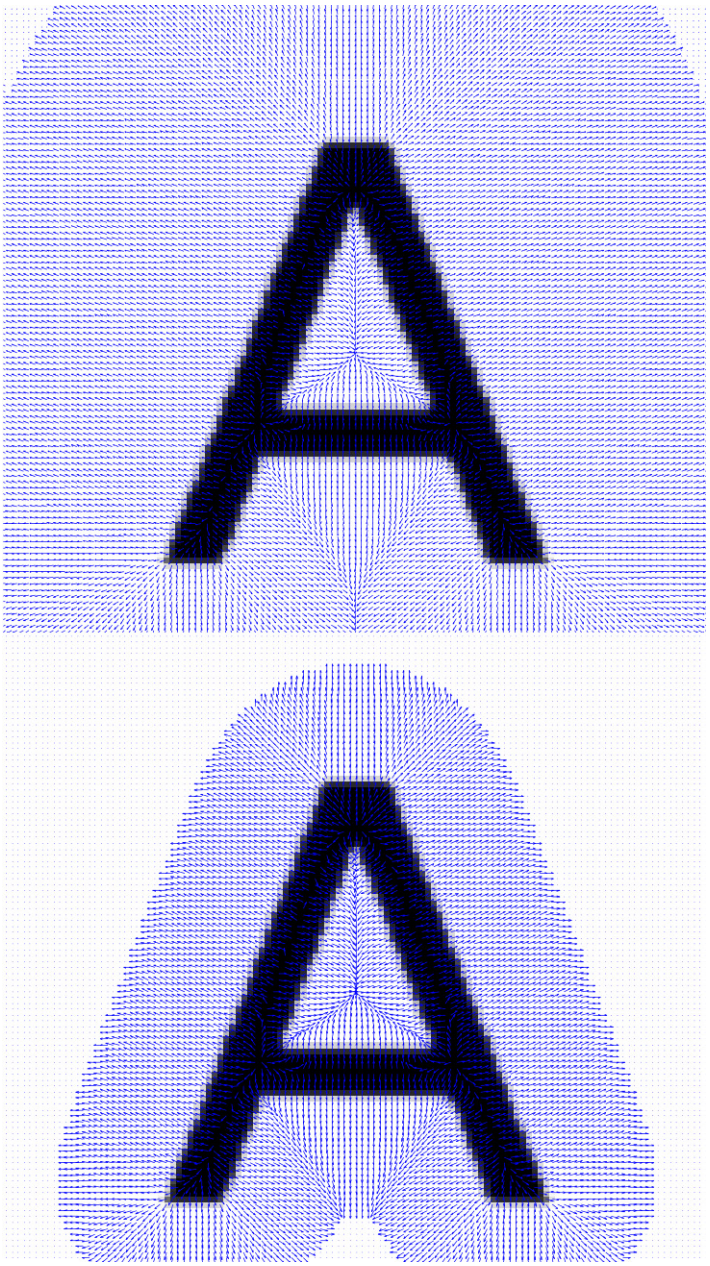

Figure 9: Normalized GVF vector field, run with the same number of iterations. The top is with 32-bit storage format and the bottom is 16-bit. These two images clearly show the reduced capture range when using 16-bit.

the 3D dataset than AMDs GPUs. The reason for this is that NVIDIA does not support writing to 3D textures in their OpenCL implementation. Thus, global memory had to be used. This memory, as we have explained earlier, is much slower than the texture memory.

Fig. 5 illustrates that the difference in execution time between using 32- and 16-bit storage formats increases as the image size increases. Thus the performance gain for 16-bit is biggest for large images and volumes, while for very small images it is almost insignif-

icant.

4.2 Memory usage

From the graph in Fig. 6, we can see that processing 2D images of typical sizes is no problem with modern GPUs that have 1GB memory and more. For 3D volumes a 1GB graphics card would manage to process a dataset, without any additional PCI express data transfer, of about 300^3 and 380^3 voxels for 32-bit and 16-bit data types respectively.

4.3 Relative accuracy

Relative accuracy tests were performed to measure the error by using the 16-bit storage format versus 32-bit. As seen in table 3 these tests showed that there was very little error in magnitude, but on average around 30 degrees angle error. The high angle errors was found to only be present for the very short vectors. In fact, the maximum magnitude of all vectors with angle error above 0.1 was $9.15 \cdot 10^{-4}$ on the 512x512 MRI brain scan image. The size of the angle error generally increases when the vector length decreases. Thus, this angle error may not be problematic for most applications. For instance, very short vectors will have very little pulling force on a snake.

Still, the capture range of using the 16-bit format is lower than 32-bit as seen in Fig. 9 where the resulting vector field has been normalized. Thus, the 16-bit storage format may not be sufficient for all applications.

5 Conclusions

In this paper, we presented a highly optimized parallel GPU implementation of Gradient Vector Flow written in OpenCL. Our implementation enables real-time execution of GVF for images of sizes up to 512^2 on modern GPUs. Since it is written in OpenCL, it can also run efficiently on multi-core CPUs. We investigated three different memory optimizations for GPUs. Our results show that using the texture memory with the 16-bit compressed floating point storage format and without shared memory is fastest on GPUs and can double the performance compared to an unoptimized GPU implementation. Relative accuracy measurements reveal that there is very little error in magnitude, but a high angle error between the 32- and 16-bit storage formats. However, the high angle errors are only present on very small vectors, and thus may not be a problem for most applications. The 16-bit storage format has also the advantage of allowing much larger volumes to reside completely in the limited memory on GPUs.

The source code of this implementation is available online at http://www.github.com/smistad/OpenCL-GVF/

Acknowledgments

Great thanks goes to the people of the High Performance Computing Lab at NTNU for all their assistance and to Mai Britt Engeness Mørk for her comments on the manuscript. 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

[1] AMD. AMD APP OpenCL Programming Guide. Technical report, AMD, 2011. http://developer.amd.com/sdks/amdappsdk/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf - Last accessed December 2011.

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

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

[4] Yujun Guo and Cheng-chang Lu. Multi-modality Image Registration Using Mutual Information Based on Gradient Vector Flow. *18th International Conference on Pattern Recognition (ICPR'06)*, pages 697–700, 2006.

[5] X Han, C Xu, and J.L. Prince. Fast numerical scheme for gradient vector flow computation using a multigrid method. *Image Processing, IET*, (1):48–55, 2007.

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

[7] Zhiyu He and Falko Kuester. GPU-Based Active Contour Segmentation Using Gradient Vector Flow. *Advances in Visual Computing*, pages 191–201, 2006.

[8] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, January 1988.

[9] NVIDIA. OpenCL Best Practices Guide. Technical report, 2009. http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf - Last accessed December 2011.

[10] Nilanjan Ray and Scott T Acton. Motion gradient vector flow: an external force for tracking rolling leukocytes with shape and size constrained active contours. *IEEE transactions on medical imaging*, 23(12):1466–78, December 2004.

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