Medical Image Analysis 20 (2015) 1–18

Contents lists available at ScienceDirect

Medical Image Analysis

journal homepage: www.elsevier.com/locate/media

Survey Paper

Medical image segmentation on GPUs – A comprehensive review

Erik Smistad [a,b,*], Thomas L. Falch [a], Mohammadmehdi Bozorgi [a], Anne C. Elster [a], Frank Lindseth [a,b]

[a] Norwegian University of Science and Technology, Sem Sælandsvei 7-9, 7491 Trondheim, Norway
[b] SINTEF Medical Technology, Postboks 4760 Sluppen, 7465 Trondheim, Norway

ARTICLE INFO

Article history:
Received 16 February 2014
Received in revised form 8 October 2014
Accepted 23 October 2014
Available online 2 December 2014

Keywords:
Medical
Image
Segmentation
GPU
Parallel

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.
© 2014 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/3.0/).

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 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. Fig. 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

* Corresponding author at: Sem Sælandsvei 7-9, 7491 Trondheim, Norway. Tel.: +47 73594475.
E-mail address: smistad@idi.ntnu.no (E. Smistad).

http://dx.doi.org/10.1016/j.media.2014.10.012
1361-8415/© 2014 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/3.0/).

GPU computing in several medical image processing areas such as image 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. Fig. 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

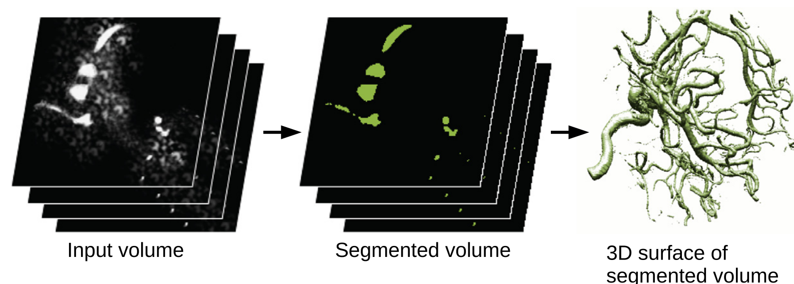

Fig. 1. Threshold-based segmentation of a computed tomography (CT) scan. The intensity of each voxel in the input volume is compared to a threshold. If it is higher than the threshold the voxel is segmented as part of the blood vessel. The segmentation result can be used to generate a 3D surface model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

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

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

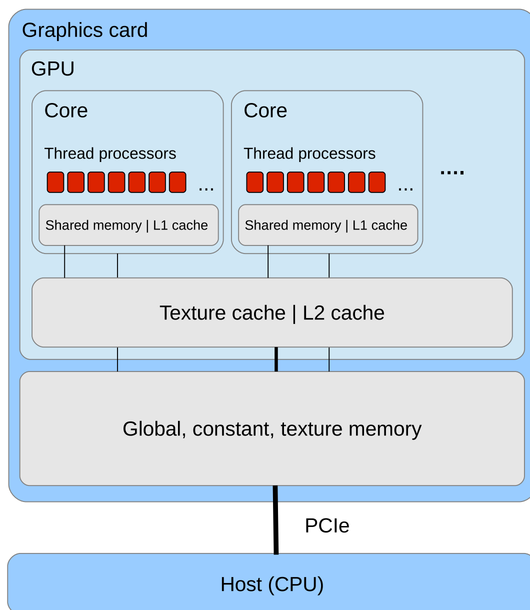

Fig. 2. General layout of a GPU and its memory hierarchy. The registers are private to each thread processor, the shared memory is private to each core, and the global, constant and texture memory is accessible from all thread processors. Note that the actual layout is much more complex and differ for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

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:

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. Fig. 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.

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.

Segmentation method	Data parallelism	Thread count	Branch div.	Memory usage	Synch.	GPU suitability
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 models						
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), Smistad et al. (2013)					
Dynamic image segmentation						
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)					

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 Fig. 1. In its simplest form, the method performs a binary segmentation using a single threshold T:

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

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.

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
```

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.

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. Fig. 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

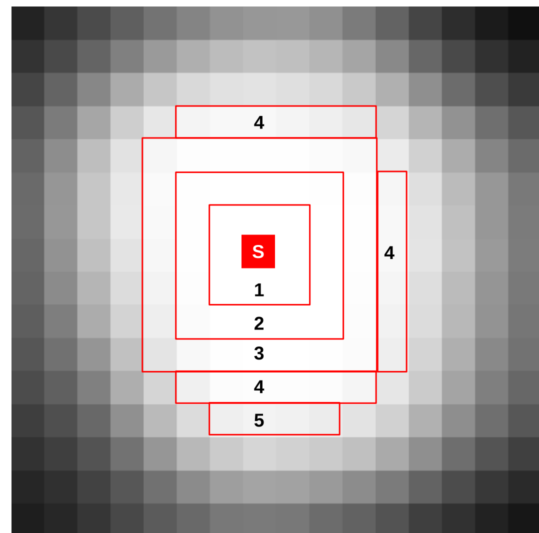

Fig. 3. Illustration of parallel region growing with double buffering. The pixel labeled S is the seed pixel. The numbers indicate at which iteration the pixels in the red regions are added to the final segmentation. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

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.

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
```

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 morphology 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 the center pixel in the structuring element is 1 as shown in Fig. 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 Fig. 5. In the resulting landscape, there are three types of points. These are determined by the 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*.

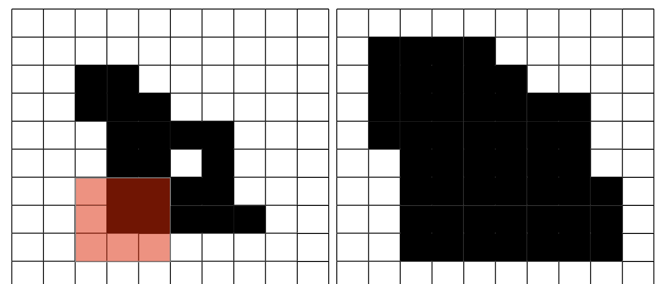

Fig. 4. Morphological dilation using a 3×3 square structuring element (shown in red to the left). Since the center pixel is 1, all 0 valued pixels under the structuring element are flipped to 1. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

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.

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.

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**
    *% **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

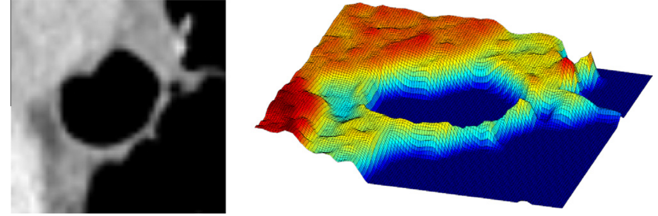

Fig. 5. Watershed segmentation. If the intensity values of the pixels of the images on the left are interpreted as heights, it will give create the landscape to the right.

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, 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 Fig. 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}:

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

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

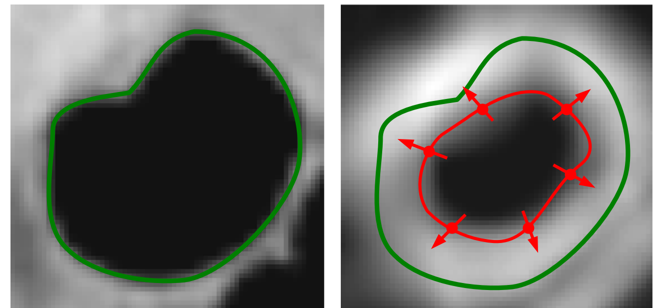

Fig. 6. Illustration of active contours. The image to the left is the input image, and the image to the right shows the gradient magnitude of the input image convolved with a Gaussian kernel. The red line superimposed on the right image is the active contour, which is driven towards the high gradient 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. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

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

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 Fig. 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 $v(s,t)$.

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

The Euler Eq. (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:

$$E(\vec{V}) = \int \mu|\nabla\vec{V}(\vec{x})|^2 + |\vec{V}(\vec{x}) - \vec{V}_0(\vec{x})|^2|\vec{V}_0(\vec{x})|^2 d\vec{x} \quad (5)$$

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 Fig. 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), 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.

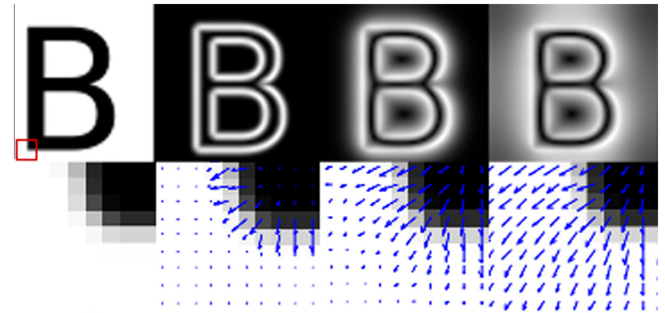

Fig. 7. Example of how gradient vector flow diffuses the gradients while preserving the large input gradients. The image to the far left is the input image. The next images depict the magnitude of the vector field after 0, 50 and 500 iterations. The bottom row shows the vector field of the zoomed area indicated by the small red square. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

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

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 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 Fig. 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 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.

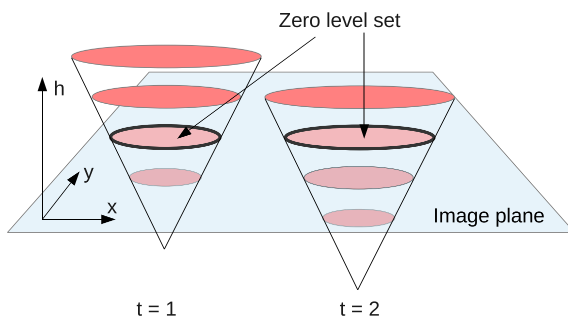

Fig. 8. Illustration of level set segmentation. A level set (hyper) surface is moved through the image plane x, y for each time step. The current contour of the segmentation is defined as the location where the height h to the (hyper) surface is zero. This is also called the zero level set. In this example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

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 x⃗ in parallel do
    Calculate first order derivatives
    Calculate second order derivatives
    Calculate gradient ∇φ(x⃗)
    Calculate curvature
    Calculate speed term F(x⃗, I)
    φ'(x⃗) ← φ(x⃗) + ΔtF(x⃗, I)|∇φ(x⃗)|
  end for
  φ = φ'
end for
for all voxels x⃗ in parallel do
  if φ(x⃗) ⩽ 0 then
    S(x⃗) ← 1
  else
    S(x⃗) ← 0
  end if
end for
```

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).

3.7.1. 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:

$$M(A,B) = H(B) - H(B|A) \quad (7)$$

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

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

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.

3.7.2. 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). 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 Fig. 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.

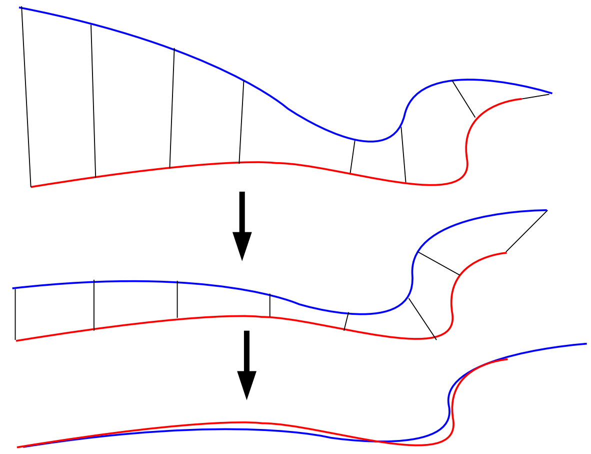

Fig. 9. Illustration of the iterative closest point method to align two lines. A set of points is chosen along each line. One of the point sets is iteratively moved and transformed to minimize the distance between each point set.

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 $\vec{x}_i = \{(x_1, y_1, z_1), \ldots, (x_N, y_N, z_N)\}$ of coordinates for each training sample i, the mean shape, $\vec{x}_{\text{mean}}$, can be calculated as the average location of each landmark:

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

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 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

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

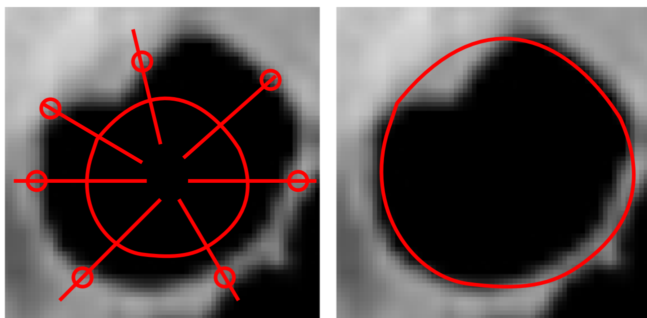

Fig. 10. The active shape model algorithm locates borders in a line search from each landmark point on the statistical shape model. A displacement is calculated and the shape is moved, scaled, rotated and deformed to best fit the identified border points. This is repeated until convergence.

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

Algorithm 6. Parallel PCA

Input: Matrix of landmarks for each shape:
$\mathbf{X} = [\vec{x}_1, \vec{x}_2, \ldots \vec{x}_M]$
Output: First c eigenvalues: $\phi_1, \phi_2, \ldots \phi_c$
$\mathbf{R} = \mathbf{X}$
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 $|\phi_k - \phi'_k| < \epsilon$
      break
    end if
    $\phi_k \leftarrow \phi'_k$
  end for
  Update residual matrix **R**
end for

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.

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 Fig. 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) 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 Fig. 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

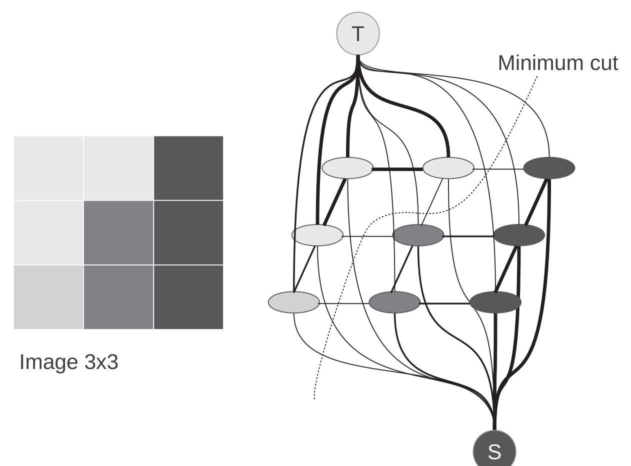

Fig. 11. Illustration of graph cut segmentation of a 3 × 3 image. The image to be segmented is shown to the left, its graph representation on the right. The thickness of the edges indicates their weight.

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 Fig. 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).

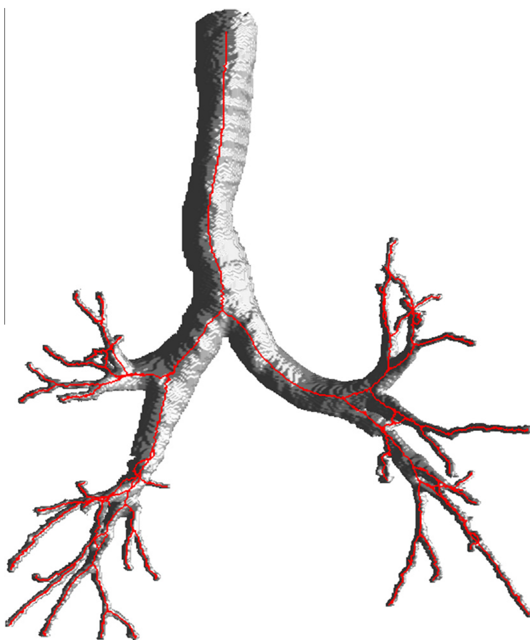

Fig. 12. Centerline, displayed in red, of the airway tree. The centerline was extracted using tube detection filters from computed tomography data and the segmentation was created using a region growing algorithm with the centerline as seeds. All the processing was done on the GPU as explained in Smistad et al. (2013). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

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 cannot 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 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 segmenta-

tion 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 320×240. Mateo Lozano and Otsuka (2008) and Lozano and Otsuka (2008) implemented face tracking on a stream of images with size 1024×768 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 findingsand 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.

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 (~60), large cache (~30 MB) and a lot of on-board memory (~16 GB). However, in contrast to GPUs, they have fewer thread processors (~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 Trans. Pattern Anal. Machine Intell. 16, 641–647. http://dx.doi.org/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. <developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/> (last accessed 12.04.13).

Ahlberg, J., 2002. An active model for facial feature tracking. EURASIP J. Appl. Signal Process., 566–571.

Alvarado, R., Tapia, J.J., Rolón, J.C., 2013. Medical image segmentation with deformable models on graphics processing units. J. Supercomput., doi:http://dx.doi.org/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, http://dx.doi.org/10.1145/1465482.146556.

Andrecut, M., 2009. Parallel GPU implementation of iterative PCA algorithms. J. Comput. Biol. 16, 1593–1599. http://dx.doi.org/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 Trans. Signal Process. 50, 174–188. http://dx.doi.org/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 Trans. Med. Imag. 21, 61–75. http://dx.doi.org/10.1109/42.993126.

Bauer, C., Bischof, H., 2008. Extracting curve skeletons from gray value images for virtual endoscopy. In: Proceedings of the 4th InternationalWorkshop 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. J. Roy. Stat. Soc. Ser. B 48, 259–302.

Besl, P.J., McKay, N.D., 1992. A method for registration of 3-D shapes. IEEE Trans. Pattern Anal. Machine Intell.

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 Trans. Visual. Comput. Graph. 18, 68–80. http://dx.doi.org/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. Med. Image Anal. 8, 217–231. http://dx.doi.org/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.F., Taylor, C., Cooper, D., Graham, K., 1995. Active shape models – their training and application. Comput. Vis. Image Understand. 61, 38–59.

Cootes, T., Edwards, G., Taylor, C., 2001. Active appearance models. IEEE Trans. Pattern Anal. Machine Intell. 23, 681–685.

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 Trans. Med. Imag. 21, 525–537. http://dx.doi.org/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, pp. 284–289. doi:http://dx.doi.org/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. Med. Image Anal. 17, 1073–1094. http://dx.doi.org/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. Comput. Methods Programs Biomed. 104, e45–e57. http://dx.doi.org/10.1016/j.cmpb.2010.10.009.

Frangi, A., Niessen, W., Vincken, K., Viergever, M., 1998. Multiscale vessel enhancement filtering. Med. Image Comput. Comput.-Assist. Intervent. 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 Trans. Pattern Anal. Machine Intell. 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 Perform. Comput., 197–208.

Hassouna, M., Farag, A., 2007. On the extraction of curve skeletons using gradient vector flow. In: IEEE 11th Int. Conf. Comput. Vis. IEEE, pp. 1–8, http://dx.doi.org/10.1109/ICCV.2007.440911.

He, Z., Kuester, F., 2006. GPU-based active contour segmentation using gradient vector flow. In: Advances in Visual Computing, pp. 191–201.

Heimann, T., Meinzer, H.P., 2009. Statistical shape models for 3D medical image segmentation: a review. Med. Image Anal. 13, 543–563. http://dx.doi.org/10.1016/j.media.2009.05.004.

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 Trans. Med. Imag. 28, 1251–1265. doi:http://dx.doi.org/10.1109/TMI.2009.2013851.

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

Holewinski, J., Pouchet, L.N., Sadayappan, P., 2012. High-performance code generation for stencil computations on GPU architectures. In: Proceedings of 26th ACM International Conference on Supercomputing – ICS '12. ACM Press, New York, New York, USA, pp. 311–320, http://dx.doi.org/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. In: 2011 IEEE 17th International Conference on Parallel and Distributed Systems, pp. 1016–1020. doi:http://dx.doi.org/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. <ark.intel.com/products/family/71840/Intel-Xeon-Phi-Coprocessors> (last accessed 03.07.14).

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 Trans. Visual. Comput. Graph. 15, 1505–1514. http://dx.doi.org/10.1109/TVCG.2009.178.

Jiménez, J., Miras, J.R.D., 2012. Three-dimensional thinning algorithms on graphics processing units and multicore CPUs. Concurr. Comput.: Practive Experience 24, 1551–1571. http://dx.doi.org/10.1002/cpe.

Jodoin, P.M., 2006. Markovian segmentation and parameter estimation on graphics hardware. J. Electron. Imag. 15, 033005. http://dx.doi.org/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). J. Real-Time Image Process. 7, 95–103. http://dx.doi.org/10.1007/s11554-010-0190-5.

Kalman, R., 1960. A new approach to linear filtering and prediction problems. J. Fluids Eng. 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:http://dx.doi.org/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. Int. J. Comput. Vis. 1, 321–331. http://dx.doi.org/10.1007/BF00133570.

Kauffmann, C., Piche, N., 2008. Cellular automaton for ultra-fast watershed transform on GPU. In: 2008 19th International Conference on Pattern Recognition, pp. 1–4. doi:http://dx.doi.org/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:http://dx.doi.org/10.1117/12.878377.

Kirbas, C., Quek, F., 2004. A review of vessel extraction techniques and algorithms. ACM Comput. Surv. 36, 81–121. http://dx.doi.org/10.1145/1031120.1031121.

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

Körbes, A., Vitor, G., 2011. Advances on watershed processing on GPU architecture. Math. Morphol. Its Appl. Image Signal Process. 6671, 260–271.

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.

Krissian, K., Malandain, G., Ayache, N., 2000. Model-based detection of tubular structures in 3D images. Comput. Vis. Image Understand. 80, 130–171. http://dx.doi.org/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. IEEE Computer Society, pp. 195–202, http://dx.doi.org/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 Trans. Visual. Comput. Graph. 10, 422–433. http://dx.doi.org/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. Med. Image Anal. 13, 819–845. http://dx.doi.org/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. In: 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, pp. 1–6. doi:http://dx.doi.org/10.1109/CVPRW.2008.4563101.

Liu, J., Sun, J., 2010. Parallel graph-cuts by adaptive bottom-up merging. In: 2010 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recogn. IEEE, pp. 2181–2188, http://dx.doi.org/10.1109/CVPR.2010.553989.

Lo, P., Ginneken, B.V., Reinhardt, J.M., de Bruijne, M., 2009. Extraction of airways from CT (EXACT'9), 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. J. Signal Process. Syst. 57, 285–295. http://dx.doi.org/10.1007/s11265-008-0250-2.

McCool, M.D., 2008. Scalable programming models for massively multicore processors. Proc. IEEE 96, 816–831. http://dx.doi.org/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, http://dx.doi.org/10.1109/CVPRW.2008.456310.

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 Trans. Med. Imag. 29, 583–597. http://dx.doi.org/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. <blogs.nvidia.com/2013/03/nvidia-ceo-updates-nvidias-roadmap/> (last accessed 16.04.14).

Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J., 2008. GPU computing. Proc. IEEE 96, 879–899. http://dx.doi.org/10.1109/JPROC.2008.917757.
Palágyi, K., Kuba, A., 1999. A parallel 3D 12-subiteration thinning algorithm. Graph. Models Image Process. 61, 199–221. http://dx.doi.org/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, http://dx.doi.org/10.1109/ITAB.2008.457054.
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, http://dx.doi.org/10.1109/CIT.2011.60.
Pham, D.L., Xu, C., Prince, J.L., 2000. Current methods in medical image segmentation. Biomed. Eng. 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 Trans. Med. Imag. 22, 986–1004. http://dx.doi.org/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. Med. Phys. 38, 2685. http://dx.doi.org/10.1118/1.3578605.
Qiu, D., May, S., Nüchter, A., 2009. GPU-accelerated nearest neighbor search for 3D registration. Comput. Vis. Syst., 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. Fundam. Inform. 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, http://dx.doi.org/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, http://dx.doi.org/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. Visual Comput. 27, 85–95. http://dx.doi.org/10.1007/s00371-010-0532-0.
Scholl, I., Aach, T., Deserno, T.M., Kuhlen, T., 2010. Challenges of medical image processing. Comput. Sci. – Res. Develop. 26, 5–13. http://dx.doi.org/10.1007/s00450-010-0146-9.
Serra, J., 1986. Introduction to mathematical morphology. Comput. Vis. Graph. and Image Process. 35, 283–305. http://dx.doi.org/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, http://dx.doi.org/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 Process. Mag. 27, 50–60. http://dx.doi.org/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. Comput. Methods Programs Biomed. 99, 133–146. http://dx.doi.org/10.1016/j.cmpb.2009.11.004.
Sherbondy, A., Houston, M., Napel, S., 2003. Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In: Visualization, VIS 2003. IEEE, pp. 171–176, http://dx.doi.org/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. Quant. Imag. Med. Surgery 2, 188–206. http://dx.doi.org/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 Trans. Med. Imag. 25, 385–405. http://dx.doi.org/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. J. Real-Time Image Process.
Smistad, E., Elster, A.C., Lindseth, F., 2013. GPU accelerated segmentation and centerline extraction of tubular structures from medical images. Int. J. Comput. Assist. Radiol. Surgery.
Song, M., Tao, D., Liu, Z., 2010. Image ratio features for facial expression recognition application. IEEE Trans. Syst. Man Cybernet. 40, 779–788.
Strandmark, P., Kahl, F., 2010. Parallel and distributed graph cuts by dual decomposition. In: IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recogn. IEEE. IEEE, pp. 2085–2092, http://dx.doi.org/10.1109/CVPR.2010.553988.
Sui, H., Peng, F., Xu, C., Sun, K., Gong, J., 2012. GPU-accelerated MRF segmentation algorithm for SAR images. Comput. Geosci. 43, 159–166. http://dx.doi.org/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 J. Select. Top. Signal Process. 6, 849–855. http://dx.doi.org/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. J. Supercomput. 58, 403–410. http://dx.doi.org/10.1007/s11227-011-0597-1.
Vincent, L., Soille, P., 1991. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. IEEE Trans. Pattern Anal. Machine Intell. 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:http://dx.doi.org/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, http://dx.doi.org/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. Comput. Vis. Image Understand. 117, 1610–1627. http://dx.doi.org/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 Trans. Image Process. 7, 359–369.
Zhang, J., 1992. The mean field theory in EM procedures for Markov random fields. IEEE Trans. Signal Process. 40, 2570–2583.
Zheng, Z., Zhang, R., 2012. A fast GVF snake algorithm on the GPU. Res. J. Appl. Sci. Eng. Technol. 4, 5565–5571.
Ziegler, G., Tevs, A., Theobalt, C., Seidel, H., 2006. On-the-fly point clouds through histogram pyramids. In: Proceedings of Vision, Modeling, and Visualization 2006. IOS Press, p. 137.